1. Podstawy matematyczne programowania grafiki 3D

Analizę zagadnień dotyczących grafiki komputerowej zaczniemy od elementów matematyki niezbędnych do zrozumienia omawianych tematów. To matematyka daje podstawy do opisu trójwymiarowego świata gier, obiektów występujących w tworzonych światach oraz relacje w nich zachodzące. W grafice komputerowej używa się wielu mechanizmów matematycznych poznanych w szkole średniej a także na wykładach z algebry. Omówimy zastosowanie w grafice macierzy, wektorów, kwaternionów, jak rozumieć układy współrzędnych, jak posługiwać się układami współrzędnych do identyfikowania obiektów w świecie, oraz relacji między nimi. Ze względu na skomplikowanie omawianych zagadnień, niektóre tematy omówimy pobieżnie, szczegółowe wyjaśnienie danego zagadnienia umieściliśmy przy konkretnym zastosowaniu.

1.1 Wektory (*Vectors*)

W grafice trójwymiarowej wektory posiadają wiele zastosowań: od przedstawiania kierunków po reprezentacje punktów w przestrzeni. Wektory, jakich będziemy używać, składają się z trzech składowych: $\begin{bmatrix} X & Y & Z \end{bmatrix}$. Można też spotkać się z wektorami z czterema składowymi $\begin{bmatrix} X & Y & Z & W \end{bmatrix}$. Ponieważ do operacji w układzie współrzędnych (przestrzeni trójwymiarowej) wystarczą nam trzy składowe, dlatego *W* będzie się równać *1*. Można w tym momencie zadać pytanie, dlaczego *W* ogóle zostało wspomniane o wektorach trójwymiarowych a wektorach czterowymiarowych. Jest to pewna niekonsekwencja towarzysząca grafice od dłuższego czasu. Tak jak wspominaliśmy, wystarczą tylko trzy komponenty do większości operacji, dopiero umiejscowienie wektora w tzw. układzie jednorodnym wymaga użycia współrzędnej *W*. Użycie zatem trójwymiarowych wektorów albo czterowymiarowych jest pozostawione programiście. My przyjmiemy, że nasze wektory są trójwymiarowe, mając w pamięci, że mogą się stać czterowymiarowe przez dodanie składowej $W = 1$.

Oto lista operacji na wektorach, które będą nam potrzebne:

- Dodawanie wektorów (*Vector Addition*). Przykładowy kod znajduje się na listingu 1.

```
CVector operator+ ( CVector& v, CVector& u )
{
          return CVector( v.m_fX + u.m_fX, v.m_fY + u.m_fY,
                               v.m_fZ + u.m_fZ );
}
```

Listing 1 – *CVector::operator+,* Źródło własne

- Odejmowanie wektorów (*Vector Subtraction*). Przykładowy kod znajduje się na listingu 2.

```
CVector operator- (CVector& v, CVector& u)
{
        return CVector( v.m_fX-u.m_fX, v.m_fY-u.m_fY, v.m_fZ-u.m_fZ );
}
```

Listing 2 – *CVector::operator-*, Źródło własne

- Iloczyn wektorowy (*Cross product*). Przykładowy kod znajduje się na listingu 3.

```
CVector vecCross(CVector &v, CVector &u)
{
        CVector vecTmp;
        vecTmp.m_fX = v.m_fY*u.m_fZ - v.m_fZ*u.m_fY;
        vecTmp.m_fY = v.m_fZ*u.m_fX - v.m_fX*u.m_fZ;
        vecTmp.m_fZ = v.m_fX*u.m_fY - v.m_fY*u.m_fX;
        return vecTmp;
}
```

Listing 3 – *vecCross*, Źródło własne

- Iloczyn skalarny (*Dot product*). Przykładowy kod znajduje się na listingu 4.

```
float vecDot(CVector &v, CVector &u)
{
        return (v.m_fX*u.m_fX + v.m_fY*u.m_fY + v.m_fZ*u.m_fZ);
}
```

Listing 4 – *vecDot*, Źródło własne

- Interpolacja liniowa (*Linear Interpolation, LERP*). Przykładowy kod znajduje się na listingu 5.

```
CVector vecLerp(CVector &v, CVector &u, float fT)
{
        CVector vecTmp;
        float fTemp = 1.0f - fT;
        vecTmp.m_fX = fTemp*v.m_fX + fT*u.m_fX;
        vecTmp.m_fY = fTemp*v.m_fY + fT*u.m_fY;
        vecTmp.m_fZ = fTemp*v.m_fZ + fT*u.m_fZ;
        return vecTmp;
}
```

Listing 5 – *vecLerp*, Źródło własne

- Obliczanie długości wektora (*Vector Length*). Przykładowy kod znajduje się na listingu 6.

```
float CVector::Length()
{
        return( (float)sqrtf(m_fX*m_fX + m_fY*m_fY + m_fZ*m_fZ) );
}
```

Listing 6 – *CVector::Length*, Źródło własne

- Normalizacja wektora (*Vector Normalization*). Przykładowy kod znajduje się na listingu 7.

```
void CVector::Normalize()
{
        float fLen = Length();

        if ( fLen == 0.0f )
            return;

        fLen = 1.0f / fLen;
        m_fX *= fLen;
        m_fY *= fLen;
        m_fZ *= fLen;
}
```

Listing 7 – *CVector::Normalize*, Źródło własne

Istotną kwestią jest zapisywanie wektorów. Na wielu wykładach matematycznych wektor jest pionowy i umieszczany za macierzą w operacji mnożenia macierzy przez wektor. W bibliotece *D3D* wektor jest „leżący” – poziomy. Co za tym idzie, kolejność przekształceń się zmienia. Mnożymy wektor przez macierz.

