# Macierze w OpenGL

1. Pojęciem macierz będziemy określali w tym tekście macierze wymiaru $4 \times 4$. Pojęciem wektor będziemy określali wektory posiadające 4 współrzędne. Przy wszystkich innych macierzach i wektorach wymiar będzie jawnie podany. Wektor może być, w sposób jednoznaczny, utożsamiany z macierzą. Można to zrobić na dwa sposoby:

   (a) *wektor wierszowy* - macierz wymiaru $1 \times 4$: $x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}$

   (b) *wektor kolumnowy* - macierz wymiaru $4 \times 1$: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$

2. Istnieją dwie podstawowe operacje w OpenGL związane z macierzami:

   (a) mnożenie macierzy przez macierz - w wyniku powstaje nowa macierz

   (b) mnożenie macierzy przez wektor w wyniku powstaje nowy wektor

   W bibliotece OpenGL oba te mnożenia wykonywane są **lewostronnie**, tzn. domnożenie do danej macierzy $M$ macierzy $N$, będzie miało postać $NM$ (a nie $MN$). Podobnie dla wektora $P$ i macierzy $M$, zostanie wykonane mnożenie $PM$, tzn. wektor $P$, musi być wektorem wierszowym[1].

$$\mathbf{M} = \begin{bmatrix} \mathbf{m_{11}} & \mathbf{m_{12}} & \mathbf{m_{13}} & \mathbf{m_{14}} \\ \mathbf{m_{21}} & \mathbf{m_{22}} & \mathbf{m_{23}} & \mathbf{m_{24}} \\ \mathbf{m_{31}} & \mathbf{m_{32}} & \mathbf{m_{33}} & \mathbf{m_{34}} \\ \mathbf{m_{41}} & \mathbf{m_{42}} & \mathbf{m_{43}} & \mathbf{m_{44}} \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{w} \end{bmatrix}$$

$$\mathbf{P} \bullet \mathbf{M} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} & \mathbf{w} \end{bmatrix} \bullet \begin{bmatrix} \mathbf{m_{11}} & \mathbf{m_{12}} & \mathbf{m_{13}} & \mathbf{m_{14}} \\ \mathbf{m_{21}} & \mathbf{m_{22}} & \mathbf{m_{23}} & \mathbf{m_{24}} \\ \mathbf{m_{31}} & \mathbf{m_{32}} & \mathbf{m_{33}} & \mathbf{m_{34}} \\ \mathbf{m_{41}} & \mathbf{m_{42}} & \mathbf{m_{43}} & \mathbf{m_{44}} \end{bmatrix}$$

   Gdyby mnożenie było prawostronne (tzn. $MP$), to wektor $P$, musiałby być wektorem kolumnowym. Lewo bądź prawostronność mnożenia jest jedynie ustaloną konwencją. W materiałach wykładowch będziemy stosowali konwencję prawostronną.

[1] Aby mnożenie macierzy $A$ wymiaru $m \times n$ przez macierz $B$ wymiaru $p \times q$ było wykonywalne w porządku $AB$ musi być spełniony warunek $n = p$, tzn. ilość kolumn macierzy $A$ musi być równa ilości wierszy macierzy $B$. W wyniku otrzymujemy macierz wymiaru $m \times q$

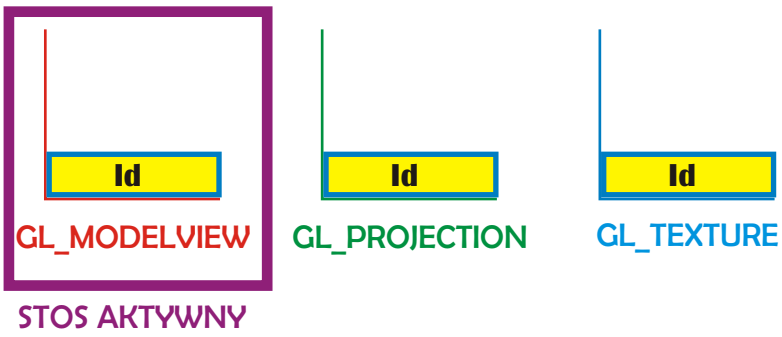
3. Macierze w OpenGL są przechowywane na stosach macierzowych. Mamy (w wersji OpenGL 1.1) trzy stosy przechowujące macierze:

   - *stos modelowania*, identyfikowany przez stałą **GL_MODELVIEW**,
   - *stos rzutowania* , identyfikowany przez stałą **GL_PROJECTION**,
   - *stos tekstur*, identyfikowany przez stałą **GL_TEXTURE**.

   Po fazie inicjacji biblioteki (**glutInit()**) na każdym z tych stosów znajduje się macierz identycznościowa **Id**.[2]

$$\mathbf{Id} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   W danej chwili czasowej dostępny jest tylko jeden stos, tzw. **stos aktywny**. Pozostałe stosy są zablokowane. Po fazie inicjacji aktywny jest stos modelowania (Rysunek 3).



