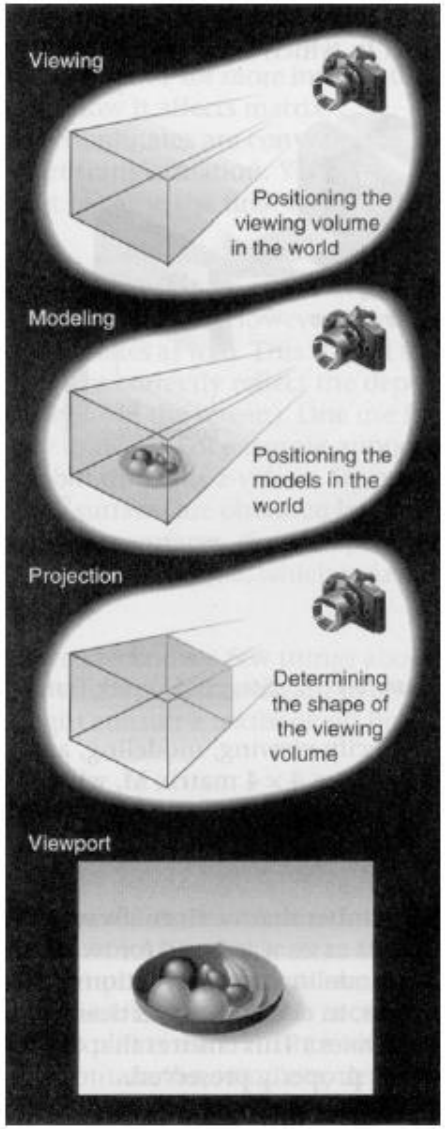
PG430 – Innføring i grafikkprogrammering

Forelesning 3

André R. Brodtkorb
Andre.Brodtkorb@nith.no

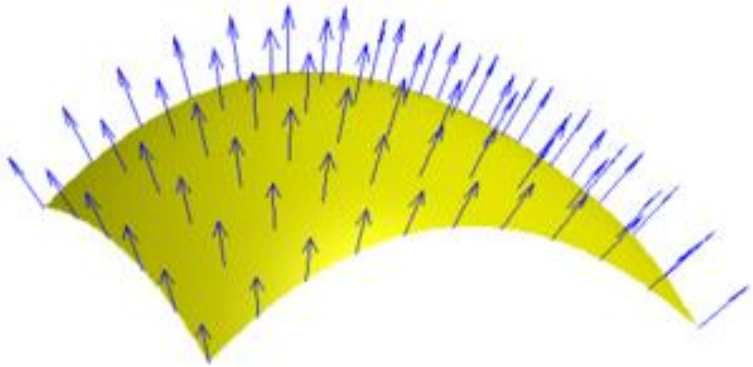
Oversikt

- Repetisjon
- OpenGL er lineær algebra
- Transformasjoner
- Projeksjoner
- Lab: Robot-arm



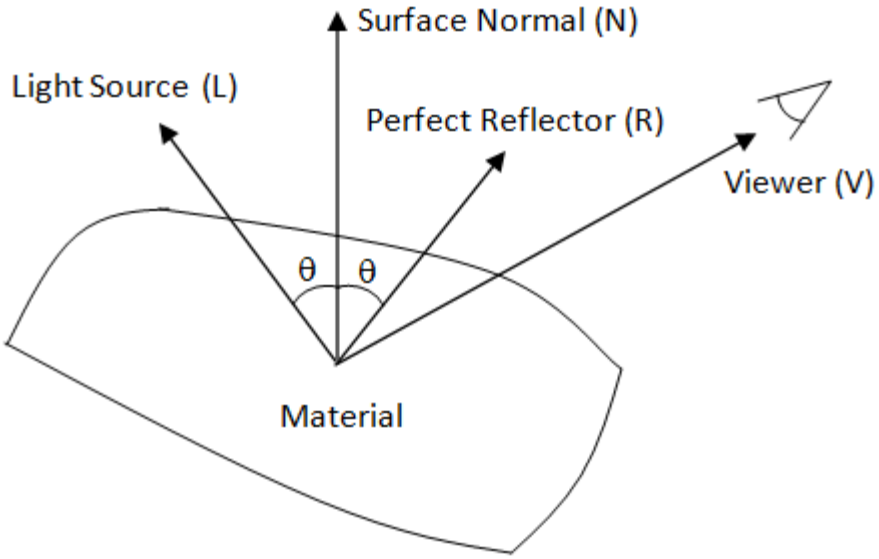
REPETISJON

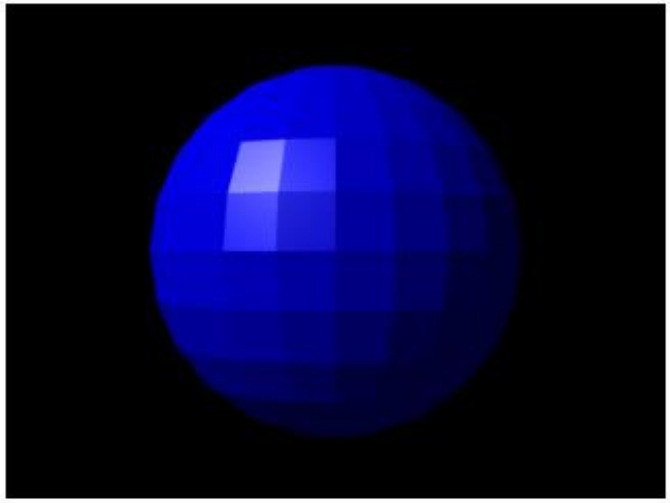
Normalvektorer



- Normalen brukes til lyssetting i OpenGL

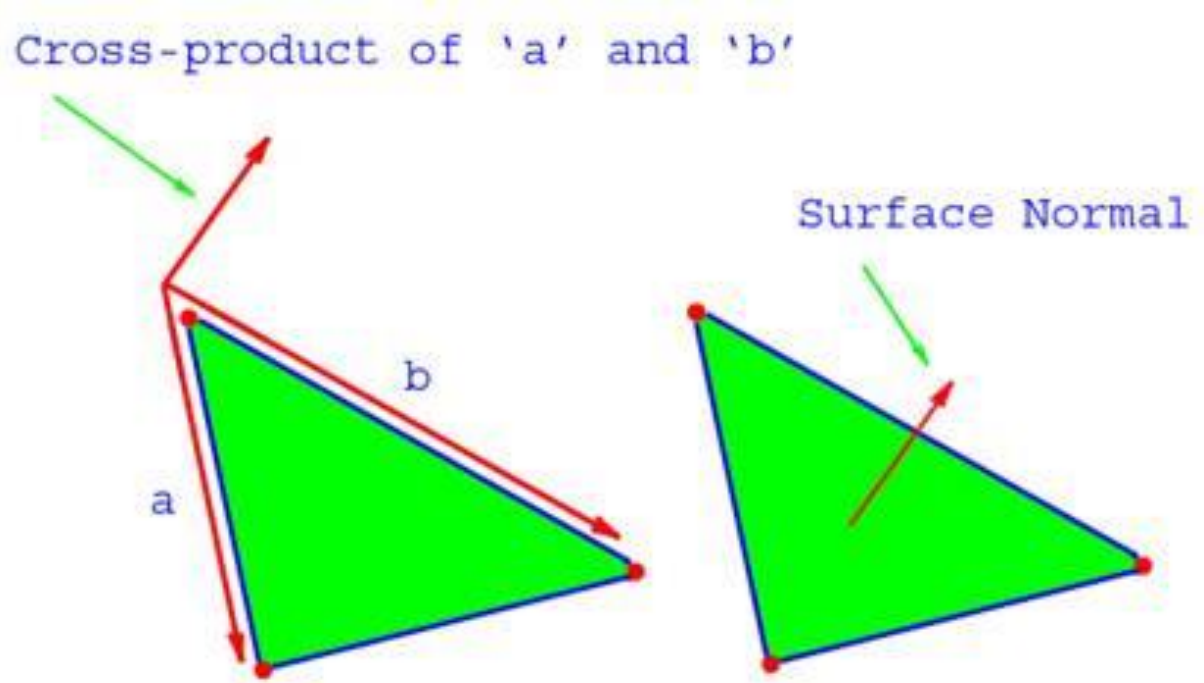
- glNormal3f(n0, n1, n2);





