En introduktion till OpenGL

Gustav Taxén
gustavt@csc.kth.se

2D1640 Grafik och Interaktionsprogrammering VT 2007

Labb 4 görs på PC!

Följ det nya labbpeket - eller:

Följ stegen i labbpeket för labb 5 för att kopiera upp katalogen med skalprogrammet från kurskatalogen, men använd katalogen "OpenGL" istället för "shaders".

Glöm inte att göra Visual Studio-inställningarna som beskrivs i peket för labb 5!

Vi börjar knyta ihop säcken...

- Signalbehandling, aliasing och filter
- Transformationer och projektioner
- Animation
- Händelsebaserad interaktion

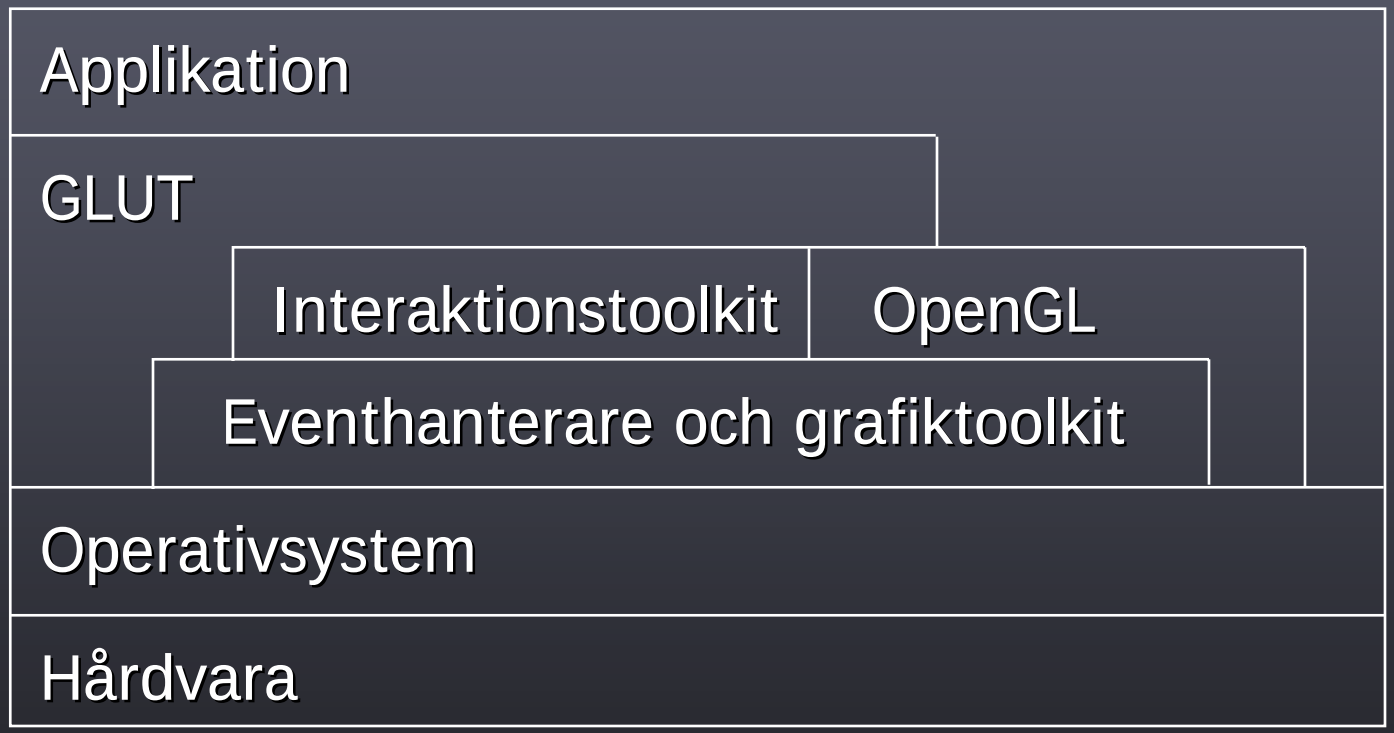
OpenGL-komponenter

- OpenGL
  - Renderingsfunktionalitet.
- GLU (OpenGL Utility Library)
  - Hjälpfunktioner för bl.a. transformationer.
- GLX / GLw
  - Sköter datautbyte mellan operativsystemet och OpenGL.
- GLUT
  - Portabelt bibliotek tredjepartsbibliotek som gömmer systemberoende delar + fönsterhantering + inenheter.

GLUT

- Mål: att **exakt** samma OpenGL-kod ska kunna kompileras på alla plattformar!
- Funktioner:
  - Starta OpenGL.
  - Öppna fönster (eller fullskärm).
  - Hantera mus, tangentbord, joystick, m.m.
  - Timer.
  - Rita boxar, sfärer, tekannor, m.m.

GLUT


Applikation
GLUT
Interaktionstoolkit
OpenGL
Eventhanterare och grafiktoolkit
Operativsystem
Hårdvara

Minimalt GLUT-program

```
#include <GL/glut.h>

void display(void) {
   /* Rita om fönster med OpenGL */
}

int main(int argc, char *argv[]) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH);
   glutCreateWindow("My window");
   glutDisplayFunc(display);
   glutMainLoop();
   return 0;
}
```

Dubbelbuffring och djupbuffert (mer senare)

Registrera callback för omritning

Events i GLUT

- Events kan indelas i tre typer:
  - Fönster
  - Menyer
  - Globala

Några GLUT-events för fönster

- Redisplay (måste fångas av alla fönster)
- Reshape
- Keypress
- Mouse button
- Passsive mouse movement
- Active mouse movement
- Entry

```
void display(void) {
    ...
}
void reshape(int width, int height) {
    ...
}

int main(int argc, char *argv[]) {
    ...
    glutCreateWindow(”My Window”);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    ...
}
```

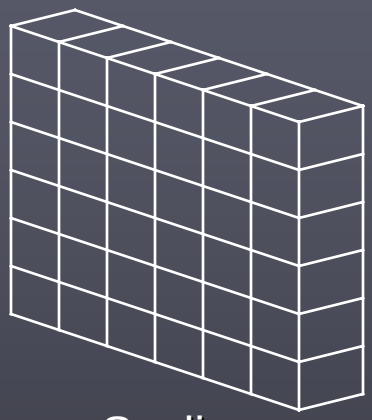
Animation

- Rita om fönstret regelbundet
- GLUT har en "idle"-event (global) som skapas automatiskt då inget annat händer
- Forcera en omritning i en idle-callback:

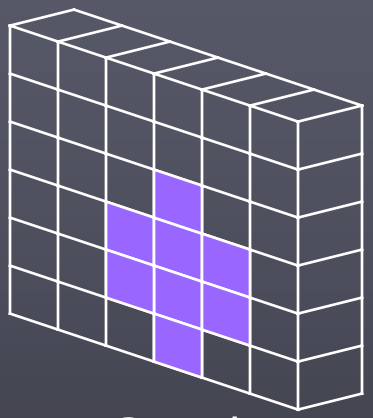
```
void idle(void) {
  ...
  glutPostRedisplay();
}

int main(...) {
  ...
  glutIdleFunc(idle);
  ...
}
```

Animation och dubbelbuffring

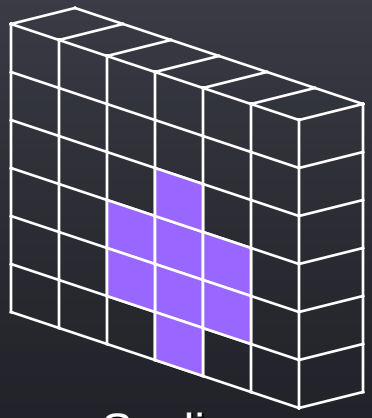
```
void display(void) {
  /* rensa fönstret */
  /* rita med OpenGL */

  ...

  glutSwapBuffers();
}
```



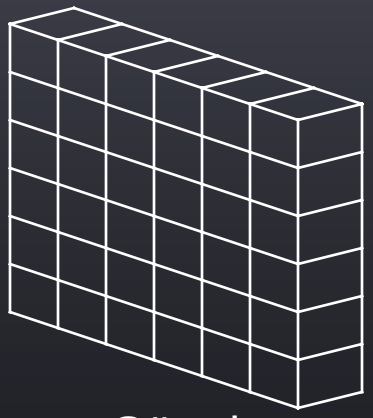
Synlig



Gömd



Synlig



Gömd

Översikt över OpenGL

TV-signal

Volym
Kontrast
Bildformat
...

Bild och ljud

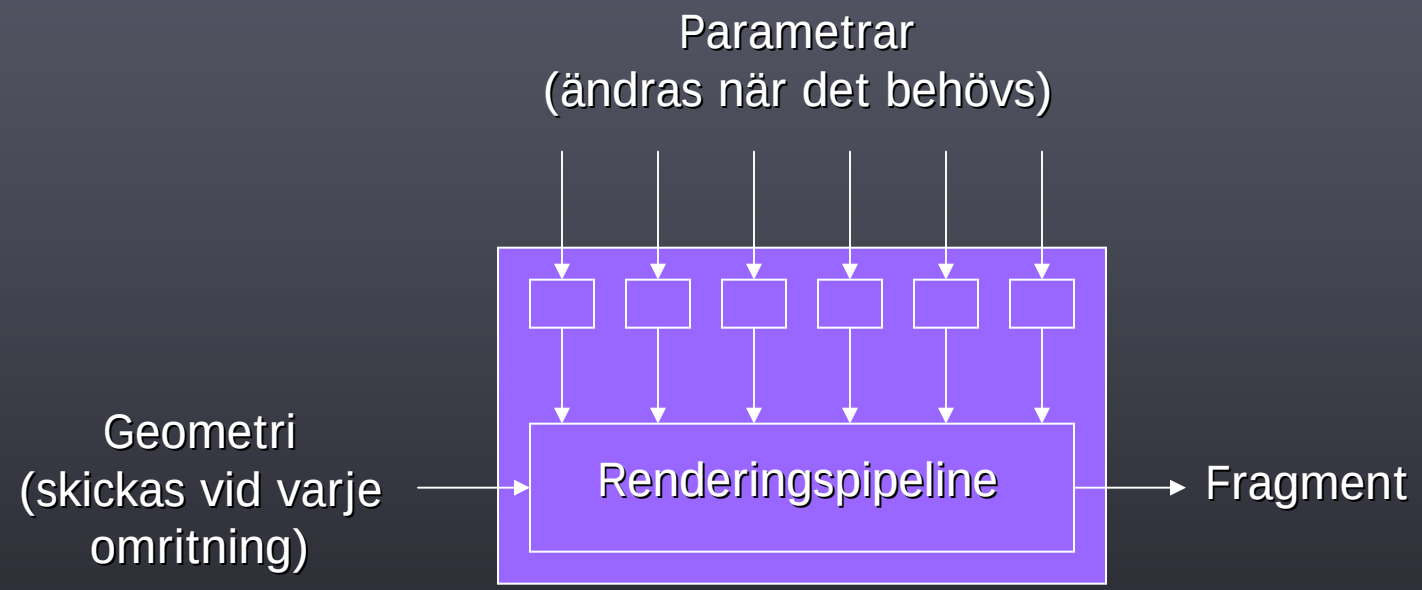
Hörn

Normal
Färg
Transformations-
matris
Projektionsmatris
...

