LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Dr. Andreas Butz | Prof. Dr. Ing. Axel Hoppe

Dipl.-Medieninf. Dominikus Baur
Dipl.-Medieninf. Sebastian Boring

Übung: Computergrafik 1

Projektionen und Transformationen

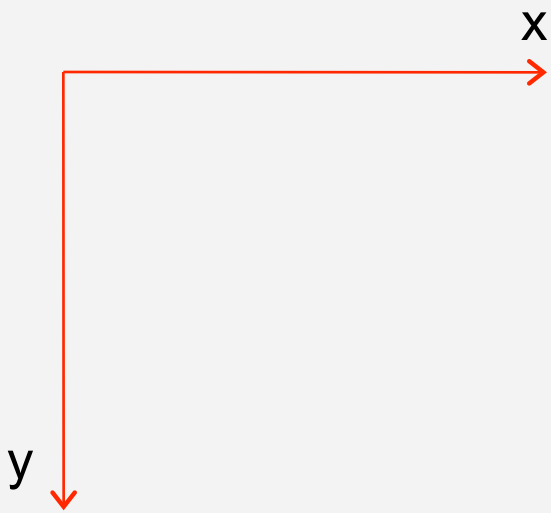
Qt Kontextmenüs

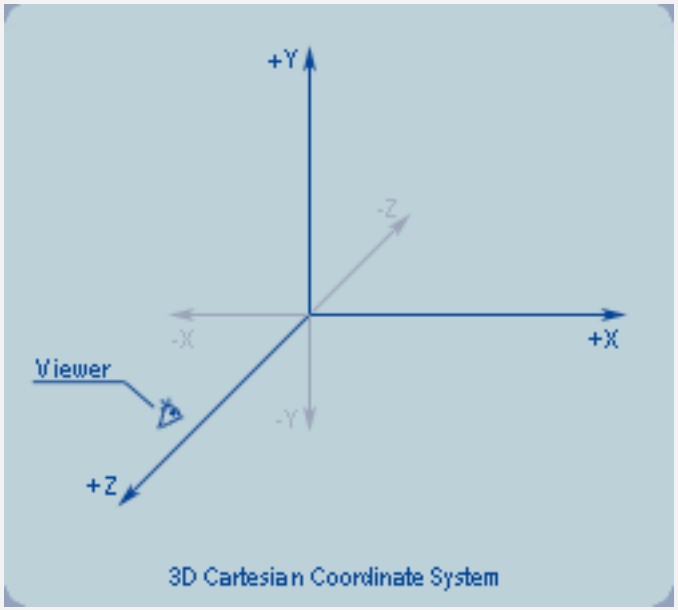
Koordinatensysteme

Koordinatensysteme

- Im allgemeinen frei wählbar
- Meistens rechtsdrehend und rechtwinklig

Beispiele aus der Computergraphik:





(Quelle: http://www.falloutsoftware.com/tutorials/gl/gl0.htm)

Koordinatensysteme II

- Koordinatensysteme werden aufgespannt durch Ursprung und Basis

$$K = B,(b_1,b_2)$$

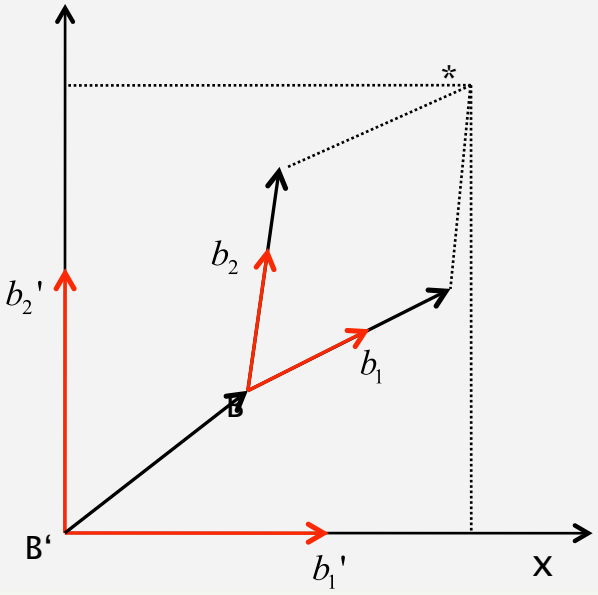
$$K' = B',(b_1',b_2')$$

Beispiel:

$$B = (0,0)^{T,K} = \left(\tfrac{3}{2},\tfrac{3}{4}\right)^{T,K'} \ldots B' = (0,0)^{T,K'},$$

$$b_1 = (1,0)^{T,K} = \left(\tfrac{1}{2},\tfrac{1}{8}\right)^{T,K'} \ldots b_1' = (1,0)^{T,K'},$$

$$b_2 = (0,1)^{T,K'} = \left(\tfrac{1}{4},1\right)^{T,K'} \ldots b_2' = (0,1)^{T,K'}.$$



Gegeben:

$$P = (2,1)^{T,K}$$

Gesucht: Darstellung von P in K‘ Koordinaten?