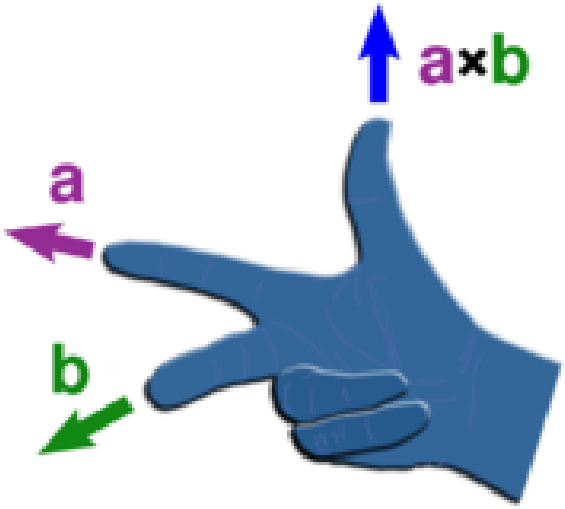
Normalvektorer

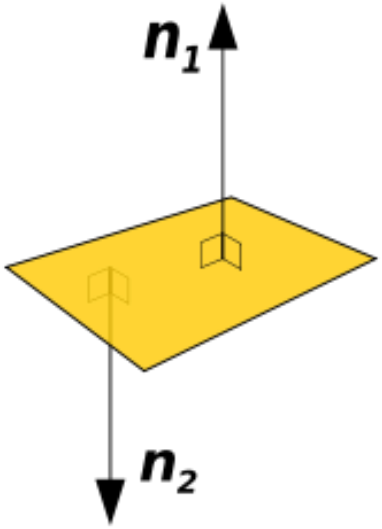
- Normalen til en flate kan regnes ut med å ta kryssproduktet mellom to vektorer i flaten.



Normalvektorer

- Rekkefølgen man krysser vektorene på bestemmer retningen.





Input til OpenGL

- Tre forskjellige typer

- Er det noen som husker dem uten å se i foilene?
  - Hva er forskjellen mellom de tre?

Immediate mode

- Typisk kode:

```
glBegin(GL_TRIANGLES);
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
glEnd();
```

Display-lister

- I funksjonen init()

```
list_id = glGenLists(1);                    // list_id er en integer

glNewList(list_id, GL_COMPILE);
   glBegin(GL_TRIANGLES);
      glNormal3f(0.0f, 0.0f, 1.0f);
      glVertex3f(0.0f, 0.0f, 0.0f);
      glVertex3f(1.0f, 0.0f, 0.0f);
      glVertex3f(1.0f, 1.0f, 0.0f);
   glEnd();
glEndList();
```

- I funksjonen render()

```
glCallList(list_id)
```

Vertex Arrays

```
//Hint: Bruk std::vector<GLfloat> i din kode!
GLfloat vertices[ ] = {0,0,1,  0,0,1,  0,0,1,  0,0,1,
                       1,0,0,  1,0,0,  1,0,0, 1,0,0,
                         0,1,0,  0,1,0,  0,1,0, 0,1,0,
                         -1,0,0,  -1,0,0, -1,0,0,  -1,0,0,
                         0,-1,0,  0,-1,0,  0,-1,0,  0,-1,0,
                         0,0,-1,  0,0,-1,  0,0,-1,  0,0,-1   };

glEnableClientState(GL_VERTEX_ARRAY);    // enable vertex arrays
glVertexPointer(3, GL_FLOAT, 0, vertices);     // sette vertex pointer
glDrawArrays (GL_QUADS, 0, 24);                // tegne 6 quads
glDisableClientState(GL_VERTEX_ARRAY);  // disable vertex arrays
```

Input til OpenGL

- Immediate mode:
  - Enkelt men lite effektivt
- Display lists:
  - Caching av geometri og states; kan gi bedre effektivitet.
- Vertex arrays:
  - Kompakt representasjon.

Vindusystemer for OpenGL

- Vindussystemer
  - Håndterer brukergrensesnittet,
  - Lager OpenGL contexter, etc.
- På windows kan man bruke:
  - Glut http://www.opengl.org/resources/libraries/glut/
  - Qt http://www.trolltech.com/products/qt
  - SDL
  - Microsoft (Win32, Microsoft .NET Framework)
  - FreeGlut (Open source Glut)
  - wxWidgets http://www.wxwidgets.org/
  - Fast Light Toolkit (FLTK) http://www.fltk.org/
  - Etc...
- Vi kommer til å bruke SDL.

DAGENS FORELESNING

Mappe 1

- Noen kommentarer eller problemer?
- Neste forelesning krasjer med opplysningstime for prosjektoppgave (14:15-15:00).
- Det vil si det blir en time lab, som gjør at det blir mindre hjelp til Mappe 1: Du burde være "nesten ferdig" til neste lab-time.
- Husk readme fil og at det skal kompilere out-of-the-box!

Readme fil

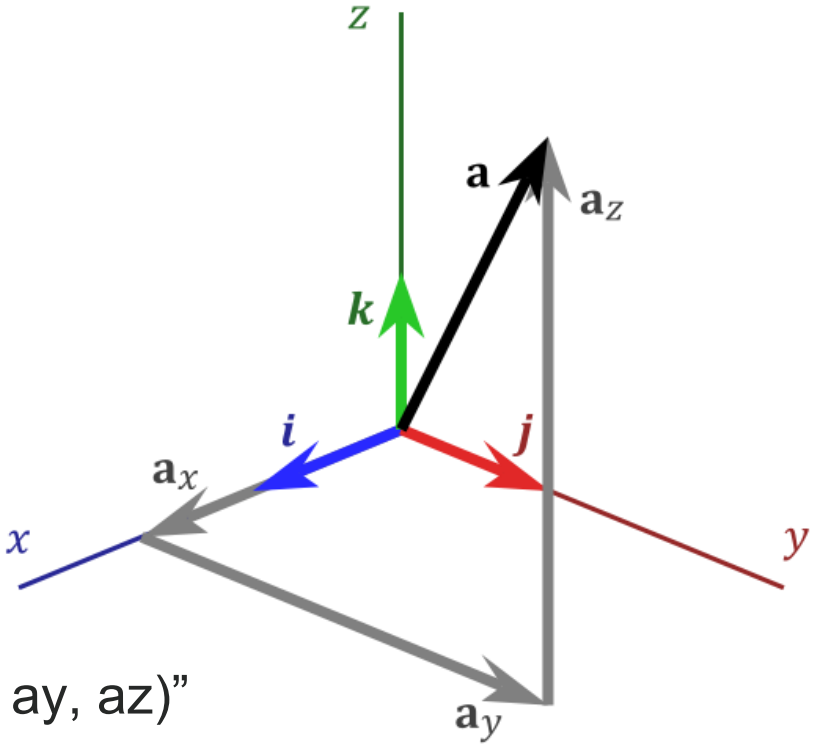
- Lag en readme fil som beskriver løsningen din
- Eksempel på readme:
  I min mappeoppgave har jeg valgt å lage en sort trekant i 3D som romskip, og bruker en quad / firkant som skudd. Piltastene styrer romskipet, og space fyrer av skudd. Jeg har satt opp lys, og bruker keyboard states for å styre skipet. For å hindre at skipet går utenfor view / frustum har jeg valgt å ha en if-test som sjekker om romskipet er nær kanten: hvis så, så setter jeg hastigheten til null for å hindre at det går utenfor.

