GPU Programmierung in OpenGL

Inhalt

- Grundlagen
- Konfiguration
- Geometrie
- Transformationen
- Viewing
- Beleuchtung
- Texturierung

GRUNDLAGEN

```
while (true) {
  read_input_devices();
  animate_scene();
  draw_scene();
}
```

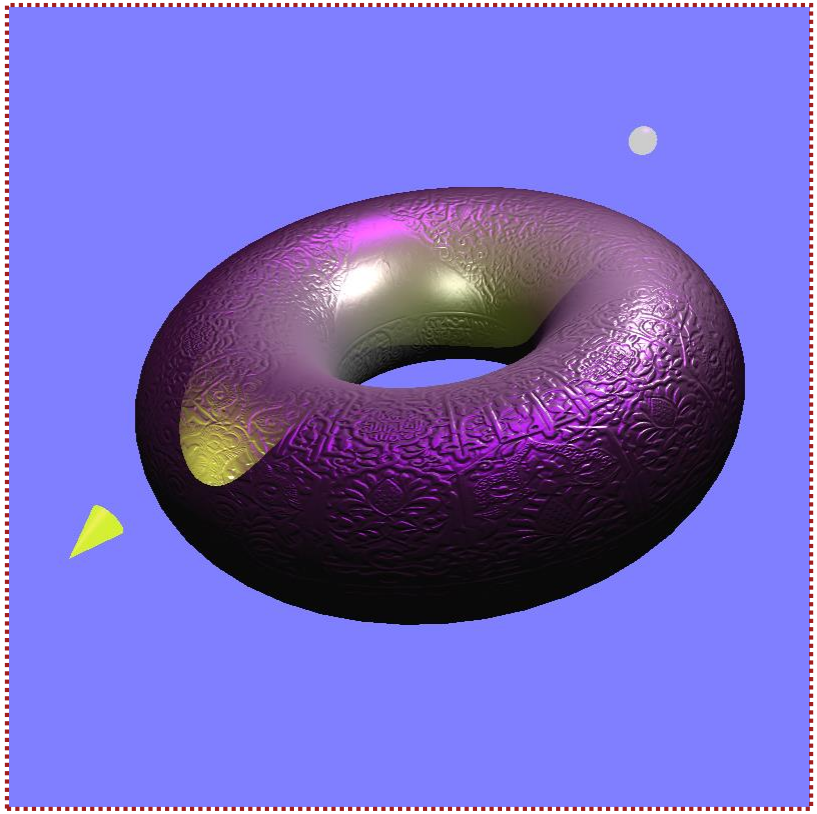
Prozessor und Graphikkarte
sind immer voll ausgelastet

Grundlagen
Ereignisverarbeitung – Schlafen

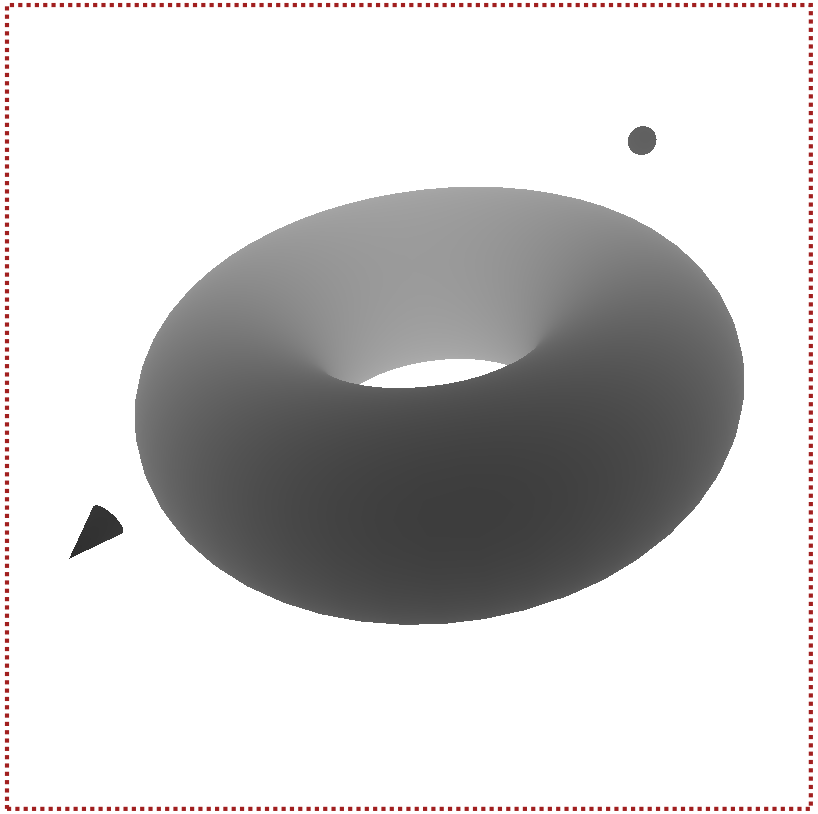

```
while (true) {
  read_input_devices();
  animate_scene();
  draw_scene();
  sleep_until_next_event();
}
```

Prozessor und Graphikkarte
werden nur bei Bedarf belastet

Grundlagen
Rasterdisplays und Bildpuffer



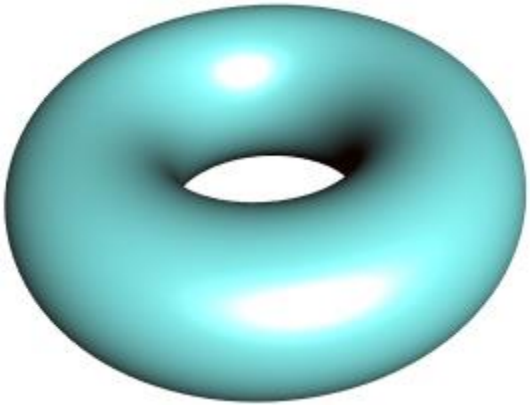
RGB[A]-Farbpuffer



Tiefenpuffer

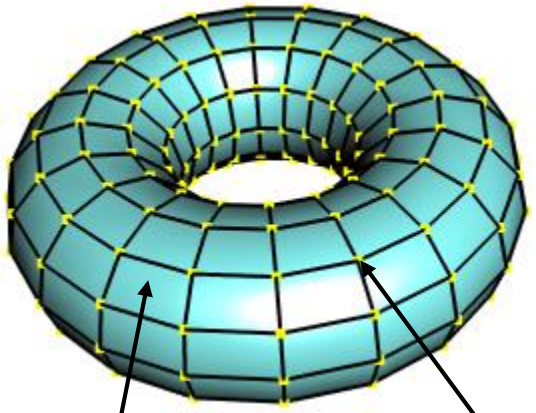
- Im Speicher der GPU werden Bereiche für verschiedene Bildpuffer reserviert, in denen die angezeigte Farbe aber auch Tiefen und weitere Werte gespeichert werden.

Grundlagen
Datenfluss in Graphiksysteme



Szene

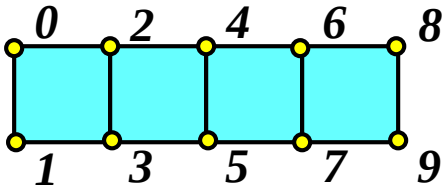
Tesselierung



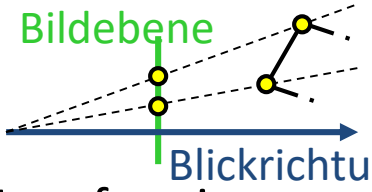
ebene Drei- oder Vierecke

Knoten mit Position, Normale, Farbe, …

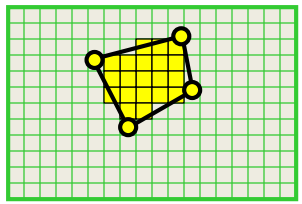
sende an GPU



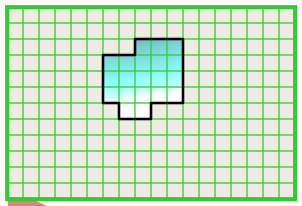
Drei- und Vierecke implizit in Knoten-reihenfolge