Wechsel des Koordinatensystems durch affine Transformation:

$$v' = A * v + d,$$

$$A = \begin{pmatrix} \frac{1}{2} & \frac{1}{4} \\ \frac{1}{8} & 1 \end{pmatrix}, d = \left(\tfrac{3}{2},\tfrac{3}{4}\right)$$

Bisher:

- Translation: Addition eines Vektors
- Skalierung, Rotation : Multiplikation des Faktors

Problem:

- Keine einheitliche Behandlung
- Zusammengesetzte Transformationen nur schwer zu realisieren
- Umkehrung von verketteten Transformationen nur schwer möglich.

Homogene Koordinatensysteme

- Jeder Punkt P ist eindeutig darstellbar durch:

$$P = (\beta_1 * b_1 + \beta_2 * b_2 + 1 * B),$$

bzw.

$$P' = (\beta_1' * b_1' + \beta_2' * b_2' + 1 * B')$$

- Das Tripel ($ß_1$,$ß_2$,1) ist eine Darstellung von P in homogenen Koordinaten.
- Die Affine Transformation lässt sich dann so darstellen:

$$\begin{pmatrix} v' \\ 1 \end{pmatrix} = \begin{pmatrix} A & d \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} v \\ 1 \end{pmatrix}$$

Zusammengesetzte Transformationen

- Hintereinanderausführen von affinen Transformationen ergibt wieder affine Transformation:

$$
\begin{aligned}
v' &= A_1 * v + d_1, \\
v'' &= A_{2*} v' + d_2 \\
&= A_2 * A_1 * v + (A_2 * d_1 + d_2)
\end{aligned}
$$

- In homogenen Koordinaten:

$$
\begin{pmatrix} v' \\ 1 \end{pmatrix} = \begin{pmatrix} A_1 & d_1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} v \\ 1 \end{pmatrix},
$$

$$
\begin{pmatrix} v'' \\ 1 \end{pmatrix} = \begin{pmatrix} A_2 & d_2 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} v' \\ 1 \end{pmatrix}
$$

$$
= \begin{pmatrix} A_2 & d_2 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} A_1 & d_1 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} v \\ 1 \end{pmatrix}
$$

- Achtung: Reihenfolge ist wichtig!

- Komplizierte Transformationen lassen sich zusammensetzen
  - Drehung um beliebigen Punkt P
  - Skalierung mit Bezug auf beliebigen Punkt P
- Transformation wird akkumuliert
  - Transformationsmatrix wird im vornherein berechnet
  - Anwendung auf alle Punkte auf einmal
- Affinität => nur Punkte müssen transformiert werden!
  - Geraden bleiben gerade
  - Parallele Kanten bleiben parallel
  - Linien und Flächen sind durch Punkte eindeutig bestimmt
  - Transformierte Objekte sind isometrisch (ähnlich) zum Original

Die allgemeine Transformationsmatrix:

$$T = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{pmatrix}$$

- Skalierung
- Rotation
- Translation

Im Speicher im “column major” Format (d.h. spaltenweise):

$a_1\ a_2\ a_3\ a_4\quad b_1\ b_2\ b_3\ b_4\quad c_1\ c_2\ c_3\ c_4\quad d_1\ d_2\ d_3\ d_4$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Wichtigste Transformationen

$$T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

$$S = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Skalierung

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um x-, y-, z-Achse

Umkehrung von Transformationen

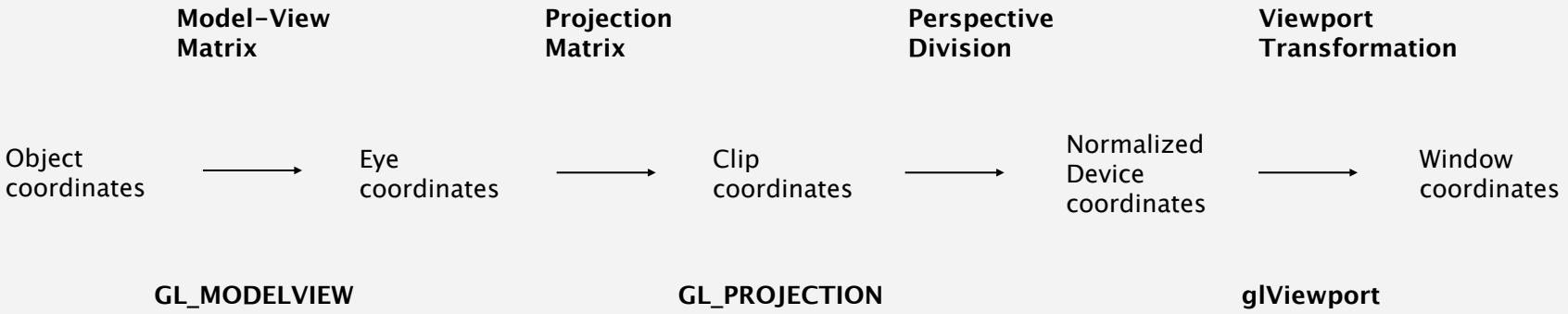
- Transformationen lassen sich rückgängig machen indem man mit der Inversen der jeweiligen Transformationsmatrix multipliziert (wegen $A * A^{-1} = 1$)
- Deutlich einfacher: Die Transformation selbst umkehren, also zurückverschieben, -drehen, -skalieren (s.Vorlesung)
- Dabei zählt wiederum die Reihenfolge:

  Eine Reihe von Transformationen T1, T2, ..., Tn muss in der Reihenfolge Tn, ..., T2, T1 rückgängig gemacht werden um das ursprüngliche Ergebnis zu erhalten