OpenGL er lineær algebra

- Lineær algebra er "operasjoner på vektorer og matriser"
- I datagrafikk (både DirectX og OpenGL) er lineær algebra en hovedkomponent for å lage 2D-bilder av 3D scener.
- Punkter og vektorer (e.g., normaler) transformeres med matriser
  - Projeksjonsmatrisen
  - Modelviewmatrisen

Lineær algebra

- Basisoperasjoner i Lineær algebra er lineære transformasjoner av vektorer

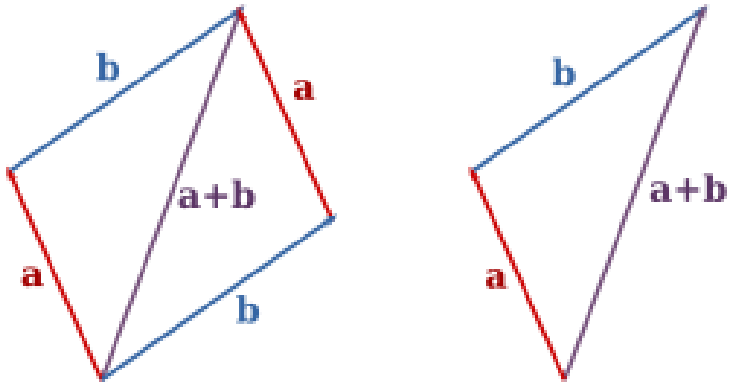


- En vektor
  - Beskriver en retning
  - ”Pilen fra punktet (0, 0, 0) til (ax, ay, az)”
  - Angis typisk som [ax, ay, az]

Vektor operasjoner

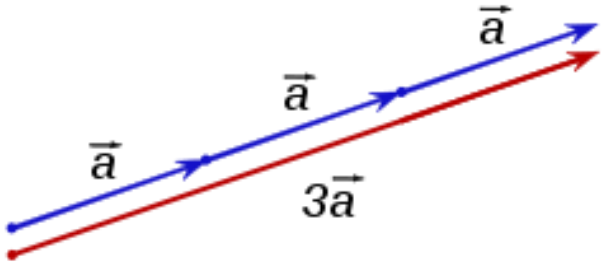
- Addisjon:

$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \\ a_z + b_z \end{pmatrix}$$



- Skalering:

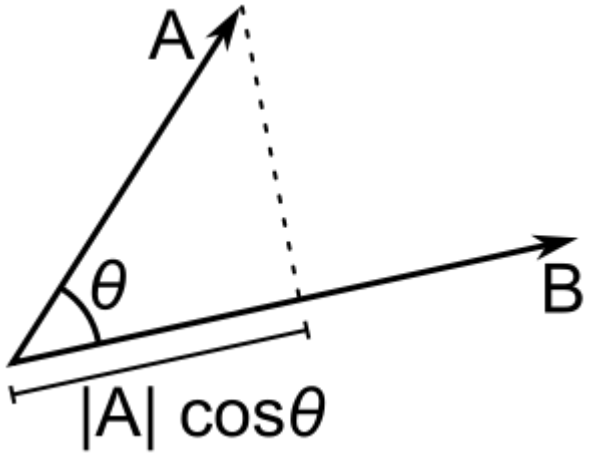
$$s\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} sa_x \\ sa_y \\ sa_z \end{pmatrix}$$



Vektor operasjoner

- Prikk-produkt (dot product):

$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \cdot \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = a_x * b_x + a_y * b_y + a_z * b_z$$

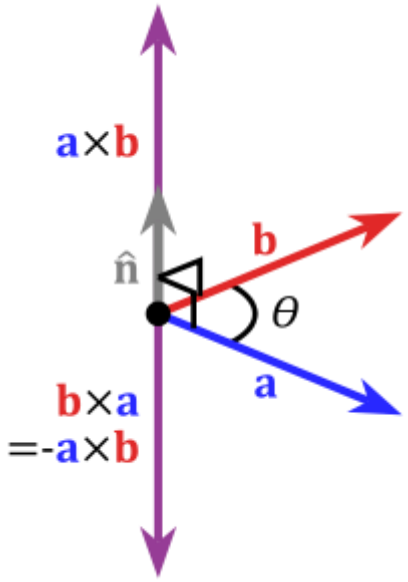


  - Kan tolkes som "lengden av A i retning B", eller "A projisert ned på B"
  - Brukes bl.a. til lyssetting.
  - Er null dersom a står vinkelrett på b

Vektor operasjoner

- Kryss produkt:

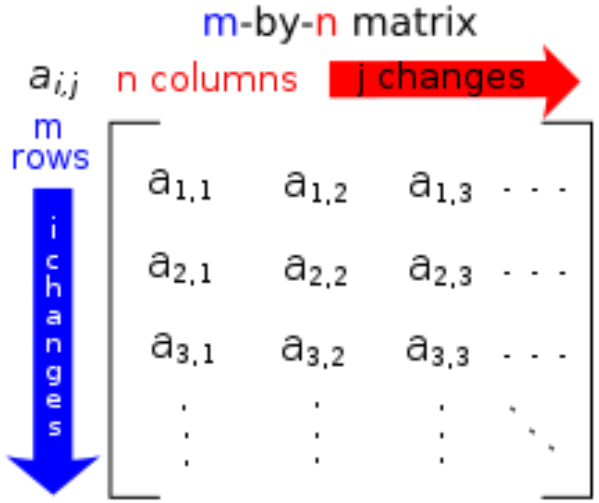
$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ -(a_x b_z - a_z b_x) \\ a_x b_y - a_y b_x \end{pmatrix}$$



  - $a \times b$ står vinkelrett på både a og b.
  - Er null dersom a og b er parallelle
  - brukes i utregning av normaler

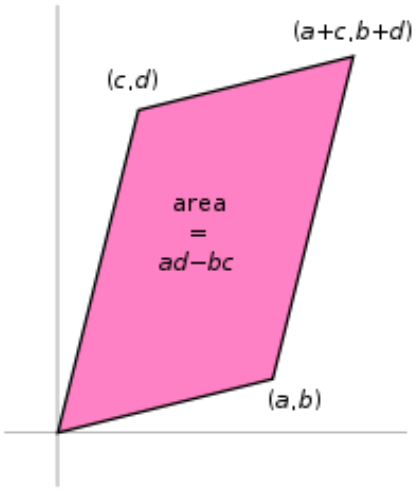
Lineær algebra

- Matriser
  - Kan brukes til å spesifisere transformasjoner



- Eksempel:
  - A kan transformere en enhetsfirkant til parallelogrammet til høyre.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$



Matrise-vektor produkt

- Matriser kan representere lineære transformasjoner som rotasjoner og skalering

$$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + dy + gz \\ bx + ey + hz \\ cx + fy + yz \end{pmatrix}$$

- Skalering:

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix}$$

Matrise-vektor produkt

- Ta transformasjonsmatrisen A, og transformer vertexene til enhetsfirkanten:

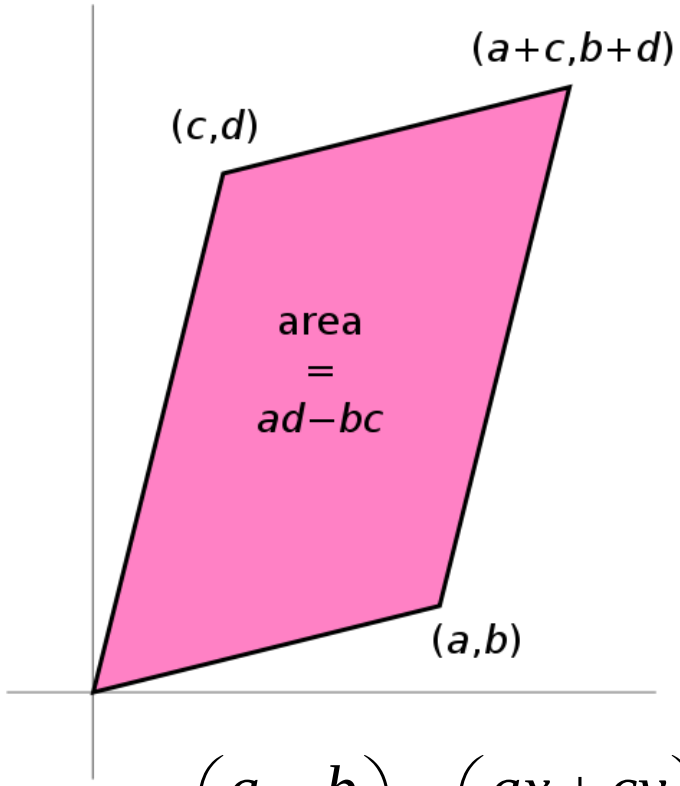
$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

- Enhetsfirkanten:

$$\begin{bmatrix} 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Matrise-vektor produktet:

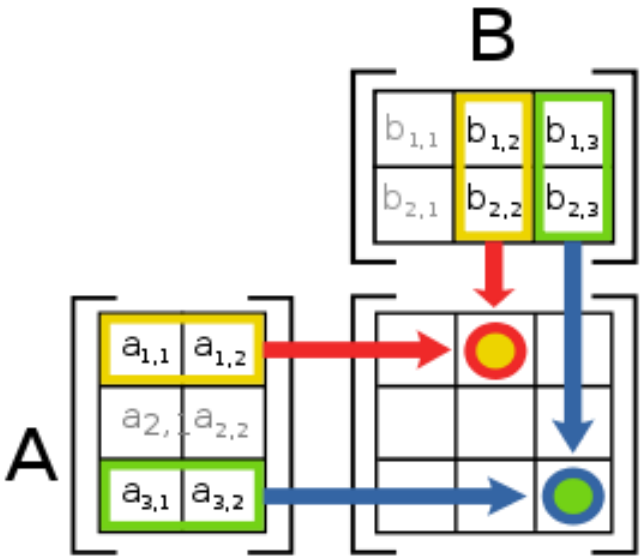
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax+by \\ cx+dy \end{pmatrix}$$



$$\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ax+cy \\ bx+dy \end{pmatrix}$$

Matrise-matrise produkt

- Produktet av to matriser er en ny matrise
  - Produktet av kvadratiske matriser er en matrise av samme størrelse
  - OpenGL bruker mange 4x4 matriser
- Assosiativ
  - ABC=(AB)C=A(BC)
- **Ikke** kommutativ
  - MN≠NM
  - Dette gjør at rekkefølgen på transformasjoner er signifikant



Homogene koordinater

- For å også tillate translasjoner bruker OpenGL 4x4 matriser:

$$\begin{pmatrix} a & d & g & t_x \\ b & e & h & t_y \\ c & f & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} ax + dy + gz + wt_x \\ bx + ey + hz + wt_y \\ cx + fy + iz + wt_z \\ w \end{pmatrix}$$

- Slike transformasjoner kalles affine
  - Lineær transformasjon plus translasjon
- For punkter er w=1
- For vektorer er w=0.
  - Dette er fordi vektorer ikke skal translateres, men er retninger

Homogene koordinater

- Dersom w=1 er x, y og z de vanlige koordinatene.

- Hvis w er ulik 1, får man de euklidske (vanlige) koordinatene ved å dele på w:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

- Hva hvis w er null? (se også forrige slide)

Transformasjoner

Modeling-transformasjoner

- Object space:
  - Vertex posisjoner og normaler er definert i et koordinatsystem som kalles *object space*.
  - Alle objekter kan defineres i et eget unikt object space.

- World space:
  - Vi trenger et felles koordinatsystem som kan inneholde en mengde forskjellige objekter. Dette koordinatsystemet kalles *world space*.

- Modeling transformation:
  - Transformasjonen mellom object space og world space kalles *modeling transformation*.

OpenGL matrisekommandoer

- glMatrixMode(GLenum mode);
  - Angir hvilken matrise som skal modifiseres av alle etteføIgende matriseoperasjoner. GL_MODELVIEW, GL_PROJECTION eller GL_TEXTURE.

- glLoadIdentity();
  - Setter valgt matrise til en 4 x 4 identitetsmatrise.

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- glLoadMatrix{fd}(const TYPE *m);
  - Setter de 16 verdiene til gjeldende matrise lik verdiene spesifisert av *m.*

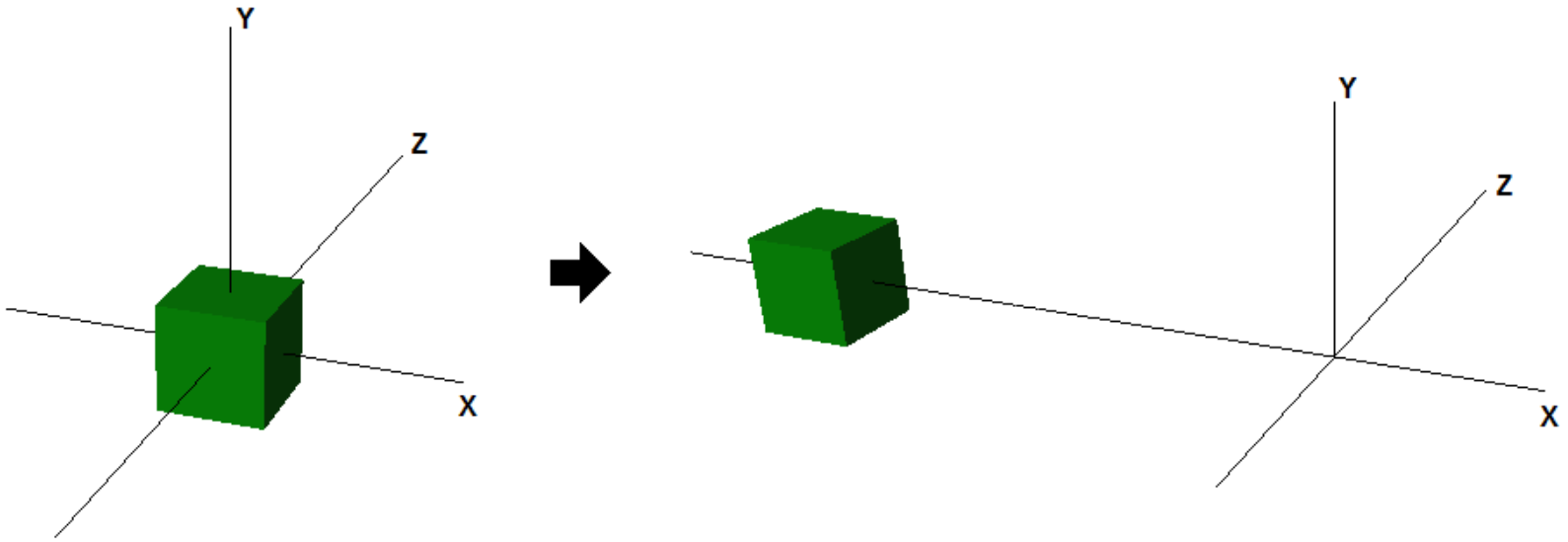
- glMultMatrix{fd}(const TYPE *m);
  - Multipliserer gjeldende matrise med verdiene spesifisert av *m.*

Translasjon

glTranslate{fd}(TYPE x, TYPE, y, TYPE z);

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

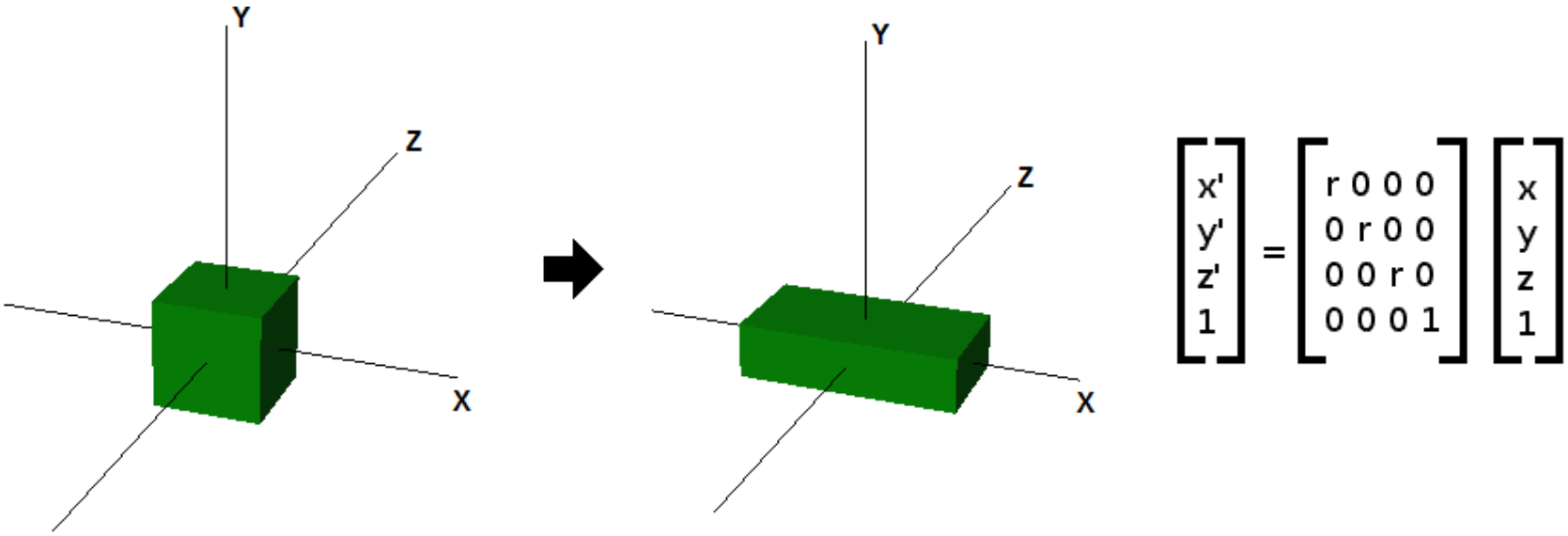
- Multipliserer gjeldende matrise med en matrise som flytter et objekt / koordinatsystem med de gitte x-, y- og z-verdiene.