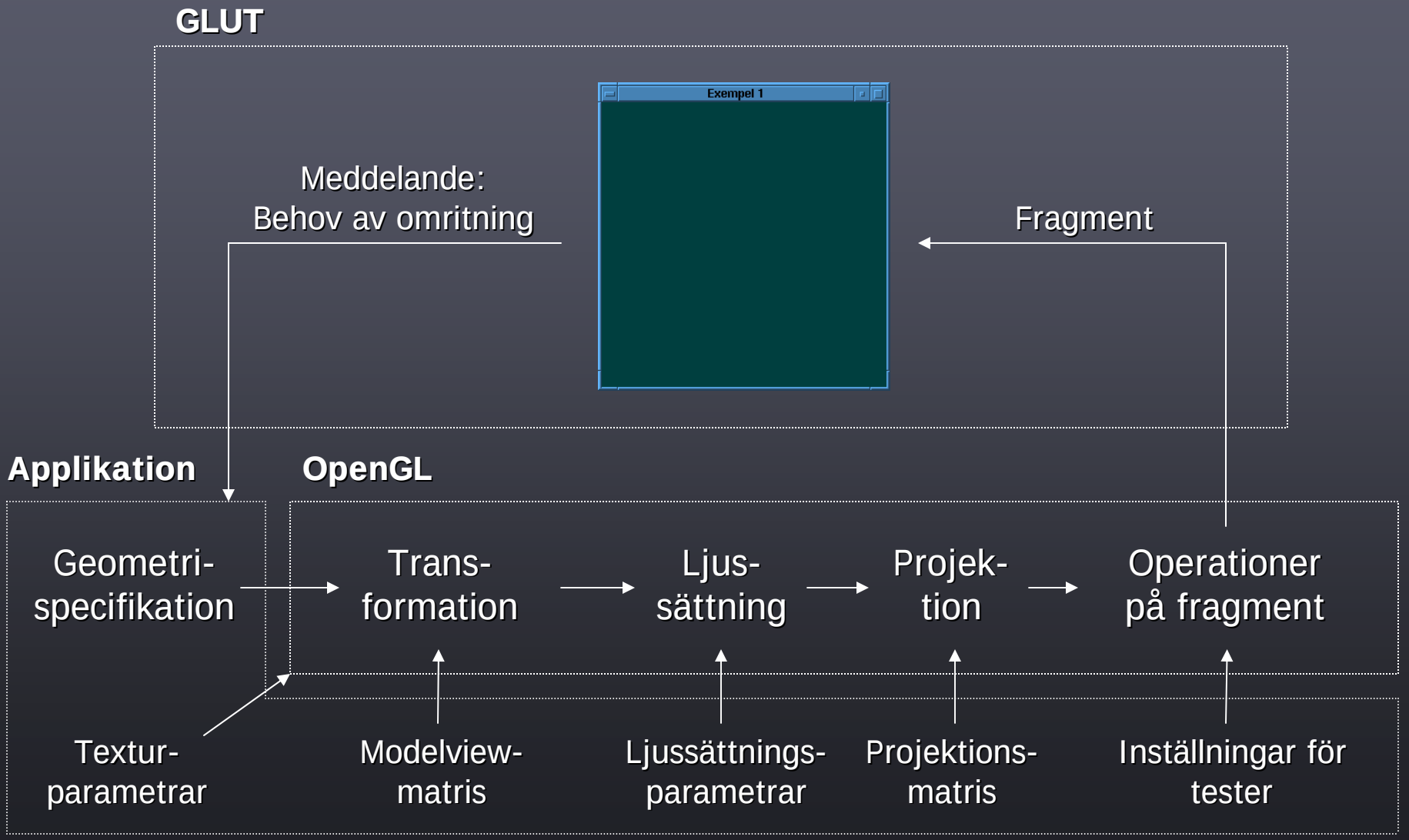
OpenGL

Fragment

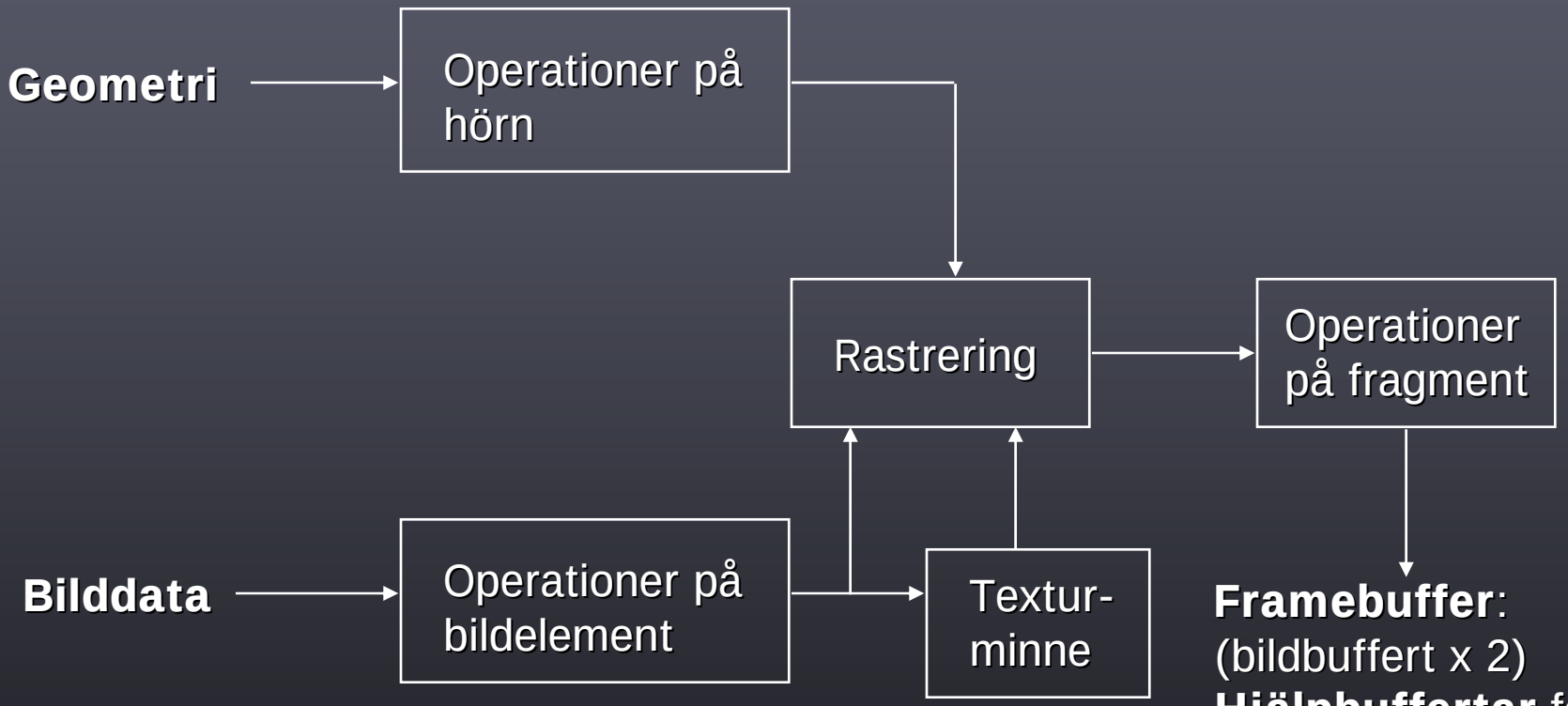
Översikt över OpenGL



Översikt över OpenGL

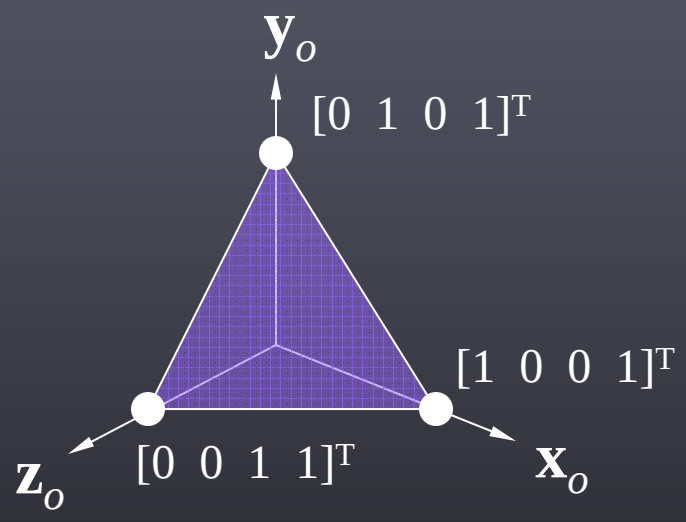


Översikt över OpenGL



Geometrispecifikation

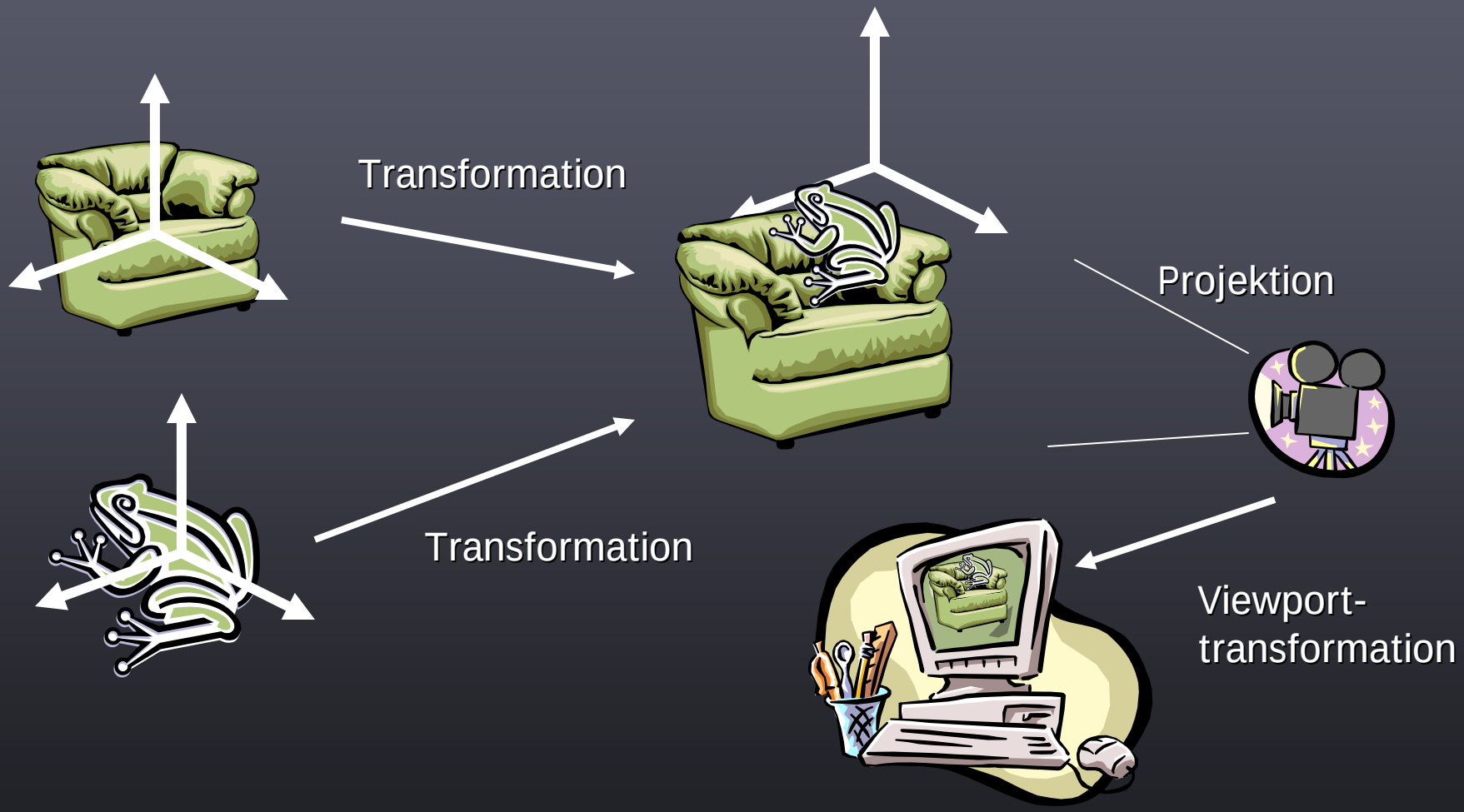
```
glBegin(GL_TRIANGLES);
glVertex3f(1.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd();
```



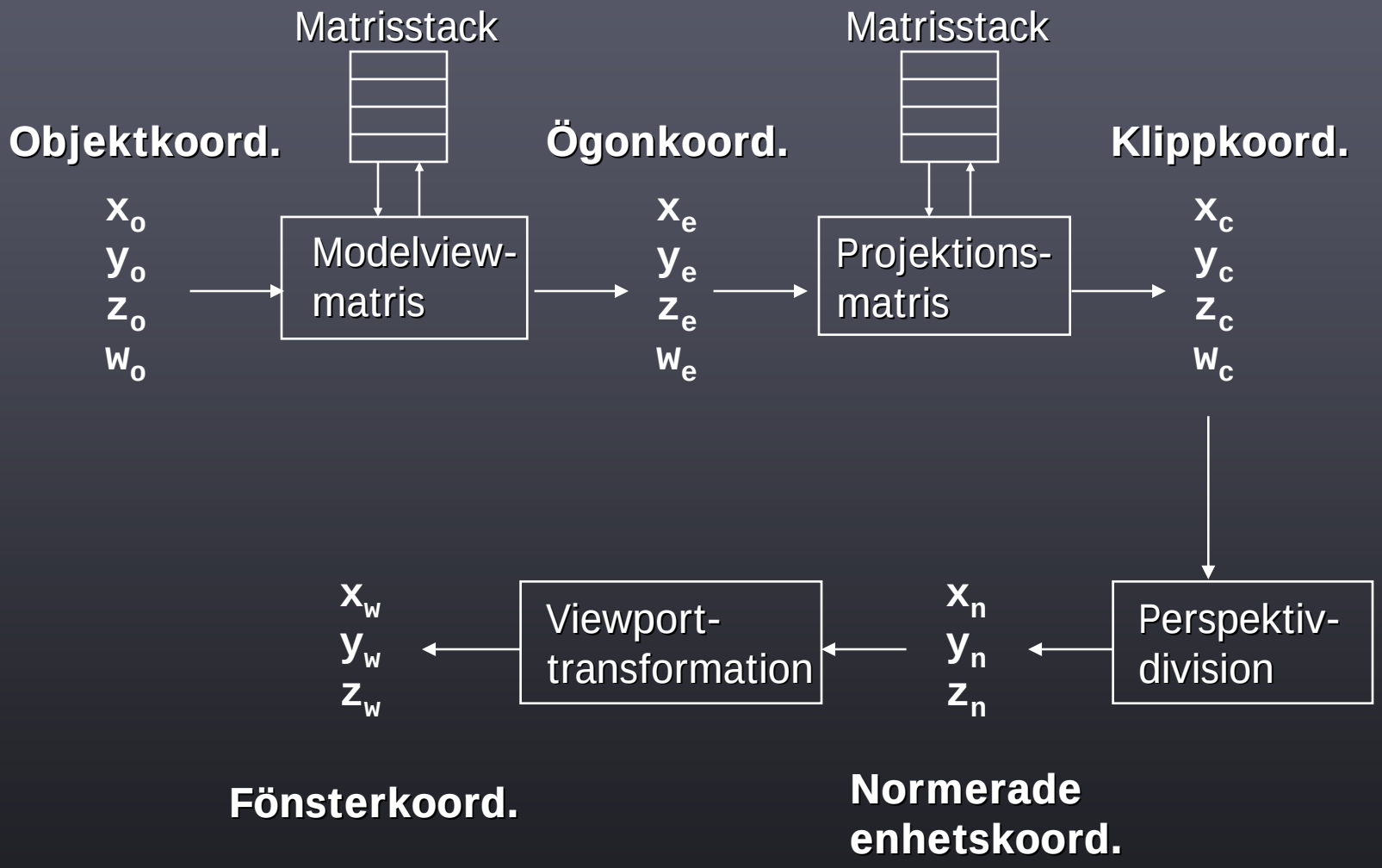
Hörn anges alltid i objektkoordinater.

Anger man ingen w-koordinat sätts den till 1.

Transformationer



Transformationer



Modelview-matrisen

- Modelview-matrisen $\mathbf{C}$ omvandlar objektkoordinater till **ögonkoordinater**: $\mathbf{v}_e = \mathbf{C}\mathbf{v}_o$
- Matrisen $\mathbf{C}$ kallade vi för $\mathbf{M}^{\mathrm{T}}$ förra föreläsningen!
- Normaler transformeras också (mer senare).
- Ljussättning görs i ögonkoordinater (mer senare).

Projektionsmatrisen

- Projektionsmatrisen $\mathbf{P}$ omvandlar **ögonkoordinater** till **klippkoordinater:** $\mathbf{v}_c = \mathbf{P}\mathbf{v}_e$
- $\mathbf{P}$ projicerar hörnen på ett bildplan och definierar samtidigt en volym (**view volume**). Alla primitiver klipps mot denna volym.
- **Kameran sitter i "världsorigo" och "tittar" längs med negativa z-axeln.**

Perspektivdivision och viewporttransformation

- Perspektivdivision:
  $[x_n \;\; y_n \;\; z_n]^{\mathrm{T}} = [(x_c / w_c) \;\; (y_c / w_c) \;\; (z_c / w_c)]^{\mathrm{T}}$
- Perspektivdivisionen ger **normerade enhetskoordinater** i intervallet [-1, 1].

- Viewport-transformation: skalning och offset av de normerade enhetskoordinaterna ger **fönsterkoordinater**.
- Skalning och offset är i 2D, så $z_w = z_n$

```
void display(void) {
    glClearColor(0.1, 0.2, 0.3, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);

    glColor3f(1.0, 1.0, 1.0);

    glBegin(GL_TRIANGLES);
    glVertex3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glVertex3f(0.0, 0.0, 1.0);
    glEnd();

    glutSwapBuffers();
}
```

Rensa fönstret

Specificera transformations-matrisen

Sätt aktuell färg

Rita en triangel

Byt plats på bakre och främre buffer

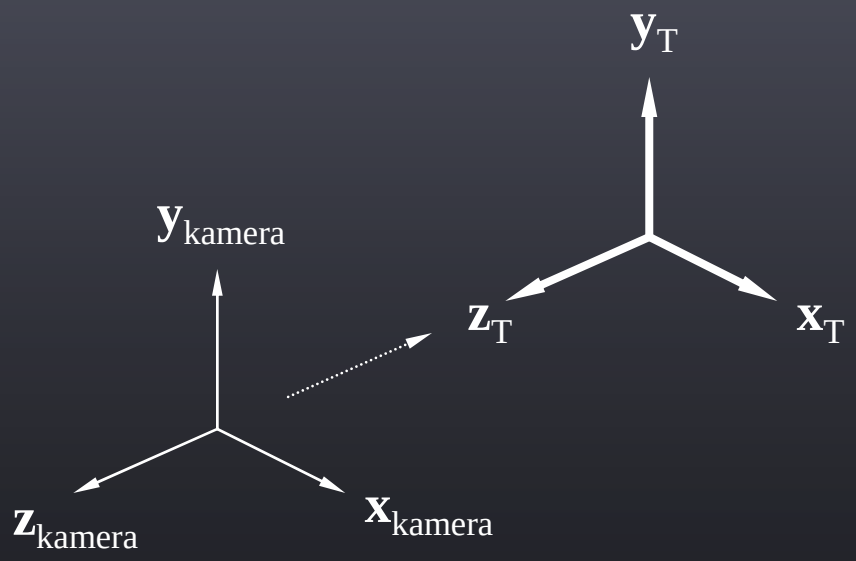
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -3.0);
```

t_x t_y t_z

C = **I** = Identitetsmatrisen

C = **CT** = **T**

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



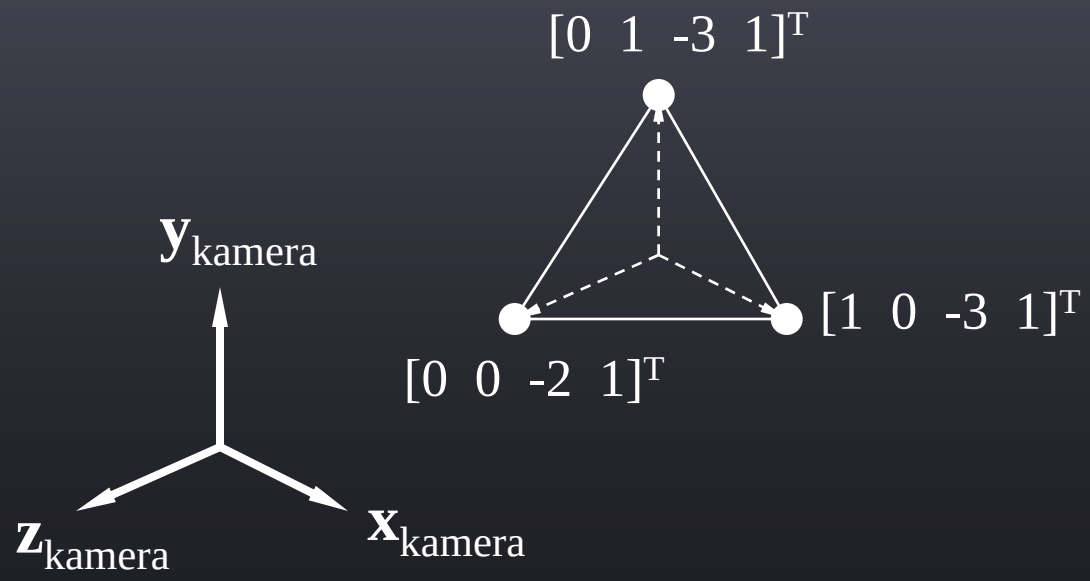
```
glBegin(GL_TRIANGLES);
glVertex3f(1.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glVertex3f(0.0, 0.0, 1.0);
glEnd();
```

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

Hörnen i ögonkoordinater...

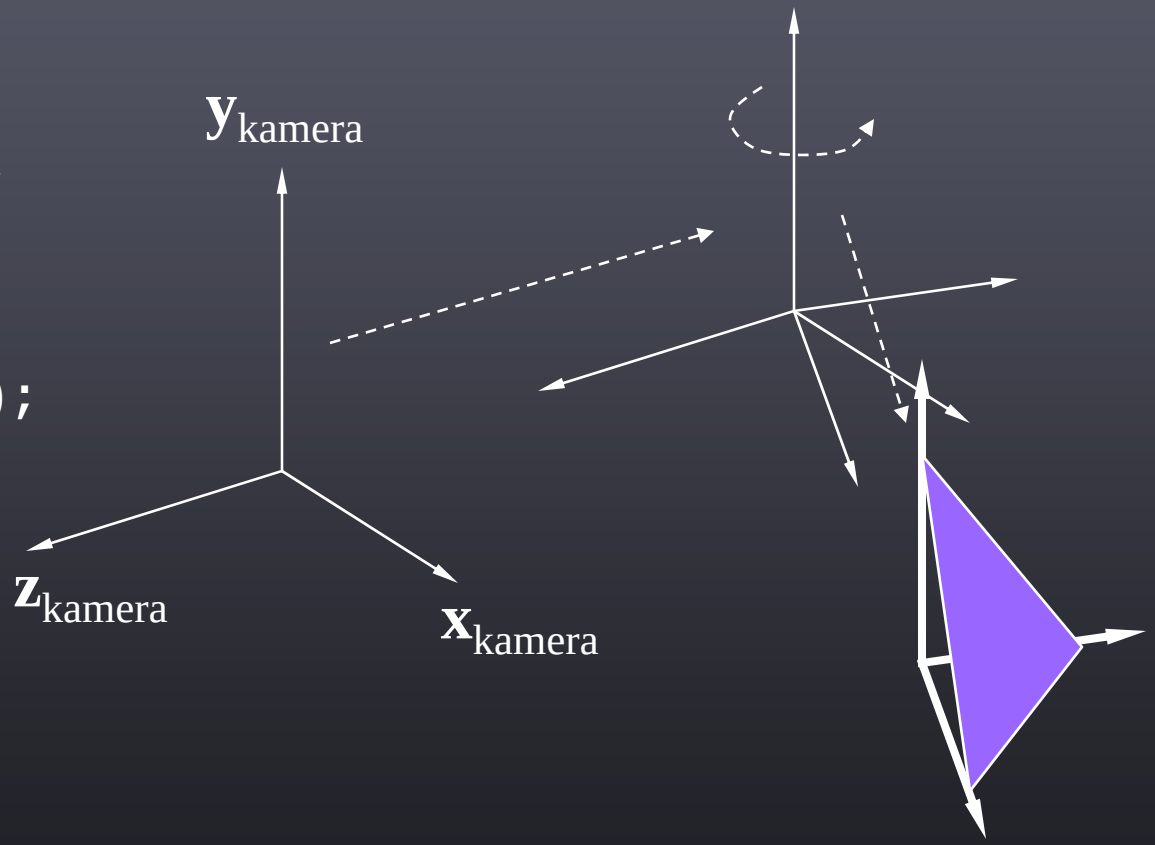
...multipliceras med $\mathbf{M}^T$...

...vilket ger triangelns transformerade hörn i **ögonkoordinater**:



Transformation är ekvivalent med byte av koordinatsystem!

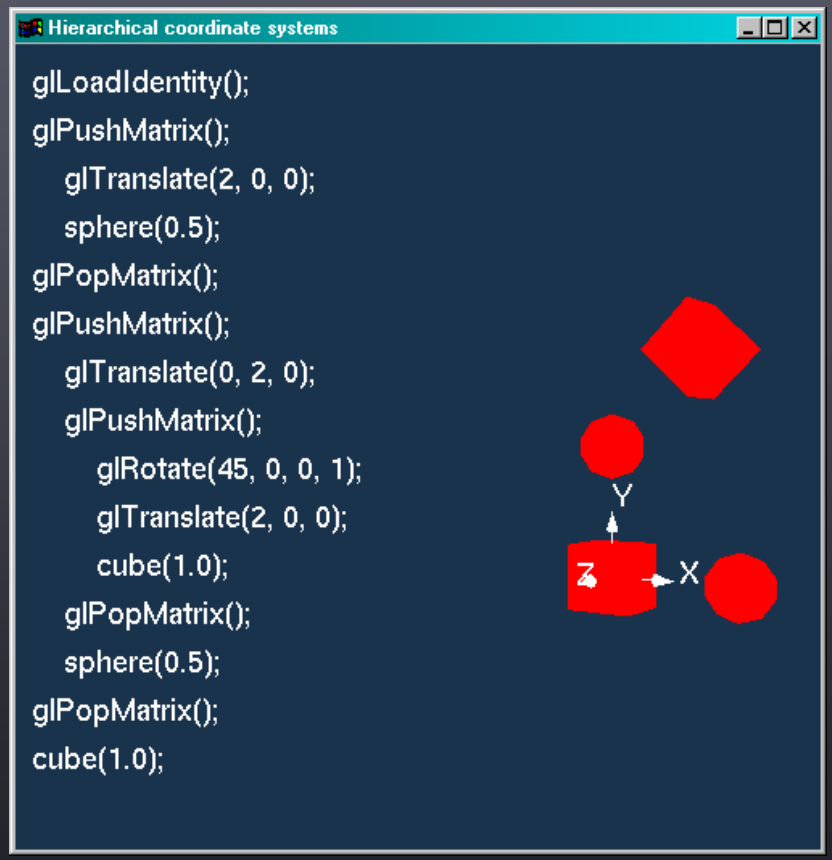
```
glLoadIdentity();
glTranslatef(0,0,-3);
glRotatef(50,0,1,0);
glTranslatef(0,0,3);