1.2 Macierze (*Matrix/Matrices*)

W grafice komputerowej używa się rożnych rodzajów macierzy. Najbardziej popularne są macierze o wymiarach *4x4*. Dlatego zajmiemy się właśnie macierzami *4x4*,

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

Macierze dają nam bardzo wygodny w użyciu system do zapisywania przekształceń.

Definicja 1 - *translacja.* Weźmy punkt $P[X \quad Y \quad Z]$ leżący w początku układu współrzędnych. Chcąc przesunąć punkt *P* o wektor $T[X \quad Y \quad Z]$zapisujemy tę operacje następująco (przyjmując, że P_x, P_y, P_z to wektor *P*, a T_x, T_y, T_z to wektor *T*):

$$x' = x + T_x$$
$$y' = y + T_y$$
$$z' = z + T_z$$

Ta samą operacje można zapisać przy pomocy macierzy.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Zatem wykonajmy operacje:

$$P' = P \cdot M$$

$$x' = (x \cdot M_{11}) + (y \cdot M_{21}) + (z \cdot M_{31}) + M_{41}$$

$$y' = (x \cdot M_{12}) + (y \cdot M_{22}) + (z \cdot M_{32}) + M_{42}$$

$$z' = (x \cdot M_{13}) + (y \cdot M_{23}) + (z \cdot M_{33}) + M_{43}$$

$$w' = (x \cdot M_{14}) + (y \cdot M_{24}) + (z \cdot M_{34}) + M_{44}$$

Jak widać, potrzebna tutaj jest współrzędna *W*. Ponieważ *W = 1* i najczęściej będzie zbędne zatem możemy je opuścić.

$$x' = (x \cdot M_{11}) + (y \cdot M_{21}) + (z \cdot M_{31}) + M_{41}$$
$$y' = (x \cdot M_{12}) + (y \cdot M_{22}) + (z \cdot M_{32}) + M_{42}$$
$$z' = (x \cdot M_{13}) + (y \cdot M_{23}) + (z \cdot M_{33}) + M_{43}$$

Zatem: $M_{11} = 1$, $M_{21} = 0$, $M_{31} = 0$, $M_{12} = 0$, $M_{22} = 1$, $M_{32} = 0$, $M_{13} = 0$, $M_{23} = 0$, $M_{33} = 1$, następnie: $M_{41} = T_x$, $M_{42} = T_y$, $M_{43} = T_z$.

Stąd:

$$x' = x \cdot 1 + T_x$$
$$y' = y \cdot 1 + T_y$$
$$z' = z \cdot 1 + T_z$$

Ostatecznie doszliśmy do zapisu wyjściowego. Przykładowy kod implementujący macierz przesunięcia znajduje się na listingu 8.

```
void CMatrix::SetMove(float x, float y, float z)
{
        LoadIdentity();
        matrix[3][0]=x;
        matrix[3][1]=y;
        matrix[3][2]=z;
}
```

Listing 8 – *CMatrix::SetMove*, Źródło własne

Przykładowe zastosowanie może być następujące:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

Definicja 2 - *skalowanie*. Weźmy sześcian położony w początku układu współrzędnych, o długości krawędzi równej *2*. Teraz zmniejszamy sześcian o połowę, tak aby długość krawędzi była *1*. Zatem, jeżeli mamy wartości skalowania zapisane w

wektorze $S(x \quad y \quad z)$ gdzie $x = y = z = 0.5$ (przyjmując że aktualnie przekształcany punkt ma współrzędne P_x, P_y, P_z a wartości skalowania to S_x, S_y, S_z), to:

$$x' = x \cdot S_x$$
$$y' = y \cdot S_y$$
$$z' = z \cdot S_z$$

A teraz to samo zapisujemy macierzą:

$$M = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Zatem:

$$P' = P \cdot M$$

Następnie obliczając:

$$x' = (x \cdot M_{11}) + (y \cdot M_{21}) + (z \cdot M_{31}) + M_{41}$$
$$y' = (x \cdot M_{12}) + (y \cdot M_{22}) + (z \cdot M_{32}) + M_{42}$$
$$z' = (x \cdot M_{13}) + (y \cdot M_{23}) + (z \cdot M_{33}) + M_{43}$$

Podstawiając wartości z macierzy dostajemy:

$$x' = x \cdot S_x + 0$$
$$y' = y \cdot S_y + 0$$
$$z' = z \cdot S_z + 0$$

czyli zapis, od którego zaczęliśmy.

Przykład:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przykładowa implementacja skalowania znajduje się na listingu 9.

```
void CMatrix::SetScale(float x,float y,float z)
{
	LoadIdentity();
	matrix[0][0]=x;
	matrix[1][1]=y;
	matrix[2][2]=z;
}
```

Listing 9 – *CMatrix::SetScale*, Źródło własne

Definicja 3 - *obroty*. Ponieważ obroty nie są tak łatwe do wyobrażenia, jak przesuwanie czy skalowanie, zatem wzory podane będziemy musieli na razie przyjąć jako pewniki. Każdy z nich zostanie w rozdziale poświęconym importowaniu geometrii/animacji szczegółowo omówiony.