Skalering

glScale{fd}(TYPE x, TYPE y, TYPE z);

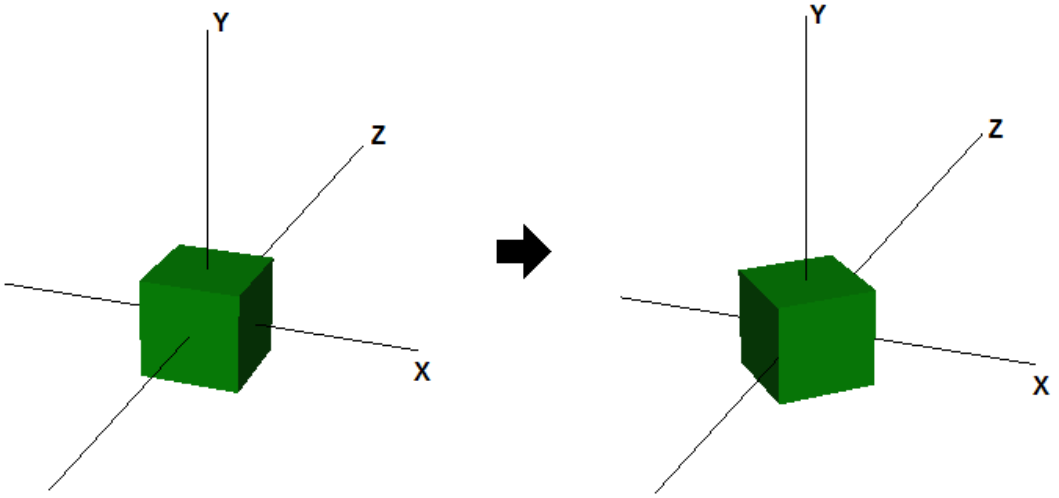
- Multipliserer gjeldende matrise med en matrise som skalerer et objekt / koordinatsystem basert på de gitte x-, y- og z-verdiene. Alle x-, y- og z-koordinatene til et objekt blir multiplisert med de tilsvarende x-, y- og z-argumentene.



Rotasjon

glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

- Multipliserer gjeldende matrise med en matrise som roterer et objekt / koordinatsystem i retning mot klokka, om en vektor mellom origo og de gitte x-, y- og z-verdiene.
- *Angle* angir rotasjonsvinkelen i grader.



$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(yaw) & 0 & \sin(yaw) & 0 \\ 0 & 0 & 0 & 0 \\ -\sin(yaw) & 0 & \cos(yaw) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

rotasjon om y-aksen

Rotasjonsmatriser

- Rotasjon rundt [1 0 0]

$$Q_{\mathbf{x}}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix},$$

- Rotasjon rundt [0 1 0]

$$Q_{\mathbf{y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix},$$

- Rotasjon rundt [0 0 1]

$$Q_{\mathbf{z}}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

Rotasjonsmatriser

- Rotasjon rundt vektoren u=[x y z]:

$$
\begin{aligned}
Q_{\mathbf{u}}(\theta) &= \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix} \sin\theta + (I - \mathbf{u}\mathbf{u}^T)\cos\theta + \mathbf{u}\mathbf{u}^T \\
&= \begin{bmatrix} (1-x^2)c_\theta + x^2 & -zs_\theta - xyc_\theta + xy & ys_\theta - xzc_\theta + xz \\ zs_\theta - xyc_\theta + xy & (1-y^2)c_\theta + y^2 & -xs_\theta - yzc_\theta + yz \\ -ys_\theta - xzc_\theta + xz & xs_\theta - yzc_\theta + yz & (1-z^2)c_\theta + z^2 \end{bmatrix} \\
&= \begin{bmatrix} x^2(1-c_\theta) + c_\theta & xy(1-c_\theta) - zs_\theta & xz(1-c_\theta) + ys_\theta \\ xy(1-c_\theta) + zs_\theta & y^2(1-c_\theta) + c_\theta & yz(1-c_\theta) - xs_\theta \\ xz(1-c_\theta) - ys_\theta & yz(1-c_\theta) + xs_\theta & z^2(1-c_\theta) + c_\theta \end{bmatrix},
\end{aligned}
$$

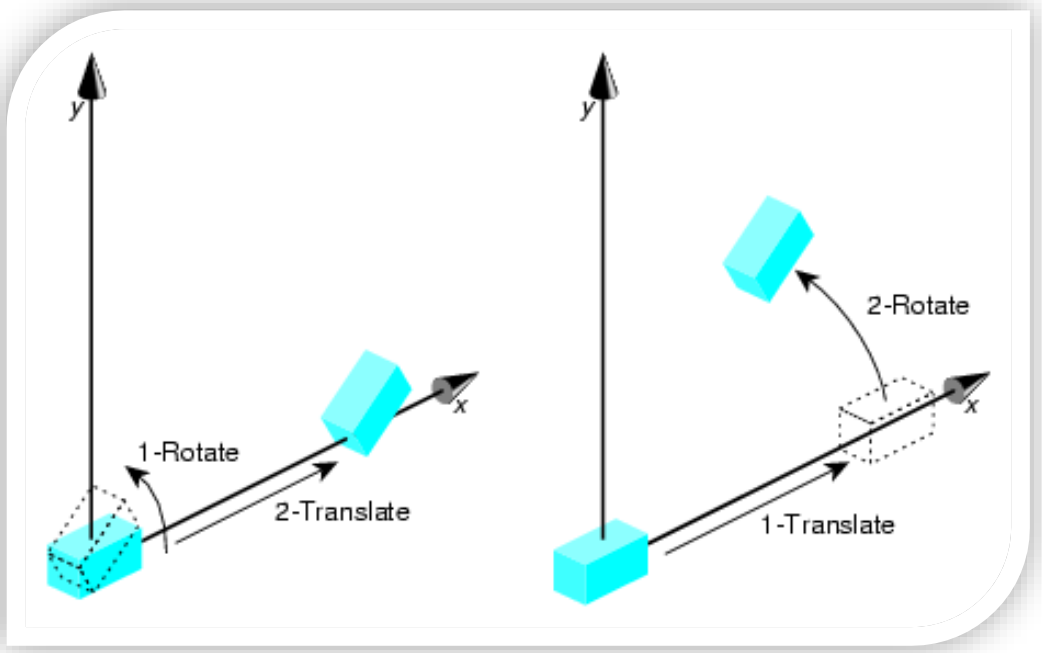
Modeling-transformasjoner

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glTranslatef(...);
glRotatef(...);
glScalef(...);

glBegin();
...
glEnd();
```

- Tutorial: Nate Robins, transformations

Modeling-transformasjoner

- Resultatet avhenger av hvordan matrisene er multiplisert med hverandre.
- Prøv å bytt mellom rotasjon/translasjon i tutorialen.



Rekkefølgen på transformasjoner

- Den siste transformasjonskommandoen i programmet er den som først anvendes på vertexene.
- Kan derfor si at man må spesifisere transformasjoner *baklengs* i koden.