glBegin(GL_TRIANGLES);
glVertex3f(1,0,0);
glVertex3f(0,1,0);
glVertex3f(0,0,1);
glEnd();
```



`glPushMatrix` *pushar aktuellt koordinatsystem på stacken.*

`glPopMatrix` *ersätter aktuellt koordinatsystem med det som finns överst på stacken.*

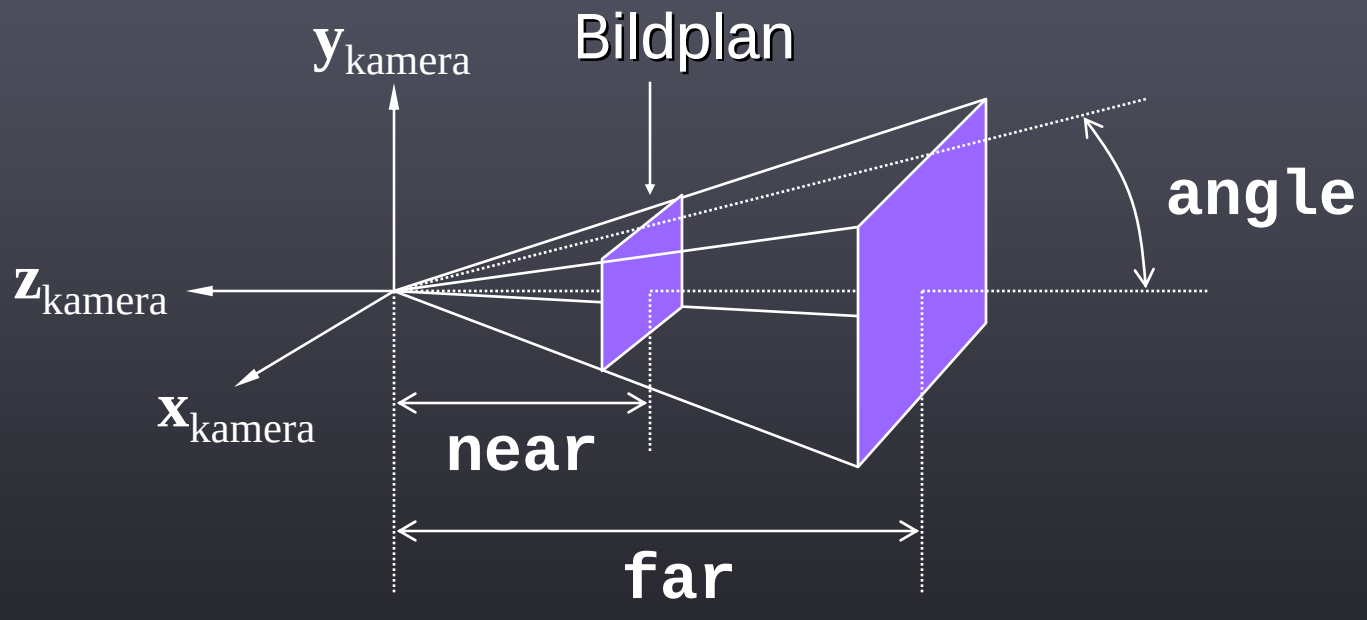
Demo


Hierarchical coordinate systems
glLoadIdentity();
glPushMatrix();
glTranslate(2, 0, 0);
sphere(0.5);
glPopMatrix();
glPushMatrix();
glTranslate(0, 2, 0);
glPushMatrix();
glRotate(45, 0, 0, 1);
glTranslate(2, 0, 0);
cube(1.0);
glPopMatrix();
sphere(0.5);
glPopMatrix();
cube(1.0);
Y
Z
X

```
void reshape(int width, int height) {
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45,
                  (GLfloat) width / (GLfloat) height,
                  0.5,
                  10.0);
   glViewport(0, 0, width, height);
}

int main(int argc, char *argv[]) {
   ...
   glutCreateWindow(“My Window");
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   ...
}
```

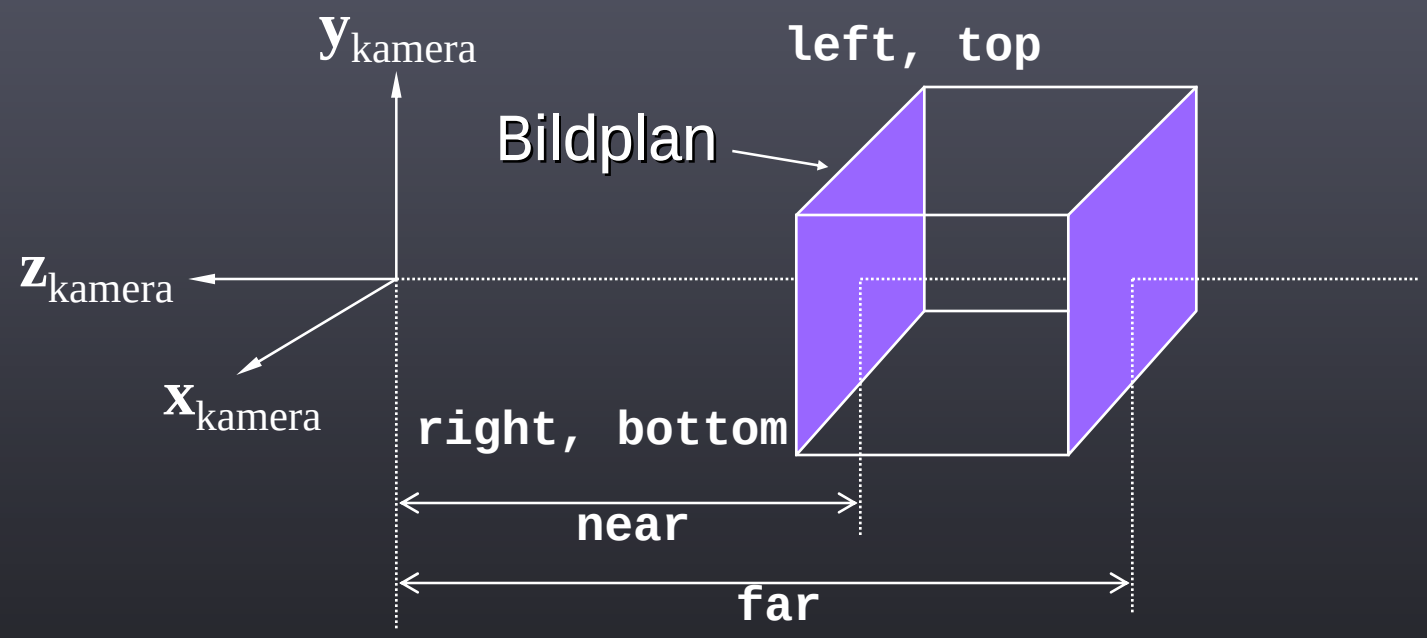
Perspektivprojektion

```
gluPerspective(angle, aspect, near, far);
```



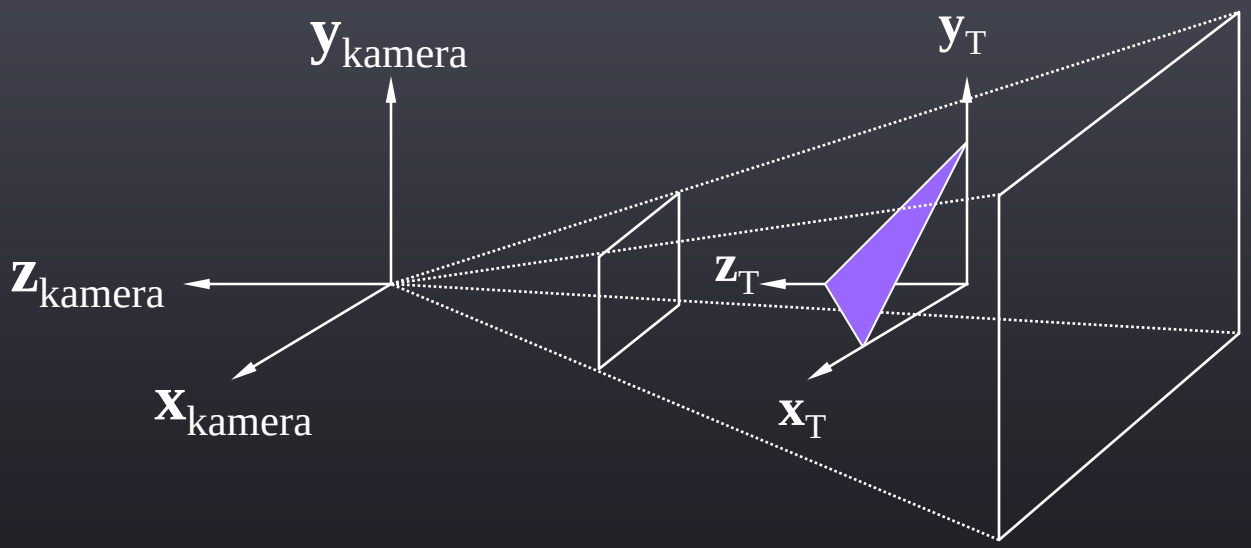
Ortogonalprojektion

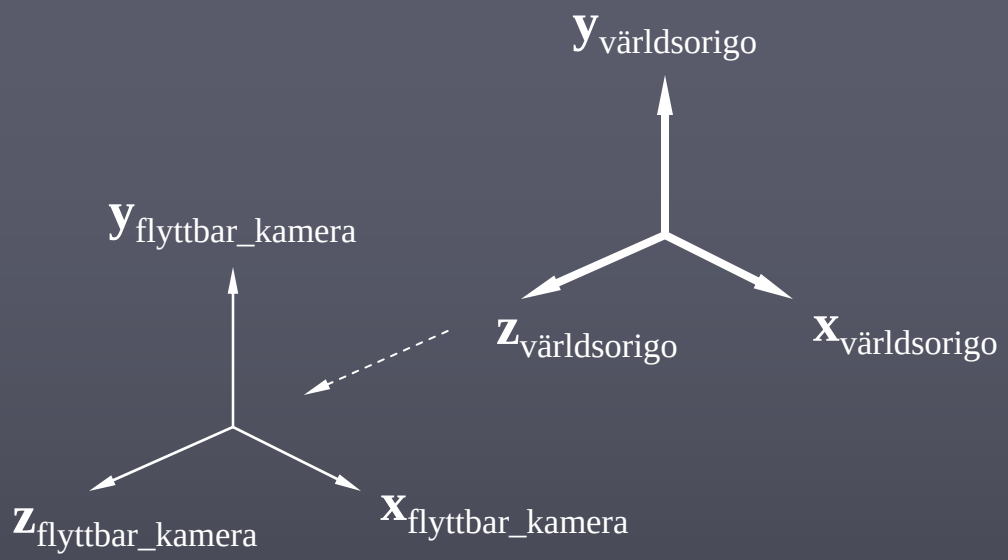
```
glOrtho(left, right, bottom, top, near, far);
```



I OpenGL flyttar man föremålen så de hamnar framför kameran!

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -3.0);
```

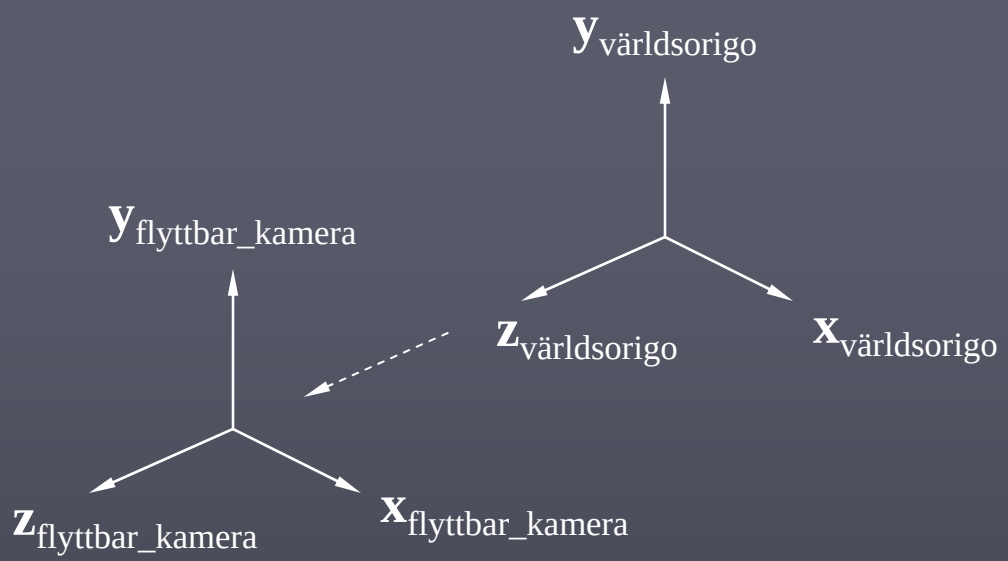




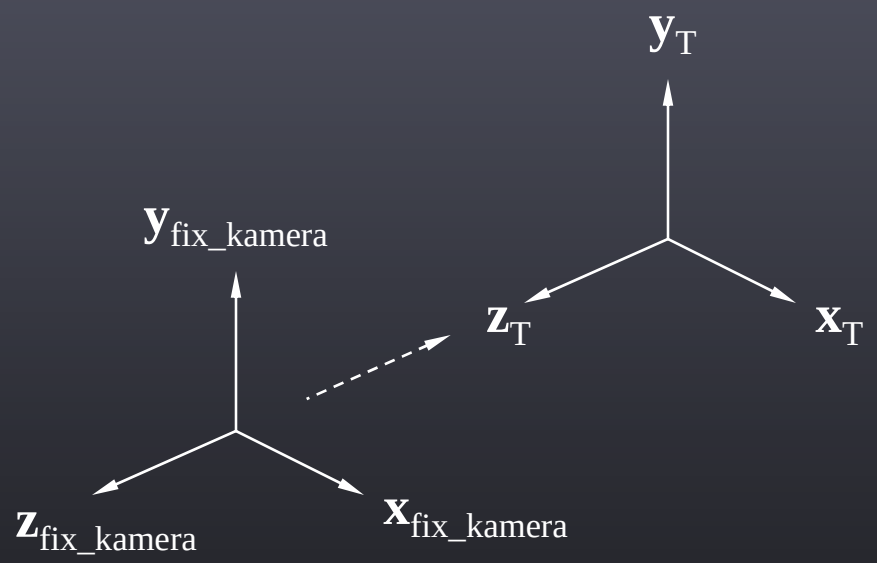
Hur gör vi om vi vill kunna flytta kameran och titta på "världens centrum" istället?

Antag att vi placerar kameran 3 enheter längs med $\mathbf{z}_{\text{världsorigo}}$.
Vi ritar nu en triangel i "världskoordinater".
Vilka "flyttbar-kamera-koordinater" får vi?

$$\begin{bmatrix} x_{flyttbar\_kamera} \\ y_{flyttbar\_kamera} \\ z_{flyttbar\_kamera} \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} x_{värld} \\ y_{värld} \\ z_{värld} \\ 1 \end{bmatrix} \longrightarrow \mathbf{K} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$\mathbf{K} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tydligen är **T** ekvivalent med att flytta kameran
+3 enheter längs med "världs-**z**"!

I allmänhet gäller att

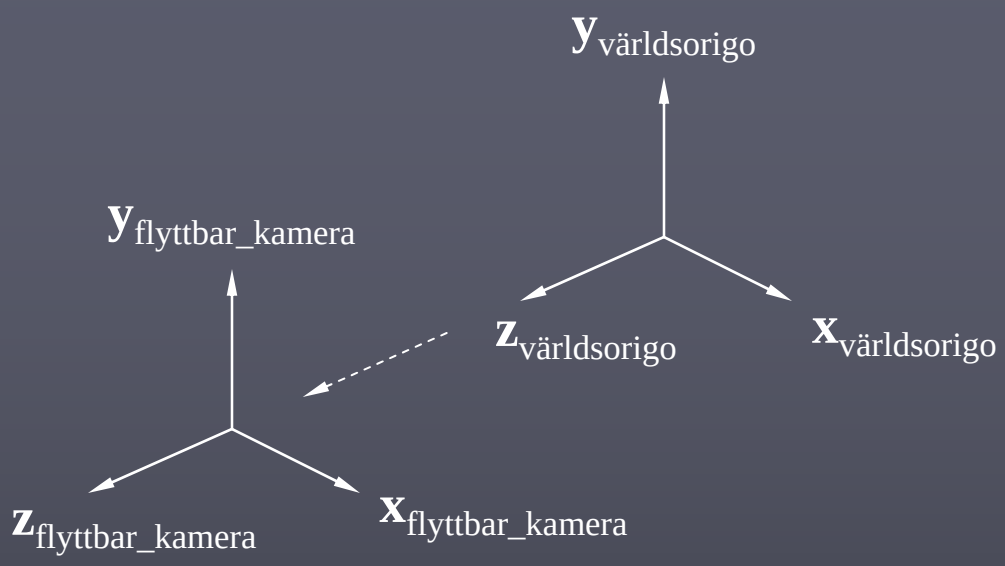
"Flytta världen framför kameran"

är ekvivalent med

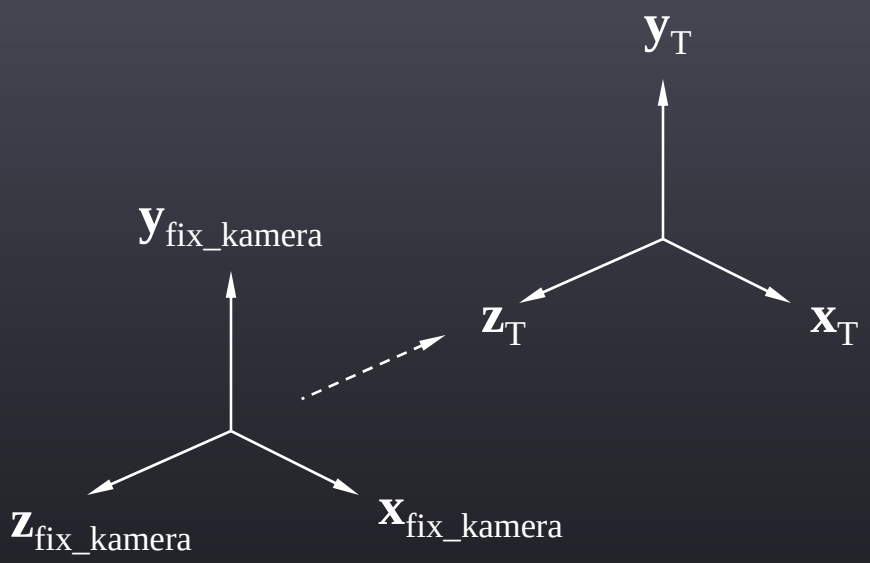
"Flytta kameran åt motsatt håll"

```
gluLookAt(campos_x, campos_x, campos_x,
          at_x, at_y, at_z,
          up_x, up_y, up_z);
```

skapar en sådan "omvänd" matris och
multiplicerar den med modelviewmatrisen.