Transformiere Knoten in Bildkoordinaten



Zerlege Polygone in Fragmente

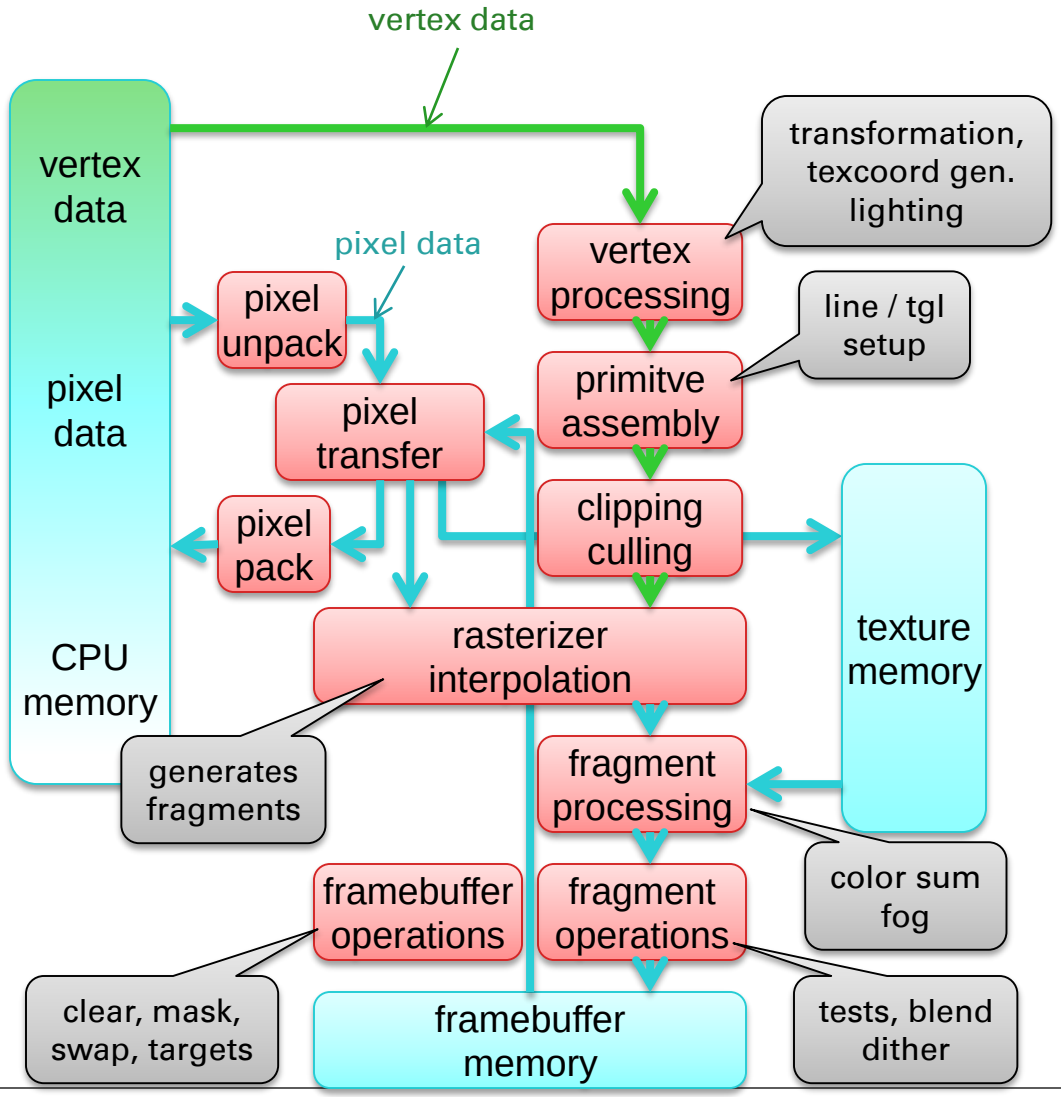


Berechne Fragment-farbe und Tiefe

GPU

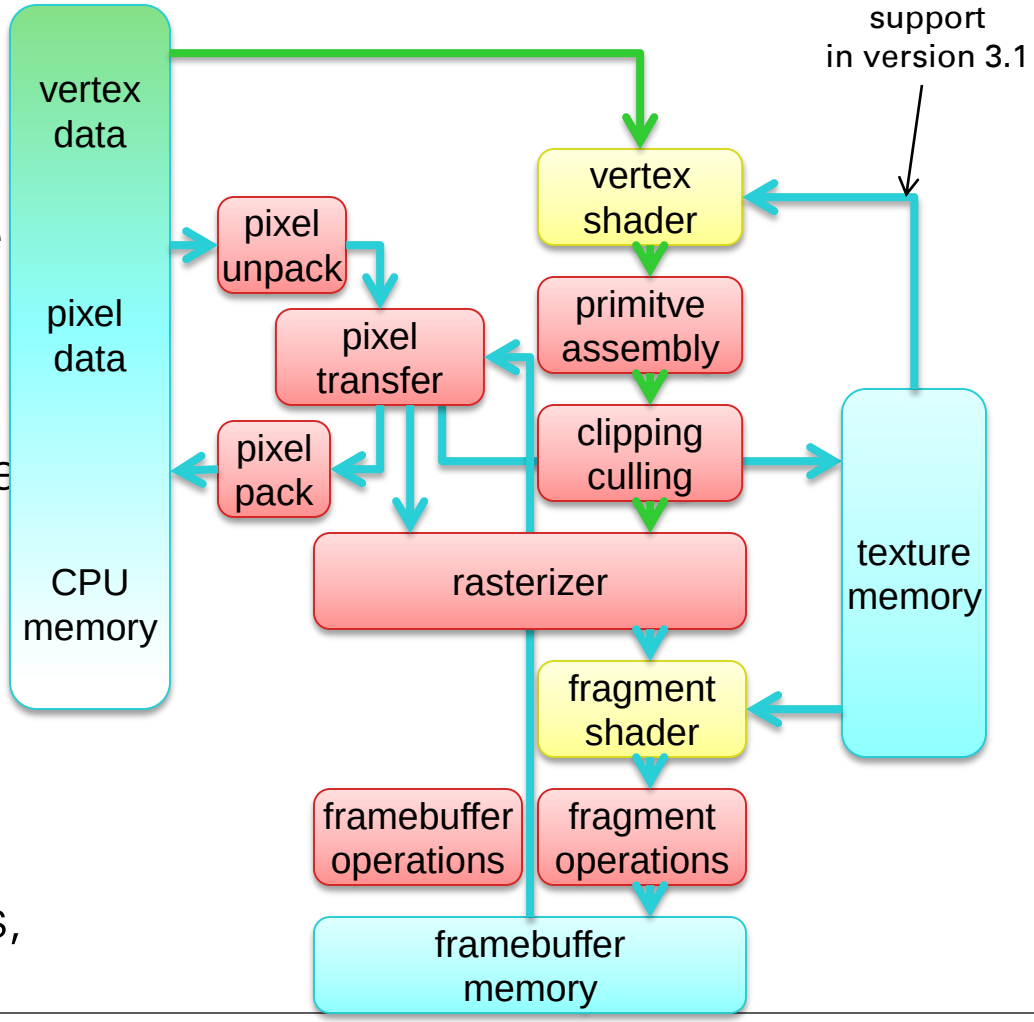
Grundlagen
OpenGL Fixed Function Pipeline

- application transmits
  - pixel data to textures or to rasterize bitmaps
  - vertex data to feed the vertex pipeline
- framebuffer consists of
  - [mutliple] rgb[a] colors
  - depth, stencil
- framebuffer data can be copied back to texture or application
- most processing is done in parallel



Grundlagen
OpenGL 2.0 Pipeline (used in WebGL)

- vertex and fragment processing can be programed with few restrictions
- programming language is a C variant
  - no pointers / references
  - all function calls by value
  - extensions for matrices
  - C++ constructors
  - no implicite type casts
  - no #include
  - access to large parts of OpenGL state (i.e. lights, materials, transforms)



Grundlagen
Übersicht

Computergraphik und Visualisierung

- Graphik-APIs (OpenGL / DirectX) … erlauben die direkte Programmierung der **Graphics Processing Unit (GPU)**
- Vorgehensweise
  - Konfiguration der GPU (Hintergrundfarbe, verwendete Bildpuffer)
  - Definition der Ansicht (Blickpunkt, Kameraparameter)
  - Definition der Beleuchtung
  - Definition von Materialparametern (Oberflächenfarbe, Texturen, Shader für die pro Pixel Auswertung eines Beleuchtungsmodells)
  - Definition der Geometriebeschreibung in der Szene
    - **Angabe der Art von Primitiven (Punkte, Linien, Dreiecke, Polygone)**
    - **Spezifikation von Oberflächennormale, Farbe, Texturposition und Raumposition pro Knoten der Primitive**
- Die GPU zerlegt die Primitive in Pixel (**Rasterisierung**)

Plattformunabhängige Graphikprogrammierung

- **OpenGL** … Bibliothek, die direkt mit Graphiktreiber kommuniziert, Daten zwischen CPU und GPU austauscht und das Darstellen von Graphikprimitiven auf der GPU initiiert.
- **GLU** (OpenGL Utility Library) … Hilfsbibliothek, mit Funktionen zur Bildmanipulation, Aufstellen von Matrizen, Triangulierung von Polygonen, Quadriken und NURBS
- **GLUT** (OpenGL Utility Toolkit) … Fenstermanagement, Ereignisverarbeitung, Overlay, Menus, Schriften, geometrische Grundprimitive (z.B. Teapot)
- Tutorials:
  - ECG-Seite: Tutorial Quellcode – Beispiel 05_glut
  - NeHe: http://nehe.gamedev.net/

KONFIGURATION

Konfiguration
Bildpuffer – Übersicht

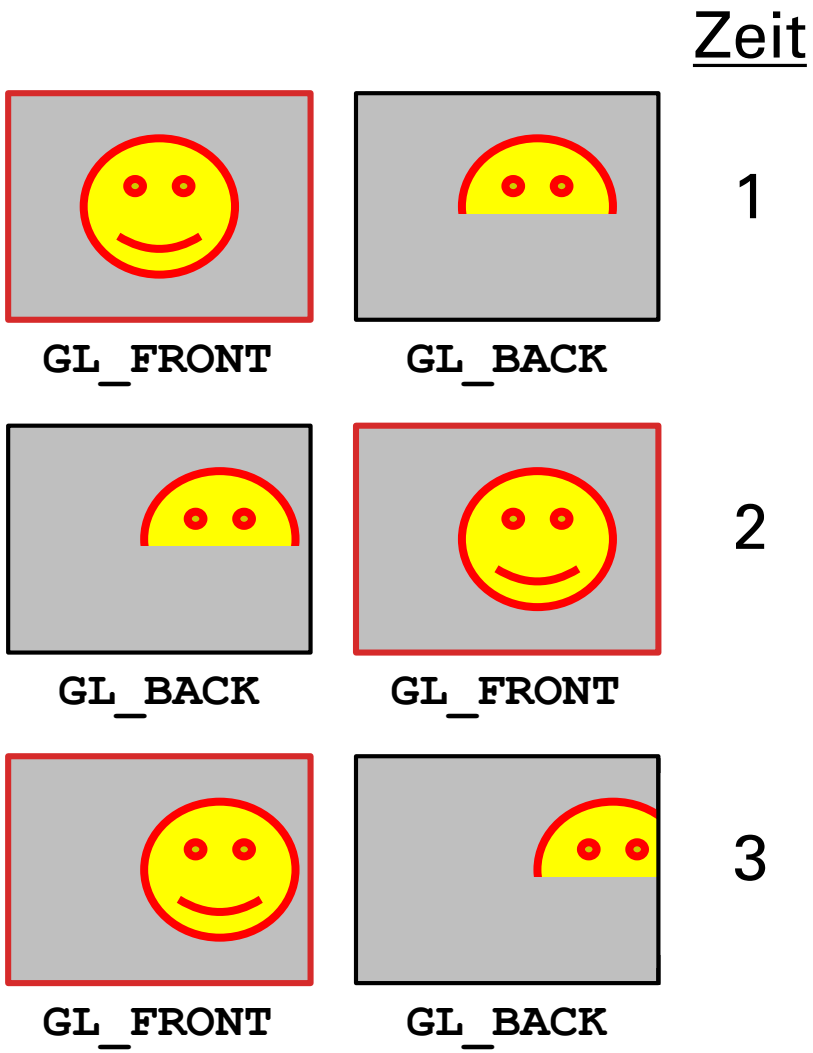
- Alpha-Kanal des Farbpuffers
  - vierter Kanal von gleichem Typ wie Farbkomponente als Transparenz oder Opazitätswerte interpretiert
  - Anwendung: Blenden von Farbwerten und Alpha-Test
- Double Buffer / Stereo:
  - Zwei bzw. vier Kopien des Farbpuffers für angezeigtes bzw. gerade erstelltes und für linkes bzw. rechtes Bild
- Tiefenpuffer / z-Puffer
  - pro Pixel ein 16/24/32-Bit Integer Tiefenwert im Intervall [0,1]
  - Anwendung: Tiefensortierung von Fragmenten mit Tiefentest
- Stencil-Puffer:
  - pro Pixel ein Bitarray als Flags oder Zähler interpretierbar
  - Anwendung: beliebige Masken, Schattenberechnung, CSG
- Accum-Puffer:
  - pro Pixel typischerweise doppelte Anzahl von Bits
  - Anwendung: Antialiasing, Motion-Blur

Konfiguration
Zustandsbasierte Programmierung

- Bei der Graphikprogrammierung (OpenGL, glut, Zeichenfenster in GUI-Programmierung, …) wird ein globaler Zustand / Kontext verwendet, der die aktuelle Konfiguration (Zeichenfarbe, Liniendicke, Lichtquellen, …) speichert.
- Vorteile
  - neue Zustandsparameter sind einfach hinzuzufügen
  - dieselbe Funktion kann mit unterschiedlichem Zustand für verschiedene Aufgaben verwendet werden
  - Der Zustand muss den Zeichenmethoden nicht übergeben werden
- Nachteile
  - Bei mehreren Fenstern muss der Zustand des Fensters aktiviert werden bevor Zeichenbefehle aufgerufen werden (besonders problematisch beim parallelen Programmieren)
  - Funktionen, die den Zustand ändern können zu unerwünschten Nebeneffekten führen. Vorgehensweise:
    - Vor dem Ändern von Zustandsparametern die aktuellen Werte zwischenspeichern und am Ende der Funktion wieder herstellen.

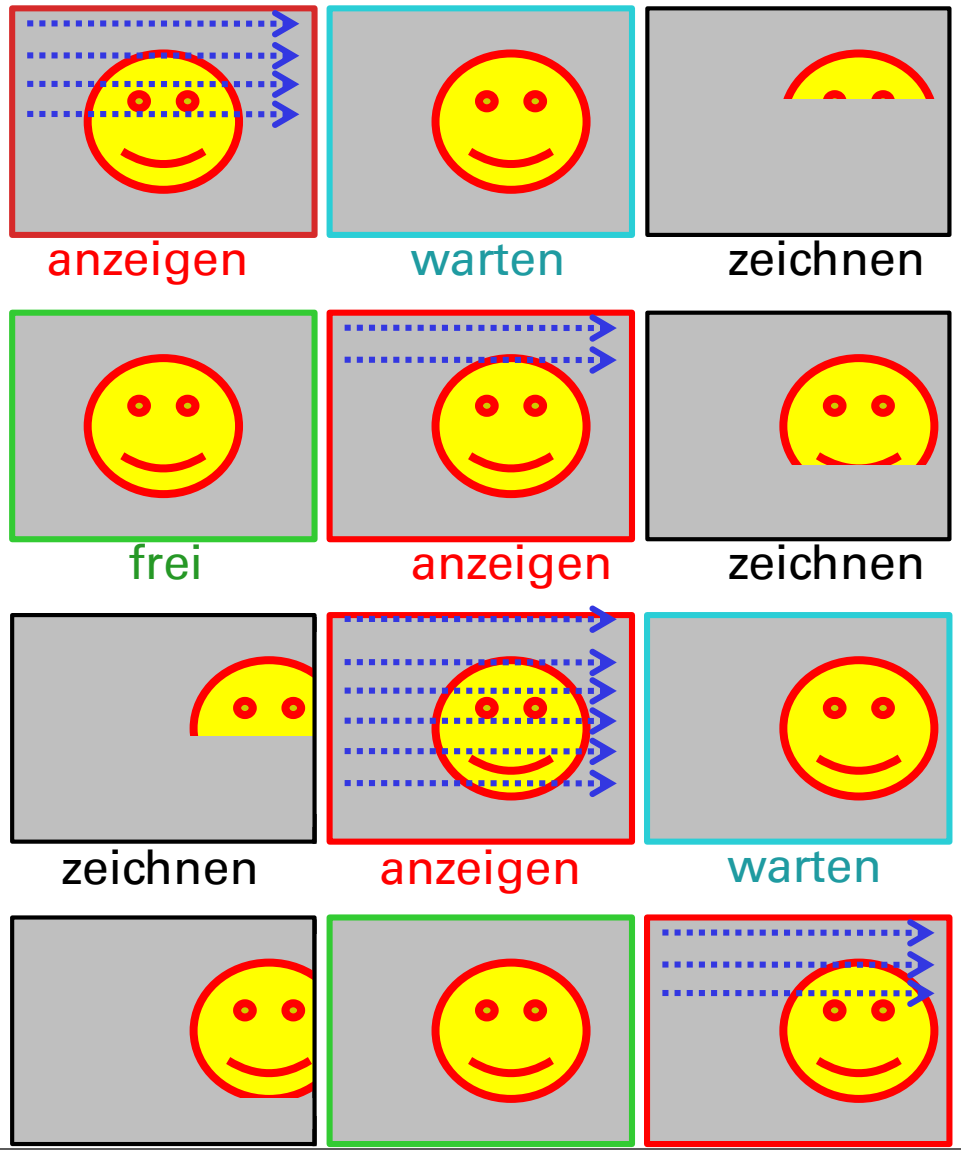
Konfiguration
Double Buffering

- Double Buffering … Um den Zeichenprozess nicht zu sehen, wird die aktuelle Graphikausgabe in einem zweiten Bildpuffer erstellt. Man unterscheidet also den sichtbaren Puffer und den Rendering Puffer. Nach dem Rendering wird der Rendering Puffer zum sichtbaren Puffer und der sichtbare zum Rendering Puffer für die nächste Darstellung
- Das Tauschen der Puffer wird in glut mit **`glutSwapBuffers`** bewerkstelligt