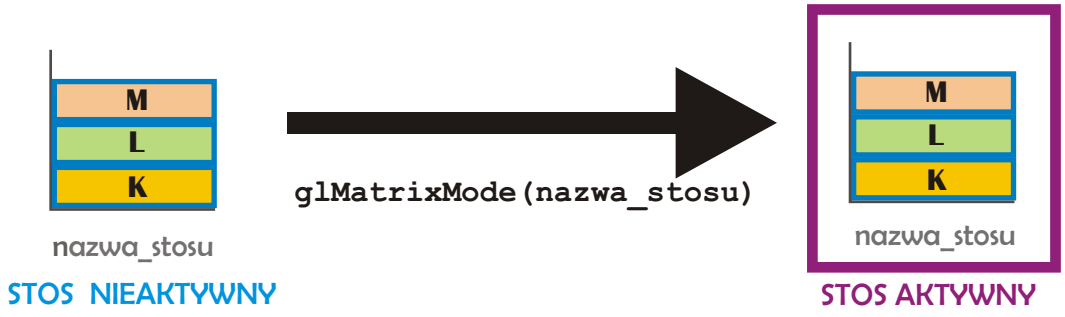

Rysunek 3: Stan stosów macierzowych po fazie inicjacji biblioteki OpenGL

4. Zmiana bieżącego stosu aktywnego na inny odbywa się przez wywolanie funkcji

   **glMatrixMode(nazwa_stosu)**,

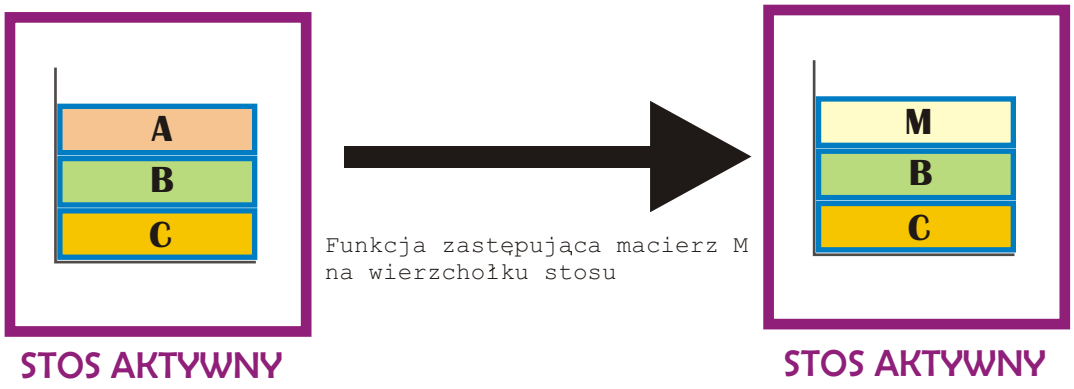
[2] Oczywiście mnożenie przez macierz identycznościową niczego nie zmienia, tzn. **Id M=M Id=M** oraz **P Id=P**.

gdzie **nazwa_stosu** przyjmuje wartość jednej ze stałych identyfikujących stosy, tzn. **GL_MODELVIEW, GL_PROJECTION** lub **GL_TEXTURE**. Po jej wywołaniu stos o nazwie **nazwa_stosu** jest nowym stosem aktywnym.




5. Nowa macierz, (nazwijmy ją M dla ustalenia uwagi) zdefiniowana przez użytkownika, może zostać przekazana przez funkcje OpenGL do aktywnego stosu (przypominamy, że pozostałe stosy są niedostępne) w dwojaki sposób:

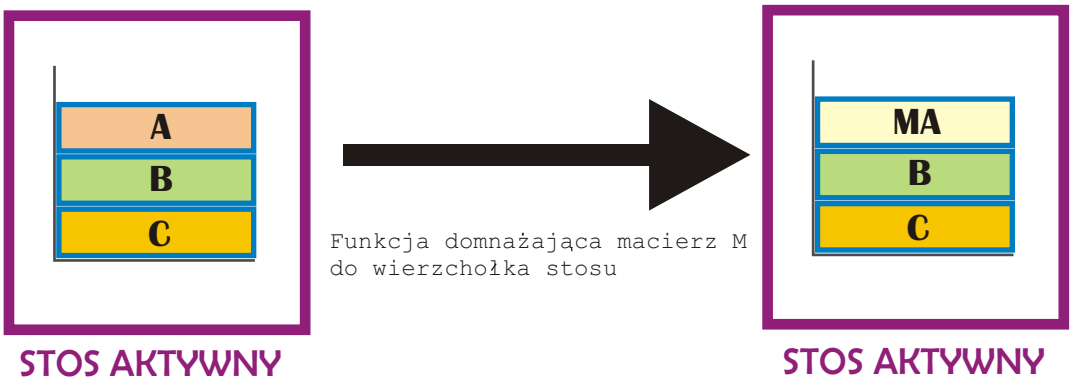
   (a) funkcja powoduje zastąpienie wierzchołka aktywnego stosu macierzą $M$




   Istnieją dwie funkcje działające według tego schematu

   - **glLoadMatrix∗(M)**, gdzie **M** jest macierzą zdefiniowaną przez programistę, a $*$ ma wartość **f** lub **d**, zależnie od używanego typu elementów macierzy.
   - **glLoadIdentity()** - szczególny przypadek funkcji **glLoadMatrix(M)**, gdzie **M=Id**, więc parametr jest zbędny.