Definicja 4 - *obrót względem osi X*. Oto wzór obracający punkt o zadany kąt jako parametr:

$$x' = x$$
$$y' = \cos\theta \cdot y - \sin\theta \cdot z$$
$$z' = \sin\theta \cdot y + \cos\theta \cdot z$$

A oto macierz reprezentująca przekształcenie obrotu względem osi *X*:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przykład:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dowód, że pierwszy zapis jest taki sam jak zapis $P' = P \cdot M$, gdzie P to punkt przekształcany P_x, P_y, P_z, będzie analogiczne do dwóch poprzednich z części poświęconych przesunięciu i skalowaniu. Przykładowy kod implementujący obrót względem osi *X* znajduje się na listingu 10.

```
void CMatrix::SetRotateX(float rad)
{
   float cosa=(float)cos(rad);
   float sina=(float)sin(rad);
   LoadIdentity();
   matrix[1][1]=cosa;
   matrix[2][2]=cosa;
   matrix[1][2]=sina;
   matrix[2][1]=-sina;
}
```

Listing 10 – *CMatrix::SetRotateX*, Źródło własne

Definicja 5 - *obrót względem osi Y*. Oto wzór obracający punkt o zadany kąt jako parametr:

$$x' = \cos\theta \cdot x + \sin\theta \cdot z$$
$$y' = y$$
$$z' = -\sin\theta \cdot x + \cos\theta \cdot z$$

oraz jego reprezentacja macierzowa:

$$M = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przykład:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Implementacja przykładowa obrotu względem osi *Y* znajduje się na listingu 11.

```
void CMatrix::SetRotateY(float rad)
{
   float cosa=(float)cos(rad);
   float sina=(float)sin(rad);
   LoadIdentity();
   matrix[0][0]=cosa;
   matrix[2][2]=cosa;
   matrix[2][0]=sina;
   matrix[0][2]=-sina;
}
```

Listing 11 – *CMatrix::SetRotateY*, Źródło własne

Definicja 6 - *obrót względem osi Z.* Ten wzór obraca punkt względem osi *Z* o zadany kąt:

$$x' = \cos\theta \cdot x - \sin\theta \cdot y$$
$$y' = \sin\theta \cdot x + \cos\theta \cdot y$$
$$z' = z$$

reprezentacja macierzowa:

$$M = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przykład:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Przykładowy kod implementujący obrót względem osi *Z* znajduje się na listingu 12.

```
void CMatrix::SetRotateZ(float rad)
{
   float cosa=(float)cos(rad);
   float sina=(float)sin(rad);
   LoadIdentity();
   matrix[0][0]=cosa;
   matrix[1][1]=cosa;
   matrix[0][1]=sina;
   matrix[1][0]=-sina;
}
```

Listing 12 – *CMatrix::SetRotateZ*, Źródło własne

Omówiliśmy podstawowe operacje, które można zapisać przy pomocy macierzy. Przyjmijmy teraz, że chcemy zmniejszyć obiekt o $\begin{bmatrix}0.5 & 0.5 & 0.5\end{bmatrix}$, obrócić o *180* stopni względem osi *Z*, a potem przesunąć o wektor $\begin{bmatrix}1.0 & -1.0 & 0.0\end{bmatrix}$. Przy zastosowaniu zwykłych wzorów uzyskalibyśmy całkiem duża liczba przekształceń. Przy przekształcaniu punkt, przy użyciu reprezentacji macierzowej przesunięcia, obrotu czy skalowania mielibyśmy tez dużą ilość operacji. Ale jest jeszcze inna możliwość. *Składanie przekształceń*. Mamy następujące operacje:

- *T* – macierz przesunięcia,
- *R* – macierz obrotu,
- *S* – macierz skalowania,
- *P* – punkt do przekształcenia.

Możemy złożyć macierze przekształceń w jedna macierz mnożąc kolejno macierze reprezentujące przekształcenia:

$$W = S \cdot R \cdot T$$

Po pomnożeniu punktu *P* przez macierz *W* dostaniemy dokładnie ten sam wynik, jakbyśmy punkt *P* najpierw pomnożyli przez macierz *S*, potem wynik przez macierz *R*, a potem wynik obu przekształceń przez macierz *T*.

Teraz dla każdego obiektu możemy przygotować jedną macierz, która będzie reprezentować wszystkie przekształcenia.

Pamiętać należy o tym, że mnożenie macierzy nie jest przemienne. Iloczyn *A* i *B* nie równa się iloczynowi *B* i *A*. Używając wektorów „poziomych" składanie macierzy powinno się zaczynać od pierwszej operacji do ostatniej. Zatem jeżeli:

$$W = S \cdot R \cdot T$$

Tak więc najpierw wykona się skalowanie *S*, potem obrót *R* a na końcu przesunięcie *T*.