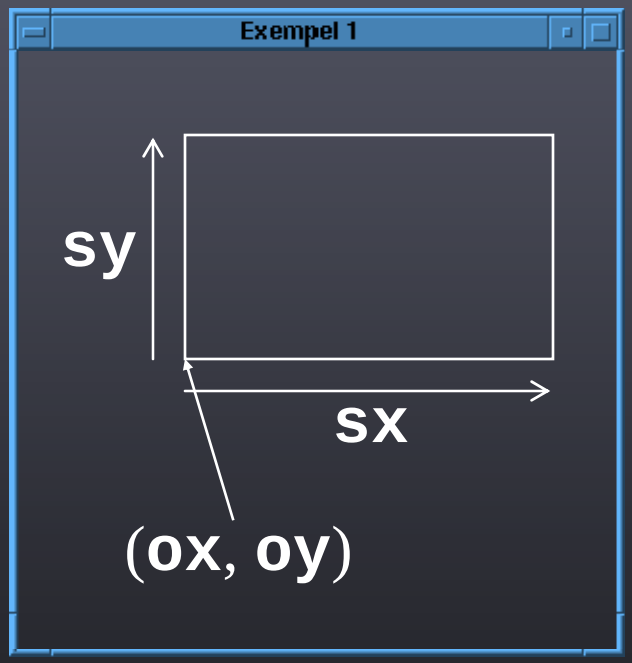

```
gluLookAt(0, 0, 3,
          0, 0, 0,
          0, 1, 0);
```

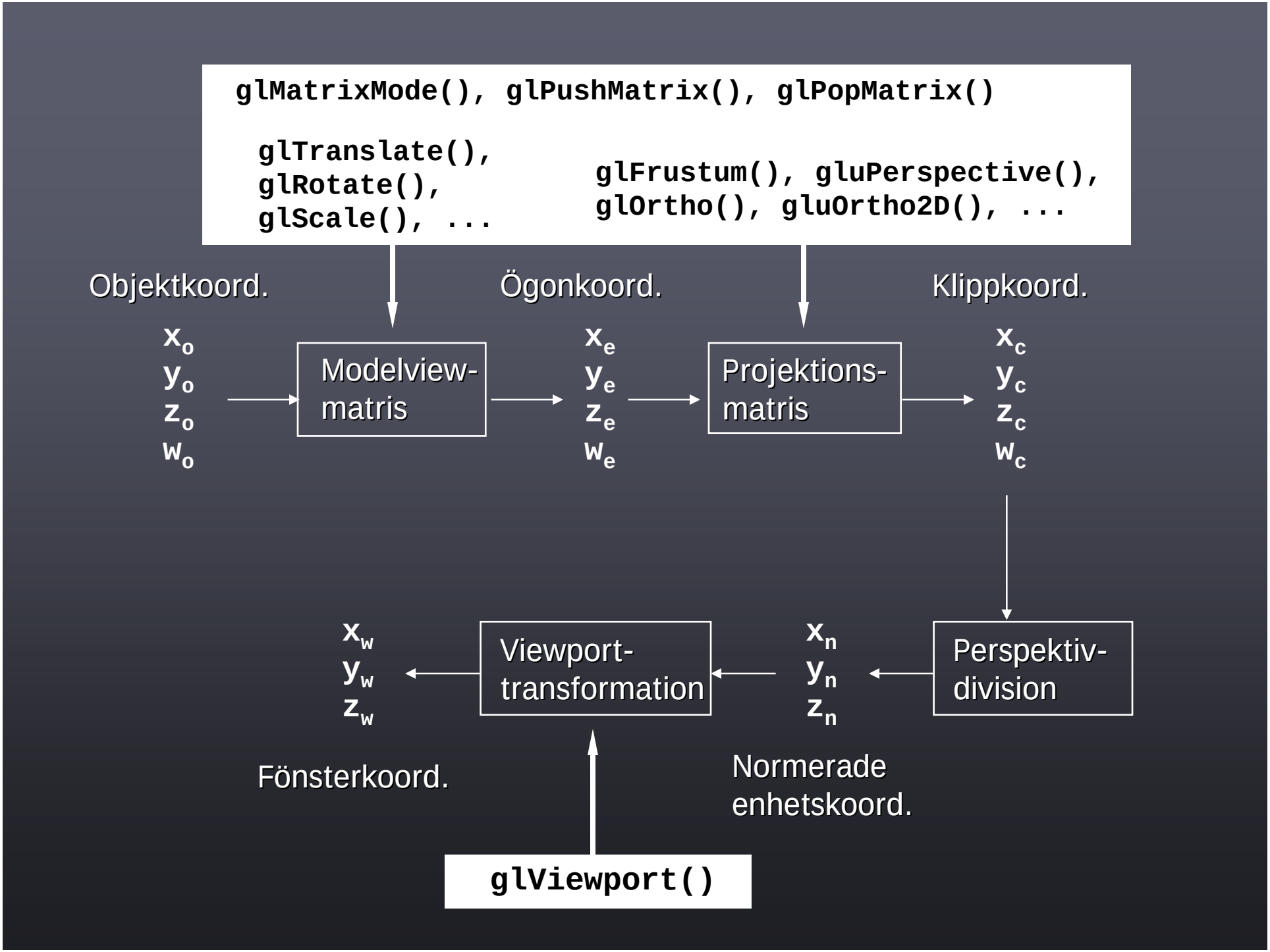


$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

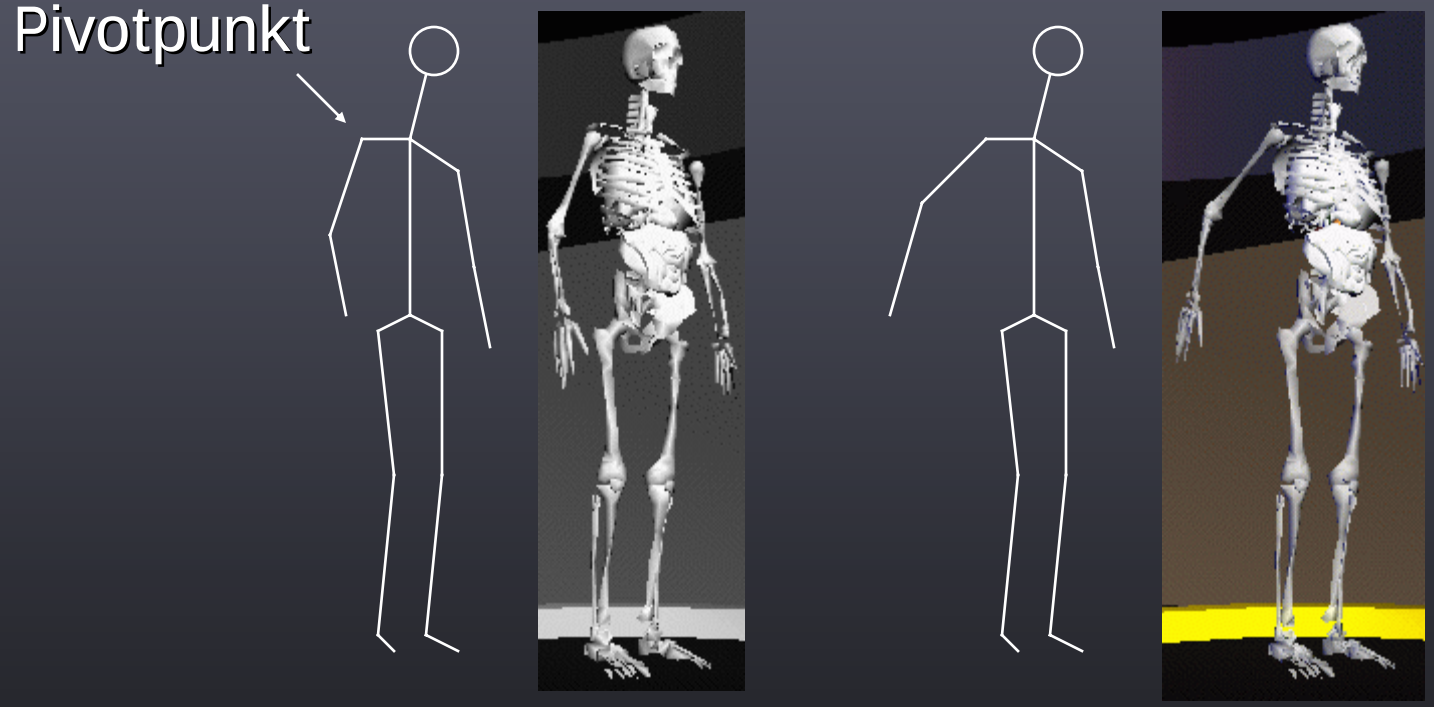
Viewport-transformation

```
glViewport(ox, oy, sx, sy);
```

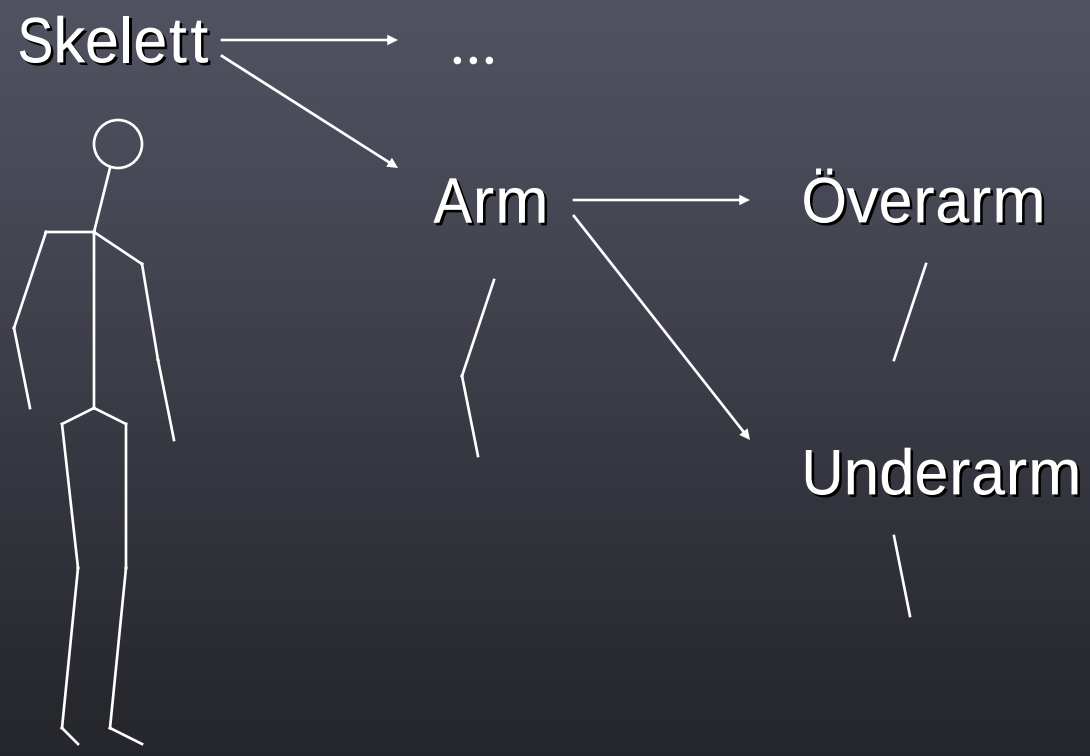



glMatrixMode(), glPushMatrix(), glPopMatrix()
glTranslate(),
glRotate(),
glScale(), ...
glFrustum(), gluPerspective(),
glOrtho(), gluOrtho2D(), ...
Objektkoord.
Ögonkoord.
Klippkoord.
x_o y_o z_o w_o
Modelview-matris
x_e y_e z_e w_e
Projektions-matris
x_c y_c z_c w_c
Perspektiv-division
x_n y_n z_n
Normerade enhetskoord.
Viewport-transformation
x_w y_w z_w
Fönsterkoord.
glViewport()

Hierarkiska modeller

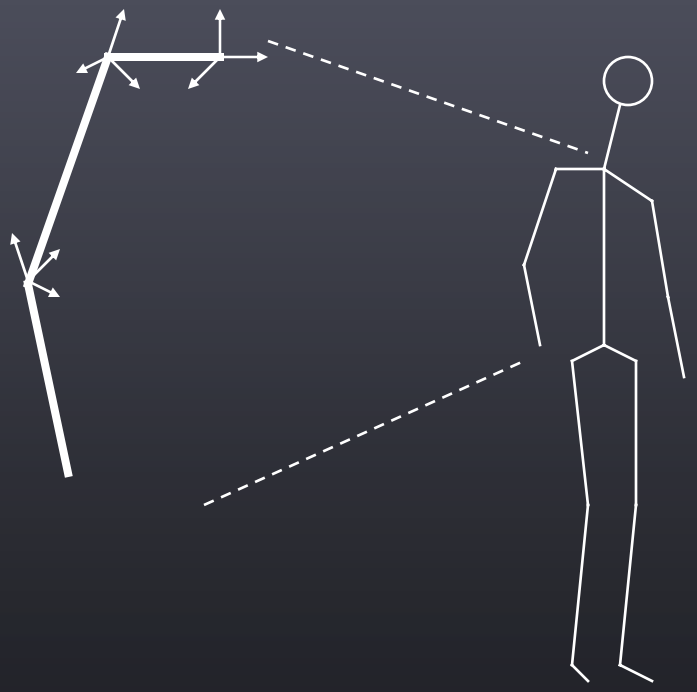


Består-av-hierarki

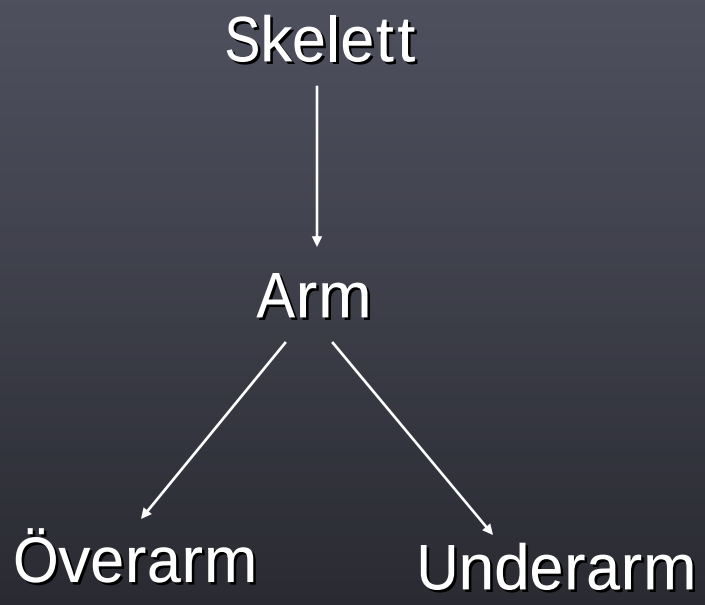


Lokala koordinatsystem

Subnodens origo positioneras m.a.p.
förälderns koordinatsystem.

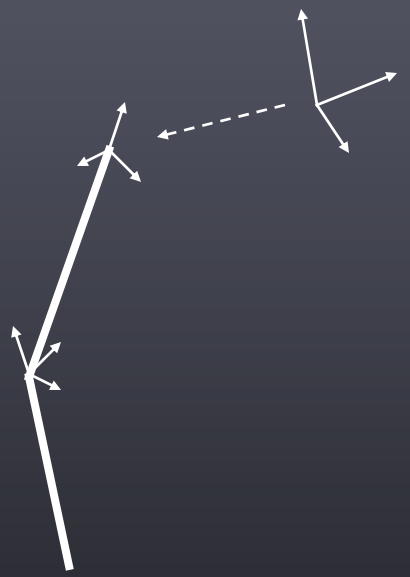


Från hierarki till kod



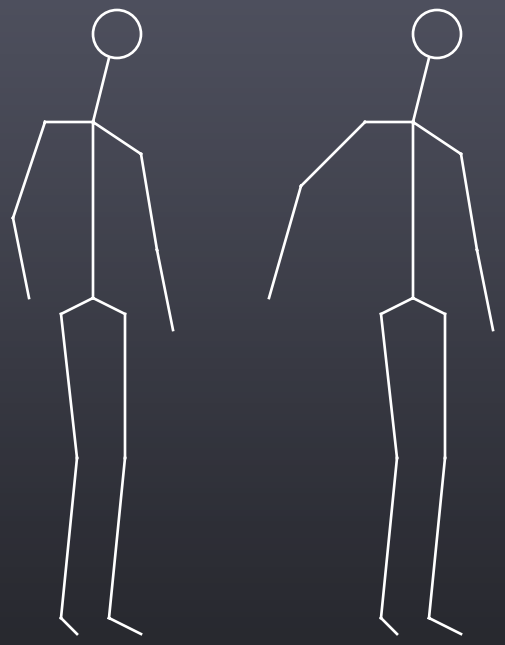

```
void drawSkeleton(void) {
   drawArm();
}

void drawArm(void) {
   drawUpperArm();
   drawLowerArm();
}

void drawUpperArm(void) {
}

void drawLowerArm(void) {
}
```

Från hierarki till kod



```
void drawSkeleton(void) {
    glPushMatrix();
    positionera armens origo;
    drawArm();
    glPopMatrix();
}

void drawArm(void) {
    glPushMatrix();
    positionera överarmens origo;
    drawUpperArm();
    glPopMatrix();

    glPushMatrix();
    positionera underarmens origo;
    drawLowerArm();
    glPopMatrix();
}
```

Från hierarki till kod

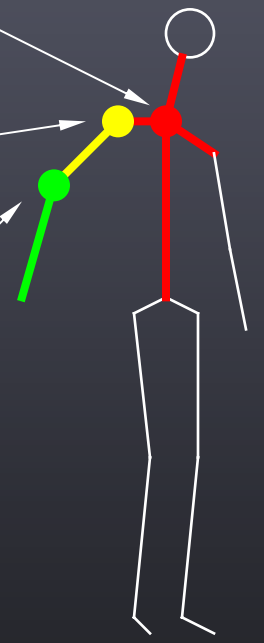


Ändra rotation här!