   (b) funkcja powoduje przemnożenie wierzchołka aktywnego stosu przez macierz $M$, tzn. jeżeli na wierzchołku znajdowała się inna macierz $A$, to po tej operacji znajduje się na nim nowa macierz, będąca iloczynem macierzy $MA$ (przypominamy, że mnożenie odbywa się lewostronnie).




Istnieje osiem funkcji działających według tego schematu

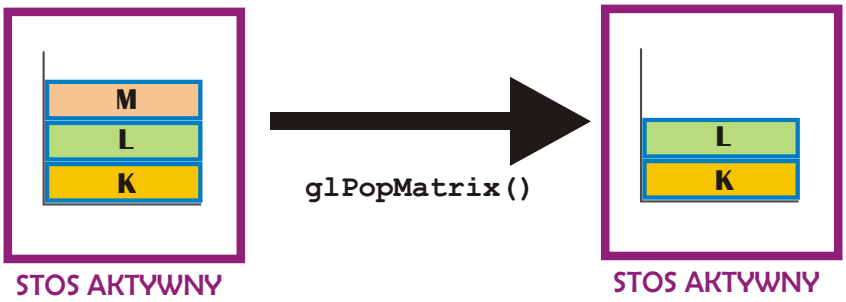
- **glMultMatrix∗(M)**, gdzie **M** jest macierzą zdefiniowaną przez programistę, a ∗ ma wartość **f** lub **d**, zależnie od używanego typu elementów macierzy,
- **glTranslate∗(tx, ty, tz)**, gdzie **tx, ty, tz** są współrzędnymi trójwymiarowego *wektora translacji*, a ∗ ma wartość **f** lub **d**, zależnie od używanego typu elementów macierzy. Na podstawie tych parametrów tworzona jest *macierz translacji* **T** (pełniąca rolę macierzy **M** w schemacie). Macierz ta wykorzystywana jest do przesuwania punktów w przestrzeni sceny (zob. niżej).
- **glRotate∗(alfa, rx, ry, rz)**, gdzie **alfa** jest *kątem obrotu* wokół *osi obrotu* określonej przez trójwymiarowy wektor [**rx, ry, rz**], a ∗ ma wartość **f** lub **d**, zależnie od używanego typu elementów macierzy. Na podstawie tych parametrów tworzona jest *macierz obrotu* `R` (pełniąca rolę macierzy **M** w schemacie). Macierz ta wykorzystywana jest do obracania punktów w przestrzeni sceny (zob. niżej).
- **glScale∗(sx, sy, sz)**, **sx, sy, sz** są *współczynnikami skali*, a ∗ ma wartość **f** lub **d**, zależnie od używanego typu elementów macierzy. Na podstawie tych parametrów tworzona jest *macierz skalowania* **S** (pełniąca rolę macierzy **M** w schemacie). Macierz ta wykorzystywana jest do skalowania wektorów identyfikujących punkty w przestrzeni sceny (zob. niżej).
- **gluPerspective(alfa, aspect, f, b)** - (opis parametrów w pliku **ogl1.pdf**) Funkcja definiuje parametry bryły widzenia w rzucie perspektywicznym i powoduje wygenerowanie *macierzy rzutowania perspektywicznego* (pełniącej rolę macierzy **M** w schemacie), która odpowiada za zrzutowanie punktów sceny na płaszczyznę obrazu (zob. niżej).
- **glFrustum(left, right, bottom, top, f, b)** - inny sposób wygenerowania bryły widzenia perspektywicznego (na razie nie będziemy się nią zajmowali). Działa jak **gluPerspective()**.
- **glOrtho(left, right, bottom, top, f, b)** - funkcja ta generuje bryłę widzenia równoległego (na razie nie będziemy się nią zajmowali). Działa podobnie do **gluPerspective()**.

- **gluLookAt(ex,ey,ez, px,py,pz, ox,oy,oz)** - (opis parametrów w pliku **ogl1.pdf**) Funkcja ta definiuje parametry obserwatora i generuje *macierz obserwatora* (pełniącą rolę macierzy **M** w schemacie), która odpowiada za przekształcenie ogólnej bryły widzenia do postaci kanonicznej, tzn. w przypadku rzutu perpektywicznego do takiego jej położenia, że obserwator jest w punkcie $(0, 0, 0)$, kierunek obserwacji wyznacza wektor **[0,0,-1]** (ujemna część osi Z), a oś pionowa na rzutni jest określona przez wektor **[0,1,0]** (dodatnia część osi Y).

6. Istnieje jeden sposób **usuwania wierzchołka** aktywnego stosu. Odpowiada za to funkcja

   **glPopMatrix().**

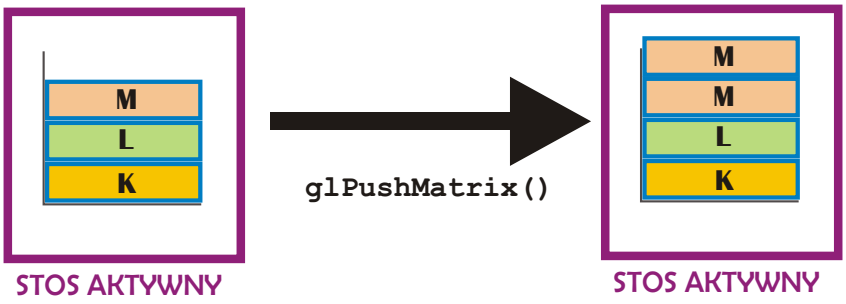
   Oczywiste jest, że nie trzeba podawać żadnego jej parametru, gdyż wierzchołek tosu aktywnego jest określony jednoznacznie.