Konfiguration
Tripel Buffering

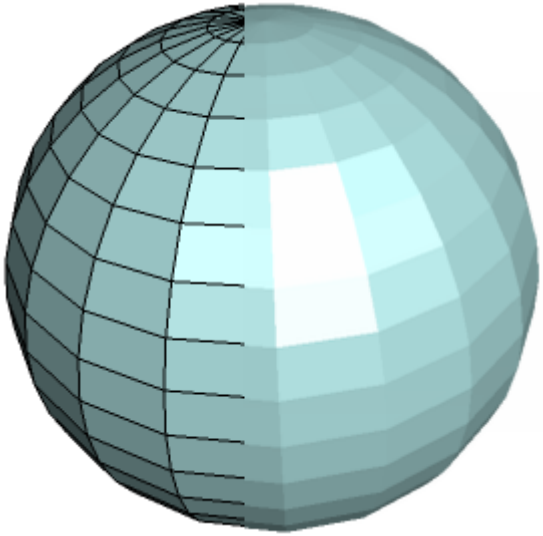
- Tripel Buffering … erfolgt das Tauschen der Puffer während die Graphikkarte den aktuellen Bildpuffer in ein Videosignal umwandelt, ergeben sich dennoch Artefakte (ein vertikaler Bildversatz der über das Bild läuft). Deshalb kann man im Graphiktreiber konfigurieren, dass der Puffertausch nur nach vollständigem Abtasten des aktuellen Bildpuffers durchgeführt wird. Um ein Warten auf den Puffertausch zu vermeiden kann man solange in einen 3. Bildpuffer rendern



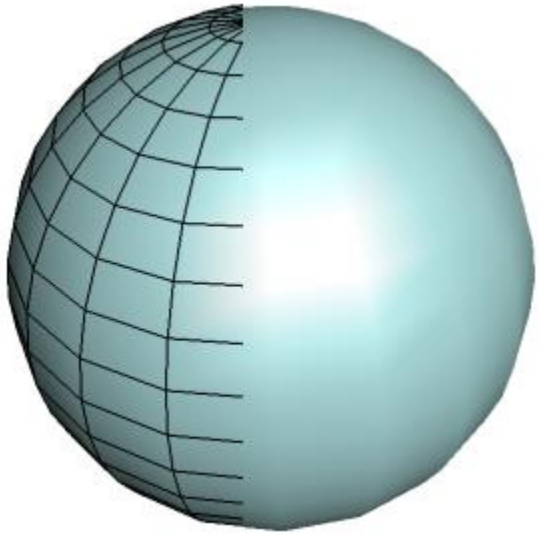
GEOMETRIE

Geometrie Grundlagen

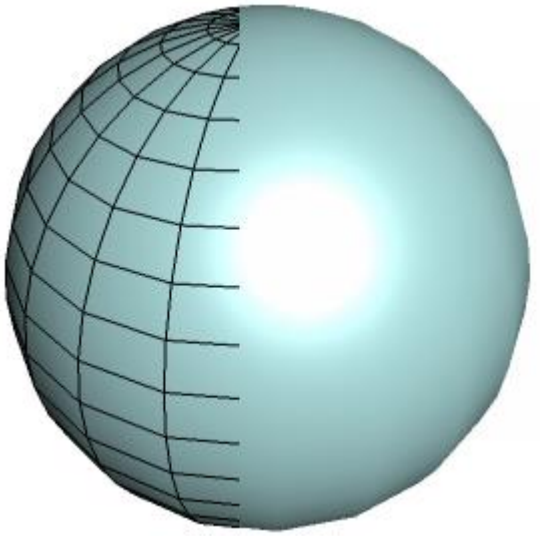
- Für die Darstellung von Oberflächen werden auf GPUs nur Drei- oder Vierecke als Zeichenprimitive unterstützt
- Glatte Flächen werden deshalb durch ebene Polygone, die man auch Facetten nennt, approximiert.
- Um die Flächen dennoch glatt erscheinen zu lassen, können für die Beleuchtung pro Knoten Oberflächennormalen spezifiziert werden
- Die Silhouetten sind jedoch unverändert



Beleuchtung auf Grund der Form
(auch Flat Shading genannt)



Beleuchtung mit Normalen
(auch Gouraud Shading genannt)

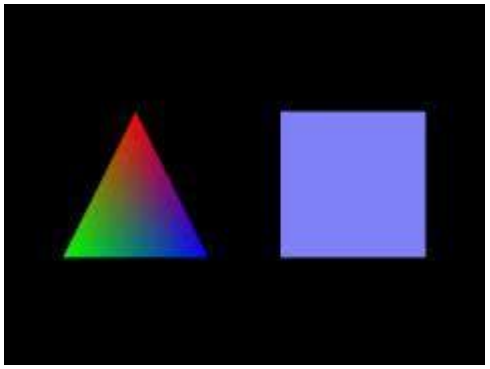


Beleuchtung pro Pixel
(auch Phong Shading genannt)

Geometrie
Beispiel zeichnen von Polygonen

Computergraphik und Visualisierung

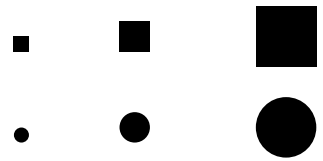
```
int DrawGLScene(GLvoid)            // Here's Where We Do All The Drawing
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Color&Depth Buffer
  glLoadIdentity();                // Reset The Current Modelview Matrix

  glTranslatef(-1.5f,0.0f,-6.0f);  // Left 1.5 Then Into Screen Six Units

  glBegin(GL_TRIANGLES);           // Begin Drawing Triangles
    glColor3f(1.0f,0.0f,0.0f);       // Set The Color To Red
    glVertex3f( 0.0f, 1.0f, 0.0f);   // Move Up One Unit From Center (Top Point)

    glColor3f(0.0f,1.0f,0.0f);       // Set The Color To Green
    glVertex3f(-1.0f,-1.0f, 0.0f);   // Left And Down One Unit (Bottom Left)

    glColor3f(0.0f,0.0f,1.0f);       // Set The Color To Blue
    glVertex3f( 1.0f,-1.0f, 0.0f);   // Right And Down One Unit (Bottom Right)
  glEnd()                          // Done Drawing A Triangle

  glTranslatef(3.0f,0.0f,0.0f);    // From Right Point Move 3 Units Right
  glColor3f(0.5f,0.5f,1.0f);       // Set The Color To Blue One Time Only
  glBegin(GL_QUADS);               // Start Drawing Quads
    glVertex3f(-1.0f, 1.0f, 0.0f);   // Left And Up 1 Unit (Top Left)
    glVertex3f( 1.0f, 1.0f, 0.0f);   // Right And Up 1 Unit (Top Right)
    glVertex3f( 1.0f,-1.0f, 0.0f);   // Right And Down One Unit (Bottom Right)
    glVertex3f(-1.0f,-1.0f, 0.0f);   // Left And Down One Unit (Bottom Left)
  glEnd();                        // Done Drawing A Quad
  return TRUE;                    // Keep Going
}
```



http://nehe.gamedev.net/

Geometrie
Graphische Primitive

- GPUs unterstützen das Rasterisierung von Punkten, Strecken, Dreiecken, konvexen Vierecken und konvexen Polygone
- Spezifikation in OpenGL 1.0
  - **`glBegin(`***PrimitiveType***`)`**
    [*PrimitiveType* ... `GL_POINTS / GL_LINES / GL_TRIANGLES / GL_QUADS / GL_POLYGON`]
  - Liste von Keilspezifikationen
    [`glColor`, `glNormal`, `glTexCoords`, `glVertex`]
    (Dabei muss `glVertex` pro Keil zuletzt aufgerufen werden)
  - **`glEnd()`**

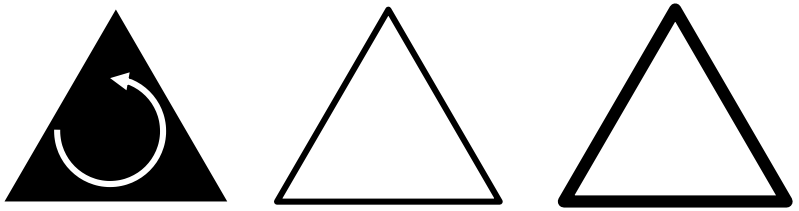


GL_POINT_SMOOTH aus

GL_POINT_SMOOTH an

Punktgröße mit **`glPointSize`**; oft gibt es maximale Punktgröße, z.B. 20 Pixel

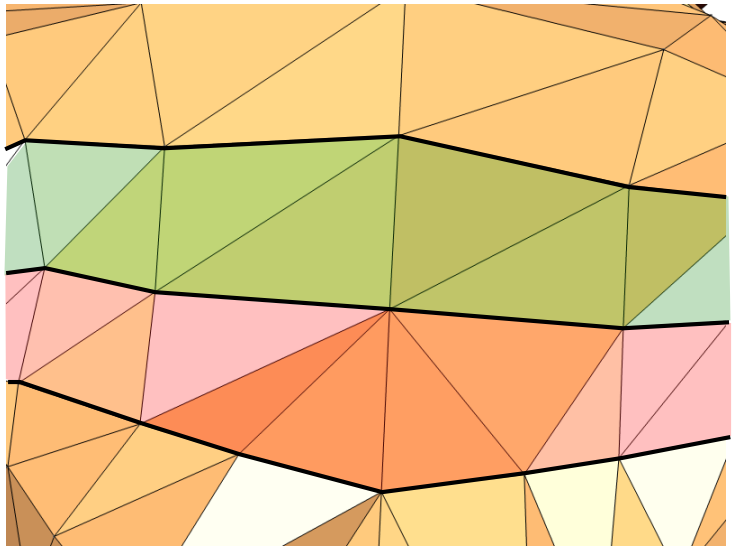
Liniendicke mit **`glLineWidth`**

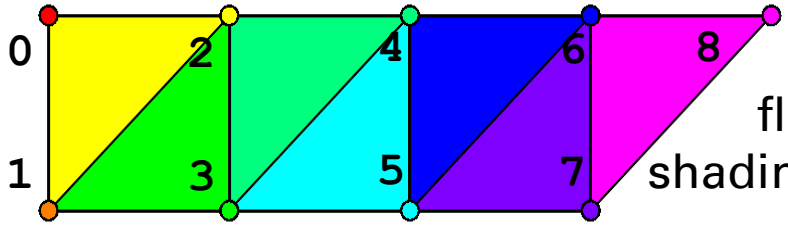


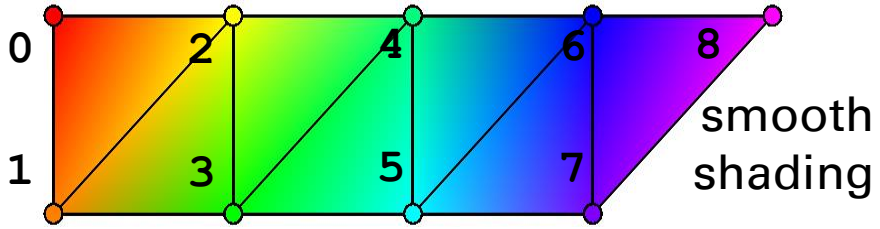
Reihenfolge bestimmt Außenrichtung, mit **`glPolygonMode`** kann Rasterisierung auf Kanten / Eckpunkte beschränkt werden

Geometrie
Dreiecks- / Viereckstreifen

- Bei geschlossenen Flächen kann die Information, die pro Keil spezifiziert wird durch den Einsatz von Dreiecks- und Viereckstreifen reduziert werden.
- Spezifikation pro Streifen über
  - **`glBegin(`**`GL_TRIANGLE_STRIP`**`)`**
  - Liste von Keilspezifikationen
  - **`glEnd()`**
- Wenn Flat Shading über **`glShadeModel`** aktiviert ist, werden pro Facette die Farbe und Normale nicht über die Facetten interpoliert sondern konst. vom letzten die Facette definierenden Keil genommen.