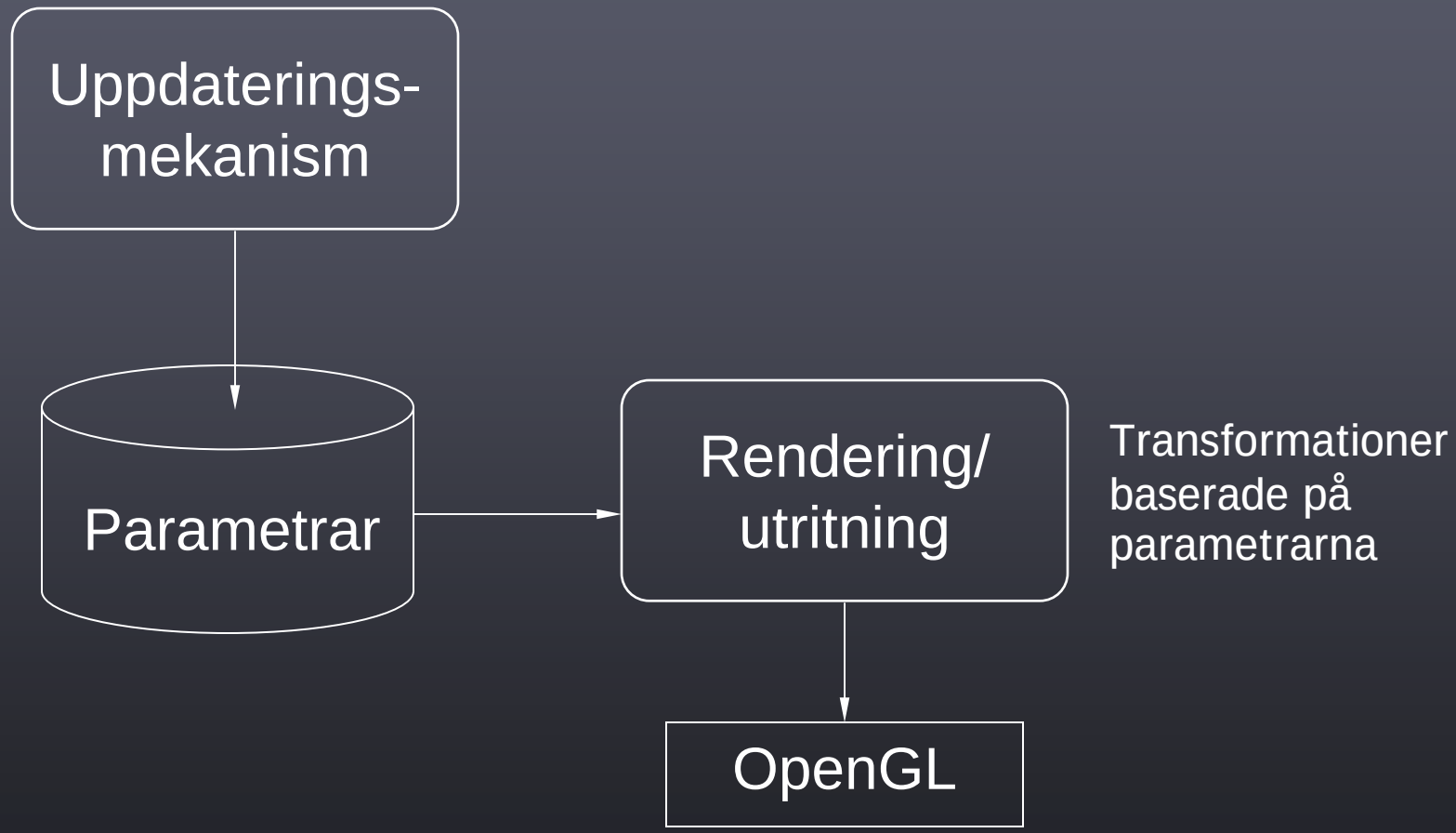
```
void drawSkeleton(void) {
   glPushMatrix ();
   positionera armens origo;
   drawArm ();
   glPopMatrix ();
}
```

Från hierarki till kod

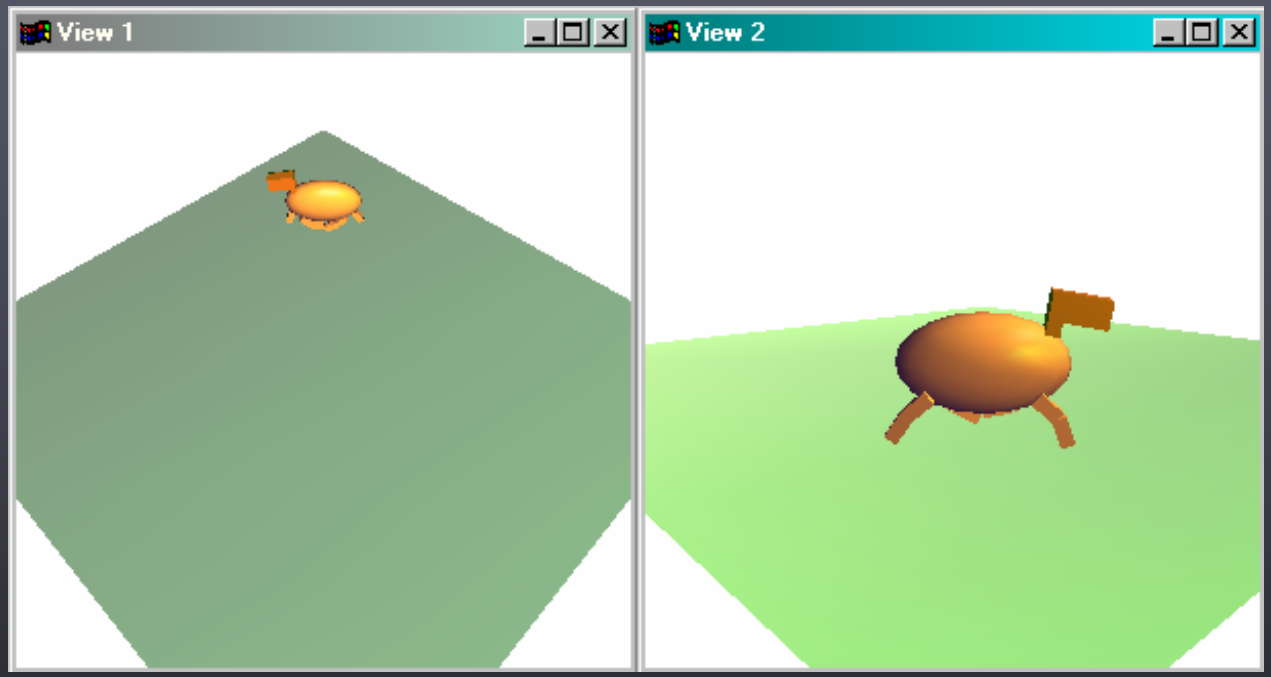
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glPushMatrix();
glTranslatef(modellens position);
rita torso;

  glPushMatrix();
  glTranslatef(pivotpunkt för överarmen);
  glRotatef(grader, rotationsaxel);
  rita överarmen;

    glPushMatrix();
    glTranslatef(pivotpunkt för underarmen);
    glRotatef(grader, rotationsaxel);
    rita underarmen;
    glPopMatrix();

  glPopMatrix();

glPopMatrix();
```



Animation


Uppdaterings-
mekanism
Parametrar
Rendering/
utritning
Transformationer
baserade på
parametrarna
OpenGL

Demo - forward kinematics

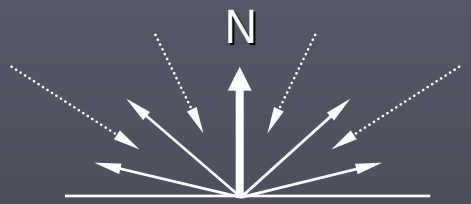


$param_0 = f_0(t)$
$param_1 = f_1(t)$
...

Kända

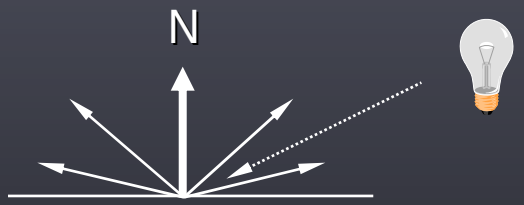
Ljussättning – Phongs ljusmodell

Ambient
(Allmänljus)



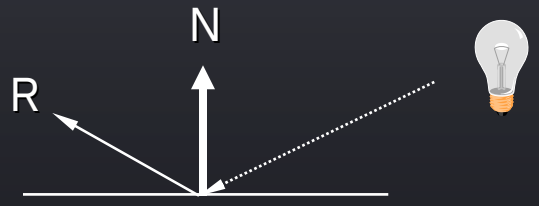
+

Diffuse
(Matt reflektion)

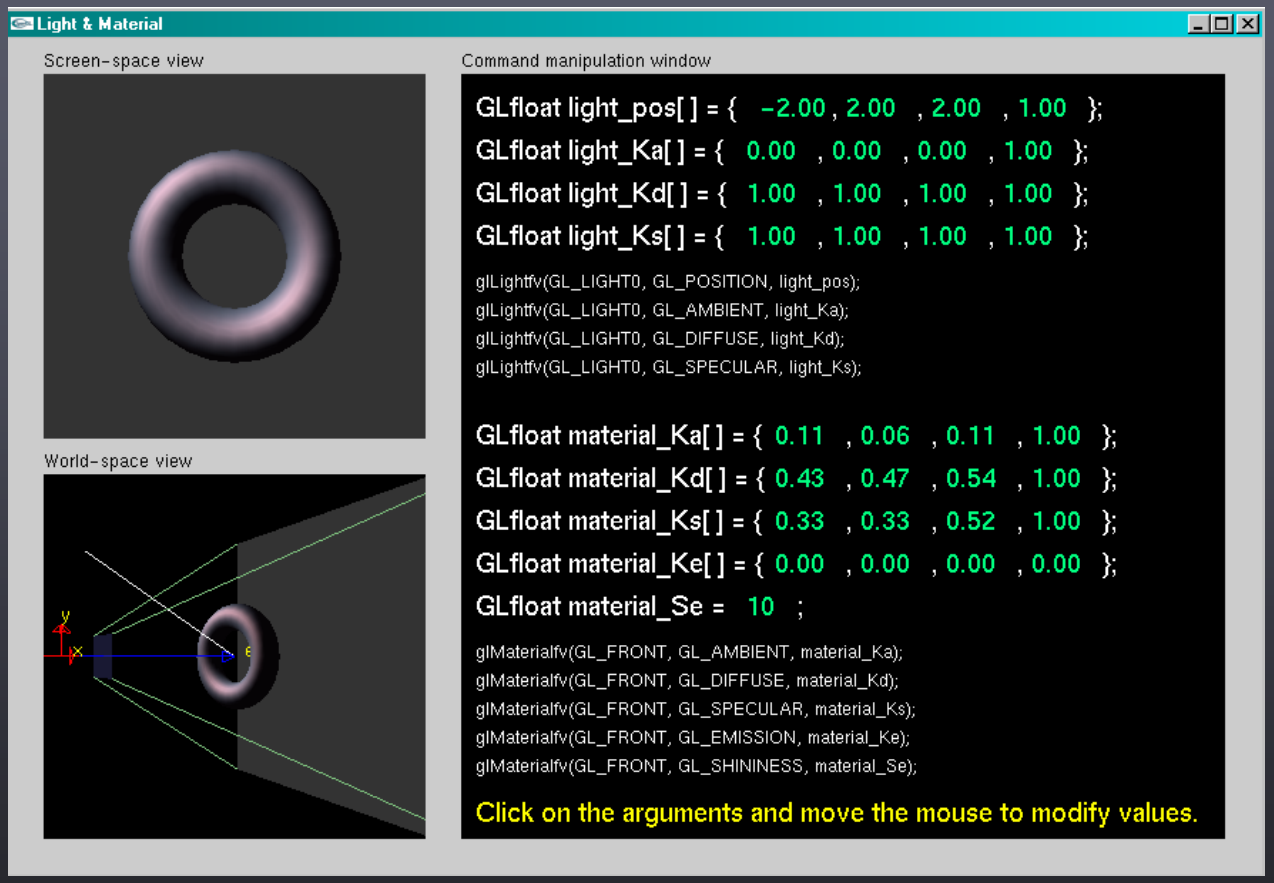


+

Specular
(Highlights)

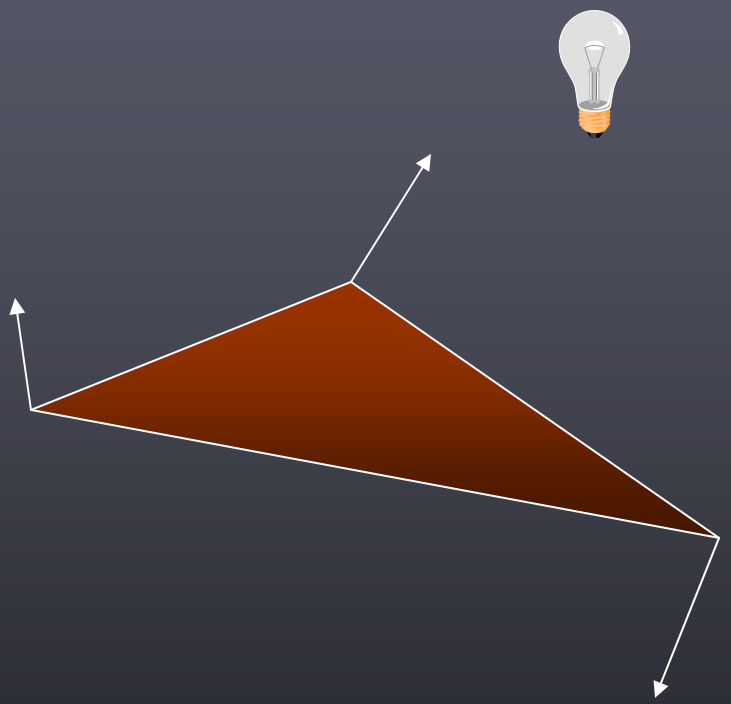


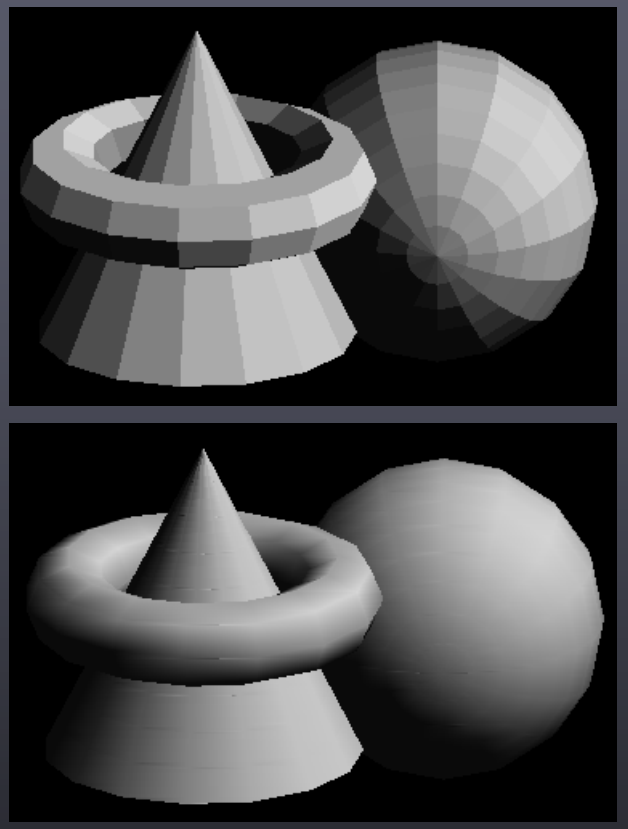
Demo



http://www.xmission.com/~nate/opengl.html

Goraud shading



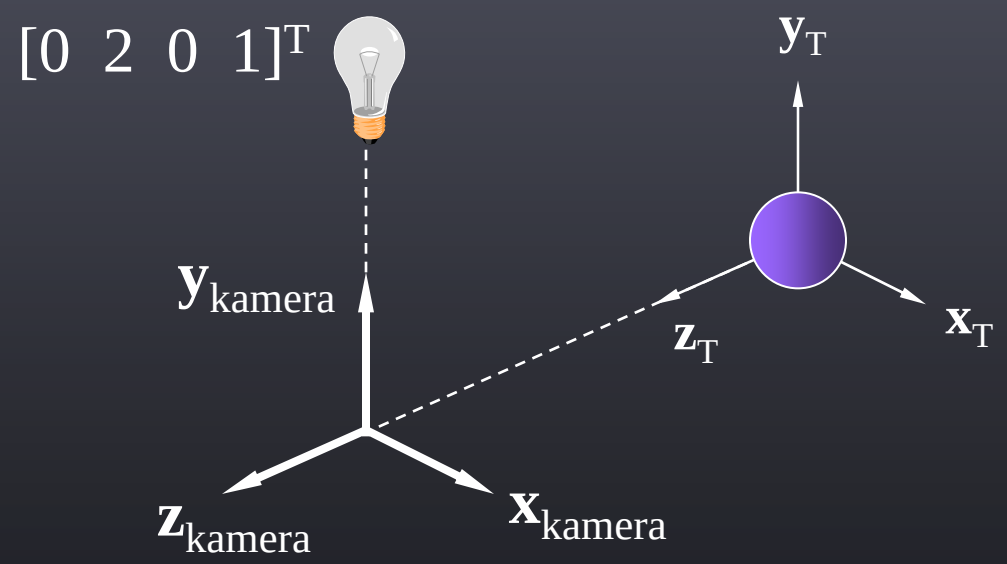


Färg beräknas i varje hörn och
interpoleras sedan linjärt över polygonen.
Normaler anges före varje hörn med
`glNormalf(nx, ny, nz)`.

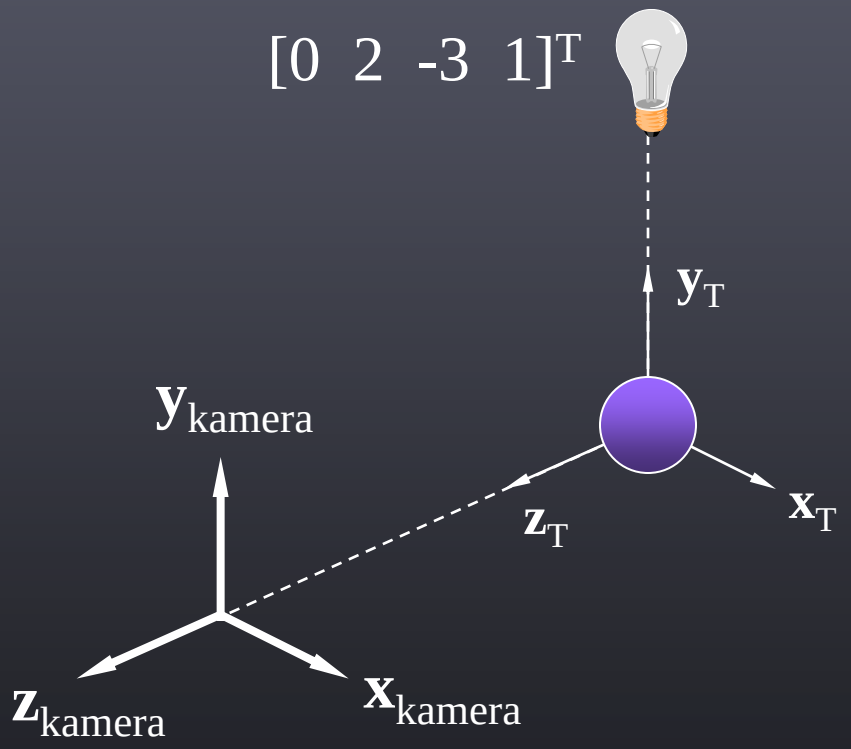
Bra att komma ihåg om ljussättning

- **Då ljussättning är aktivt har aktuell färg (`glColor`) ingen effekt.**
- Då ljuskällans position anges multipliceras den med aktuell modelviewmatris.