Projektionen in OpenGL

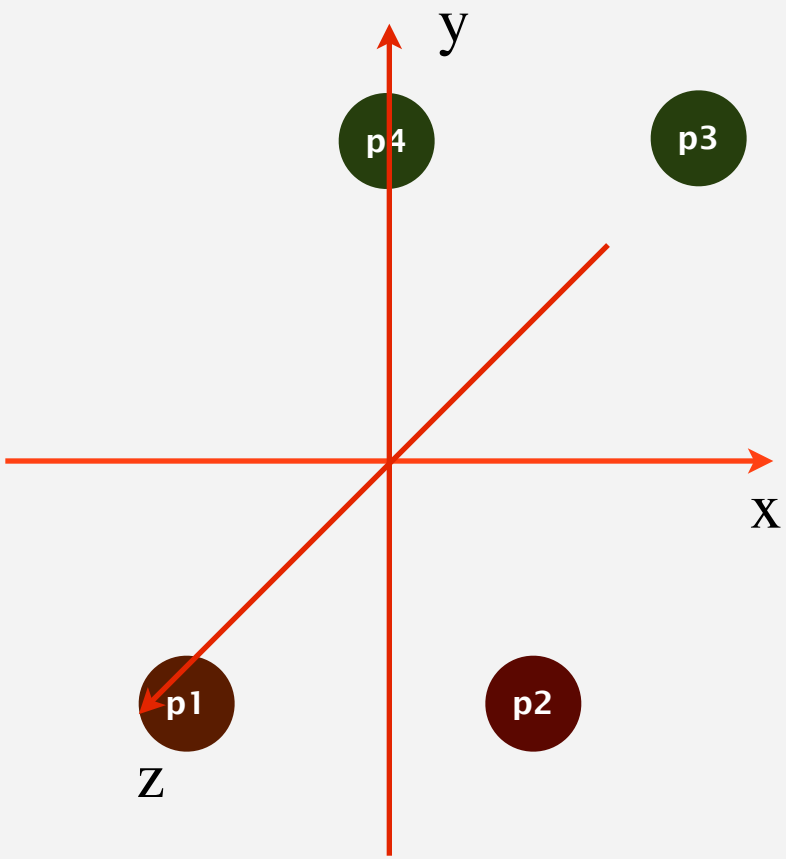
OpenGL Transformationen

- Feste Kette von Matrixmultiplikationen zur Transformation/Projektion
- Objekte in Object/World coordinates ->
  Farbige Pixel in Window coordinates



(Quelle: http://www.opengl.org/resources/faq/technical/transformations.htm)

OpenGL Object coordinates




```
glBegin(GL_POLYGON);
    glColor3f(1.0f, 0.0f, 0.0f); // red
    glVertex3f(-1.0f, -1.0f, 1.0f); // p1
    glVertex3f(1.0f, -1.0f, 1.0f);  // p2
    glColor3f(0.0f, 1.0f, 0.0f); // green
    glVertex3f(1.0f, 1.0f, -1.0f);  // p3
    glVertex3f(-1.0f, 1.0f, -1.0f); // p4
glEnd();
```

- `glVertex` erzeugt Punkte in Object coordinates
- OpenGL gibt keine Einheiten für Object coordinates vor

(Quelle i.F.:http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf)

OpenGL Eye coordinates

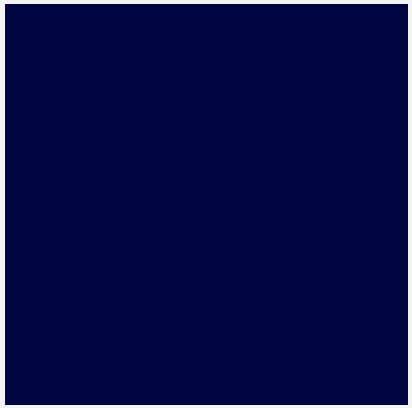
$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$$

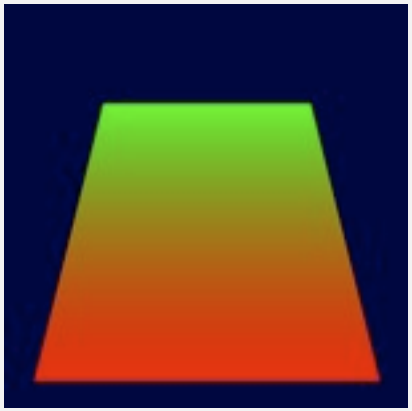
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0, 0, -4);
glRotatef(45, 0, 1, 0);
```

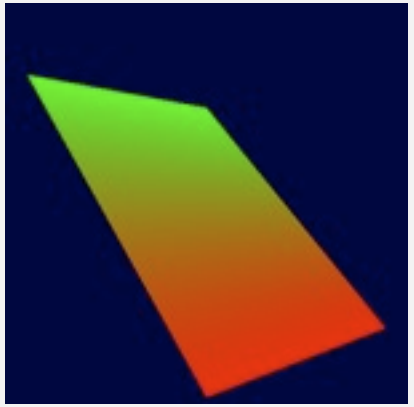
```
1.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0
0.0  0.0  1.0  0.0
0.0  0.0  0.0  1.0
```

```
1.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0
0.0  0.0  1.0 -4.0
0.0  0.0  0.0  1.0
```