Geometrie Datenübertragung

Computergraphik und Visualisierung

Direct Mode

- ist eine nicht indizierte Spezifikation von Geometrie
- Attribute von jedem Keil werden einzeln spezifiziert
- Sehr flexibel aber auch langsamer und nicht für sehr große Modelle geeignet

Display Lists

- Geometrie und weitere Befehle können für häufige Verwendung in Display Listen verwaltet werden
  - Erzeugung
    `glGenLists`
    `glNewList`
    :
    `glEndList`
  - Verwendung
    `glCallList/s`
- Der Aufbau ist typischerweise zeitaufwendig weil der Treiber die Befehle optimiert

Vertex Arrays

- pro Attribute ein Array in Hauptspei.
- gemeinsame Indizierung aller Arrays
- Die Listen werden mit
  `glVertexPointer`,
  `glNormalPointer`,
  `glColorPointer`
  definiert und mit
  `glEnableClientState`
  aktiviert
- Indizes werden einzeln mit
  `glArrayElement`
  oder als Block mit
  `glDrawArrays`
  spezifiziert

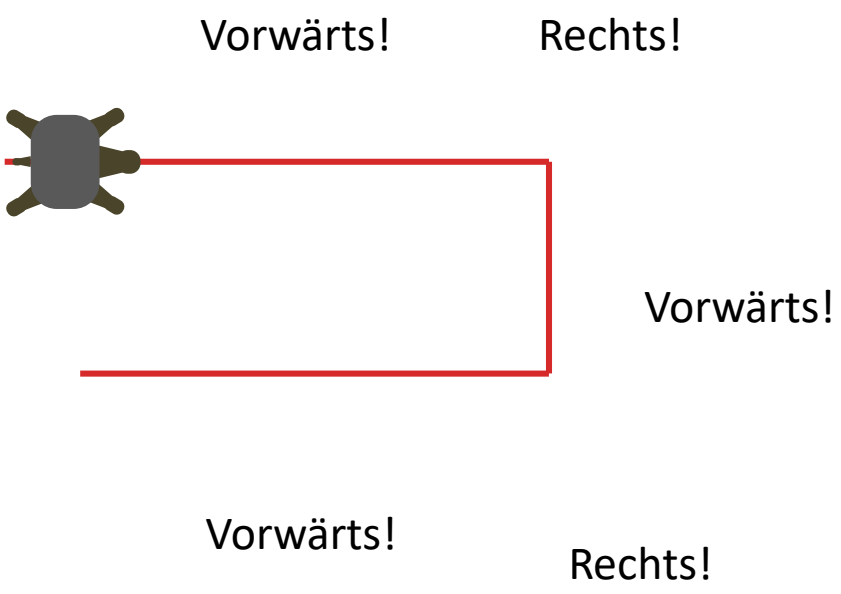
Vertex Buffer Objects

- Attribute können in einem (interleaved) oder mehreren Buffern auf GPU gespeichert werden
- nur gemeinsame Indizierung aller Attribute
- Buffer erzeugen
  `glGenBuffer`,
  `glBindBuffer`,
  `glBufferData`
- Buffer anbinden:
  `glVertexAttribPointer`
  und aktiviert:
  `glEnableVertexAttribArray`
- Rendern mit
  `glArrayElement`
  und als Block mit
  `glDrawArrays`

TRANSFORMATIONEN

Transformationen
Turtle-Graphik

- Transformationen dienen zum Positionieren und deformieren von Objekten
- In OpenGL wird das Prinzip der Turtle-Graphik angewandt. Dabei beschreiben Transformationsbefehle Bewegungen der Schildkröte
- Vorsicht! Bei der Turtle Graphik bewegt und dreht sich das Referenzkoordinatensystem (Schildkröte) mit.
- In OpenGL gibt es drei Grundtransformationen
  - Rotieren (drehen um Achse durch aktuellen Ursprung)
  - Translieren (verschieben)
  - Skalieren (Größe ändern)

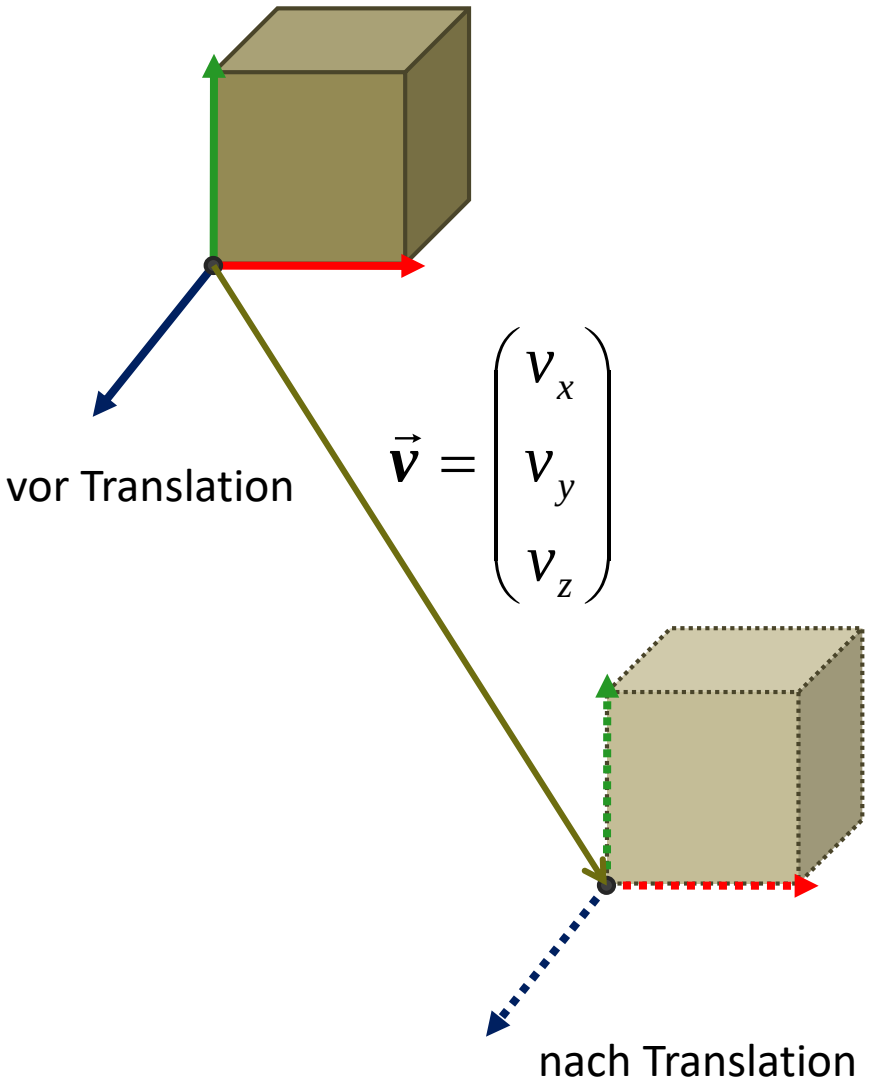



```
glRotated(angle,ax,ay,az)

glTranslated(vx,vy,vz)

glScaled(sx,sy,sz)
```

Transformationen
Translation

- Eine Translation wird mit Hilfe von einem Vektor $\vec{v}$ definiert, mit dem alle Objekte und das Koordinatensystem verschoben werden (Vektoraddition)
- Transformationen, die nach der Translation durchgeführt werden, nutzen als Referenz-koodinatensystem, das verschobene Koordinaten-system