```
glLoadIdentity();
glMultMatrixf(M);
glMultMatrixf(N);
glBegin(GL_POINTS);
   glVertex3f(v);
glEnd();
```

Rekkefølgen på transformasjoner

`glMatrixMode(GL_MODELVIEW);`	switch to `MODELVIEW`
`glLoadIdentity();`	`MODELVIEW` matrix M is I
`glMultMatrix*(`M_n`)`	$M = M_n$
`glMultMatrix*(`M_{n-1}`)`	$M = M_n M_{n-1}$
$\vdots$	$\vdots$
`glMultMatrix*(`M_2`)`	$M = M_n M_{n-1} \cdots M_2$
`glMultMatrix*(`M_1`)`	$M = M_n M_{n-1} \cdots M_2 M_1$
`glVertex*(p)`	$\mathbf{p}' = M_n M_{n-1} \cdots M_2 M_1 \mathbf{p}_1$

- Første transformasjon som utføres på vertexen er siste matrise angitt i koden
- Gjelder også glTranslate*, glRotate* ...

Viewing-transformasjoner

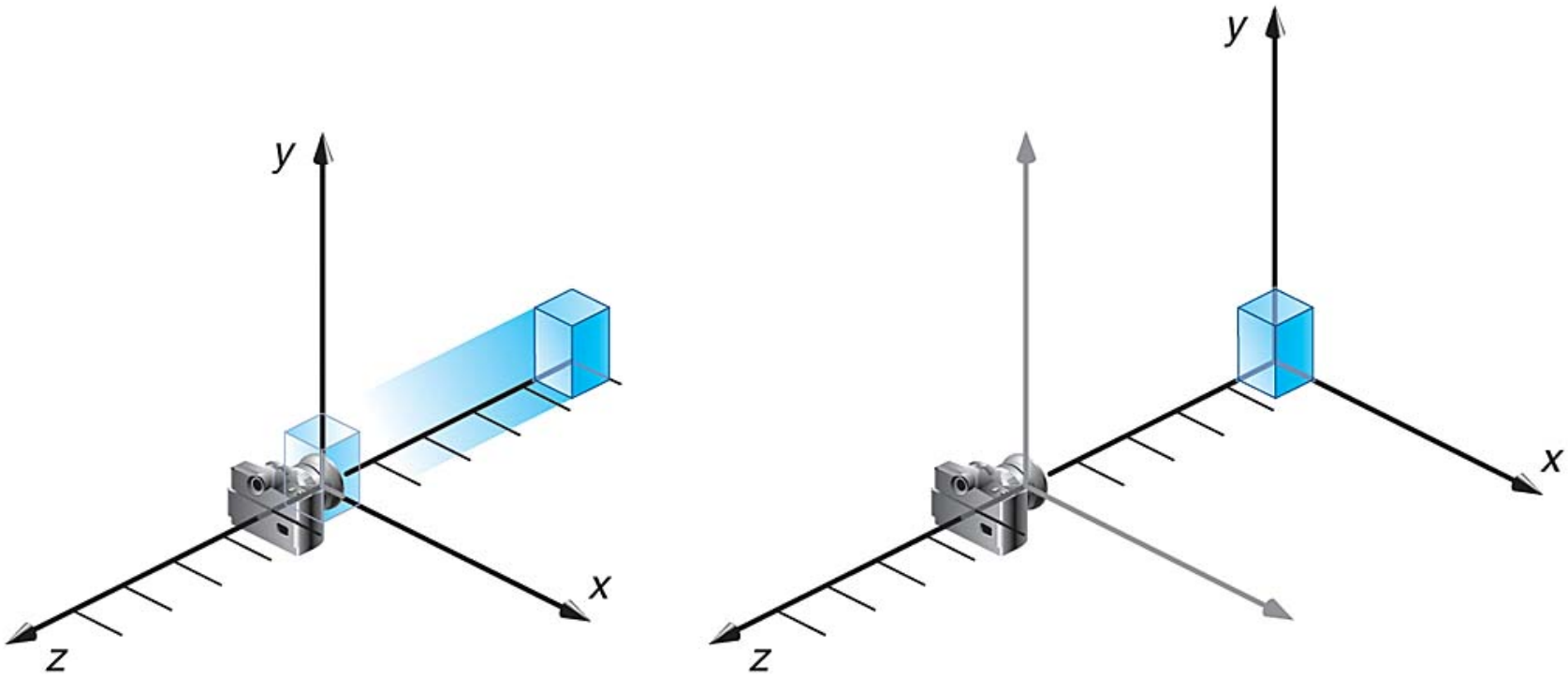
- Eye space:
  - Kameraet er definert i et koordinatsystem som kalles *eye space*.

- Viewing-transformation:
  - Transformasjonen mellom *world space* og *eye space* kalles *viewing transformation*.

- I OpenGL er *model matrix* og *viewing matrix* kombinert i en og samme matrise som kalles *modelview matrix*.

Kamera

- Det er to måter å se på modelview-transformasjoner i OpenGL.
  - Flyttes kamera?
  - Flyttes objektene?



Kamera

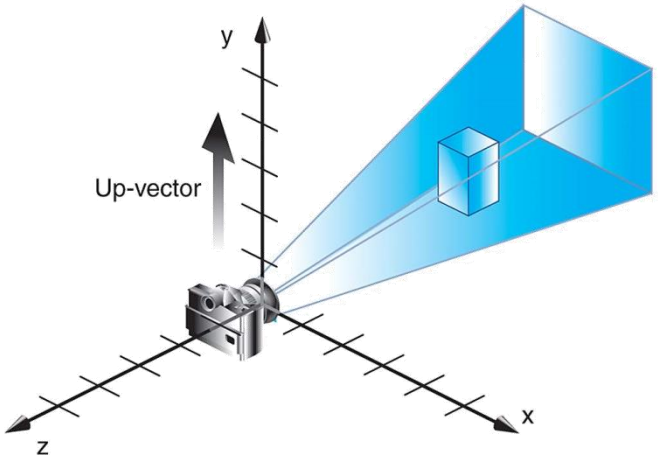
- Kan utføre modelview-transformasjoner på forskjellige måter:
  - Bruk av glTranslate of glRotate
  - Bruk av kommandoen gluLookAt()
  - Håndtere matrisene selv og sette transformasjonen med glLoadMatrix()

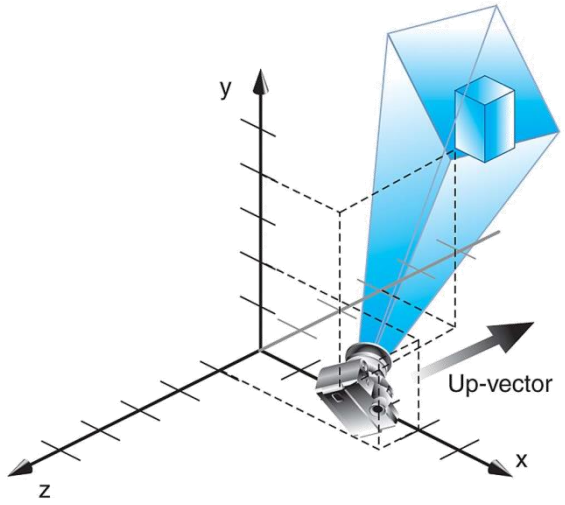
- Default er kamera plassert i origo, peker ned langs negativ z-akse og har en opp-vektor lik (0,1,0).

- Viewing-transformasjoner må utføres før modeling-transformasjoner slik at modeling-transformasjoner utføres på objekter først.

Kamera

gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
Gldouble centerx, GLdouble centery, GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);





- Definerer en viewing-matrise og multipliserer denne med gjeldende matrise.
- *Eye* definerer posisjon til kamera, *center* definerer hva kamera peker mot og *up* definerer oppvektoren.

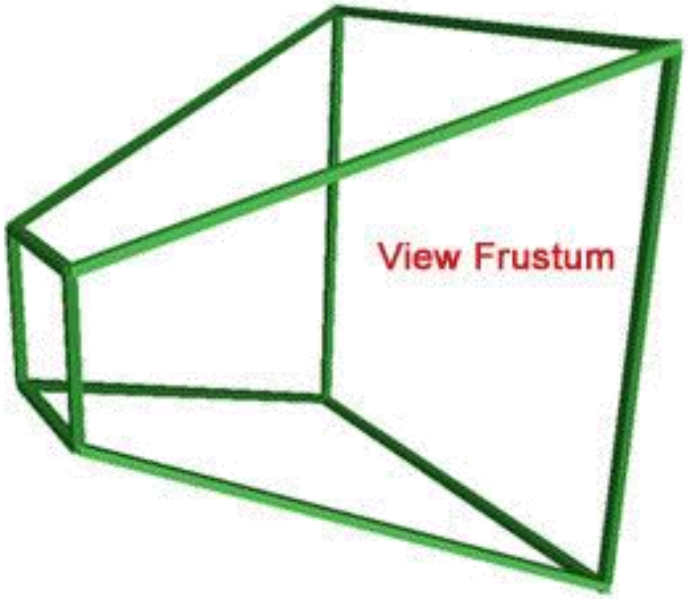
PROJEKSJONER

Viewing Volume, Clip Space

- Beskrives av projection matrisen
  - Har ikke noe å gjøre med modelview matrisen.
- Viewing volume:
  - Et *viewing volume* er den regionen som blir synlig på det endelige bildet som tegnes.
- Projection matrix:
  - Et *viewing volume* blir definert ved å spesifisere projeksjonsparametere for en matrise som kalles *projection matrix*.
- Clip space:
  - En projeksjonsmatrise brukes til å transformere objekter som befinner seg i et *viewing volume* inn i et nytt koordiantsystem som kalles *clip space.*

Viewing Volume, Frustum

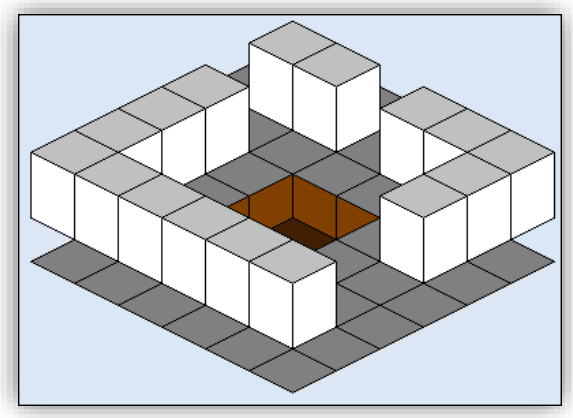
- Funksjoner som brukes for å spesifisere *viewing volume*:
  - glOrtho(...);
  - glFrustum(...):
  - gluPerspective(...);

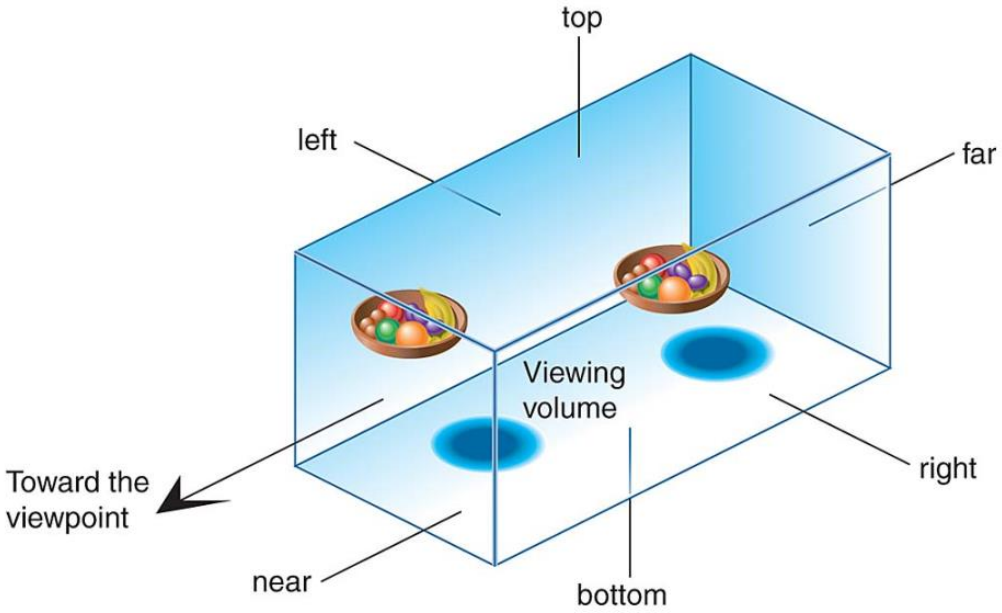


Ortografisk projeksjon

- Ved ortografisk projeksjon har *viewing volume* form som en boks.

glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble near, GLdouble far);

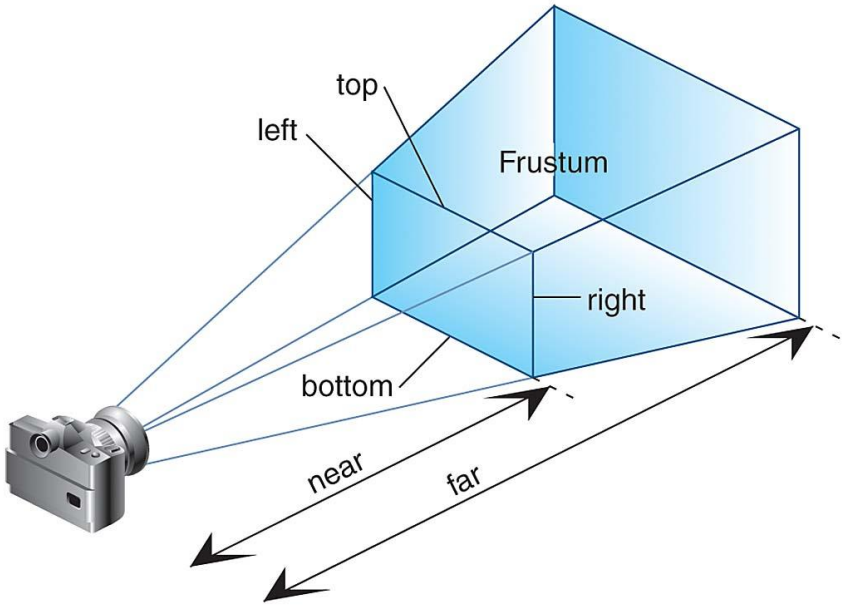




Perspektiv projeksjon

- Objekter som er nært kamera blir større enn objekter som er lengre unna kamera.

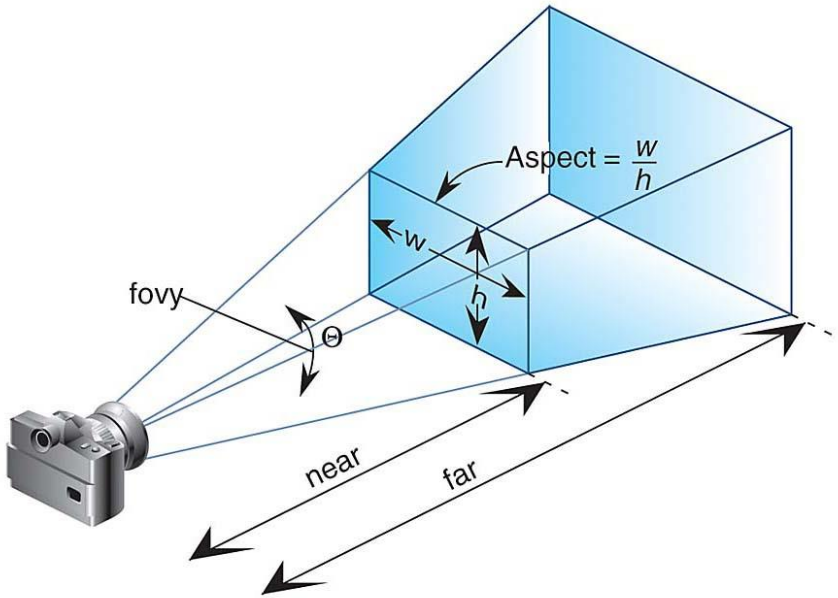
glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
GLdouble top, GLdouble near, GLdouble far);



Perspektiv projeksjon

- Om man synes glFrustum er vanskelig å bruke kan man bruke følgende kommando fra *glu* i steden for:

```
gluPerspective(GLdouble fovy, GLdouble aspect,
      GLdouble near, GLdouble far);
```



Projeksjon

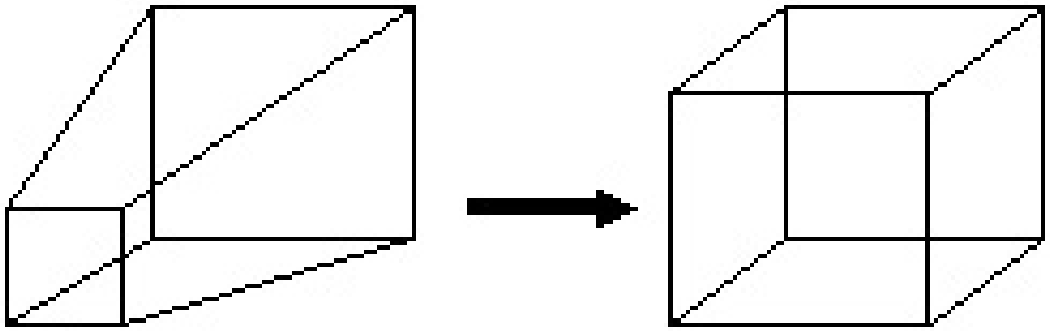
- Må sette rett matrix mode før man forandrer på projeksjonsmatrisen:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(...);
glMatrixMode(GL_MODELVIEW);
```

- Tutorial: Nate Robins, projection
  - Bytt mellom glOrtho, glFrustum og gluPerspective og modifiser verdiene.
  - Prøv gjerne dette i deres egen space shooter også.