```
 0.7071  0.0000  0.7071  0.0000
 0.0000  1.0000  0.0000  0.0000
-0.7071  0.0000  0.7071 -4.0000
 0.0000  0.0000  0.0000  1.0000
```







OpenGL Clip coordinates

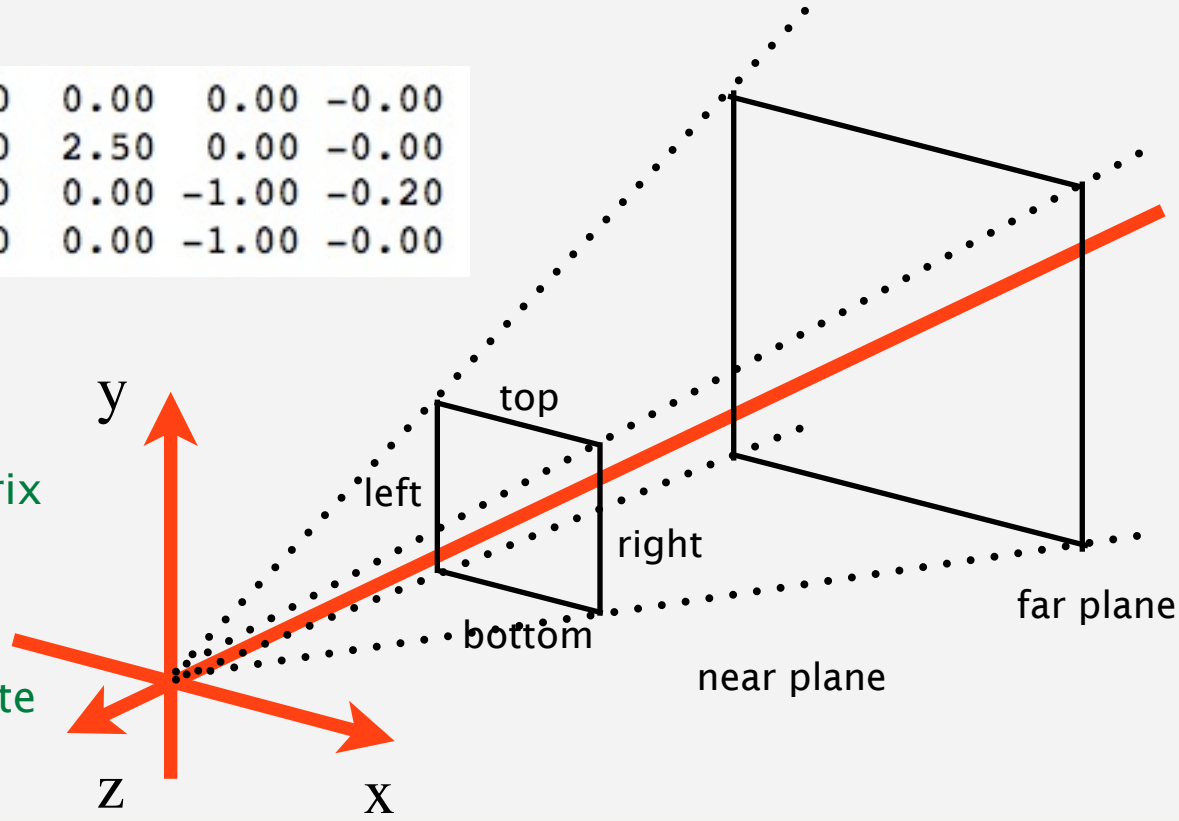
$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-0.04f, 0.04f, -0.04f, 0.04f, 0.1f, 100);
```

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

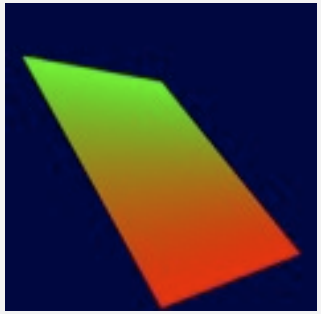
2.50	0.00	0.00	-0.00
0.00	2.50	0.00	-0.00
0.00	0.00	-1.00	-0.20
0.00	0.00	-1.00	-0.00

- `glFrustum (left, right, bottom, top, near, far)` multipliziert die aktuelle Matrix mit einer perspektivischen Matrix
- `gluPerspective` ist die benutzerfreundlichere Variante



GL_PROJECTION Perspektivisch

```
2.50  0.00  0.00 -0.00
0.00  2.50  0.00 -0.00
0.00  0.00 -1.00 -0.20
0.00  0.00 -1.00 -0.00
```



$$\begin{bmatrix} \frac{2nearVal}{right-left} & 0 & A & 0 \\ 0 & \frac{2nearVal}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- `glFrustum` (s. rechts) und `gluPerspective` produzieren Matrizen die nicht mehr dem Schema $a_4 = b_4 = c_4 = 0$, $d_4 = 1$ folgen!
- Nach der Multiplikation mit GL_PROJECTION entspricht die homogene Komponente w_c der Punkte nicht mehr unbedingt 1

$$A = \frac{right+left}{right-left}$$

$$B = \frac{top+bottom}{top-bottom}$$

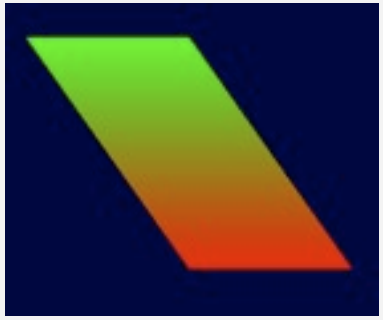
$$C = -\frac{farVal+nearVal}{farVal-nearVal}$$

$$D = -\frac{2\,farValnearVal}{farVal-nearVal}$$

(Quelle:http://www.opengl.org/sdk/docs/man/xhtml/glFrustum.xml)