7. Istnieje jeden sposób na zwiększenie ilości elementów na aktywnym stosie (żadna powyżej opisana operacja tego nie robi, a po fazie inicjacji na każdym stosie znajduje się tylko jedna macierz). Odpowiada za to funkcja

   **glPushMatrix(),**

   która wykonuje **kopię wierzchołka** stosu aktywnego.




   Nie ma żadnej funkcji wkładającej na stos dowolną, zdefiniowaną przez programistę macierz.

8. Punkty w trójwymiarowej przestrzeni sceny są opisywane w czterowymiarowym układzie *współrzędnych jednorodnych* `XYZW`, przez wektor **[x y z w]**, gdzie współrzędna **w** zwykle przyjmuje wartość równą 1. Punkty te mogą być poddawane różnym przekształceniom (obroty, translacje itp.) opisywanym przez macierze. Każde przekształcenie odbywa się przez **pomnożenie** wektora punktu **P=[x y z w]** przez macierz przekształcenia **M**, tzn. wykonywana jest operacja **PM**, dająca w wyniku nowy punkt (obraz punktu **P**) **P'=[x' y' z' w']** (szczegóły zostaną wyjaśnione na wykładzie).

9. Wiadomo już (ćwiczenia nr 1), że w bibliotece OpenGL wystarczy określić co się dzieje z punktami, będącymi wierzchołkami wielokątów, gdyż one definiują struktury bardziej złożone (wielokąty i obiekty). Zatem kluczowa jest odpowiedź na pytanie:

   *Co biblioteka OpenGL robi w momencie, gdy pojawia się w kodzie komenda* **glVertex3f(x,y,z)**[3] ?

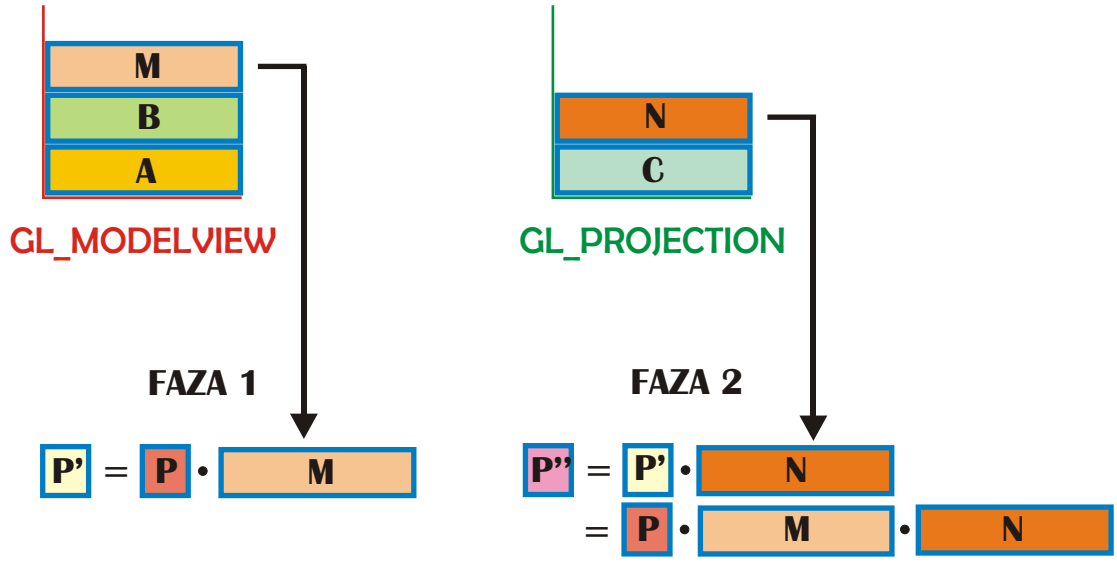
   Z poprzedniego punktu wynika, że tak zdefiniowany wierzchołek jest przechowywany w wektorze **P=[x y z 1]**. Dla ustalenia uwagi przyjmijmy, że na wierzchołu stosu modelowania znajduje się macierz **M**, a na wierzchołku stosu rzutowania jest macierz **N**.


   Reakcja biblioteki dzieli się na dwie fazy:

   (a) **FAZA 1**: Punkt `P` jest **mnożony przez wierzchołek stosu modelowania**. W wyniku otrzymujemy nowy punkt **P'=PM**.

   (b) **FAZA 2**: Wynik FAZY 1 poddawany jest **mnożeniu przez wierzchołek stosu rzutowania**. W wyniku otrzymujemy nowy punkt **P"=P'N=(PM)N =P(MN)**.

[3] lub jakikolwiek jej wariant




Jest to **WSZYSTKO**, co biblioteka OpenGL robi z danym punktem. Niczego innego już się potem nie da zrobić! Widać zatem, że kluczowe jest takie ustawienie wierzchołków obu stosów, aby przed wywołaniem komendy **glVertex3f(x,y,z)** znalazły się tam przekształcenia realizujące (w dwóch macierzach!) wszystkie operacje, którym chcemy poddać dany wierzchołek.