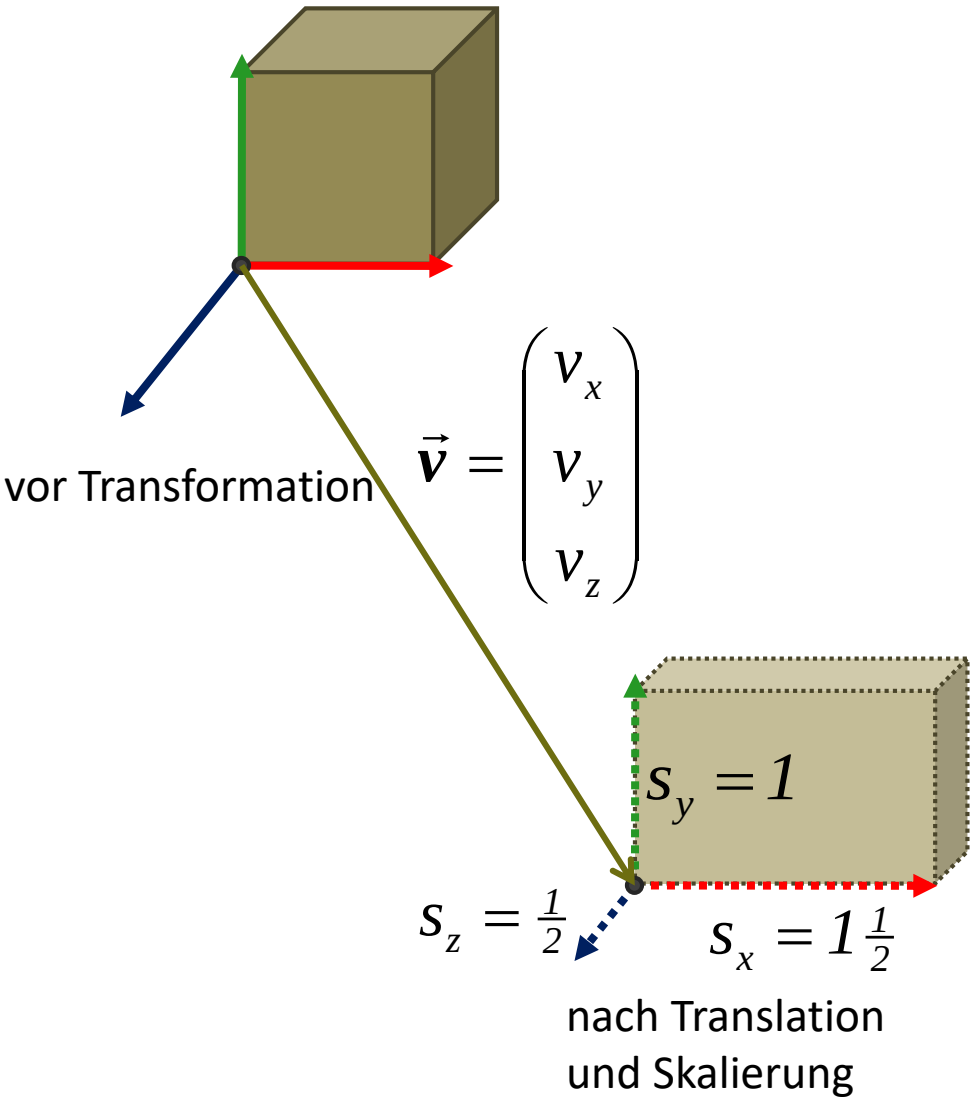
```
glTranslated(vx,vy,vz)
```



Transformationen
Skalierung

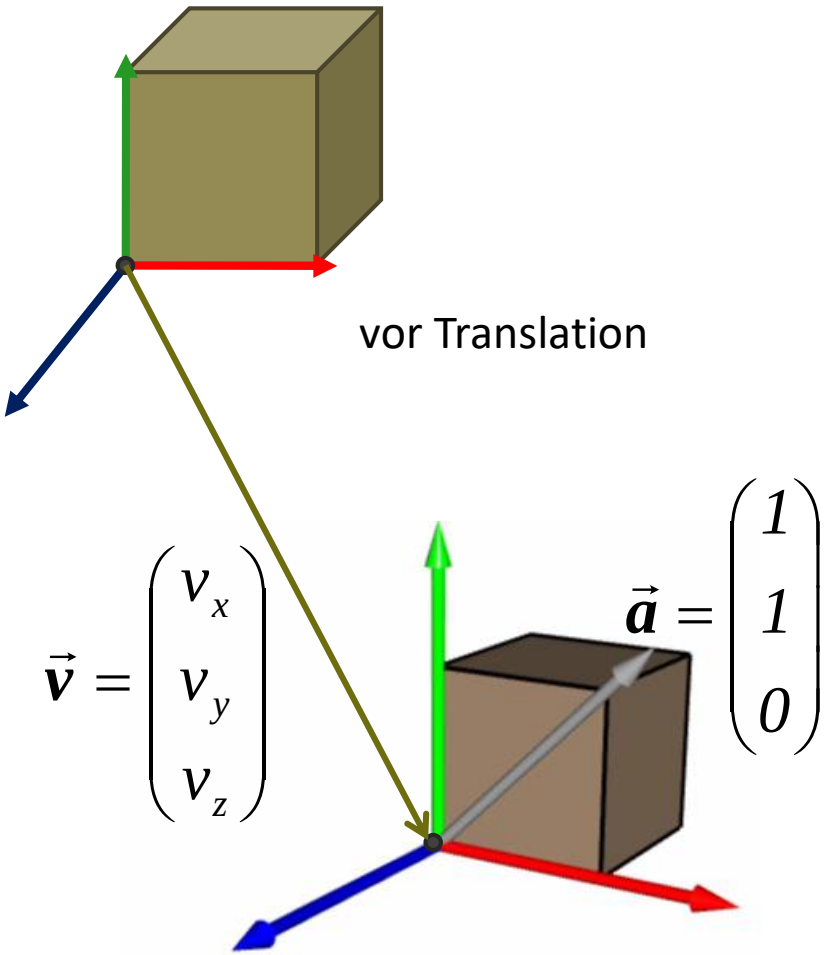
- Eine Skalierung wird durch drei Skalierungsfaktoren definiert, die an die Koordinaten der drei Achsen multipliziert werden.
- In Abbildung wird zusätzlich transliert, damit Urbild und Bild nicht übereinander liegen.
- Ist der Skalierungsfaktor
  - $s_? > 1$ … wird gestreckt
  - $0 < s_? < 1$ … wird gestaucht
  - $s_? = 0$ … wird geplättet
  - $s_? < 0$ … wird gespiegelt

```
glScaled(sx,sy,sz)
```



Transformationen
Rotation

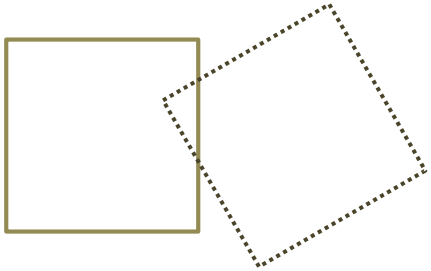
- Eine Rotation wird durch einen Rotationswinkel in Grad und drei Koordinaten eines Vektors der die Richtung der Rotationsachse definiert.
- Die Rotationsachse wird vom Richtungsvektor und dem Koordinatenursprung aufgespannt
- Gedreht wird gemäß der rechten Handregel: Daumen entlang der Achse, Finger zeigen Rotationsrichtung
- Umkehrung durch Inversion der Achse oder Negierung des Winkels



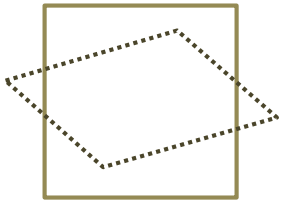

```
glRotated(angle,ax,ay,az)
```

Transformationen
Verkettung

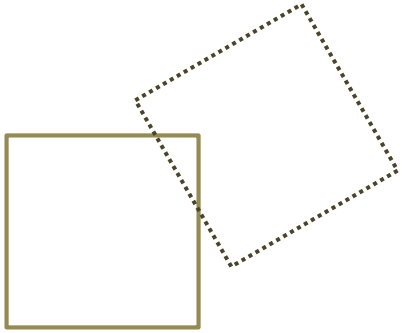
- Bei Nacheinanderausführung mehrerer Transformationen ist Ergebnis nicht immer intuitiv verständlich:



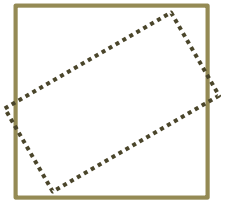
```
glTranslated(1,0,0);
glRotated(30,0,0,1);
```



```
glScaled(1,½,1);
glRotated(30,0,0,1);
```



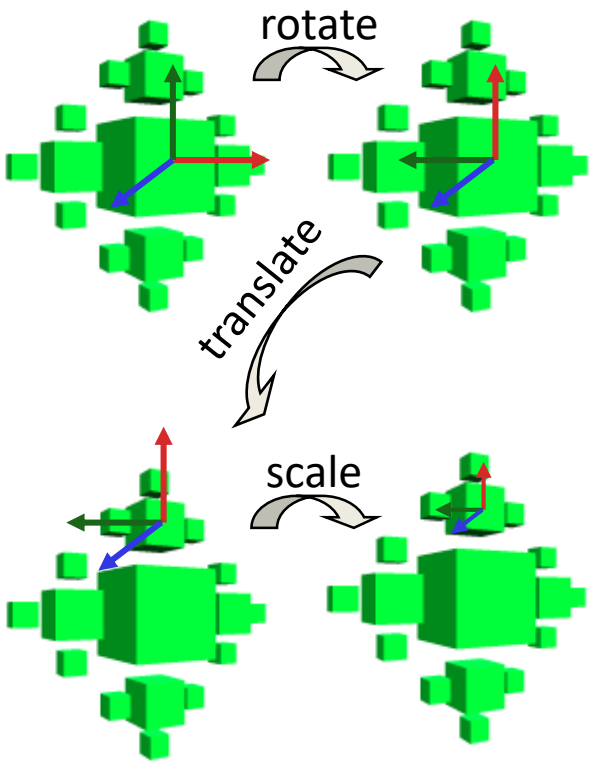
```
glRotated(30,0,0,1);
glTranslated(1,0,0);
```



```
glRotated(30,0,0,1);
glScaled(1,½,1);
```

Transformationen rekursives Beispiel

- Beispiel: rekursives Zeichnen eines Würfelbaums:
  - Die aktuelle Modelviewtransformation konvertiert von Welt- in Kamerakoordinaten
  - Vor der Rekursion beim Zeichnen des Baumes wird das aktuelle Koordinatensystem rotiert, verschoben und skaliert.




```
/// draw a cube tree of given depth in the current coordinate system
void draw_cube_tree(context& ctx, unsigned int depth, int nr_children = 3)
{
    ctx.tesselate_unit_cube();
    if (depth < rec_depth)
        // iterate children
        for (int i=0; i<nr_children; ++i) {
            // remember current coordinate system
            glPushMatrix();
                // rotate around z -axis by -90, 0, 90 or 180 degrees
                glRotated(i*90-90, 0, 0, 1);
                // move along x axis by 2 units
                glTranslated(2,0,0);
                // shrink child cube by a factor of 1/2
                glScaled(0.5,0.5,0.5);
                // recursively draw child cube
                draw_cube_tree(ctx, depth+1);
            // restore coordinate system before moving on to next child cube
            glPopMatrix();
        }
}
```

Transformationen
Speicherformat

- Vorsicht das Speicherformat für homogene Matrizen hat in OpenGL eine andere Reihenfolge, wie z.B. in DirectX:

`GLdouble m[16]` ... $\tilde{\boldsymbol{M}} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$

- Oft findet man auch die Interpretation, dass die homogenen Vektoren als Zeilen von links multipliziert werden. Das ist äquivalent, da $\left(\tilde{\boldsymbol{M}}\tilde{\boldsymbol{p}}\right)^T = \tilde{\boldsymbol{p}}^T \tilde{\boldsymbol{M}}^T$
- **`glMultMatrix(`**$\tilde{A}$**`)`** ersetzt die aktuelle Matrix $\tilde{\boldsymbol{M}}$ durch $\tilde{\boldsymbol{M}}\tilde{\boldsymbol{A}}$
- entsprechend werden Translation, Skalierung und Rotation von rechts an die aktuelle Matrix multipliziert.

Transformationen
Matrixstapel

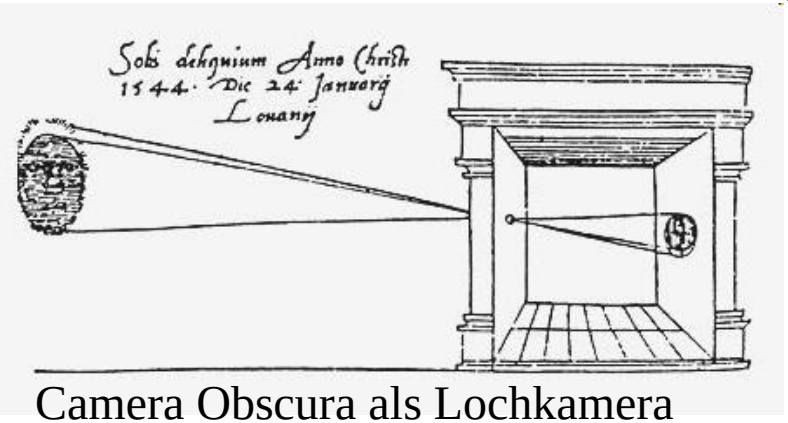
Computergraphik und Visualisierung

- Alle Transformationen werden mit homogenen Matrizen dargestellt, alle Punkte mit einem homogenen Vektor.
- es gibt drei aktuelle Transformationsmatrizen:
  - GL_PROJECTION $\tilde{\boldsymbol{P}}$... Transformation von Kamera- oder Weltkoordinaten (je nach gusto) in Bildkoordinaten
  - GL_MODELVIEW $\tilde{\boldsymbol{M}}$.. Transformation von Objektkoordinaten in Kamera- oder Weltkoordinaten
  - GL_TEXTURE $\tilde{\boldsymbol{T}}$... Transformation wirkt auf Texturkoordinaten
- pro Matrix gibt es einen Matrixstapel
- Es kann immer nur ein mit **`glMatrixMode`** gewählter Matrizentyp verändert werden
  - **`glLoadIdentity, glLoadMatrix, glMultMatrix`**
  - **`glRotate, glScale, glTranslate`**
  - **`glPushMatrix, glPopMatrix`**
- Jeder Knoten wird mit Modelview- und Projektionsmatrix in Bildkoordinaten umgerechnet: $\tilde{\boldsymbol{p}}' = \tilde{\boldsymbol{P}}\tilde{\boldsymbol{M}}\tilde{\boldsymbol{p}}$

VIEWING

Viewing
Lochkamera – Idee

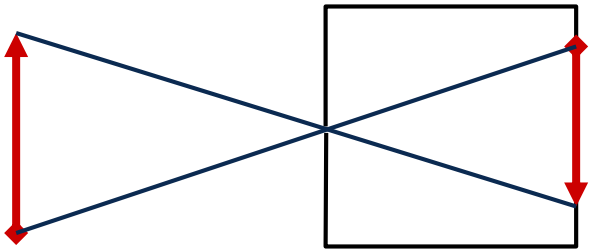
- Die Abbildung mit Hilfe einer Lochkamera (Camera Obscura) wurde 1544 von Frisius in Holzstich festgehalten