Transformasjoner

- Normalized device space:
  - Hver komponent av *clip space* koordinatene divideres med den homogene koordinaten.
  - Dette kalles *perspective divide* siden den homogene koordinaten blir forandret i perspektivtransformasjonen.
  - Alle synlige primitiver blir transformert inn i en kubisk region, hvor hver komponent vil befinne seg i [-1, 1], [-1, 1], [-1, 1].



Transformasjoner

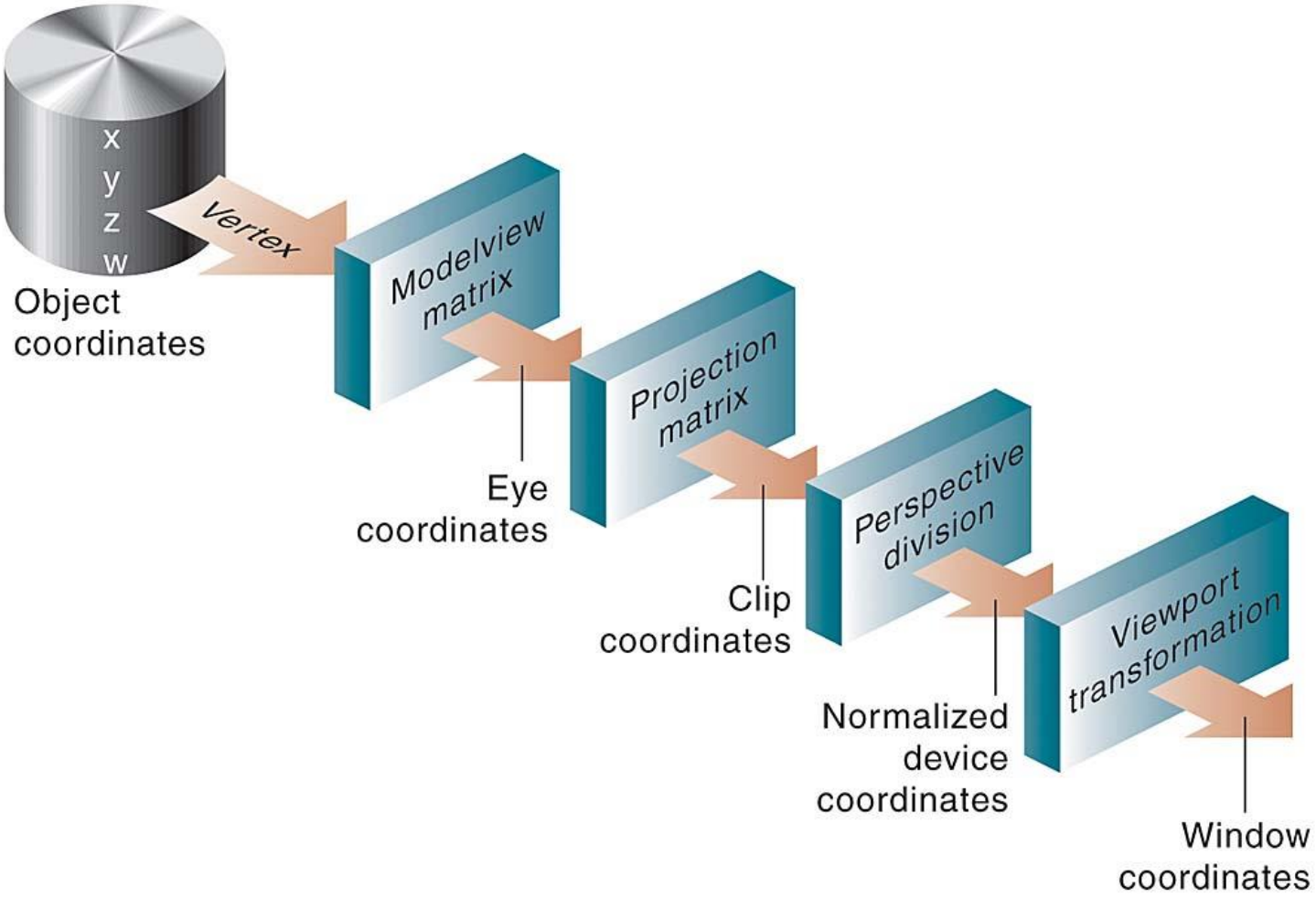
- Window space.
  - Det siste transformasjonssteget er transformasjonen fra *normalized device space* til *window space*.
  - Window space strekker seg fra
    [0, bredde-1] i x-retning og

    [0, høyde-1] i y-retning.

- Denne transformasjonen spesifiseres ved funksjonskallet:
  - glViewport(GLint x, GLint y, GLsizei width, GLsizei height);

    x og y spesifiserer nedre venstre hjørne mens width og height angir størrelsen på viewport-regionen.

Transformasjoner forts.

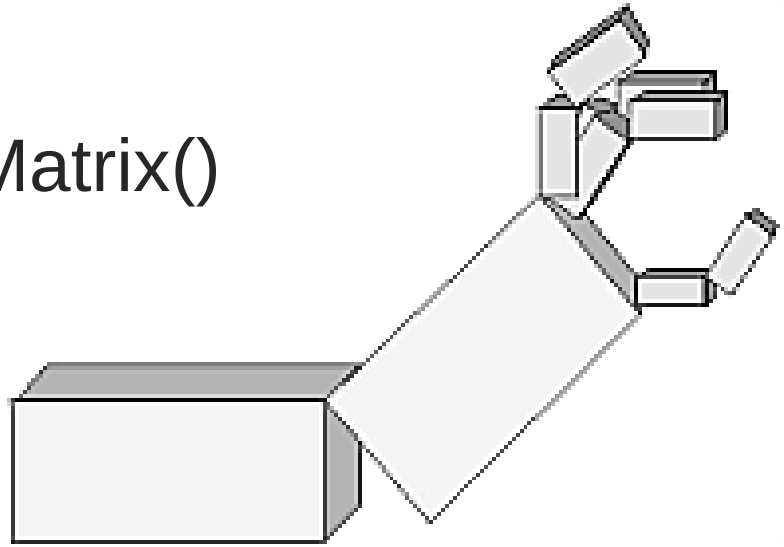


Koordinatsystemer og transformasjoner i OpenGL

PAUSE – 15 MIN

Lab: Robot-arm

- Ta utgangspunkt i lab-kildekoden på ITSL.
- Implementer en "robot-arm" i OpenGL.
  - Lag som vist på figuren: overarm, underarm, og fingre
- Bruk farger for å identifisere de forskjellige leddene
- Bruk glPushMatrix()/glPopMatrix()
- Pass på at boksene ikke går inn i hverandre men er hengslet riktig



Matrise-stack

- Bruk OpenGLs matrisestack

- glPushMatrix();
  - Skyver alle matrisene i matrise-stacken ned ett nivå. Den øverste matrisen blir kopiert, slik at innholdet blir duplisert i både den øverste og nest øverste matrisen.

- glPopMatrix();
  - Fjerner den øverste matrisen fra matrise-stacken slik at innholdet blir slettet. Verdiene i nest øverste matrise blir skrevet til den øverste matrisen i stacken.

Lab: Flere oppgaver

- Spesifiser en rotasjonsmatrise, en translasjonsmatrise, og en skaleringsmatrise og bruk flere glMultMatrix(...)
  for å utføre transformasjonene.

- Spesifiser en rotasjonsmatrise rundt aksen [1 1 1].

- Sjekk at du får samme resultat som med funksjonskallene glRotate, glTranslate og glScale.