10. Z powyższego wynika, że zanim ustawimy oba interesujace nas stosy, musimy odpowiedzieć na pytanie:

    *Jakiego typu przekształceniom muszą zostać poddane punkty sceny, aby znalazły się na płaszczyźnie obrazu* ?

    Z poprzednich punktów wynika, że przekształcenia te można podzielić na dwie grupy:

    (a) Przekształcenia unikalne dla danego wierzchołka (wielokąta, obiektu) - np. określona translacja, obrót itp.

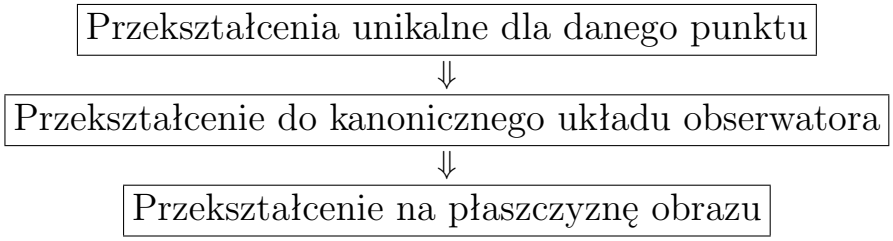
    (b) Przekształcenia wspólne dla wszystkich punktów w scenie:

        i. przekształcenie całej sceny do kanonicznego położenia obserwatora (macierz generowana przez funkcję **gluLookAt()**),

        ii. przekształcenie wszystkich punktów na płaszczyznę obrazu (macierz generowana przez funkcję **gluPerspective()**[4] )

    Jasne też jasne, że ostatnim przekształceniem dla każdego punktu musi być jego przeniesienie na płaszczyznę obrazu. W związku z tym wszystkim potok przekształceń realizowanych w trójwymiarowej grafice komputerowej zwykle ma następującą kolejność:

[4]lub funkcje **glFrustum(), glOrtho()**




Załóżmy, że wypadkowa macierz (tzn. prawidłowo uporządkowany iloczyn macierzy składowych) realizująca przekształcenia unikalne dla wierzchołka opisanego wektorem **P** nazywa się **K**, macierz generowana przez funkcję **gluLookAt()** nazywa się **M**, natomiast macierz generowana przez funkcję **gluPerspective()** to macierz **N**. Wówczas, ponieważ mnożymy lewostronnie, musi zostać wykonane przekształcenie

$$\mathbf{((PK)M)N=P(KMN)}.$$

Jak trzeba ustawić stosy modelowania i rzutowania, żeby ten schemat został zrealizowany ? Rozwiązań jest kilka (trzeba pamiętać, że po fazie inicjacji biblioteki na wierzchołkach obu z nich są macierze identycznościowe):

Rozwiązanie 1

I. Domnożenie na wierzchołek stosu modelowania macierzy **N**. Na wierzchołku znajduje się teraz iloczyn **N Id=N**.

II. Domnożenie na wierzchołek stosu modelowania macierzy **M**. Na wierzchołku znajduje się teraz iloczyn **MN**.

III. Domnożenie na wierzchołek stosu modelowania macierzy **K**. Na wierzchołku znajduje się teraz iloczyn **KMN**.

W wyniku reakcji biblioteki (punkt 9) zostaje wykonane przekształcenie **P K M N Id=PKMN**

Rozwiązanie 2 To samo co w Rozwiązaniu 1, tylko wykonane na stosie rzutowania. W wyniku reakcji biblioteki zostaje wykonane przekształcenie **P Id K M N=PKMN**.

Zastanówmy się jakie wady mają dwa powyższe rozwiązania. Rozpatrzmy przypadek kodu definiującego dwa obiekty. Każdy z obiektów składa się z wielokątów, które definiowane są poprzez wierzchołki. Załóżmy, że kody definiujące geometrię obu obiektów znajdują się odpowiednio w funkcjach: **Obiekt1(), Obiekt2()**. Dlaczego zostało tu napisane "geometrię" ? Wyobraźmy sobie, że obiektem pierwszym ma być sześcian o boku długości 1, bokach równoległych do płaszczyzn układu (tzn. płaszczyzn **XY, YZ, XZ**) taki, że najbliższy w stosunku do zera układu wierzchołek znajduje się w punkcie (10,5,4). Jest jasne, że łatwiej byłoby zdefiniować jego wierzchołki tak, że jeden z nich znajduje się w punkcie (0,0,0), a krawędzie są rozpięte wzdłuż osi układu i dopiero tak zdefiniowany obiekt przesunąć o wektor [10,5,4]. Wtedy funkcja **Obiekt1()** zawierałaby kod łatwiej definiowalnego sześcianu. Co trzeba zrobić, żeby znalazł się on we właściwym miejscu ? Jeżeli przedstawione powyżej przykładowe **Rozwiązania** są zrozumiałe, to powinno być jasne, że wystarczy, aby "najbardziej lewą" operacją na wierzchołku stosu modelowania była macierz translacji **T** generowana przez wywołanie funkcji **glTranslatef(10,5,4)**

(macierz ta pełni rolę macierzy **K** w powyższym przykładzie). Kod realizujący ten kawałek składałby się z dwóch linii:

```
glTranslate(10,5,4);
Obiekt1();
```