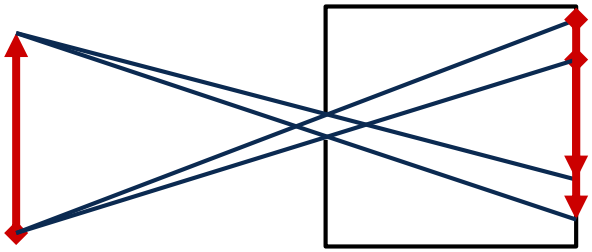


Camera Obscura als Lochkamera

Regnier Gemma Frisius

- Bild auf dem Kopf und je größer das Loch desto unschärfer

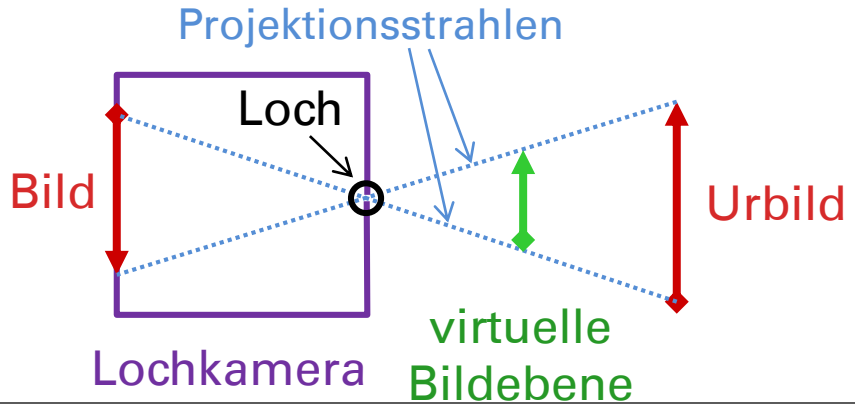




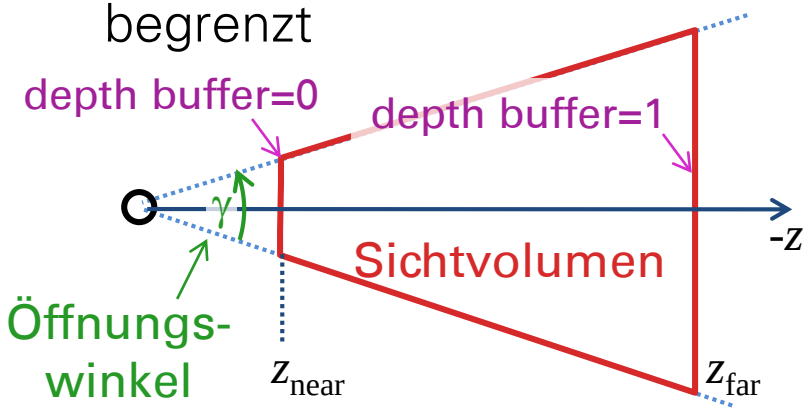
- In der CG meist idealisiertes Kameramodell:
  - unendlich kleines Loch ➔ unendliche Tiefenschärfe
  - Bild wird vor Projektionszentrum erzeugt

Viewing
Lochkamera – Spezifikation

- Die Ansicht auf die Szene wird durch Definition einer Kamera und deren Position und Orientierung in der Szene spezifiziert
- Es wird das Modell einer Loch-kamera verwendet bei dem alle Lichtstrahlen durch ein punktförmiges Loch fallen, das Projektionszentrum oder Augposition genannt wird.



- Um zu vermeiden, dass das Bild auf dem Kopf steht, wird es auf einer virtuellen Bild-ebene vor dem Projektions-zentrum aufgenommen.
- Das sichtbare Szenenvolumen wird durch Öffnungswinkel in Bild-x- und -y-Richtung definiert und für die Skalierung der z-Werte im Tiefenpuffer durch z_{near} und z_{far} Clipping Ebenen begrenzt



Viewing
Lochkamera in OpenGL

- Die Kamera wird in drei Schritten definiert (in glut-Tutorial im resize-Callback):

1. Bildausschnitt im Fenster in Pixelkoordinaten

```
glViewport(0, 0, W, H);
```

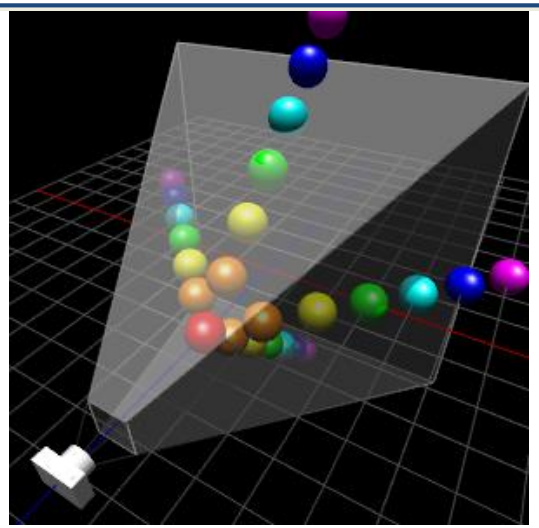
2. Für Projektion verwendetes Sichtvolumen relativ zum Augkoordinatensystem

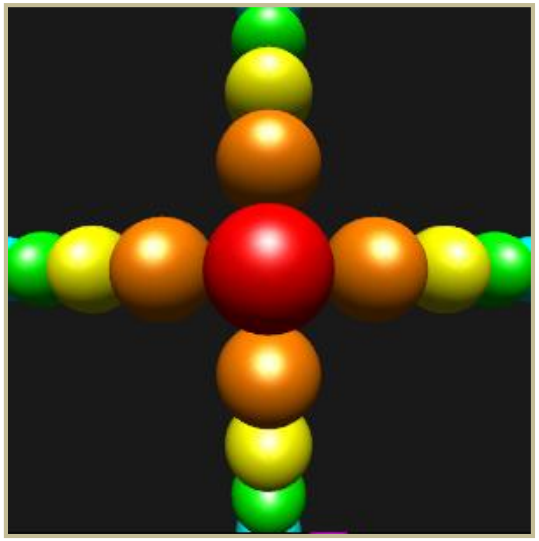
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(45,(float)W/H,znear,zfar);
```

3. Positionierung der Kamera im Koordinatensystem der Szene

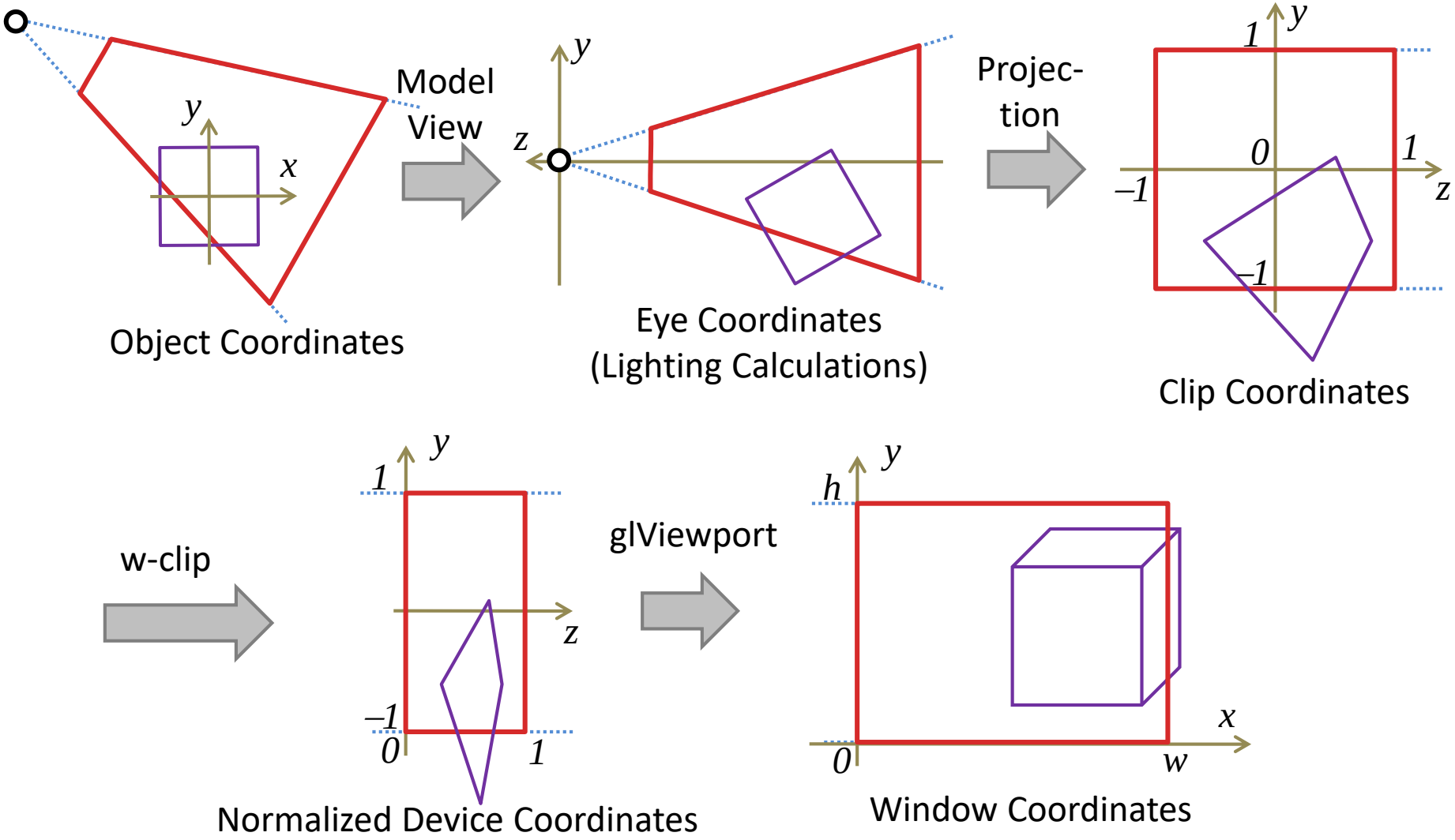
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0,0,5,0,0,0,0,1,0);
```