```
GLfloat lpos[4] = { 0, 2, 0, 1 };
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, lpos);
glTranslatef(0, 0, -3);
glutDrawSolidSphere(...);
```



```
GLfloat lpos[4] = { 0, 2, 0, 1 };
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -3);
glLightfv(GL_LIGHT0, GL_POSITION, lpos);
glutDrawSolidSphere(...);
```

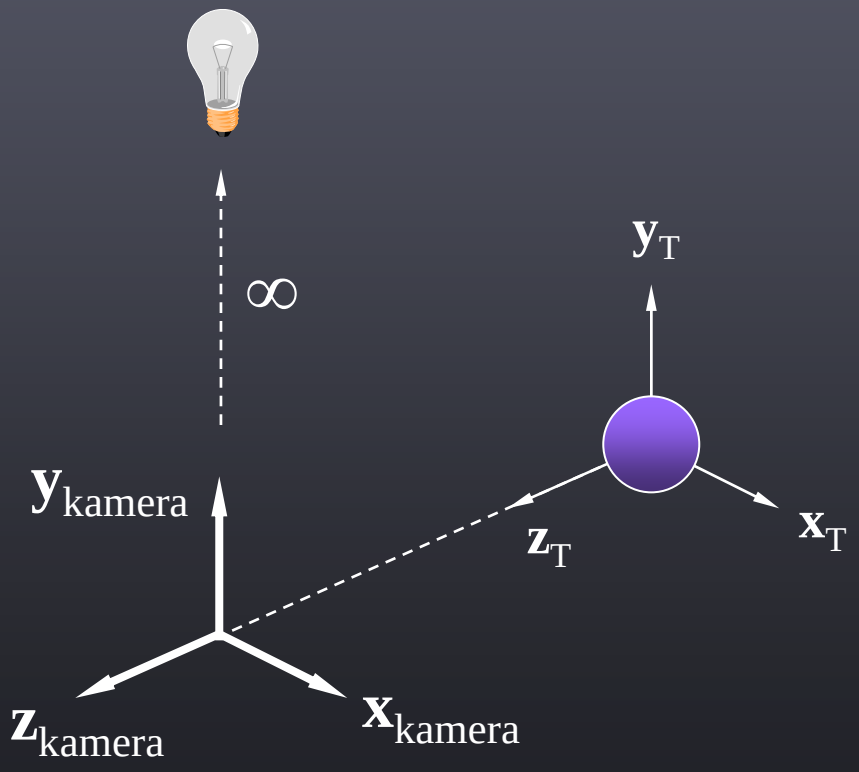


Ljuskällans position är nu relativt sfären ist.f. världsorigo!

```
GLfloat lpos[4] = { 0, 2, 0, 0 };
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, lpos);
glTranslatef(0, 0, -3);
glutDrawSolidSphere(...);
```

Om w-koordinaten är 0 betyder det att ljuskällan är oändligt långt bort i den angedda riktningen.

Men se till att modelviewmatrisen är identitetsmatrisen, annars blir det ofta oväntade resultat!



Normaler

Normalen i en punkt på en yta definieras av det **plan** som är parallelt med ytan i punkten:

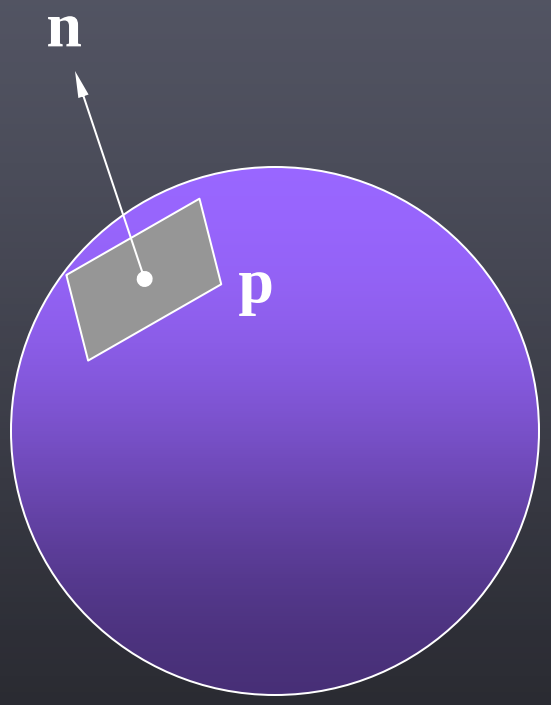
$$\mathbf{p} = [\underbrace{a \quad b \quad c}_{\mathbf{n}} \quad d]$$

Om punkten P med koordinaterna

$$\mathbf{a} = [x \;\; y \;\; z \;\; 1]^{\mathrm{T}}$$

ligger på detta plan gäller att

$$\mathbf{pa} = 0$$



Normaler

Antag nu att vi har en transformationsmatris $\mathbf{C}$.
Punktens koordinater transformeras enligt

$$\mathbf{a}' = \mathbf{Ca}$$

Vi har att

$$\mathbf{pa} = 0$$
$$\Leftrightarrow$$
$$\mathbf{pC}^{-1}\mathbf{Ca} = 0$$

Så $\mathbf{Ca}$ ligger på planet $\mathbf{pC}^{-1}$, dvs.
$\mathbf{pC}^{-1}$ är det transformerade planet.

Normaler

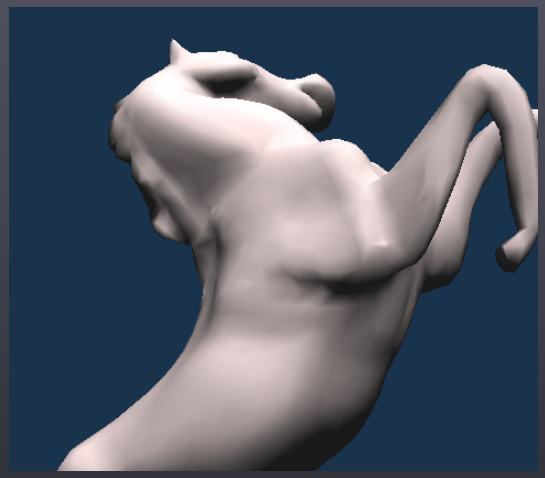
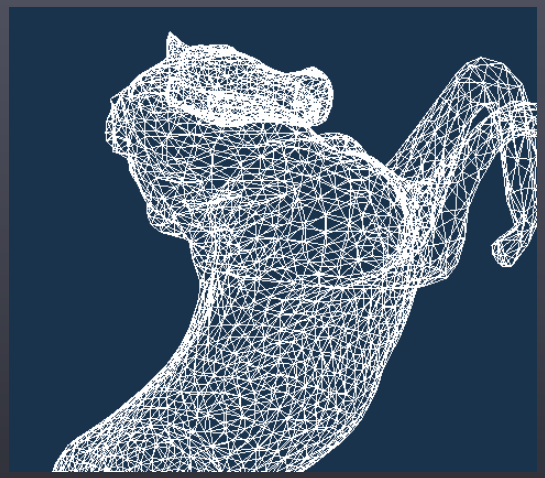
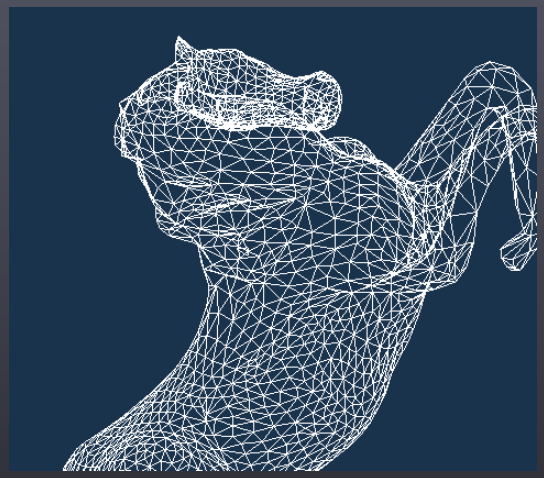
Eftersom vi vill postmultiplicera alla vektorer
vänder vi på uttrycket:

$$\mathbf{pC}^{-1} = (\mathbf{C}^{-1})^{T}\mathbf{p}^{T}$$

Så normaler multipliceras med
den matris man får
om man först inverterar modelview-matrisen och sedan
transponerar den!

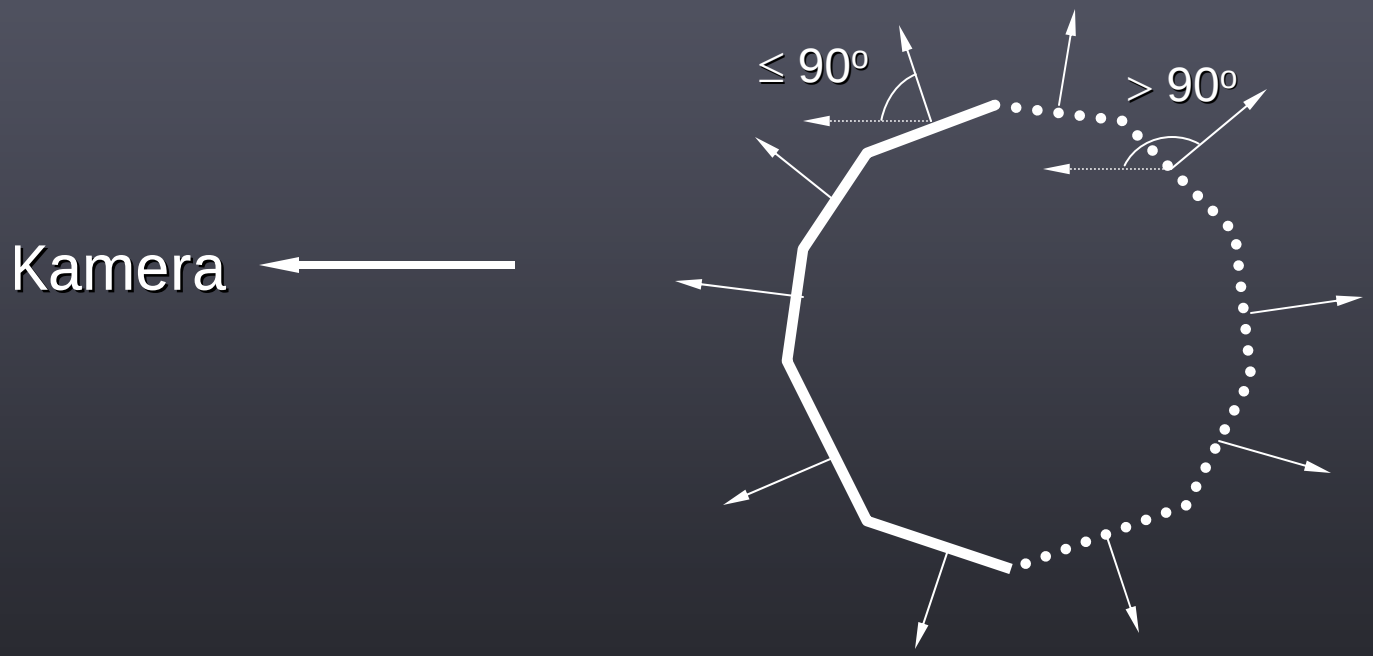
I OpenGL "vet" man att normalen alltid har $w = 0$,
så av effektivitetsskäl specificerar man alltid
normaler med tre komponenter.

Back face culling

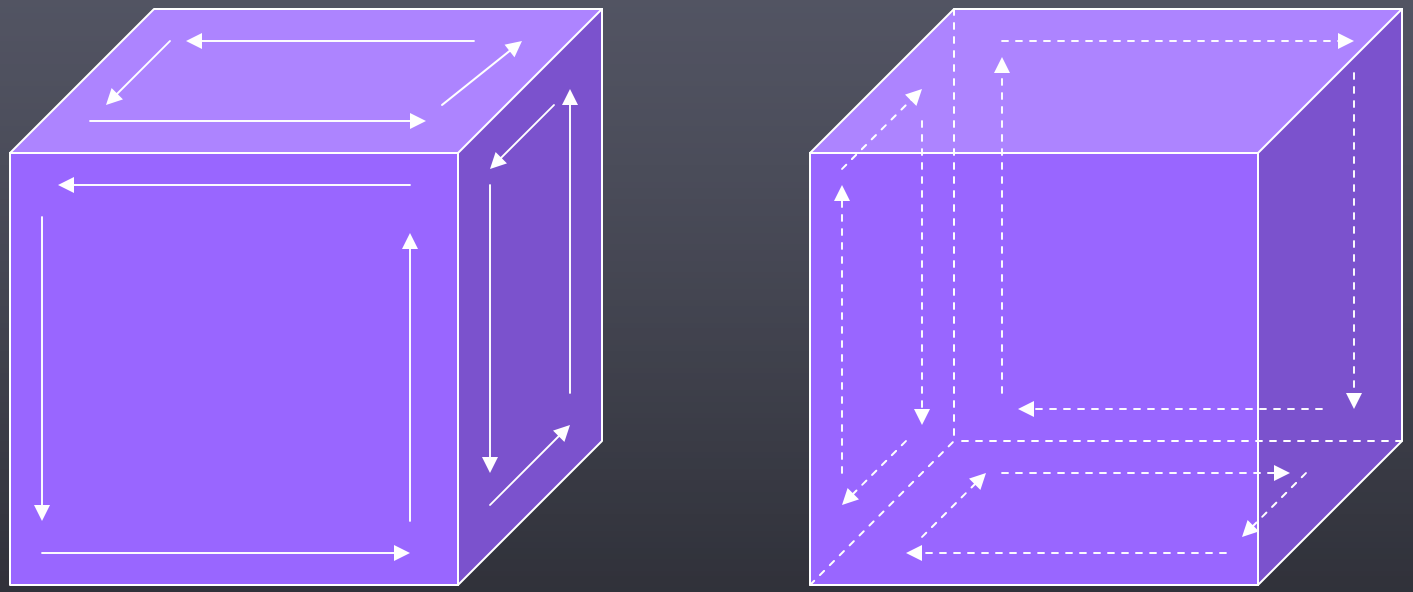