1.3 Kwaterniony (*Quaternions*)

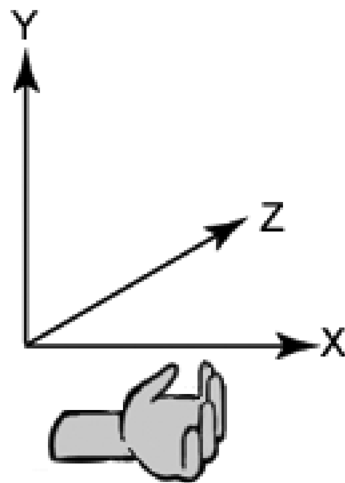
Kwaterniony w zapisie są podobne do wektorów. Służą one do zapisywania obrotu. Reprezentują oś obrotu i kąt obrotu wokół osi. Zapis $\begin{bmatrix}\begin{bmatrix}x & y & z\end{bmatrix} & \alpha\end{bmatrix}$ oznacza: $\begin{bmatrix}x & y & z\end{bmatrix}$ to oś obrotu, a α to kąt obrotu. Rotację macierzową względem trzech osi można zapisać jako kwaternion i odwrotnie: rotacje zapisana przy pomocy kwaternionu możemy zapisać jako macierz. Wiele programów do edycji i animowania obiektów przestrzennych używa kwaternionów do zapisu obrotów. My wybraliśmy reprezentacje

macierzową ponieważ interpolacja obrotów przed zapisaniem ich w macierzy jest prostsza do implementacji i zrozumienia niż interpolacja kwaternionów.

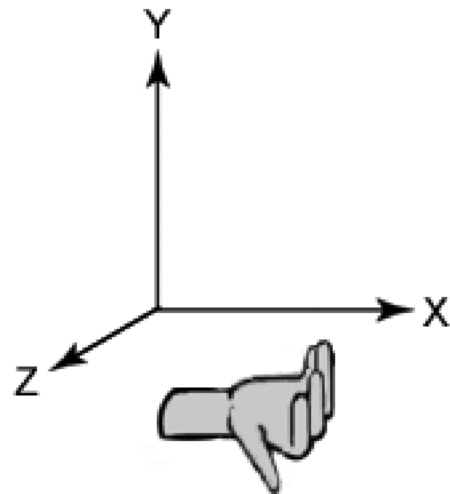
1.4 Układy współrzędnych

W grafice komputerowej wyróżnia się dwa rodzaje układów współrzędnych:

- Lewoskrętny układ współrzędnych *(Left-handed Cartesian Coordinates)* który przedstawiony jest na rysunku 1.
- Prawoskrętny układ współrzędnych *(Right-handed Cartesian Coordinates)* który przedstawiony jest na rysunku 2.



Rysunek 1 - *Left-handed*,



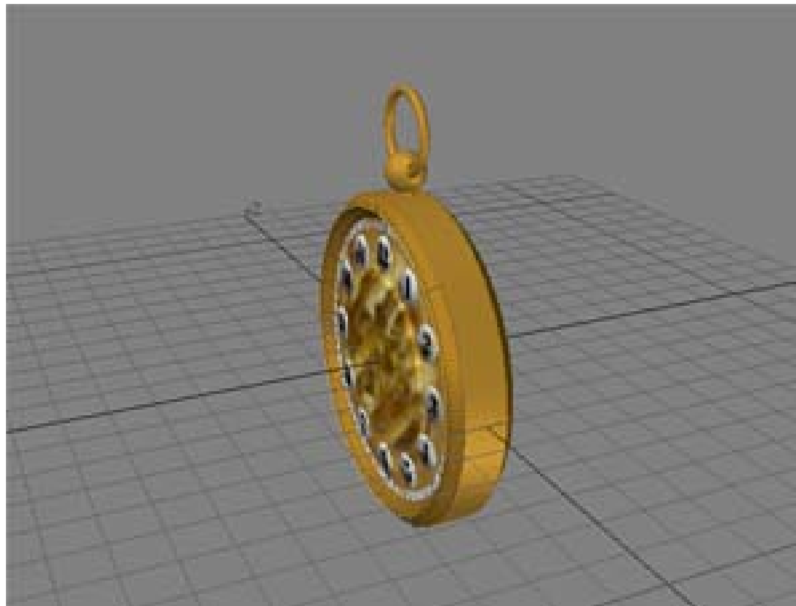
Rysunek 2 - *Right-handed,*

W większości systemów graficznych używa się lewoskrętnego układu. Czyli oś *Z* jest skierowana „od" obserwatora. Biblioteka *D3D*, którą będziemy się zajmować w kolejnych rozdziałach pracy, używa domyślnie lewoskrętnego układu.

W grafice każdy obiekt ma swój własny układ współrzędnych. Zazwyczaj umieszczony jest w jego początku co przedstawia ilustracja 1.