GL_PROJECTION Orthographisch

```
0.50   0.00  -0.00   0.00
0.00   0.50  -0.00   0.00
0.00   0.00  -0.02  -1.00
0.00   0.00  -0.00   1.00
```



```
glOrtho(-2, 2, -2, 2, 0.1f, 100);
```

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{farVal-nearVal} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

$$t_x = -\frac{right+left}{right-left}$$

$$t_y = -\frac{top+bottom}{top-bottom}$$

$$t_z = -\frac{farVal+nearVal}{farVal-nearVal}$$

- `glOrtho` (s. rechts) erzeugt Matrizen, die dem Schema $a_4 = b_4 = c_4 = 0$, $d_4 = 1$ folgen!
- Nach der Multiplikation mit GL_PROJECTION ist die homogene Komponente w_c der Punkte weiterhin 1

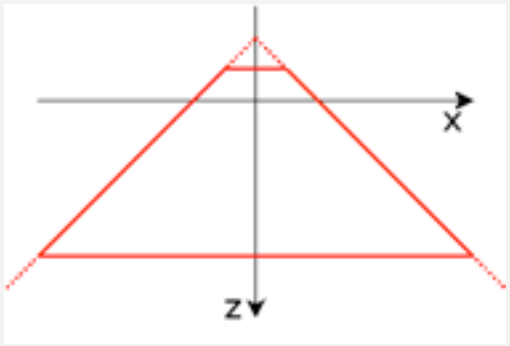
(Quelle:http://www.opengl.org/sdk/docs/man/xhtml/glOrtho.xml)

Normalized Device Coordinates

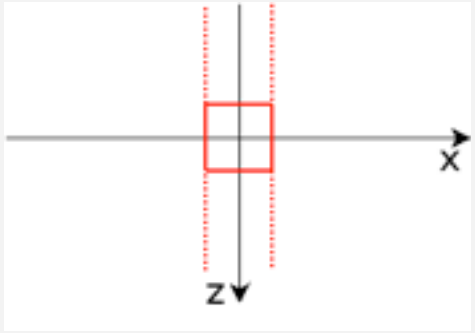
$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \end{pmatrix}$$

- Clip coordinates liegen im Wertebereich ($-w_c$; $+w_c$)
- Um die Darstellung einheitlich zu machen wird durch w_c geteilt, um die Koordinaten in den Wertebereich (-1; 1) zu bringen

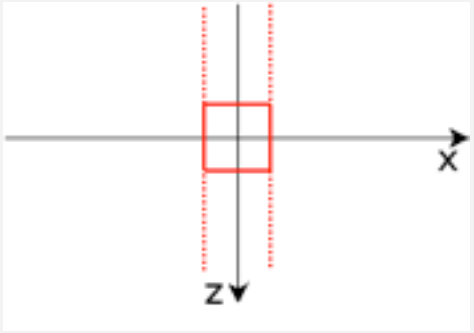
eye coordinates



clip coordinates



normalized device coordinates



(Quelle:http://homepages.uni-paderborn.de/prefect/glmath/spaces.html)

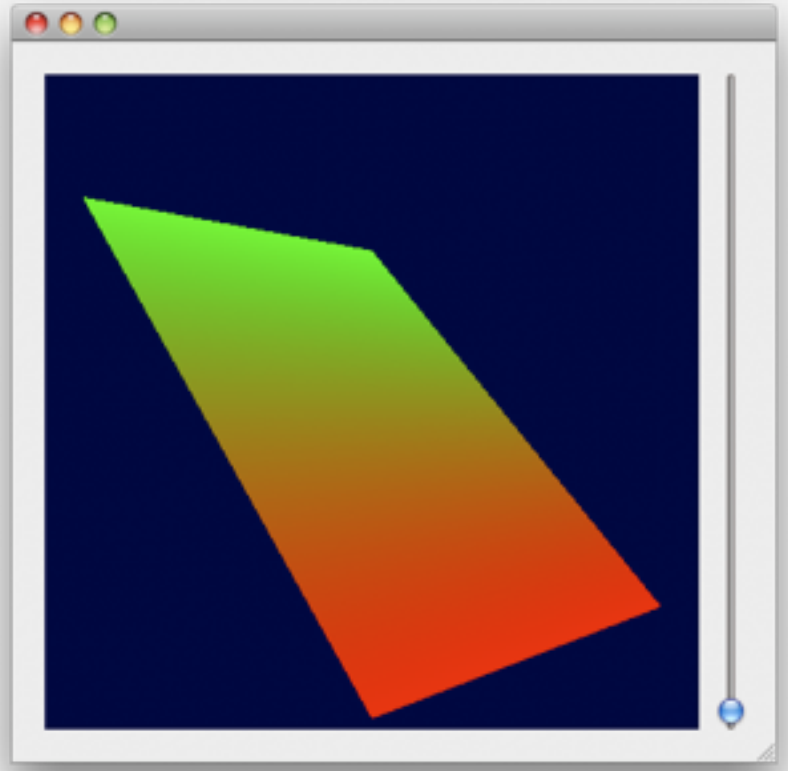
OpenGL Window coordinates

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} (p_x/2)x_d + o_x \\ (p_y/2)y_d + o_y \\ [(f-n)/2]z_d + (n+f)/2 \end{pmatrix}$$

```
glViewport(0, 0, (GLint)width, (GLint)height);
```