```
glEnable(GL_CULL_FACE);
```

Back face culling



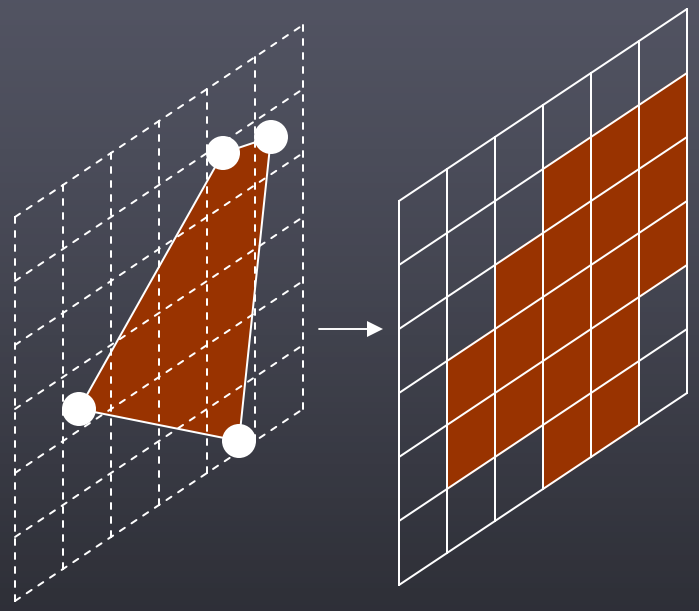
Fram- eller baksida?



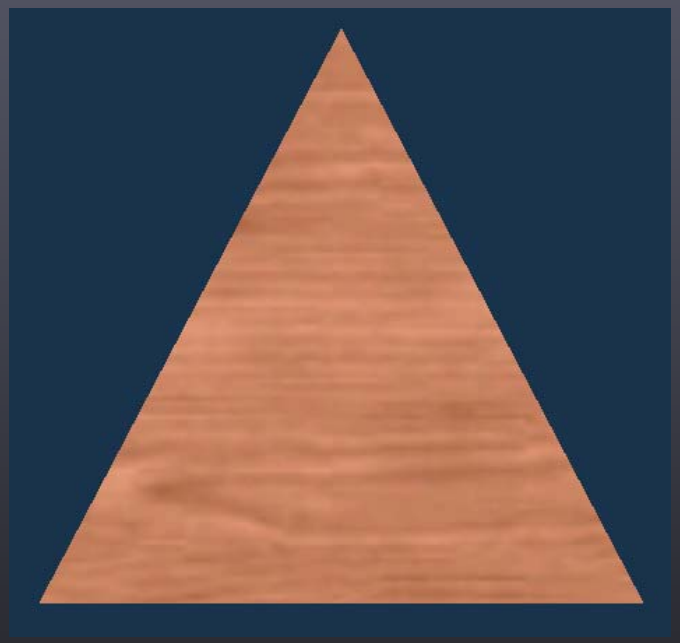
Definieras av hörnens ”ordning” sett från kameran.
Konventionen är att **motsols är riktat mot kameran**.

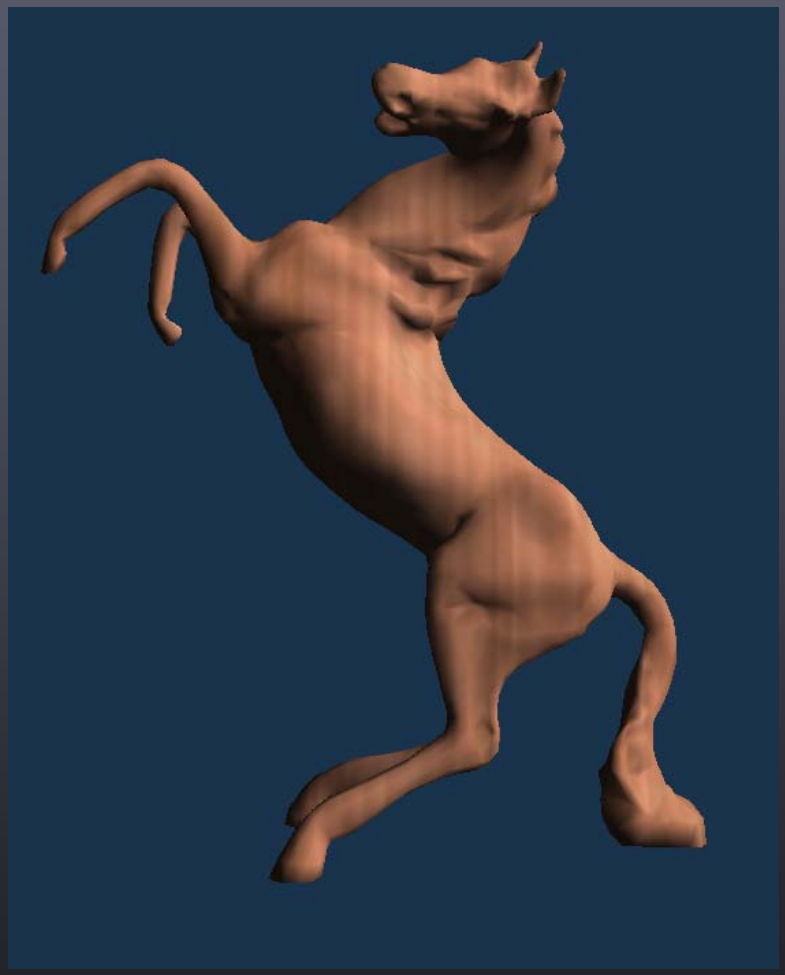
Rastrering

- "Översätter" från hörn till pixelpunkter.
- Ett fragment består av
  - **Position**:
    (x, y, z) i normerade enhetskoordinater
  - **Färg**:
    (R, G, B, A)
  - **Texturkoordinater**:
    (s, t, q, r)



Texturer

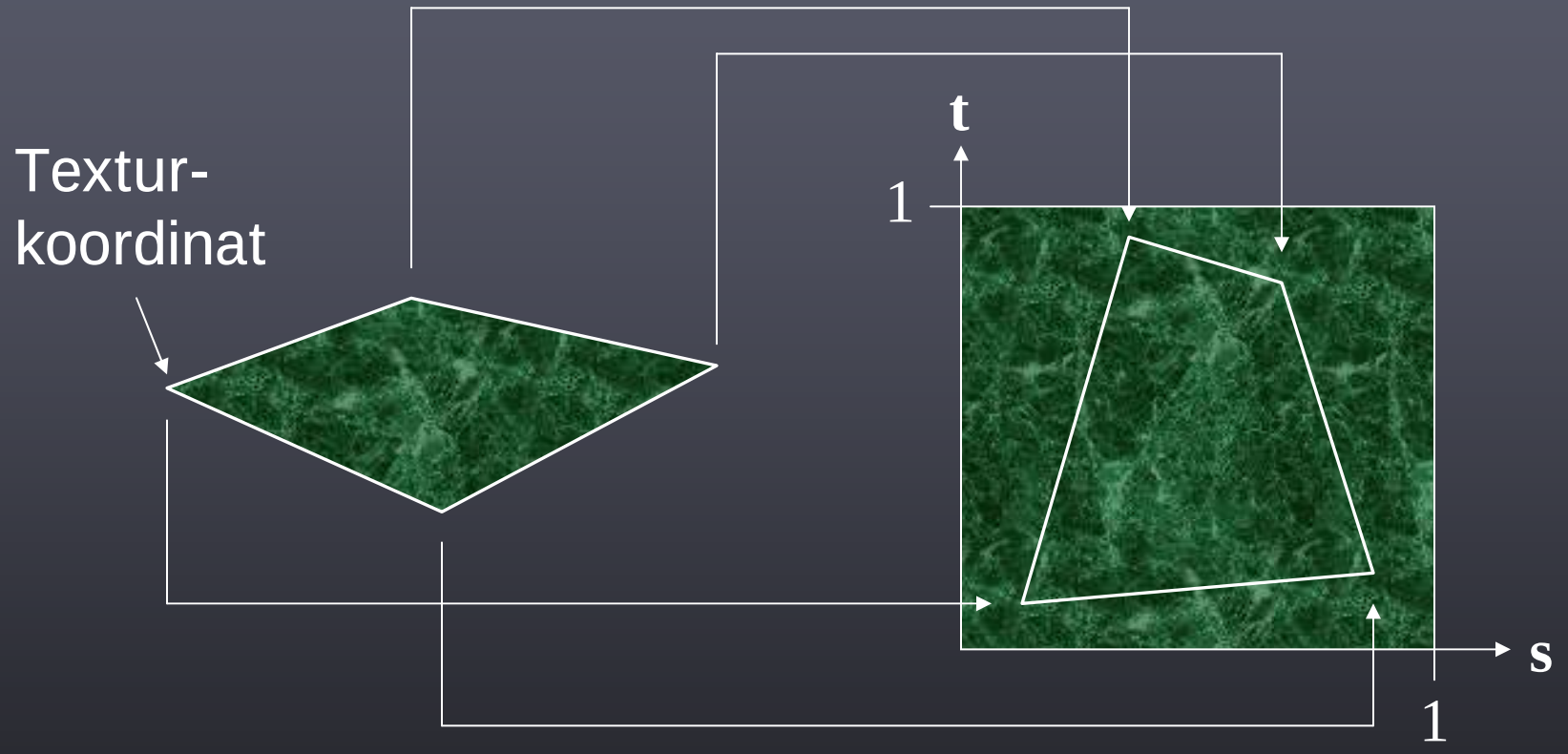




Texturer

- OpenGL har ingen filformatstolk för bilder.
- För labben finns **libjpeg** förberett.
- Eller hämta ett gratisbibliotek från webben.
- Ett bra alternativ är **Corona** (http://corona.sourceforge.net/).
- (Eller skriv en egen tolk...)

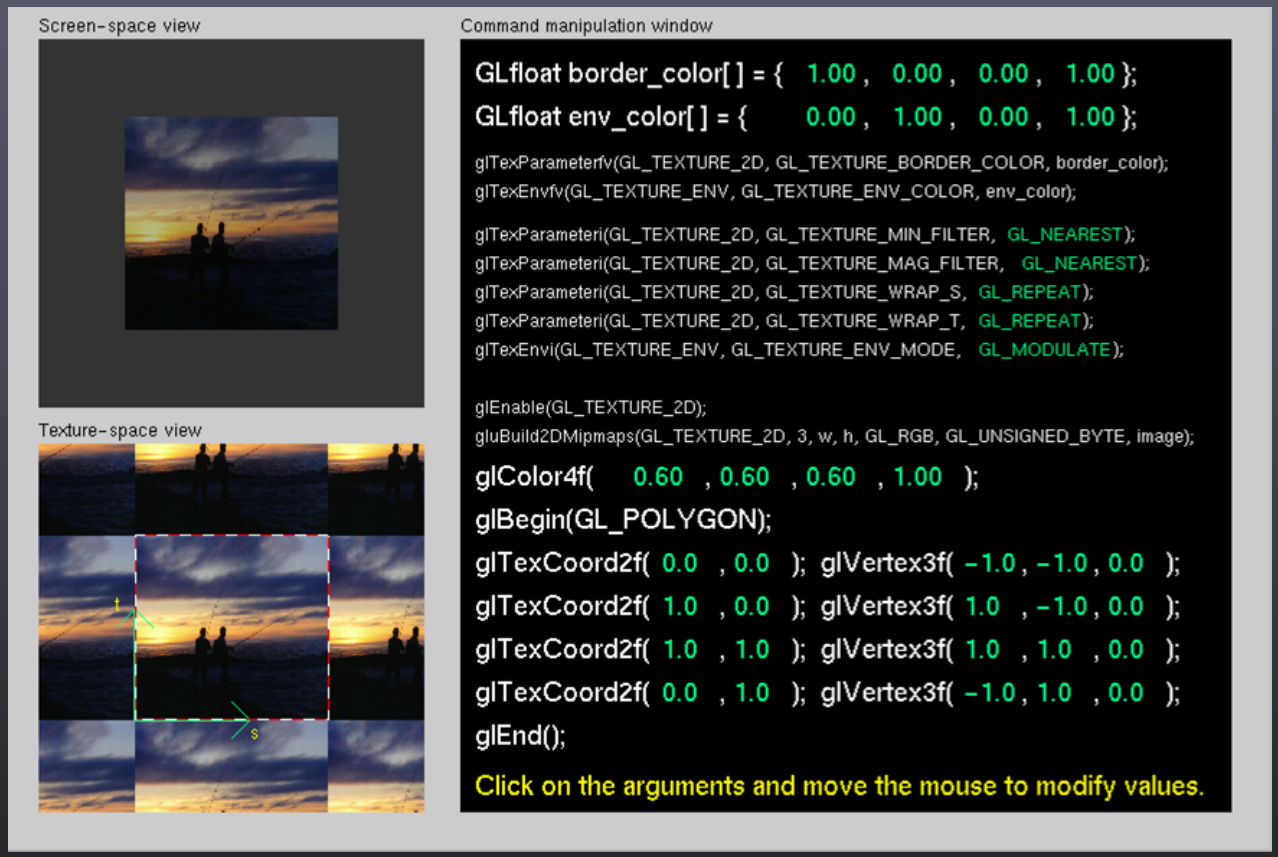
Texturkoordinater



`glTexCoord2f(s, t);`

Anges före **`glVertex`** (precis som för normaler).

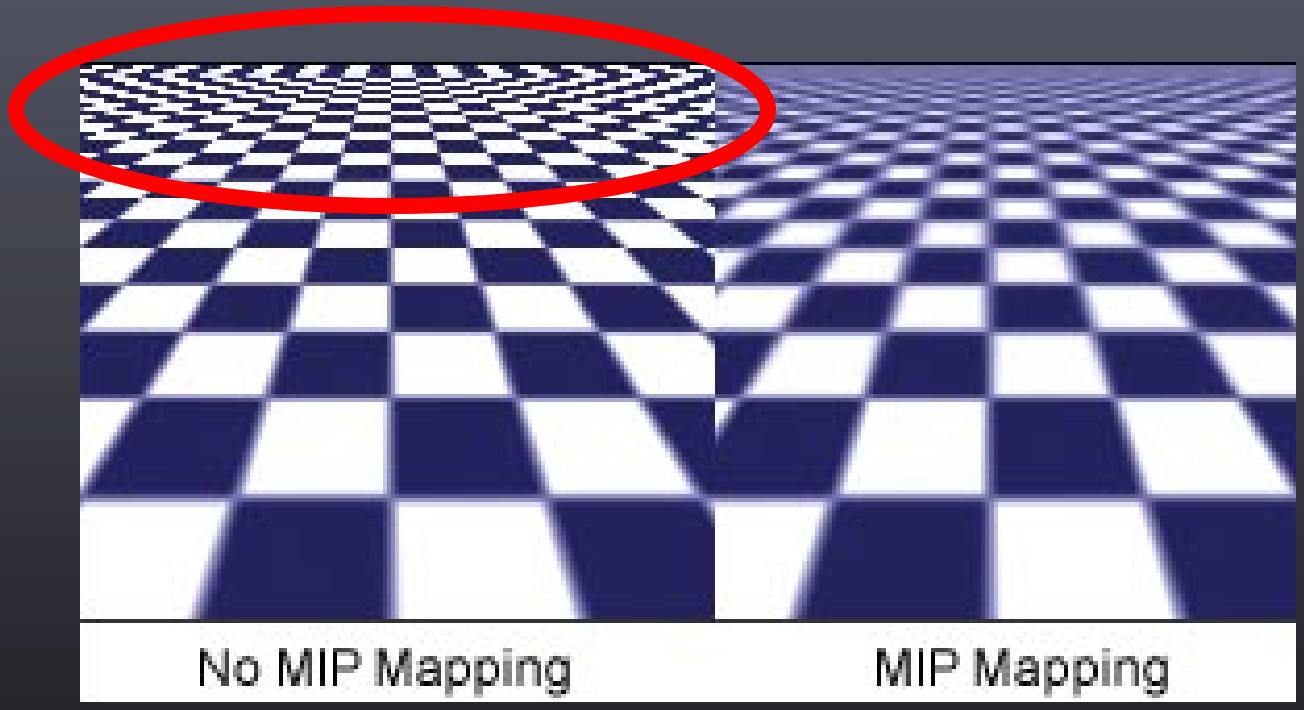
Demo



http://www.xmission.com/~nate/opengl.html

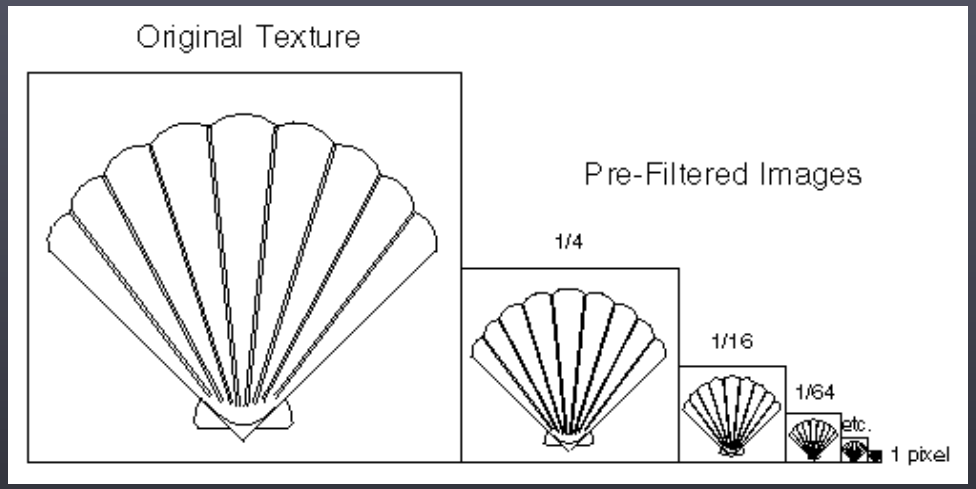
MIP-mapping

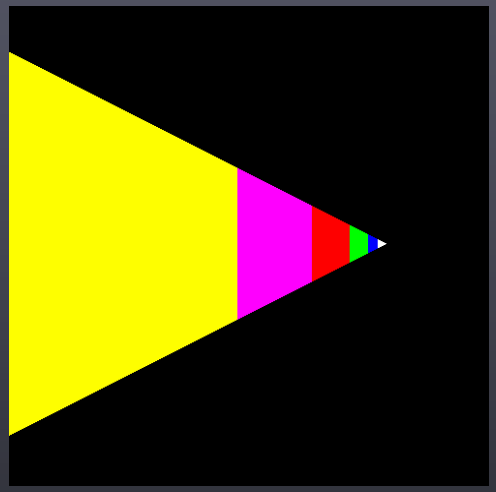
Teknik för att undvika aliasing-problem.



MIP-mapping

Filtrera ett antal texturer i förväg och lagra dem effektivt.

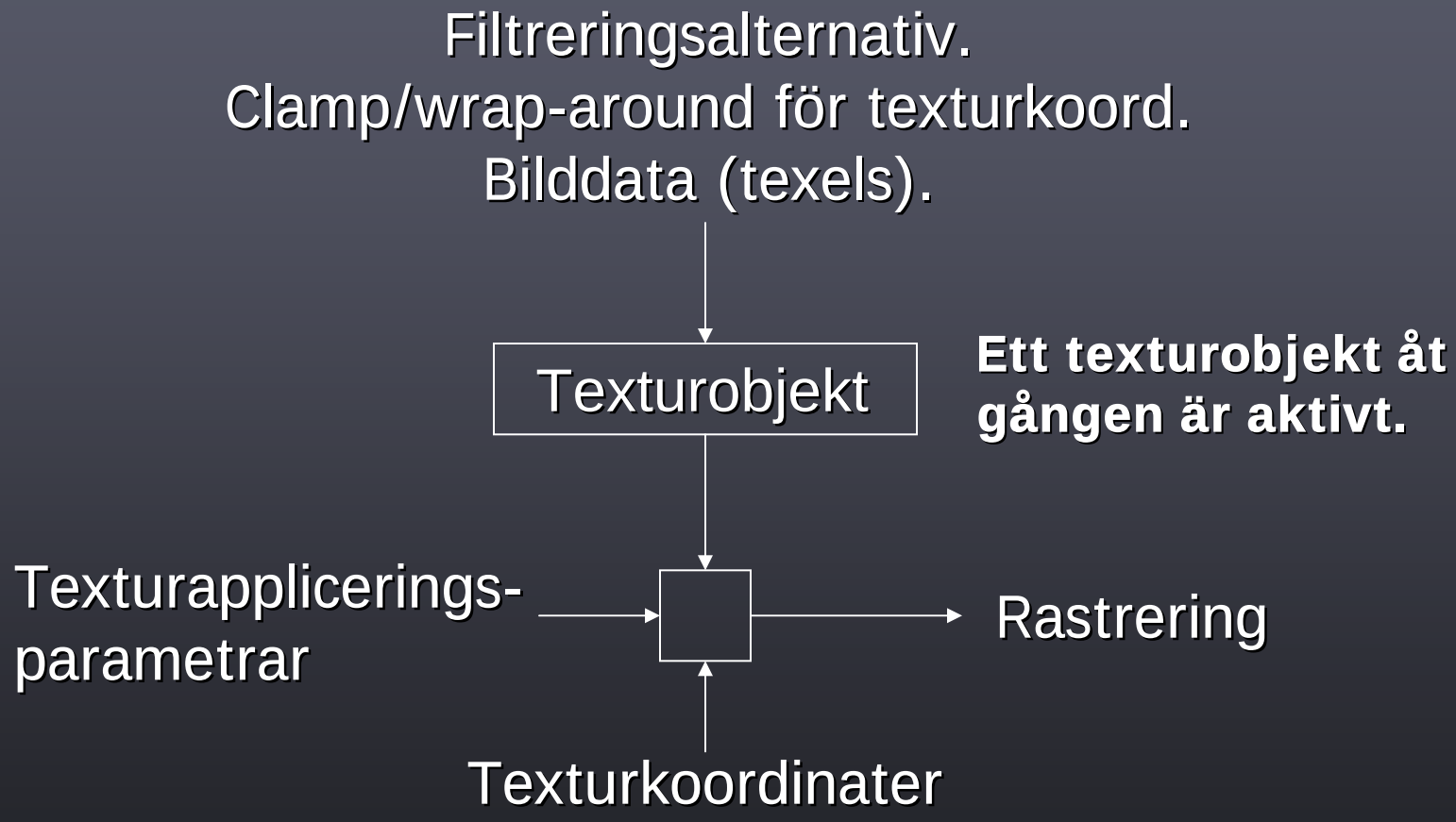




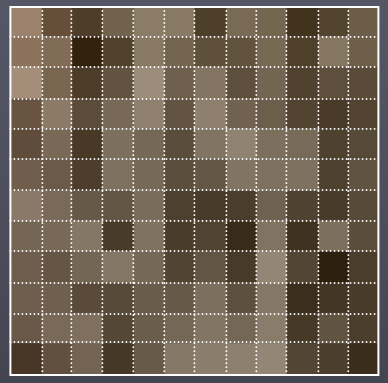
Vid texturering väljs automatiskt den textur som passar bäst m.a.p. avstånd från kameran (om MIP-mapping är aktiverat).

Många grafikkort har andra former av antialiasing också.

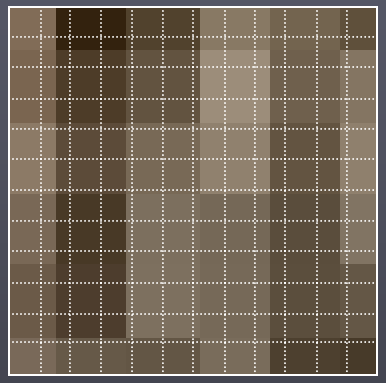
Texturering i OpenGL



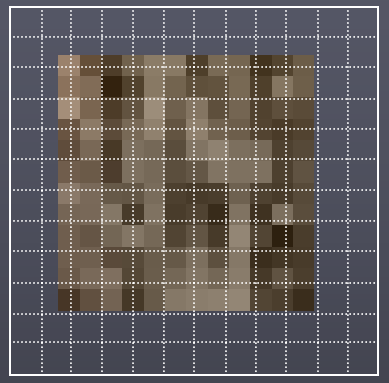
Förstoring och förminskning



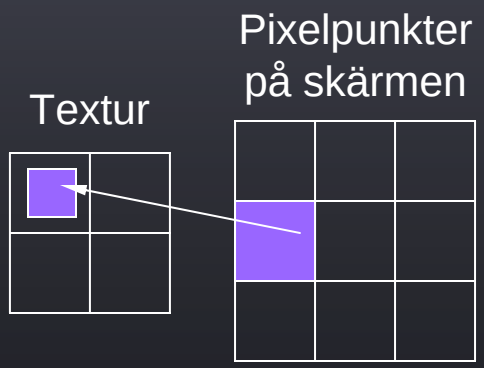
Texelstorlek = pixelstorlek
(inträffar i princip aldrig)

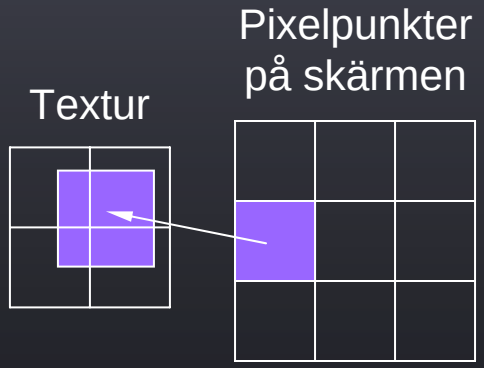


Förstoring

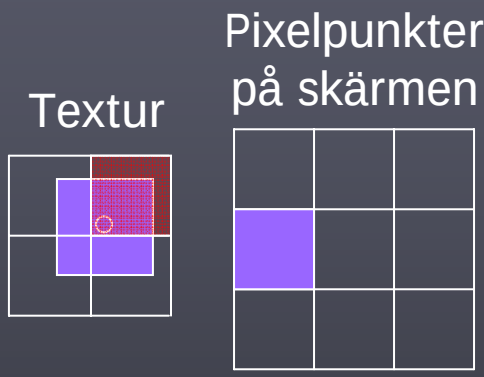


Förminskning

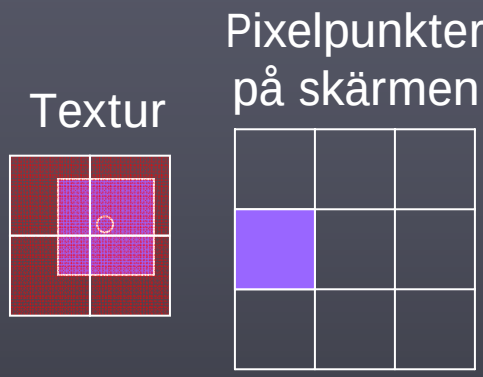




Filtreringsalternativ