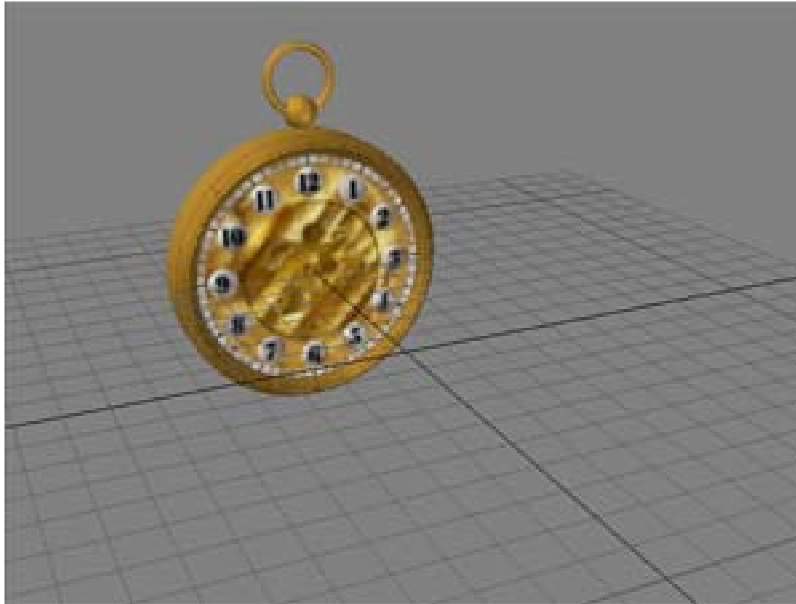
Ilustracja 1 - Obiekt w początku układu współrzędnych, Źródło własne

Obiekt zawsze obracany jest względem początku układu współrzędnych w którym się aktualnie znajduje. Przykład obrotu znajduje się na ilustracji 2.



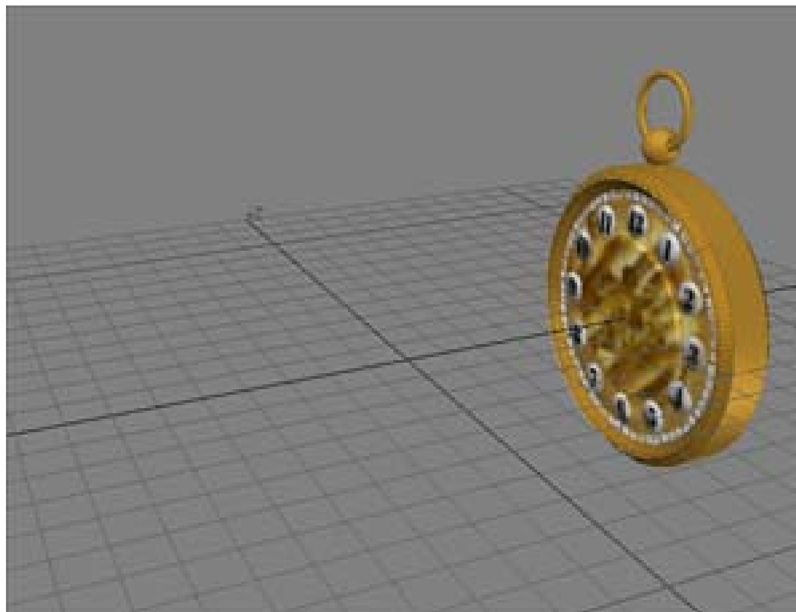
Ilustracja 2 - Obiekt w początku układu współrzędnych po obrocie, Źródło własne

Rozpatrzmy teraz przypadek kiedy obiekt nie znajduje się w początku układu współrzędnych co przedstawia ilustracja 3.



Ilustracja 3 - Obiekt przesunięty względem środka układu współrzędnych, Źródło własne

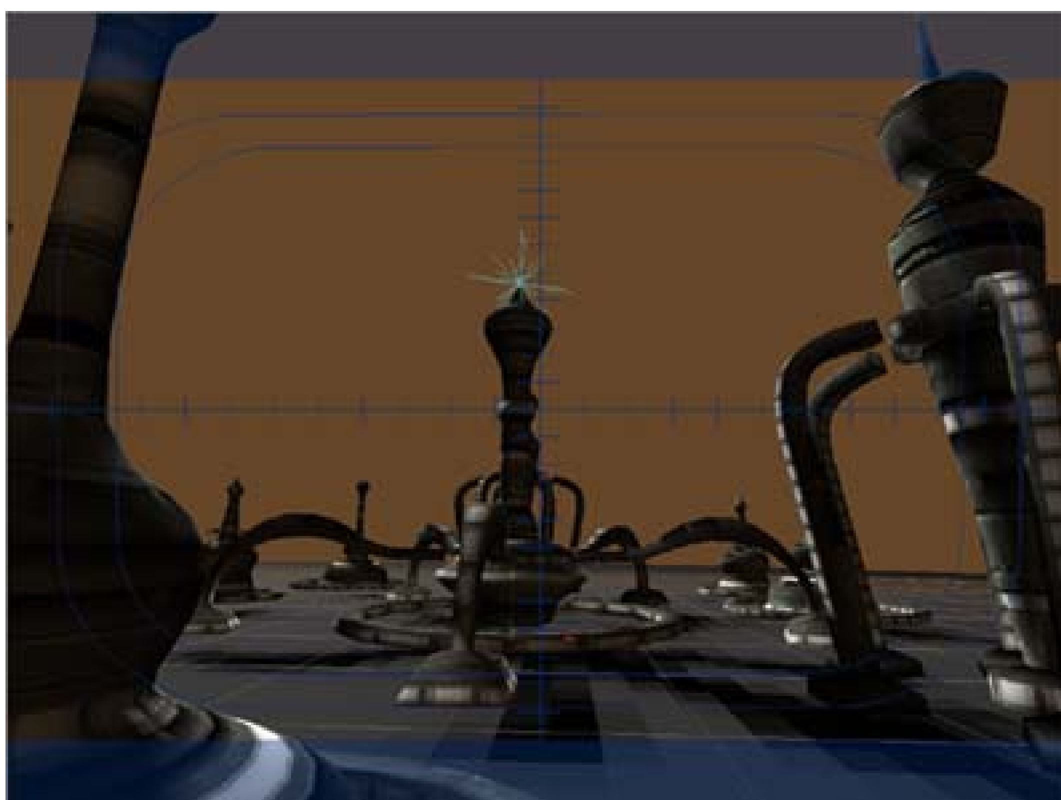
Sytuacje po obrocie przedstawia ilustracja 4.