- p_d und p_d sind Breite und Höhe, o_x und o_y der Mittelpunkt des Ausgabefensters
- `glViewport(x0, y0, width, height)` berechnet aus den Ecken des Fensters automatisch p_d, p_d, o_x und o_y
- `n` und `f` sind initial auf 0 und 1 gesetzt, können mit `glDepthRange` verändert werden und haben Auswirkungen auf den Depth Buffer (s.u.)

OpenGL Transformationen - Beispiel



```
void GlTest::resizeGL(int width, int height){
    printf("%d, %d", width, height);
    glViewport(0, 0, (GLint)width, (GLint)height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-0.04f, 0.04f, -0.04f, 0.04f, 0.1f, 100);
    glMatrixMode(GL_MODELVIEW);
}

void GlTest::paintGL(){
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0, 0, -4);
    glRotatef(45, 0, 1, 0);
    glBegin(GL_POLYGON);
        glColor3f(1.0f, 0.0f, 0.0f); // red
        glVertex3f(-1.0f, -1.0f, 1.0f); // p1
        glVertex3f(1.0f, -1.0f, 1.0f);  // p2
        glColor3f(0.0f, 1.0f, 0.0f); // green
        glVertex3f(1.0f, 1.0f, -1.0f);  // p3
        glVertex3f(-1.0f, 1.0f, -1.0f); // p4
    glEnd();

}
```

OpenGL Matrixoperationen

- `glMatrixMode` Wechselt die aktuelle Matrix (`GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE`)

- `glLoadIdentity` Lädt die Identitätsmatrix
- `glMultMatrixf (GLdouble* m)` Multipliziert die aktuelle Matrix mit m

OpenGL Matrixoperationen

- OpenGL hat für jede Matrix einen Stack
- Matrizen können “gesichert” werden, um sie nach weiteren Transformationen wieder laden zu können
- Ohne Stack müssten alle Transformationen rückgängig gemacht werden (-> hoher Aufwand)
- Der Modelview-Stack kann mindestens 32, die beiden anderen mindestens zwei aufnehmen

OpenGL Matrixoperationen

- Speichern einer Matrix auf dem Stack:

  ```
  void glPushMatrix();
  ```

- Laden einer Matrix vom Stack:

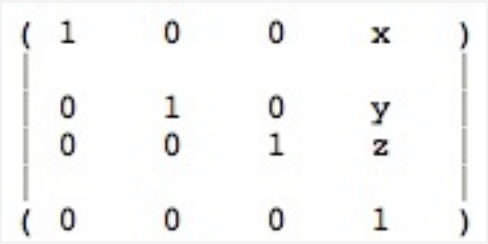
  ```
  void glPopMatrix();
  ```

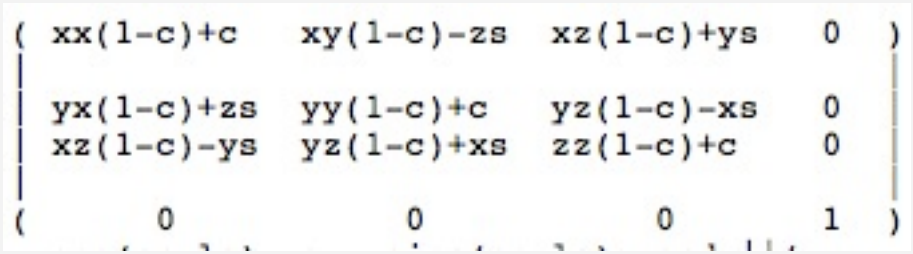
- Dadurch lassen sich Transformationen ohne großen Aufwand rückgängig machen