```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
```

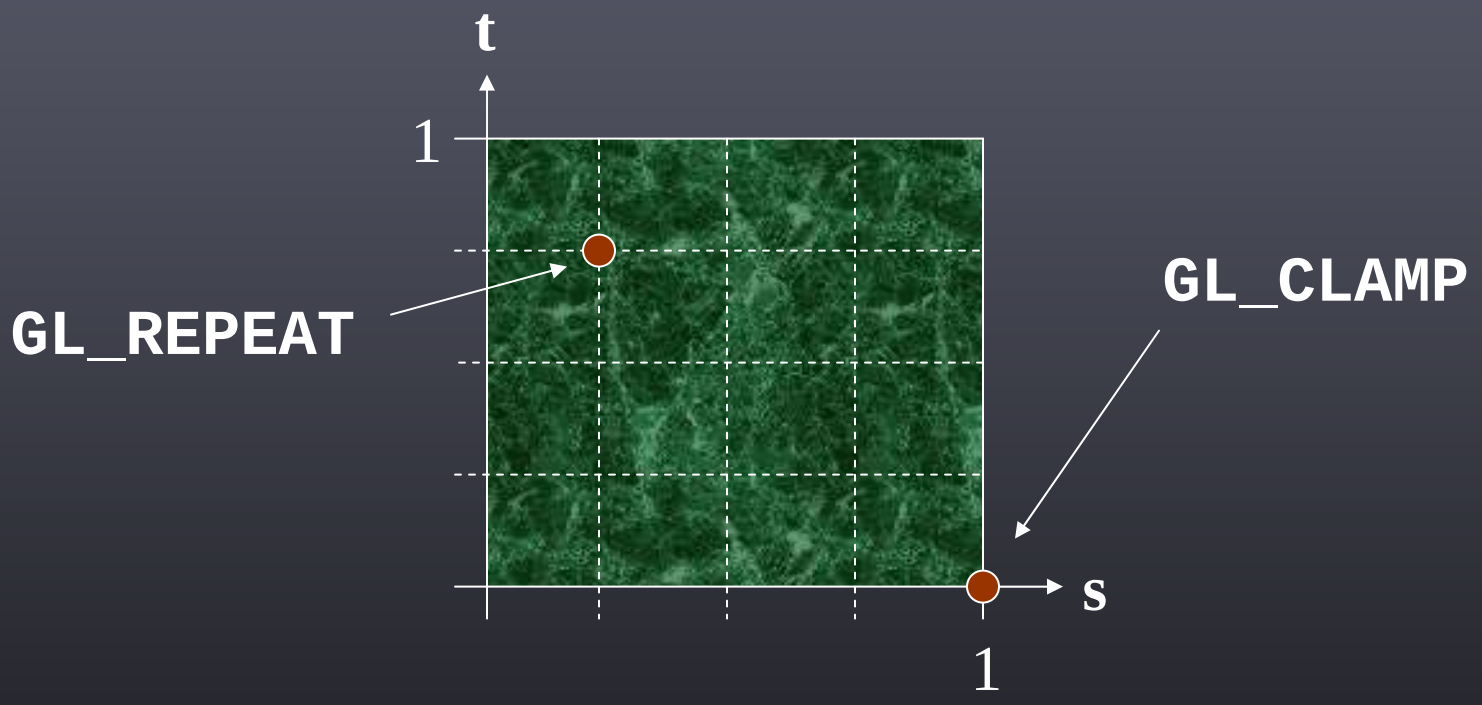



```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

Labbtips:
GL_NEAREST_MIPMAP_LINEAR är default,
men vi använder inte MIP-maps...
Glöm heller inte att sätta parametern för förstoring!

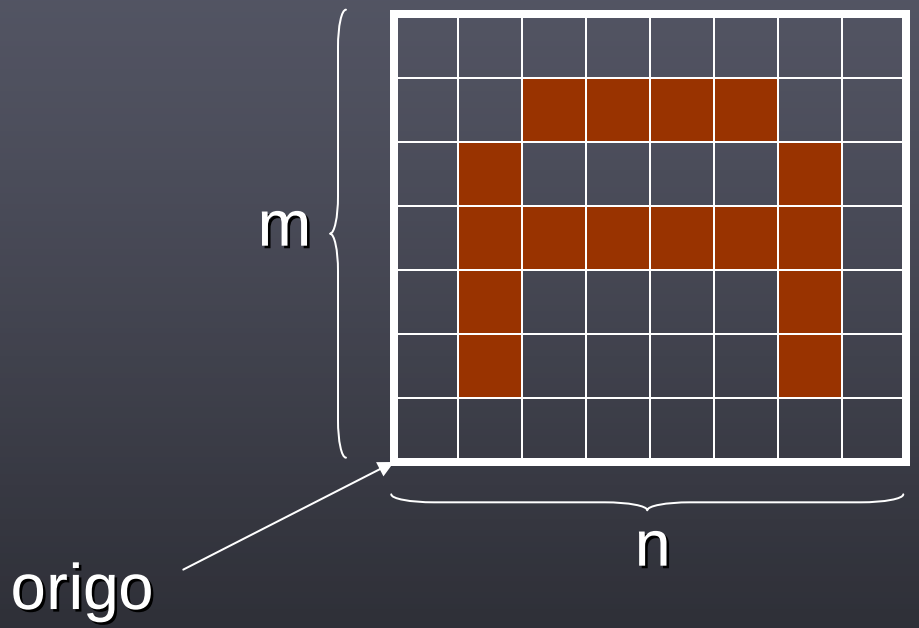
Clamp / wrap around

(s, t) = (1.25, -2.75)




```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
```

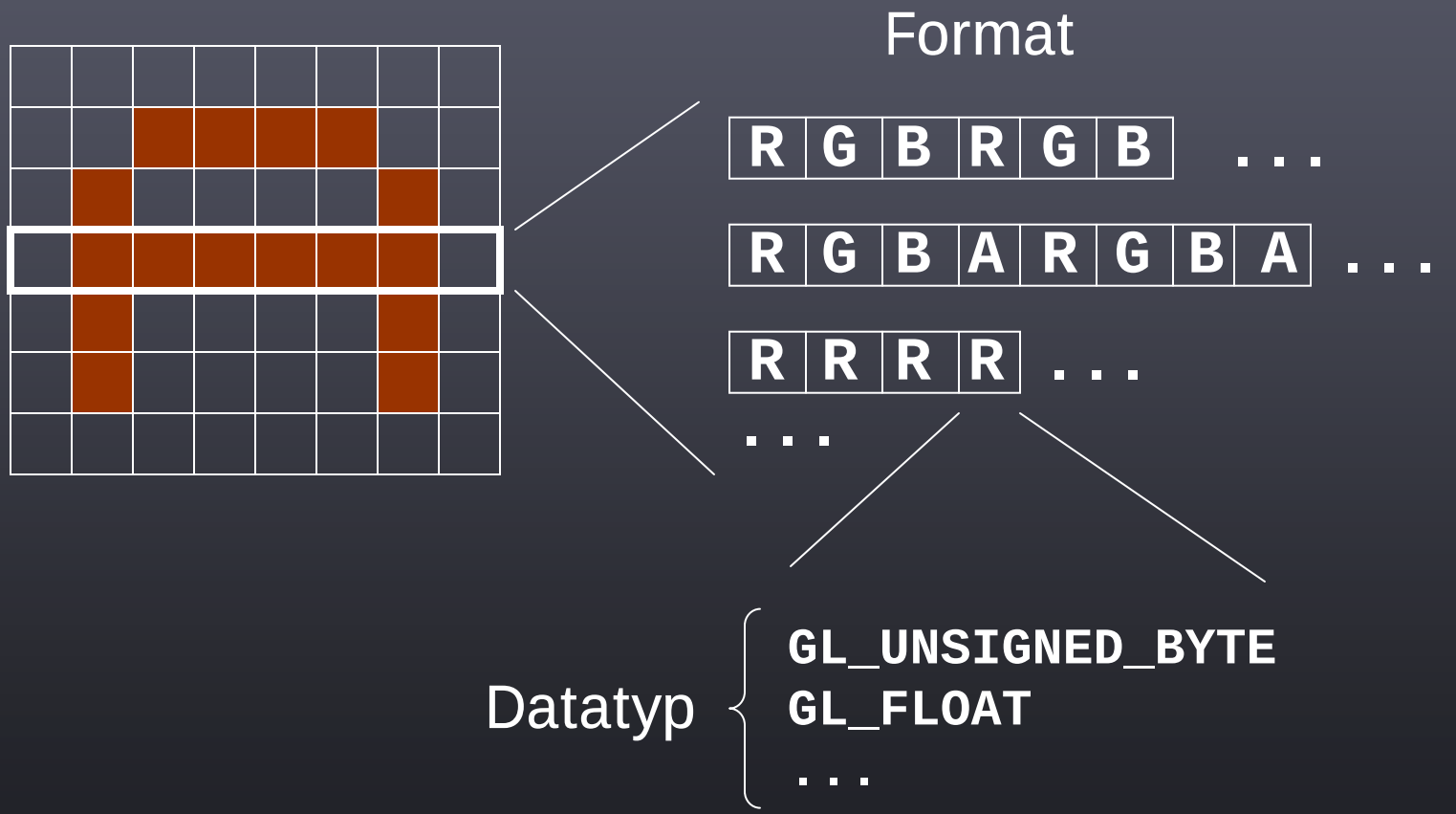
Bilddata: Texels



(**`width`**, **`height`**) = (2^n, 2^m)

där m, n är heltal ≥ 0

Bilddata: Format och datatyper



Specifikation av bilddata

```
glTexImage2D(GL_TEXTURE_2D,
             level,
             internalformat,
             width, height, 0,
             format,
             type,
             texels);
```

MIP-map-nivå (sätt till 0)

Sätt till samma som **type**

GL_RGB eller **GL_RGBA**

Datatyp

Namn på array med texels

Använder du **loadJPEG()**-funktionen
i skalprogrammet för labben,
sätt **format** till **GL_RGB** och **type** till **GL_UNSIGNED_CHAR**.

Texturappliceringsparametrar

```
glTexEnvi(GL_TEXTURE_ENV,
          GL_TEXTURE_ENV_MODE,
          mode);
```

där **mode** är

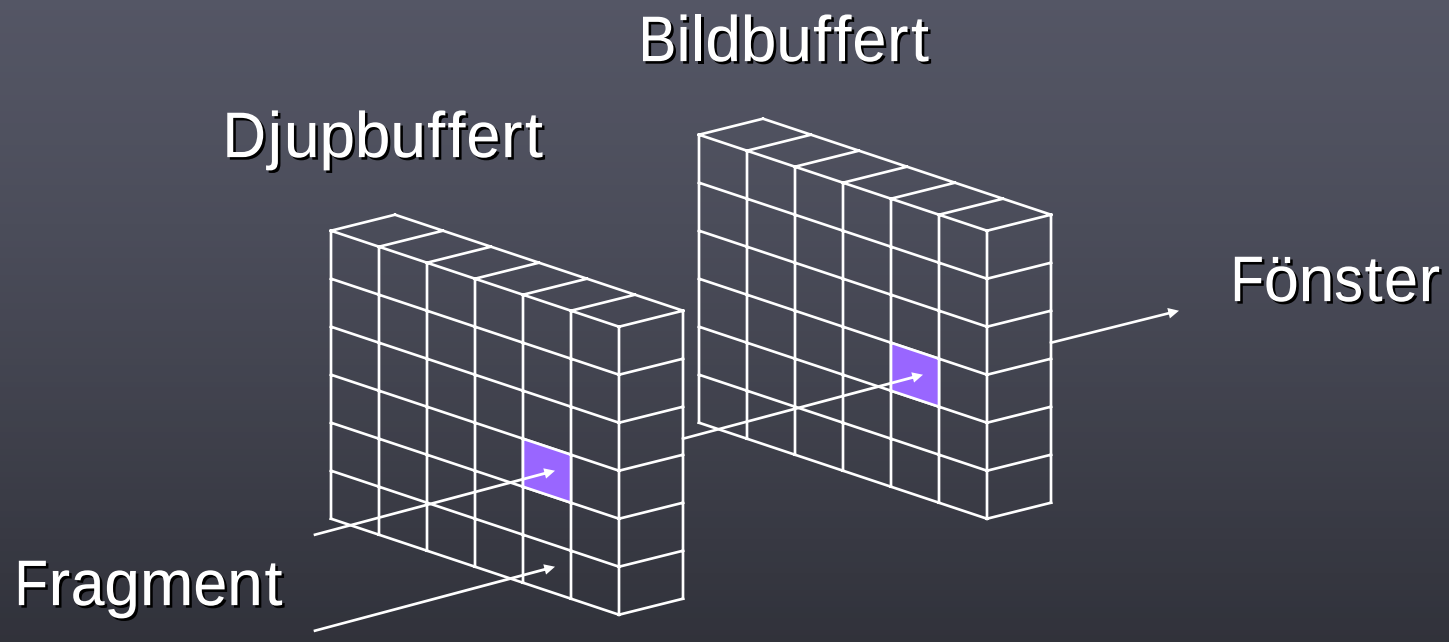
GL_DECAL = Ersätt RGB med texturdata, spara A.
GL_REPLACE = Ersätt RGBA med texturdata.
GL_MODULATE = Kombinera texturdata med ljussättning/färg.

Lagras inte i texturobjekt!

Operationer på fragment

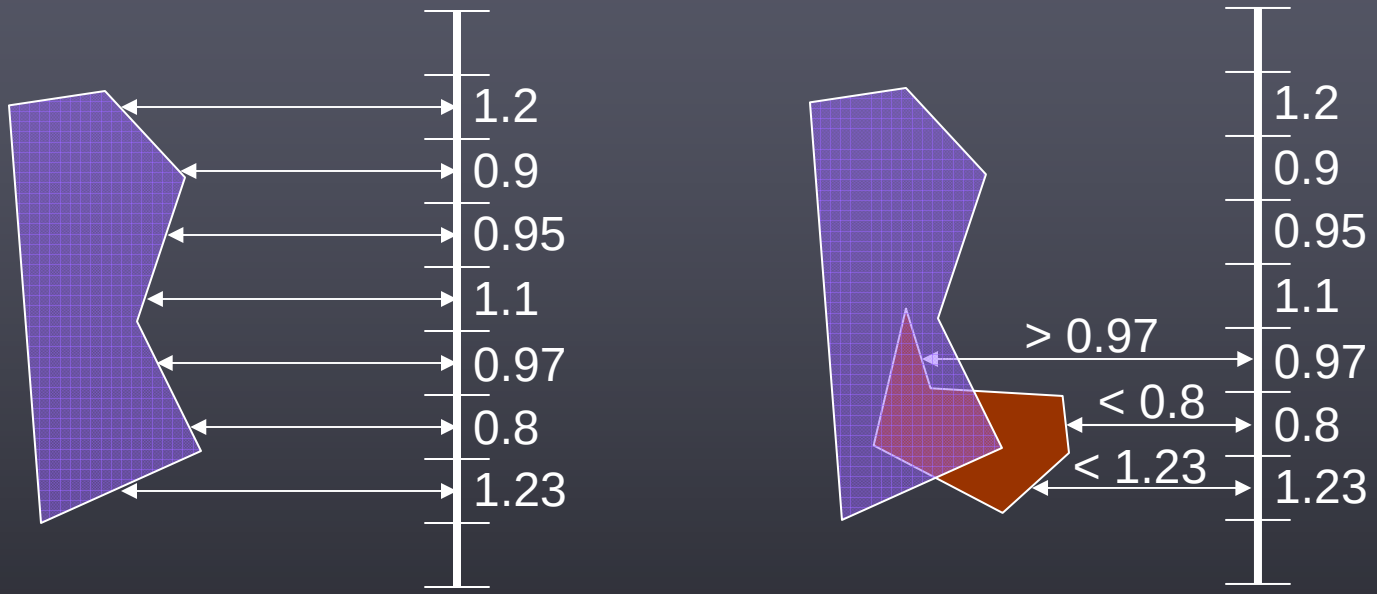
- Huvudsakligen ett antal **tester** som avgör om fragmentet ska skickas till bildbufferten eller inte.
- Testerna genomförs alltid i samma ordning.
- Ordningen är definierad i OpenGL-specifikationen.
- Används för borttagning av skymda ytor, dynamiska skuggor, specialeffekter, m.m.

Djuptest



Vanlig bugg på labben: du glömde tala om för GLUT att du ville ha en djupbuffert!

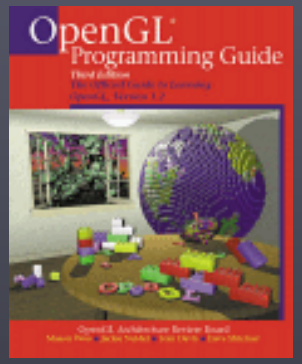
Djuptest (Z-buffring)



Testa om anländande fragment har större eller mindre djupvärde än det som redan finns i bufferten. Om värdet är mindre, ersätt med det nya fragmentet.

Mer info

- GLUT/OpenGL-kompendiet i kursbunten.
- Woo, Neider, Davis: **The OpenGL Programming Guide**, Addison Wesley Developer Press. Upplagan för OpenGL 1.1 finns som PDF.
- Segal, Akeley: **The OpenGL Graphics System: A Specification**. Finns på http://www.opengl.org
- Plus en mängd andra böcker...



- Se länkar på kurshemsidan!

Mer info

Gå till

http://www.opengl.org/

Nästa gång...

...skriver vi ett komplett OpenGL-program från scratch!