Ilustracja 4 - Obiekt obrócony względem środka układu współrzędnych, Źródło własne

Kiedy budujemy świat *3D*, każdy obiekt jest w początku swojego układu współrzędnych. Ale kiedy umiejscowimy obiekty w konkretnej przestrzeni to wtedy znajdują się one w układzie współrzędnym tej przestrzeni. Dalej przestrzeń wraz z obiektami, źródłami światła, kamerami będziemy nazywać światem. Przykład świata przedstawia ilustracja 5.

Ilustracja 5 - Świat: obiekty, kamera, światło w jednym układzie współrzędnych, Źródło własne

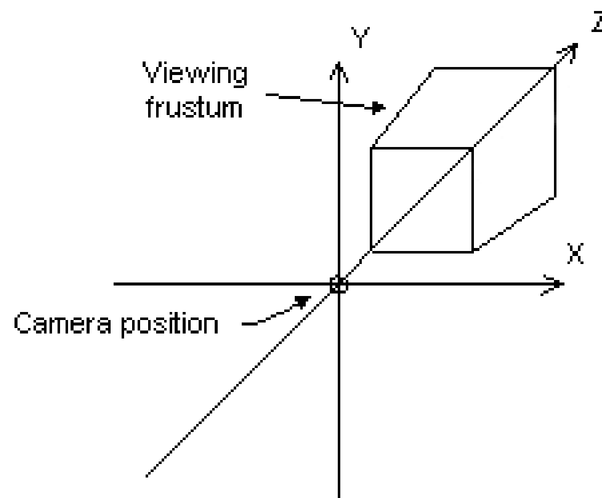


Ilustracja 6 – Świat z punktu widzenia kamery, Źródło własne

Następnym ważnym układem współrzędnych jest układ kamery. Kamera reprezentuje widza który patrzy na nasz świat. Pozycja kamery to pozycja widza. To co „widzi" kamera zaznaczona zielonym kolorem na ilustracji 5 widać na ilustracji 6.

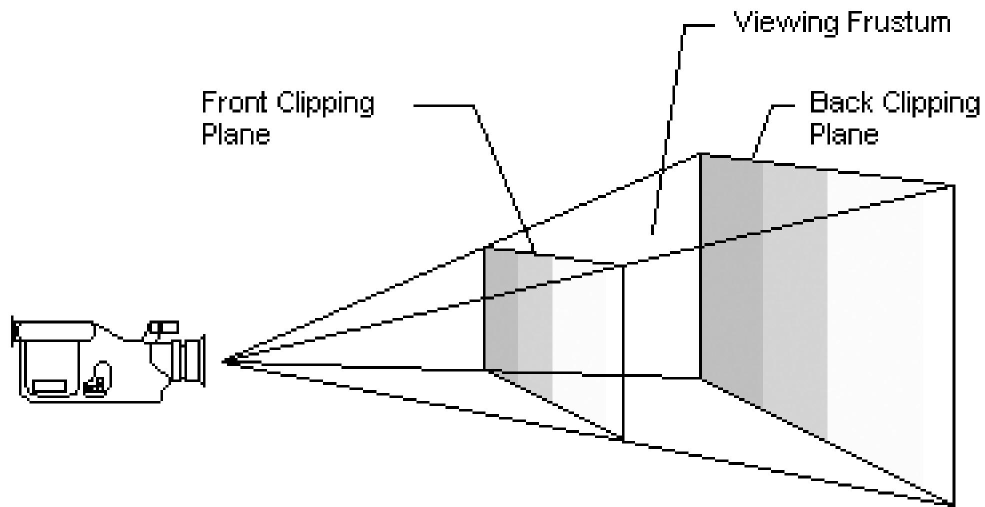
1.5 Kamera

W układzie współrzędnych kamery, kamera – obserwator, stoi dokładnie w początku układu współrzędnych czyli w punkcie *(0, 0, 0)*. Rysunek 3 przedstawia układ współrzędnych kamery oraz ostrosłup widzenia.