Zapewnia to, że przed pojawieniem się każdego wierzchołka obiektu pierwszego stos modelowania będzie tak ustawiony, że biblioteka najpierw wykona na każdym wierzchołku (a więc i na całym obiekcie) translację. Aby zrealizować cały potok graficzny dla obiektu pierwszego kod musialby wyglądać następująco (nie podajemy dla prostoty kodu parametrów funkcji rzutu i obserwatora):

```
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glTranslate(10,5,4); //macierz T
Obiekt1();
```

Proszę zwrócić uwagę, że nie ustawiamy jawnie stosu aktywnego, więc wszystkie macierze odkładaja się na stosie modelowania. Stos rzutowania zawiera tylko macierz identycznościową. Załóżmy teraz, że po zdefiniowaniu geometrii obiektu drugiego w funkcją **Obiekt2()**, obiekt ten ma być poddany tylko przeskalowaniu wartościami (2,2,8), odpowiednio w kierunkach **X,Y,Z**. Wtedy realizacja prawidłowo określonego potoku graficznego dla tego obiektu wyglądałaby następująco:

```
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glScalef(2,2,8); //macierz S
Obiekt2();
```

Jeżeli oba obiekty są w tej samej scenie, to któryś z dwóch powyższych kawałków kodów musiałby być najpierw wykonany. Widać jednak, że icj dwie pierwsze linie są identyczne. Oczywiście nie chcemy generować zbednego kodu. Spróbujmy więc tak:

```
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glTranslate(10,5,4); //macierz T
Obiekt1();
glScalef(2,2,8); //macierz S
Obiekt2();
```

Widać, że dla obiektu pierwszego potok graficzny zostanie poprawnie zrealizowany (przekształcenie **PTMN**). Jednak przed wywołaniem funkcji skalującej stan stosu modelowania wynosi **TMN**. Zatem po jej wywołaniu zmiania się na **STMN** i obiekt drugi zostanie poddany dodatkowo niepożądanej translacji. Rozwiązania tego problemu są różne. Podajmy dwa z nich:

Rozwiązanie 1

I. Zmiana stosu aktywnego na stos rzutowania
II. Domnożenie na wierzchołek stosu rzutowania macierzy **N**. Na wierzchołku stosu rzutowania znajduje się teraz iloczyn **N Id=N**.
III. Domnożenie na wierzchołek stosu modelowania macierzy **M**. Na wierzchołku stosu rzutowania znajduje się teraz iloczyn **MN**.
IV. Zmiana stosu aktywnego na stos modelowania
V. Domnożenie na wierzchołek stosu modelowania macierzy **T**. Na wierzchołku znajduje się teraz **T Id=T**.
VI. Wowołanie funkcji **Obiekt1()**
VII. Zastąpienie wierzchołka stosu modelowania macierzą identycznościową. Na wierzchołku stosu modelowania znajduje sie teraz macierz **Id**.
VIII. Domnożenie na wierzchołek stosu modelowania macierzy **S**. Na wierzchołku znajduje się teraz iloczyn **S Id=S**.
IX. Wowołanie funkcji **Obiekt2()**

Kod realizujący to rozwiązanie wygląda następująco:

```
glMatrixMode(GL_PROJECTION);
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glMatrixMode(GL_MODELVIEW);
glTranslate(10,5,4); //macierz T
Obiekt1();
glLoadIdentity(); // macierz Id
glScalef(2,2,8); //macierz S
Obiekt2();
```

Istotą tego rozwiązania jest odłożenie operacji wspólnych (wykonywanych na końcu) na stos rzutowania.

Rozwiązanie 2

I. Domnożenie na wierzchołek stosu modelowania macierzy **N**. Na wierzchołku stosu znajduje się teraz iloczyn **N Id=N**.
II. Domnożenie na wierzchołek stosu modelowania macierzy **M**. Na wierzchołku znajduje się teraz iloczyn **MN**.
III. Skopiowanie wierzchołka stosu modelowania. Na stosie znajdują się teraz dwa identyczne elementy będące iloczynami **MN**.
V. Domnożenie na wierzchołek stosu modelowania macierzy **T**. Na wierzchołku znajduje się teraz iloczyn **TMN**. Element "poniżej wierzchołka" ma nadal postać **MN**.
VI. Wowołanie funkcji **Obiekt1()**
VII. Usunięcie wierzchołka stosu modelowania. Na stosie znajduje się tylko jeden element - wierzchołek stosu postaci **MN**.
VIII. Domnożenie na wierzchołek stosu modelowania macierzy **S**. Na wierzchołku znajduje się teraz iloczyn **SMN**.
IX. Wowołanie funkcji **Obiekt2()**

Kod realizujący to rozwiązanie wygląda następująco:

```
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glPushMatrix();
  glTranslate(10,5,4); //macierz T
  Obiekt1();
glPopMatrix();
glScalef(2,2,8); //macierz S
Obiekt2();
```

Wywołanie **glPushMatrix()** pozwala "zapamiętać" wspólne operacje, przed ich modyfikacji operacjami unikalnymi dla danych obiektów. Wowołanie **glPopMatrix()** ustawia zapamiętane operacje jako bieżące. Widać, że gdyby w kodzie były trzy obiekty z unikalnymi dla nich operacjami, to funkcja skalowania i wywołanie kodu obiektu drugiego również musiałyby być objęte parą wywołań **glPushMatrix() ... glPopMatrix()**.