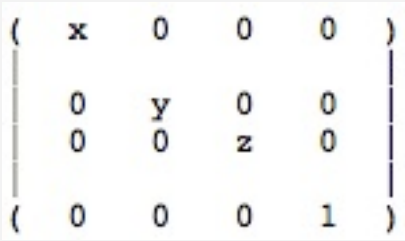
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0f, 0.0f, -10.0f); // pos(1)
glPushMatrix(); // speichern von pos(1)
glTranslatef(2.0f, 0.0f, 0.0f);
glRotatef(90.0f, 1.0f, 0.0f, 0.0f);
// zeichne etwas
glPopMatrix(); // zurück zu pos(1)
glTranslatef(-2.0f, 0.0f, 0.0f);
glRotatef(-90.0f, 1.0f, 0.0f, 0.0f);
// zeichne etwas
```

Transformationen (GL_MODELVIEW)

- Verschiedene Möglichkeiten Objekte zu transformieren:
  - `glTranslatef(float x, float y, float z)`
    - Verschiebt alle nachfolgenden Objekte entlang der drei Koordinatenachsen (Translation)
  - `glRotatef(float angle, float x, float y, float z)`
    - Rotiert alle nachfolgenden Objekte um Winkel `angle` (in Grad) um eine beliebige Achse (Rotation)
  - `glScalef(float x, float y, float z)`
    - Skaliert alle nachfolgenden Objekte entlang der drei Koordinatenachsen (Skalierung)







(Quelle: http://www.opengl.org)

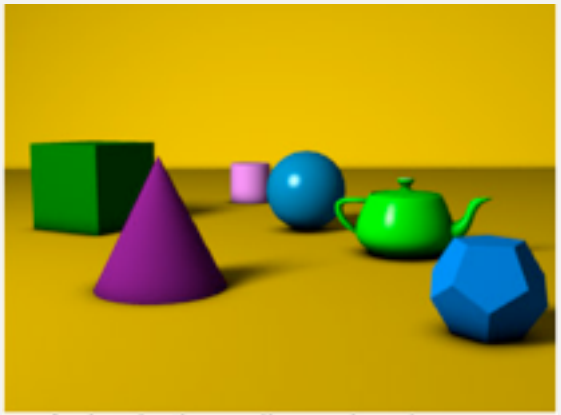
OpenGL Matrixausgabe

```
GLint viewport[4];
glGetIntegerv(GL_VIEWPORT, viewport);
printf("viewport: (%d, %d, %d, %d)\n",
       viewport[0], viewport[1], viewport[2], viewport[3]);

printf("modelview:\n");
GLdouble modelview[16];
glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
for(int i = 0; i < 4; i++){
    printf("%3.4f %3.4f %3.4f %3.4f\n",
           modelview[i], modelview[i + 4], modelview[i + 8], modelview[i + 12]);
}
printf("\n");

printf("projection:\n");
GLdouble projection[16];
glGetDoublev(GL_PROJECTION_MATRIX, projection);
for(int i = 0; i < 4; i++){
    printf("%3.4f %3.4f %3.4f %3.4f\n",
           projection[i], projection[i + 4], projection[i + 8], projection[i + 12]);
}
```

Depth Buffering

Depth Buffering

- Durch die Projektion von 3D nach 2D werden Objekte verdeckt
- Depth Buffering (Z-Buffering) führt dazu, dass es die richtigen trifft (z-Culling)
- Implementierung: Matrix in Bildgröße die z-Werte enthält



A simple three dimensional scene

Z-buffer representation

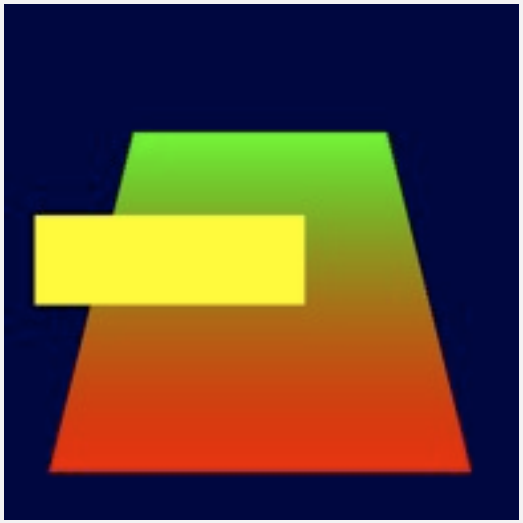
(Quelle: http://en.wikipedia.org/wiki/File:Z-buffer.jpg)

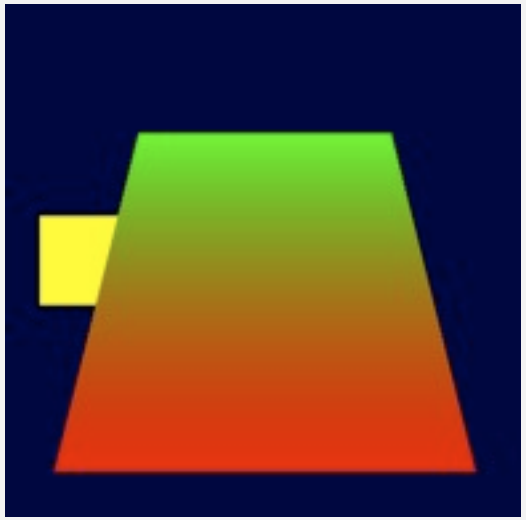
OpenGL Depth Buffer

- Wird der Depth Buffer in OpenGL nicht aktiviert können Objekte im Hintergrund solche im Vordergrund verdecken (Zeichenreihenfolge!)
- `glEnable(GL_DEPTH_TEST)` aktiviert den Depth Buffer
- `glClear(GL_DEPTH_BUFFER_BIT)` löscht den Depth Buffer (am Besten bei jedem Neuzeichnen:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

)





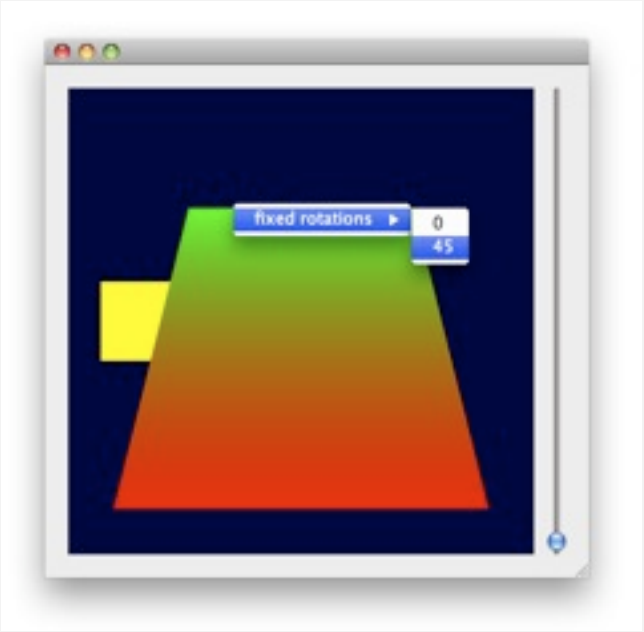
OpenGL Depth Buffer

- Weitere OpenGL Depth Buffer Funktionen:
- `glDepthRange (nearClip, farClip)` setzt n und f (s. Folie 20) auf Werte zwischen 0 und 1 damit nicht der gesamte Depth Buffer ausgenutzt wird
- `glDepthFunc` legt fest wann ein Pixel gezeichnet wird (Gültige Werte: `GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS)` (Default: `GL_LESS)`
- `glClearDepth` bestimmt bis zu welcher Tiefe der Depth Buffer beim Aufruf von `glClear` gelöscht wird (Default: 1)

Kontextmenüs in Qt

Qt Kontextmenüs

- Werden über `QMenu` und `QAction` realisiert.
- Zum Aufruf muss `QtWidget::contextMenuEvent(QContextMenuEvent* e)` überschrieben werden
- Um QActions auszuführen muss auf `connect` zurückgegriffen werden



Qt Kontextmenüs

gltest.h