Augpunkt Fokus Obenrichtung





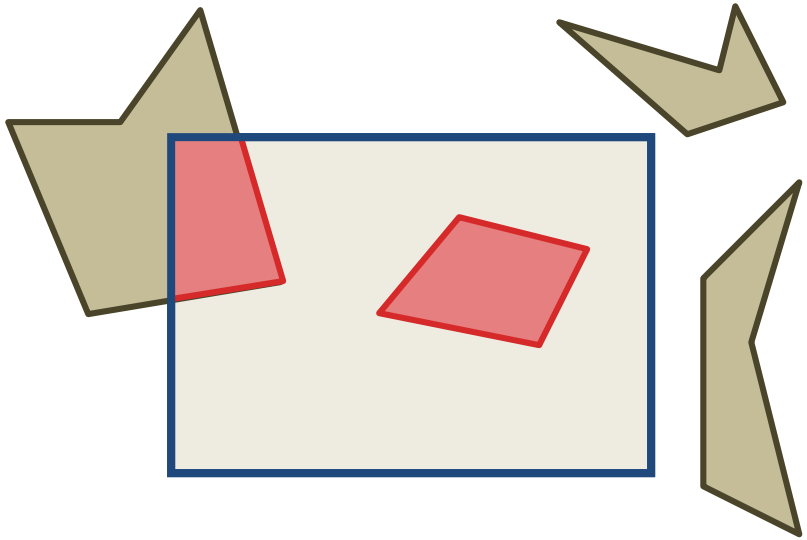
Viewing
Koordinatensysteme in OpenGL

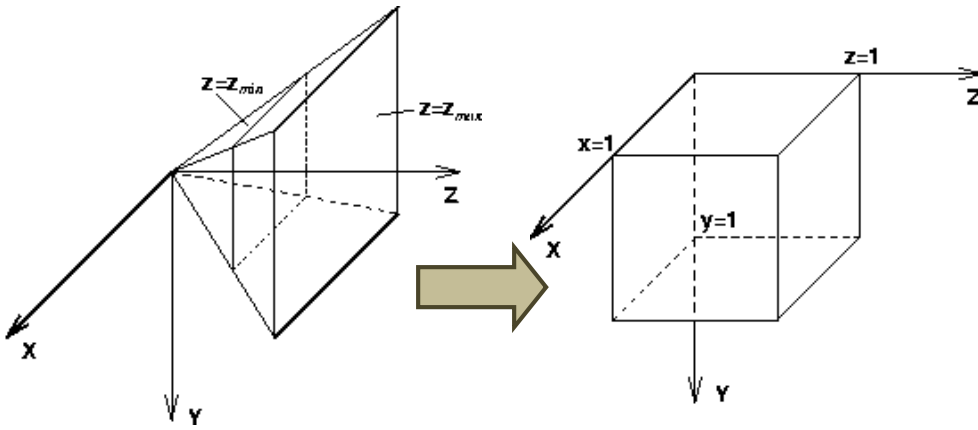


Object Coordinates

Viewing
Culling – Clipping

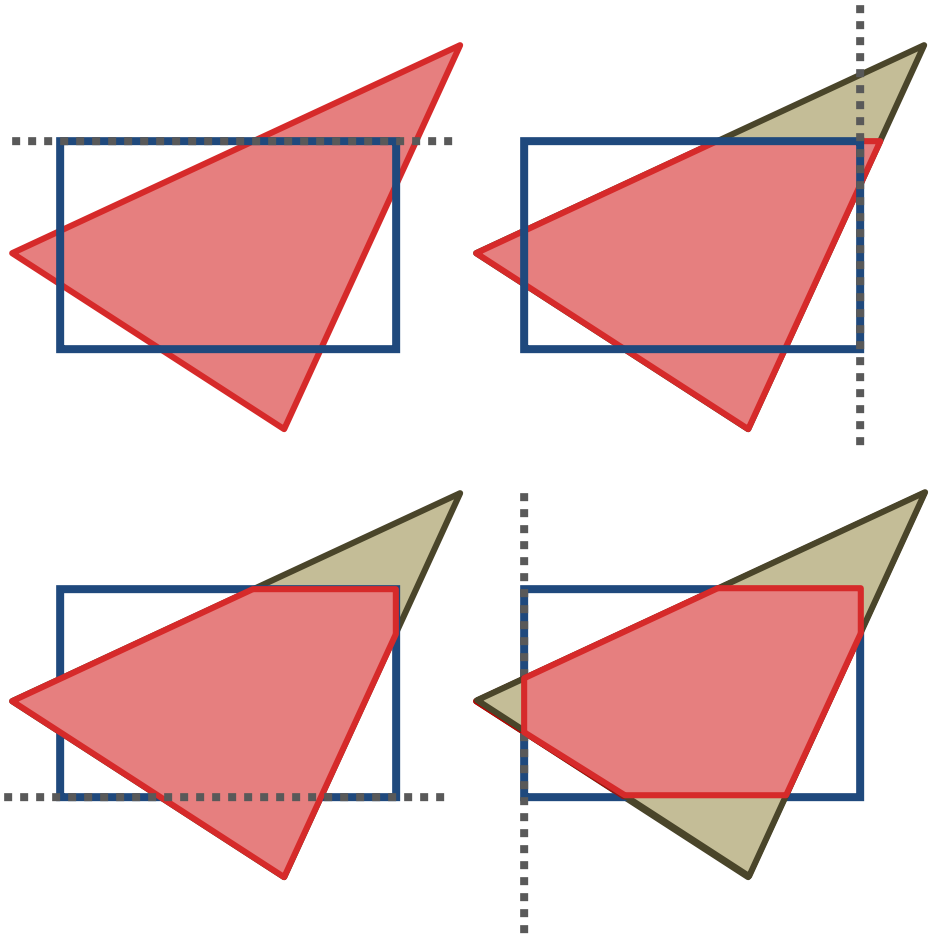
- Elimination von Objekten, die nicht auf den Bildschirm projizieren
- Bereichsgarantie der Bildschirmkoordinaten für die Rasterisierung
- Äquivalent zu Schnittmenge von Bildschirm mit Objekten
- Im 3D wird an Sichtpyramide ge-clipped.
  - Rückführung auf Quader-Clipping mittels perspektivischer Transform





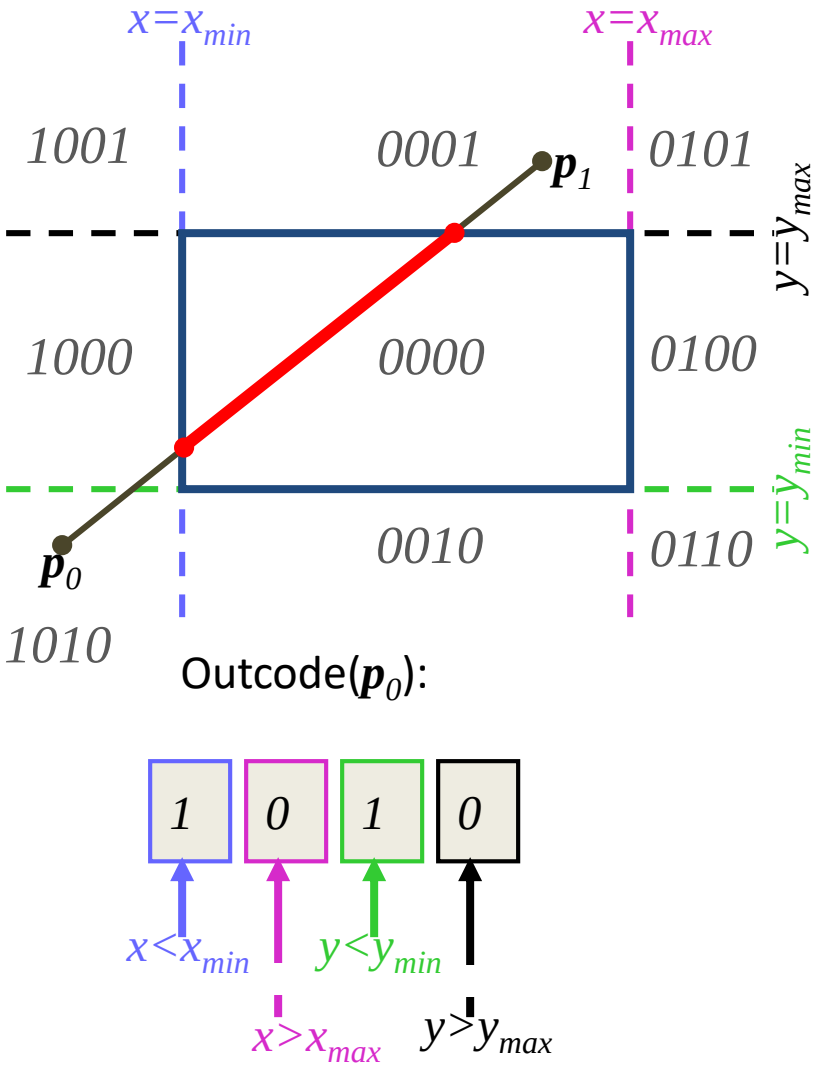
Viewing
Clipping – Sutherland-Hodgman

- Schnitt von Bildschirm mit Polygon teilt sich in
  - Teilsegmente von Polygonkanten (maximal ein Segment pro Kante)
  - Teilsegmente des Bildschirmrandes (mehrerer Segmente pro Kante möglich)
- Sutherland-Hodgman:
  - clipped Polygon sukzessive an den vier Randkanten des Bildschirms



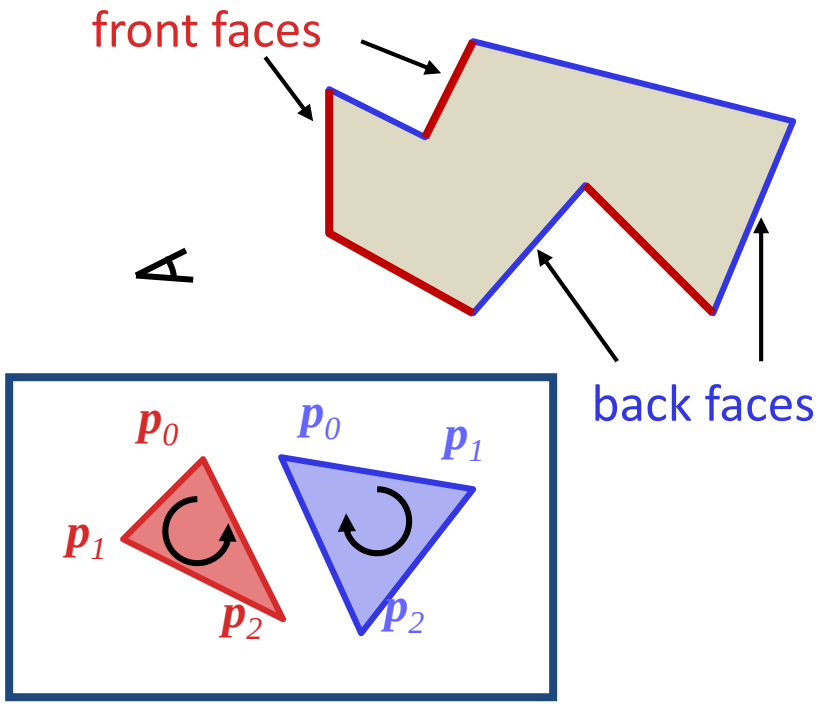
Viewing
Clipping – Cohen-Sutherland

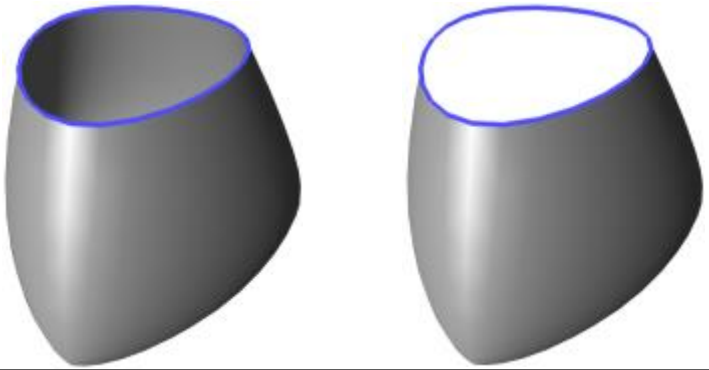
- **Outcode:** pro Begrenzungsebene ein Bit aus Koordinatenvergleich
- innen, bei Outcode *0000*
- außen, wenn Outcodes von $\boldsymbol{p}_0$ und $\boldsymbol{p}_1$ in einer Eins übereinstimmen (aus logischem Und)
- sonst Schnittberechnung, die maximal ein Segment ergibt (folgt aus Konvexität)



Viewing
Culling – Rückseitenelimination

- circa 50% aller Facetten sind dem Beobachter abgewandt
- bei geschlossenen Objekten sind diese nicht sichtbar
- beim back face culling werden abgewandte Facetten aufgrund ihres Umlaufsinnes auf dem Bildschirm eliminiert
- Vorsicht bei Objekten mit Rand / Löchern





Viewing
Vertigo Effekt bzw. Dolly Zoom

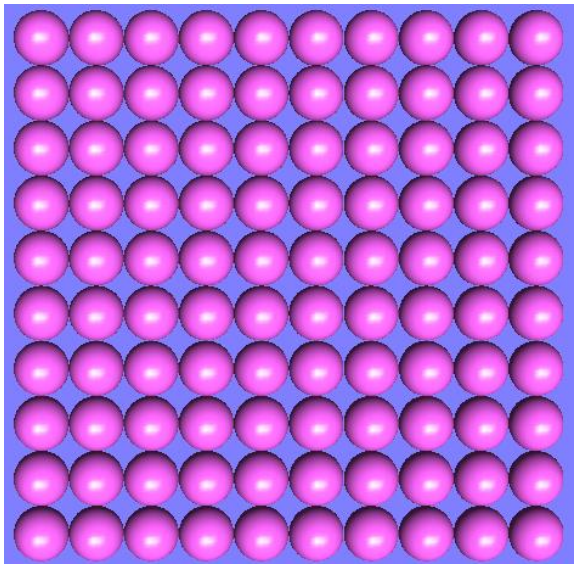
- Idee: gleichzeitiges wegbewegen und hineinzoomen (Verkleinerung des Öffnungswinkels), so dass Fokusobjekt gleich groß bleibt

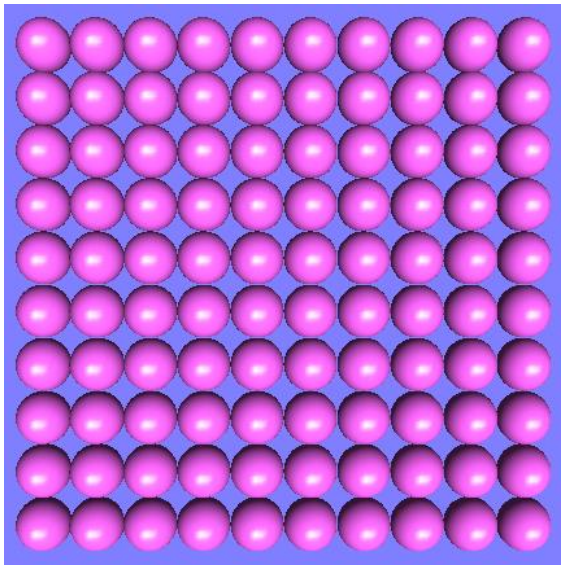
Hitchcock, Vertigo (1958)

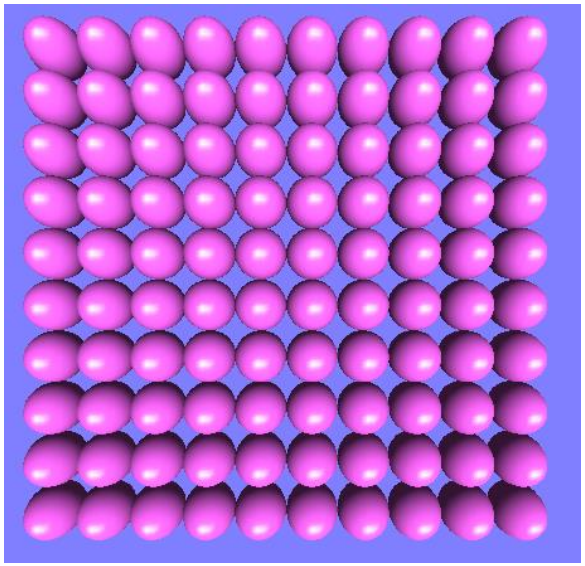
Menders, Road to Perdition (2002)