11. Kilka zasad ogólnych w scenach statycznych (animacji będą poświęcone ćwiczenia nr 3):

    (a) Operacja rzutowania pozostaje zwykle stała podczas procesu generowania obrazu sceny, a ponadto jest wykonywana jako ostatnia. W związku z tym można ją bezpiecznie odłożyć na stos rzutowania na początku kodu.

    ```
    glMatrixMode(GL_PROJECTION);
    gluPerspective();
    ```

    (b) Operacja transformacji obserwatora również pozostaje stała w scenach nieanimowanych, więc tak samo można ją odłożyć na stos rzutowania, ale tak aby była wykonana przez bibliotekę jako przedostatnia. Po tym można przejść na stos modelowania opisując na nim całą geometrię sceny.

    ```
    glMatrixMode(GL_PROJECTION);
    gluPerspective(); //macierz N
    gluLookAt(); ////macierz M
    glMatrixMode(GL_MODELVIEW);
    ...
    opis obiektów i związanych z nimi operacji
    ...
    ```

    (c) Brak ruchu w scenie nie zapewnia jeszcze, że powyższy schemat jest zawsze poprawny. Wyobraźmy sobie, że przesuwamy okno na ekranie lub zmieniamy jego rozmiar przy pomocy myszki. Wtedy biblioteka okienkowa wywołuje standardową funkcję obsługującą te zdarzenia. Funkcja ta automatycznie powoduje odrysowanie sceny w oknie, poprzez ponowne wywołanie funkcji zawierającej opis sceny. Co się wówczas stanie jeżeli jej kod jest taki jak w punkcie 11b? Po pierwszym wywołaniu stan stosu rzutowania to iloczyn **MN** na wierzchołku. Po ponownym wywołaniu stan tego stosu będzie wynosił **MNMN**. Podobna sytuacja wystąpi na stosie modelowania. Rozwiązaniem jest załadowanie na

wierzchołki obu stosów macierzy identycznościowej przed domnożeniem na nie innych macierzy, tzn.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(); //macierz N
gluLookAt(); //macierz M
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
...
opis obiektów i związanych z nimi operacji
...
```

Oczywiście dwa dodatkowe wywołania są zbędne przy pierwszym narysowaniu sceny, ale konieczne przy jakimkolwiek jej odrysowaniu. Można by co prawda zrealizować to inaczej, ale ilość wywołań byłaby o trzy większa:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
  gluPerspective(); //macierz N
  gluLookAt(); ////macierz M
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
  ...
  opis obiektów i związanych z nimi operacji
  ...

glPopMatrix();//na stosie modelowania ponownie będzie macierz Id
glMatrixMode(GL_PROJECTION);
glPopMatrix();//na stosie rzutowania ponownie będzie  macierz Id
```

W tym przypadku kopiujemy wierzchołki zawierające macierze identycznościowe.

12. Konkluzja jest następująca: Mamy dwa stosy do dyspozycji i potok graficzny, który musimy zrealizować dla każdego wierzchołka każdego obiektu. Jak pokazały powyższe przykłady poprawnych rozwiązań jest zwykle co najmniej kilka. Niemniej powinno być jasne, że przy każdym wywołaniu jakiejkolwiek funkcji w kodzie musimy być pewni jaki jest bieżący stan obu stosów. W przeciwnym przypadku działanie kodu jest czysto losowe.

13. *Przykład - konstrukcja sześcianu o środku w punkcie* (0,0,0)*, ścianach równoległych o scian układu współrzędnych i boku długości* 1*.*

    Zakładamy, że funkcja generująca scenę nazywa się **display()**. Musimy włączyć algorytm zasłaniania, gdyż generujemy bryły (zob. plik **ogl1.pdf**)

- Rozwiązanie najprostsze

```
void display()
{
glClearColor(1.0,1.0,1.0,1.0);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// "czyszczenie"' tła okna i bufora głębokości

glEnable(GL_DEPTH_TEST); //włączenie algorytmu zasłaniania
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, 1.0, 0.1, 10.0); //bryła widzenia perspektywicznego
gluLookAt(2.0,2.0,2.0, 0.0,0.0,0.0, 0.0,1.0,0.0);//obserwator
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glBegin(GL_QUADS);

  //ściana: y = 0.5

  glColor3f(1.0, 0.0, 0.0);
  glVertex3f( 0.5,  0.5,  0.5);
  glVertex3f( 0.5,  0.5, -0.5);
  glVertex3f(-0.5,  0.5, -0.5);
  glVertex3f(-0.5,  0.5,  0.5);

  //ściana: y = -0.5

  glColor3f(0.0, 1.0, 0.0);
  glVertex3f( 0.5, -0.5,  0.5);
  glVertex3f( 0.5, -0.5, -0.5);
  glVertex3f(-0.5, -0.5, -0.5);
  glVertex3f(-0.5, -0.5,  0.5);

  //ściana: x = 0.5

  glColor3f(0.0, 0.0, 1.0);
  glVertex3f( 0.5,  0.5,  0.5);
  glVertex3f( 0.5,  0.5, -0.5);
  glVertex3f( 0.5, -0.5, -0.5);
  glVertex3f( 0.5, -0.5,  0.5);

  //ściana: x = -0.5
```

```
  glColor3f(1.0, 1.0, 0.0);
  glVertex3f(-0.5,  0.5,  0.5);
  glVertex3f(-0.5,  0.5, -0.5);
  glVertex3f(-0.5, -0.5, -0.5);
  glVertex3f(-0.5, -0.5,  0.5);

  //ściana: z = 0.5

  glColor3f(1.0, 0.0, 1.0);
  glVertex3f( 0.5,  0.5,  0.5);
  glVertex3f( 0.5, -0.5,  0.5);
  glVertex3f(-0.5, -0.5,  0.5);
  glVertex3f(-0.5,  0.5,  0.5);

  //ściana: z = -0.5

  glColor3f(0.0, 1.0, 1.0);
  glVertex3f( 0.5,  0.5, -0.5);
  glVertex3f( 0.5, -0.5, -0.5);
  glVertex3f(-0.5, -0.5, -0.5);
  glVertex3f(-0.5,  0.5, -0.5);

glEnd();

glFlush();
}
```