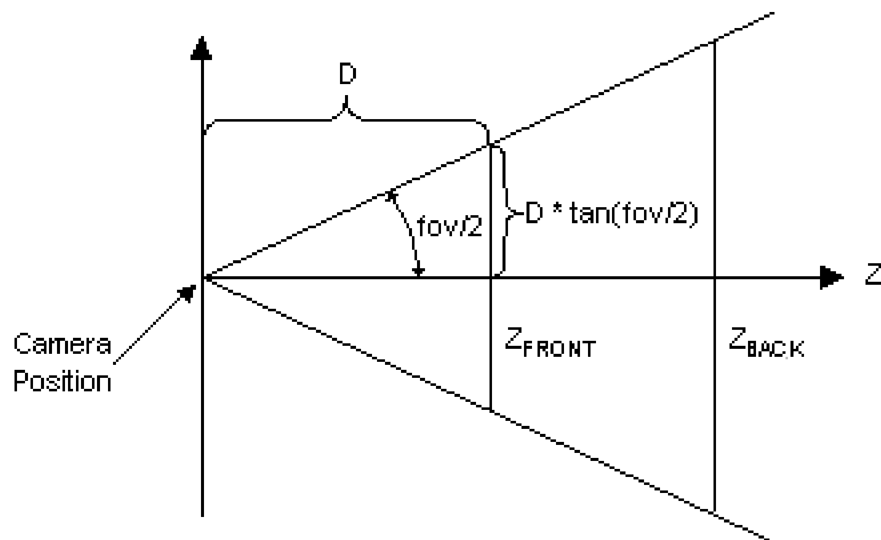


Rysunek 3 – Kamera i ostrosłup widzenia, Źródło: *DirectX9 SDK*

Kamera „widzi" wszystko co jest zawarte w ostrosłupie widzenia (*Viewing Frustum)*. Rysunek 4 przedstawia kamerę oraz opisuje ostrosłup widzenia. Rysunek 5 opisuje matematycznie ostrosłup widzenia.



Rysunek 4 – Kamera i ostrosłup widzenia, Źródło: *DirectX9 SDK*



Rysunek 5 – Matematyczny opis ostrosłupa widzenia, Źródło *DirectX SDK*

Definiując kamerę podaje się także oprócz pozycji takie parametry jak:

- aspekt, czyli stosunek wysokości kamery do szerokości (np. dla *640x480* będzie to *480/640* czyli *0.75*),
- pole widzenia (ang. *Field of view – FOV*), kąt, który widzi kamera,
- bliska płaszczyzna obcinania (*Near clipping plane* albo *Front clipping plane*) – odległość od kamery po której następuje obcięcie geometrii,

- daleka płaszczyzna obcinania (*Far clipping plane* albo *Back clipping plane*) – odległość od kamery przed która następuje obcięcie geometrii.

Definiowanie ostrosłupa widzenia jest odwzorowaniem rzeczywistości. Człowiek nie widzi wszystkiego, co się dzieje dookoła niego. Zazwyczaj jest to jakaś część horyzontu np. ograniczona kątem *150* stopni.

Definiowanie płaszczyzn obcinania ma na celu ustalenie dokładności obliczeń związanych z odległością obiektu od kamer. Mała odległość między płaszczyznami obcinania to duża dokładność. Duża odległość równa się małej dokładności. Dokładniej ten aspekt omówimy sobie dalej, analizując algorytmy znajdowania widocznych powierzchni.

1.6 Przechodzenie między układami współrzędnych

Omówiliśmy więc układ obiektu, układ świata i układ kamery, a teraz powiemy jak wszystkie układy razem ze sobą współpracują. Jeżeli mamy obiekt, który w jego własnym układzie np. w punkcie *(0, 0, 0)*, a chcemy, aby w układzie świata znajdował się np. w punkcie *(-1, 0, 10)* i był obrócony o *180* stopni, to musimy sobie zbudować macierze:

- T – przesunięcie o $\begin{bmatrix}-1 & 0 & 10\end{bmatrix}$,
- R – obrót o *180* stopni,

składamy przekształcenia:

$$W = R \cdot T$$

najpierw jest obrót a potem przesunięcie, inaczej obiekt obróciłby się względem innego punktu, tak jak było pokazane na ilustracjach wcześniej. Macierz W przenosi nam obiekt do układu świata. To samo robimy z kamerą, światłami itd.

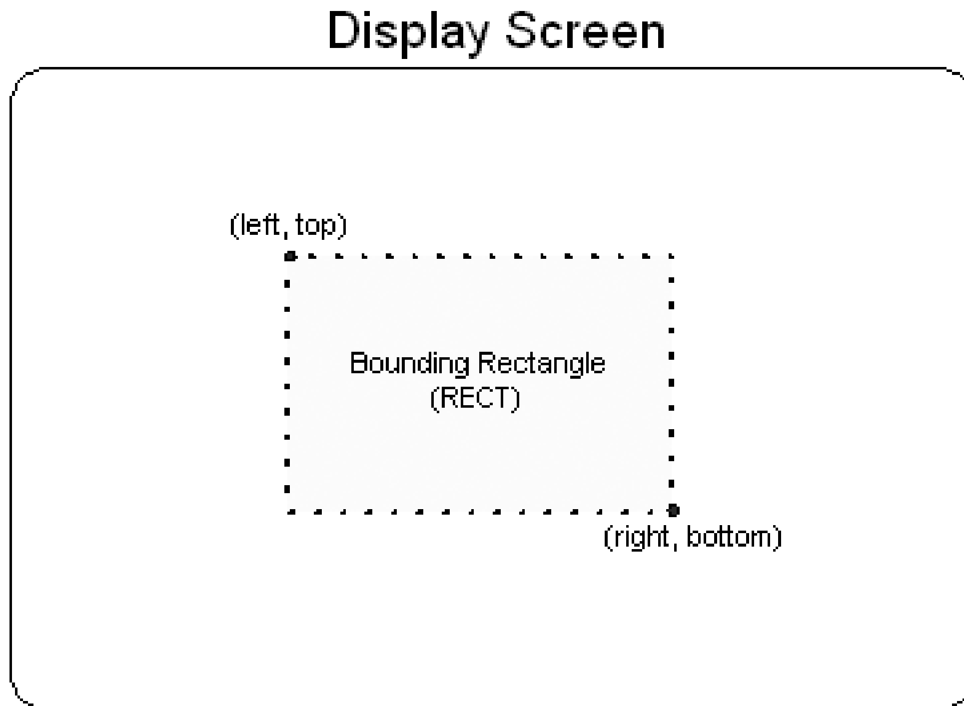
Teraz, kiedy chcemy aby przenieść układ świata do układu kamery, musimy odwrócić macierz kamery. Zatem macierz W musimy pomnożyć przez macierz kamery K tyle, że odwróconą:

$$W = W \cdot K.Invert()$$

Macierz W przenosi obiekt z układu obiektu do układu kamery.

1.7 Układ współrzędnych ekranu i układ współrzędnych jednorodnych

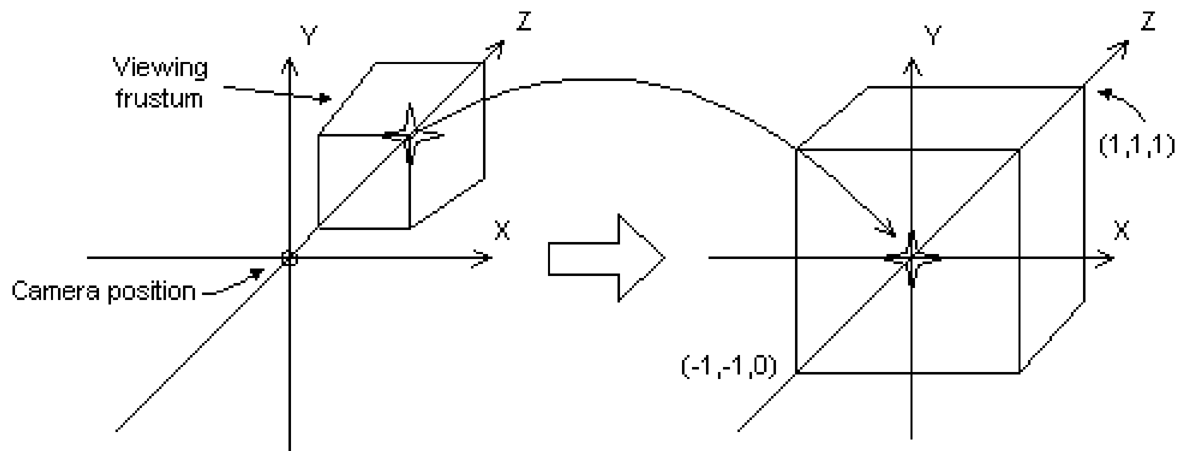
Układ współrzędnych ekranu. Ponieważ monitor potrafi wyświetlać tylko obraz dwuwymiarowy, musimy nasz świat zrzutować na płaski ekran:



Rysunek 6 - Ekran monitora i okno wyświetlania, Żródło *DirectX SDK*

Jak widać na rysunku 6 który przedstawia ekran monitora (*Display Screen)*, miejsce w którym chcemy wyświetlić nasz obraz jest definiowane przez prostokąt (*Bounding Rectangle*) opisane przez lewy górny punkt (*left* top) i prawy dolny (*right bottom*). W bibliotece *D3D* struktura definiująca obszar wyświetlania nazywa się *Viewport*.

Układ współrzędnych jednorodnych. Czasem może się zdarzyć, że obiekt wyjdzie po za ostrosłup widzenia. Wtedy musimy dokonać obcięcia obiektu tak, aby mieścił się on w ostrosłupie. Biblioteki graficzne zazwyczaj używają do tego celu układu współrzędnych jednorodnych. Operacja polega na przekształceniu ostrosłupa widzenia do prostopadłościanu szerokiego od *–1* do *1* i wysokiego od *–1* do *1* oraz głębokiego na *1*. W takim układzie można prosto sprawdzić, czy któraś współrzędna wyszła poza sześcian. Rysunek 7 przedstawia przekształcenie ostrosłupa widzenia to układu współrzędnych jednorodnych.



Rysunek 7 - Ostrosłup widzenia a układ współrzędnych jednorodnych, Źródło *DirectX SDK*

Wzór macierzy reprezentującej te przekształcenie ma postać:

$$M = \begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

Gdzie:

$$w = ctg\left(\frac{fov_w}{2}\right) \qquad h = ctg\left(\frac{fov_h}{2}\right) \qquad Q = \frac{Z_f}{Z_f - Z_n} \qquad w = \frac{2 \cdot Z_n}{V_w} \qquad h = \frac{2 \cdot Z_n}{V_h}$$

Oraz:

- *FOVw* – kąt widzenia w poziomie,
- *FOVh* – kąt widzenia w pionie,
- *Zf* – daleka płaszczyzna obcinania,
- *Zn* – bliska płaszczyzna obcinania,
- *Vw* – szerokość widzenia kamery (np. *640*),
- *Vh* – wysokość widzenia kamery (np. *480*).