```
class GlTest : public QGLWidget
{
    Q_OBJECT

public:
    GlTest(QWidget *parent = 0);
    ~GlTest();
    QSize sizeHint() const;

public slots:
    void rotateObject(int nuval);
    void rotate45();
    void rotate0();

protected:
    // overrides:
    void initializeGL();
    void paintGL();
    void resizeGL(int width, int height);

    void contextMenuEvent(QContextMenuEvent* e);

private:
    float rotaY;
    QMenu* popup;
};
```

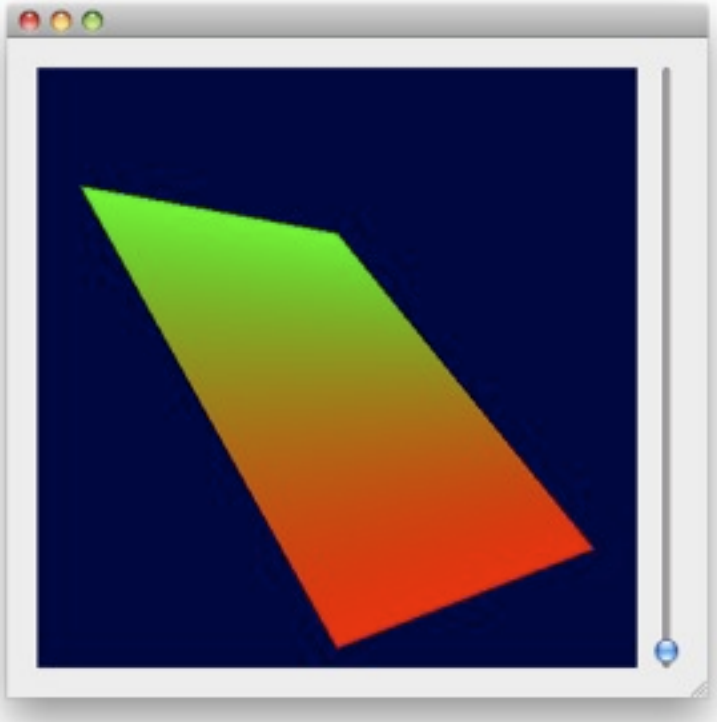
```
GlTest::GlTest(QWidget *parent)
    : QGLWidget(parent)
{
    popup = new QMenu(this);

    QAction* r0 = new QAction("0", this);
    QAction* r45 = new QAction("45", this);
    QAction* r90 = new QAction("90", this);
    r90->setDisabled(true);
    popup->addAction(r0);
    popup->addAction(r45);
    popup->addAction(r90);

    connect(r45, SIGNAL(triggered()), this, SLOT(rotate45()));
    connect(r0, SIGNAL(triggered()), this, SLOT(rotate0()));

    rotaY = 0;

}
```

gltest.cpp

Qt Kontextmenüs

```
void GlTest::contextMenuEvent(QContextMenuEvent* e){
    popup->exec(mapToGlobal(e->pos()));
}

void GlTest::rotate45(){
    this->rotateObject(45);
}
void GlTest::rotate0(){
    this->rotateObject(0);
}
```

gltest.cpp



Weiterführende Literatur

- http://www.opengl.org/sdk/docs/man/
- James Van Verth, Lars Bishop: Essential Mathematics for Games and Interactive Applications: A Programmer's Guide
- OpenGL ‘Redbook’: http://fly.srk.fer.hr/~unreal/theredbook/
- NeHe OpenGL Tutorials: http://nehe.gamedev.net/
- Greg Sidelnikov Tutorials: http://www.falloutsoftware.com/programming.php4