Każda ściana sześcianu jest geometrycznie takim samym kwadratem. Ponadto każdy wierzchołek należy do trzech ścian, więc w powyższym kodzie jego kod występuje trzykrotnie.

- Rozwiązanie nieco inne

```
//kwadrat o środku (0,0,0), leżący w płaszczyźnie XY
//o krawędziach długości 1, równoległych do osi X i Y

void sciana()
{
glBegin(GL_QUADS);
  glVertex3f( 0.5,  0.5,  0.0);
  glVertex3f( 0.5, -0.5,  0.0);
  glVertex3f(-0.5, -0.5,  0.0);
  glVertex3f(-0.5,  0.5,  0.0);
glEnd();
}
```

```
void display()
{
glClearColor(1.0,1.0,1.0,1.0);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// "czyszczenie"' tła okna i bufora głębokości

glEnable(GL_DEPTH_TEST); //włączenie algorytmu zasłaniania
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0, 1.0, 0.1, 10.0); //bryła widzenia perspektywicznego
gluLookAt(2.0,2.0,2.0, 0.0,0.0,0.0, 0.0,1.0,0.0);//obserwator
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

  //ściana: y = 0.5

  glColor3f(1.0, 0.0, 0.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi Y o 0.5
    glTranslatef(0.0, 0.5, 0.0);
    //obrót wokół osi X o 90 stopni
    glRotatef(90.0, 1.0, 0.0, 0.0);
    sciana();
  glPopMatrix();

  //ściana: y = -0.5
  glColor3f(0.0, 1.0, 0.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi Y o -0.5
    glTranslatef(0.0, -0.5, 0.0);
    //obrót wokół osi X o 90 stopni
    glRotatef(90.0, 1.0, 0.0, 0.0);
    sciana();
  glPopMatrix();

  //ściana: x = 0.5

  glColor3f(0.0, 0.0, 1.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi X o 0.5
    glTranslatef(0.5, 0.0, 0.0);
    //obrót wokół osi Y o 90 stopni
    glRotatef(90.0, 0.0, 1.0, 0.0);
    sciana();
```

```
  glPopMatrix();

  //ściana: x = -0.5

  glColor3f(1.0, 1.0, 0.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi X o -0.5
    glTranslatef(-0.5, 0.0, 0.0);
    //obrót wokół osi Y o 90 stopni
    glRotatef(90.0, 0.0, 1.0, 0.0);
    sciana();
  glPopMatrix();

  //ściana: z = 0.5

  glColor3f(1.0, 0.0, 1.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi Z o 0.5
    glTranslatef(0.0, 0.0, 0.5);
    sciana();
  glPopMatrix();

  //ściana: z = -0.5

  glColor3f(0.0, 1.0, 1.0);
  glPushMatrix();
    //przesunięcie wzdłuż osi Z o -0.5
    glTranslatef(0.0, 0.0, -0.5);
    sciana();
  glPopMatrix();

glFlush();
}
```

14. *Zadania*

    (a) Przeczytać ten materiał tyle razy, aż stanie się oczywisty.

    (b) Zdefiniować krzesło takie, że siedzenie, oparcie i nogi są prostopadłościanami. W tym celu zdefiniować funkcję

    **prostopadloscian(a, b, c)**

    generującą prostopadłościan o wierzchołku w punkcie (0,0,0) i krawędziach rozpiętych wzdłuż dodatnich osi układu współrzędnych. Długości krawędzi wzdłuż osi X, Y, Z są przekazywane odpowiednio przez parametry **a,b,c**. Posługując

się funkcją **prostopadloscian()** zdefiniować funkcję

**nogi()**

a następnie funkcję

**krzeslo()**

konstruującą krzesło przy pomocy funkcji **nogi()** i **prostopadloscian()**. Funkcja **display()** wywołuje funkcję krzesło.

(c) Zdefiniować funkcję **stol()** w podobny sposób jak funkcję krzesło.

(d) Przy pomocy funkcji **prostopadloscian()**, **krzeslo()** i **stól()** zdefiniować funkcję **pokoj()** konstruującą pokoj zawierający stół i cztery krzesła.