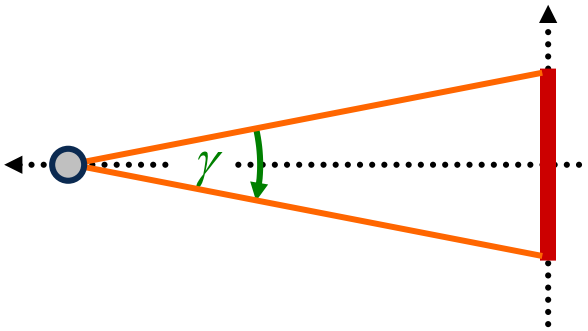
http://www.hdm-stuttgart.de/festschrift/Grusstexte/Fuxjaeger/GrusstextFuxjaeger.htm

Viewing
Öffnungswinkel

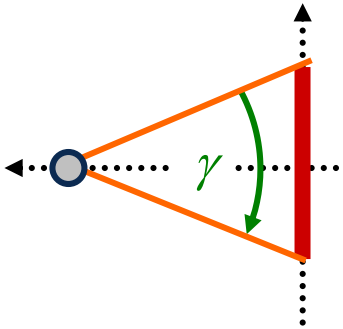




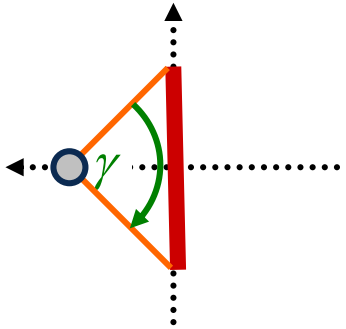




$\gamma = 22°$



$\gamma = 45°$

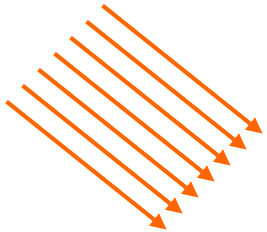


$\gamma = 90°$

BELEUCHTUNG

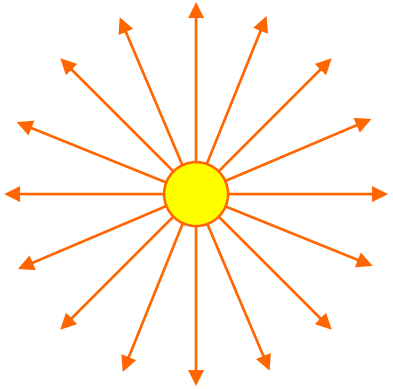
Beleuchtung
Lichtquellen

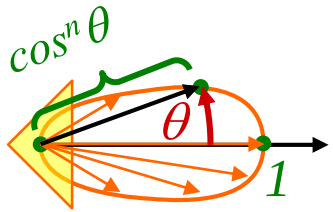
- Es ist die Spezifikation von Licht-quellen (meist vor dem Darstellen der Szene) und Oberflächenerschei-nung (während dem Darstellen der Szene vor Spezifikation der Form)
- OpenGL unterstützt bis zu acht Lichtquellen der folgenden Typen:
  - Richtungslichtquellen
    (definiert durch Richtung und farbige Intensität)
  - Punktlichtquellen
    (definiert durch Position, farbige Intensität und Intensitätsabfall mit wachsender Distanz)
  - gerichteter Strahler
    (erweitert Punktlichtquelle um Abstrahlrichtung, Bündelung und Öffnungswinkel)



Richtungs-
lichtquelle

Punkt-
lichtquelle

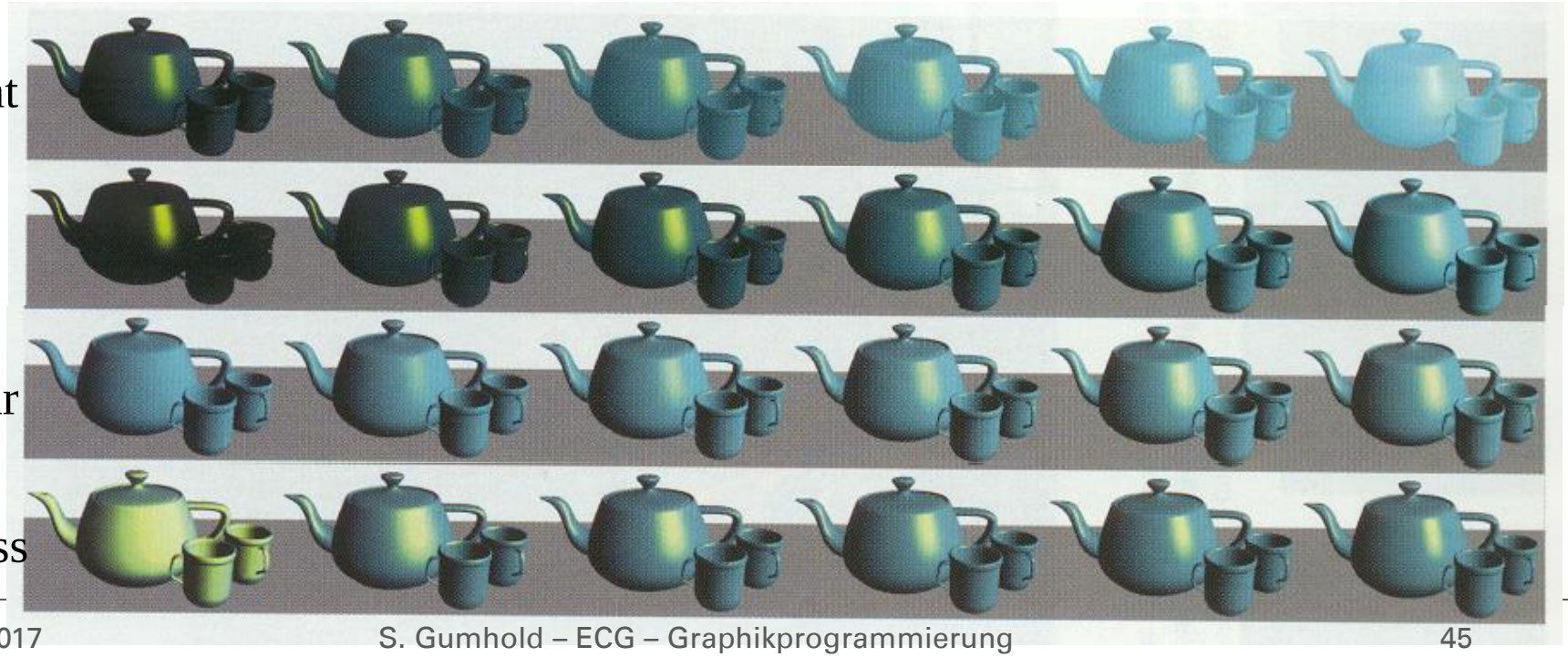




Strahler

Beleuchtung
Materialien

- Bei Lichtquellen und Materialien können drei Beleuchtungsanteile als RGB Farben angegeben werden.
- ambient … Richtungsunabh. Beitrag der Streulicht emuliert
- diffuse … abh. von Orientierung der Oberfl. zur Lichtquelle (am hellsten bei senkrechter Bestrahlung)
- spekular … zusätzlich von Beobachterposition abh. (am hellsten bei spiegelnder Reflektion – emuliert Glanzlichter deren Fokusierung über die shininess einstellbar ist)



Beleuchtung OpenGL

- Zuerst muss Beleuchtungsrechnung im allgemeinen und die verwendeten Lichtquellen im einzelnen angeschalten werden
- Dann werden die verschiedenen Licht und Materialparameter spezifiziert mit **glLightf** und **glMaterialf**
- Lichtpositionen sind relativ zum aktuellen Koordinatensystem und werden durch die Befehle glTranslate, glRotate, glScale transformiert.

```
void init_application()
{
    :
    // turn on lighting with one light and
    // where material is based on current color
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_NORMALIZE);
    glEnable(GL_COLOR_MATERIAL);
    // specify light source
    const GLfloat light_ambient[]  = { 0, 0, 0, 1 };
    const GLfloat light_diffuse[]  = { 1, 1, 1, 1 };
    const GLfloat light_specular[] = { 1, 1, 1, 1 };
    const GLfloat light_position[] = { 2, 5, 5, 0 };
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    // specify default material
    const GLfloat mat_ambient[]    = {.7f,.7f,.7f,1 };
    const GLfloat mat_diffuse[]    = {.8f,.8f,.8f,1 };
    const GLfloat mat_specular[]   = {1.f,1.f,1.f,1 };
    const GLfloat high_shininess[] = {};
    glMaterialfv(GL_FRONT, GL_AMBIENT,   mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,   mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR,  mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, 100.0f);
}
```

TEXTURIERUNG

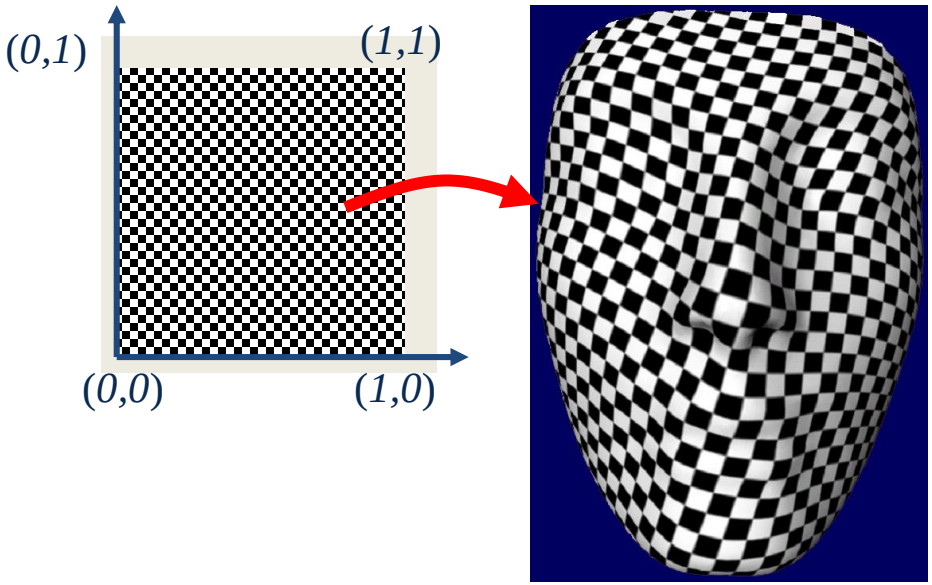
Texturierung Übersicht

- Erzeugung der Textur

```
glGenTextures(1,&id)
glBindTexture(GL_TEXTURE_2D,id);
glTexImage2D(
      GL_TEXTURE_2D, 0,
      GL_RGB, 512, 512, 0
      GL_RGB, GL_UNSIGNED_BYTE,
      dataPointer);
```

oder

```
gluBuild2DMipmaps(
      GL_TEXTURE_2D, 3,
      512, 512,
      GL_RGB, GL_UNSIGNED_BYTE,
      dataPointer);
```

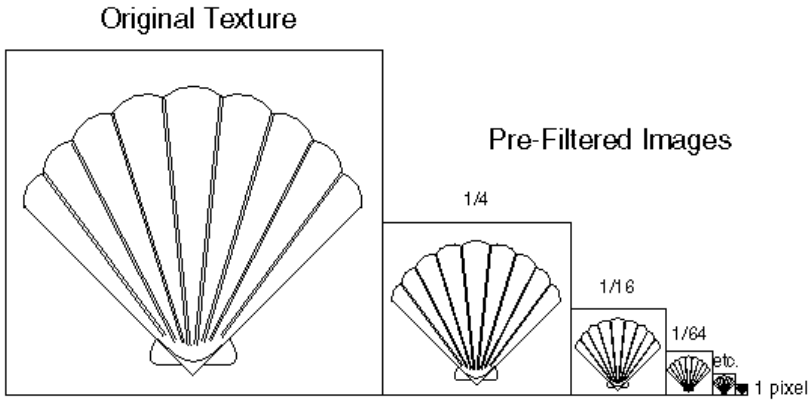


- Aktivierung

```
glBindTexture(GL_TEXTURE_2D,id);
glEnable(GL_TEXTURE_2D)
```

- Texturkoordinaten

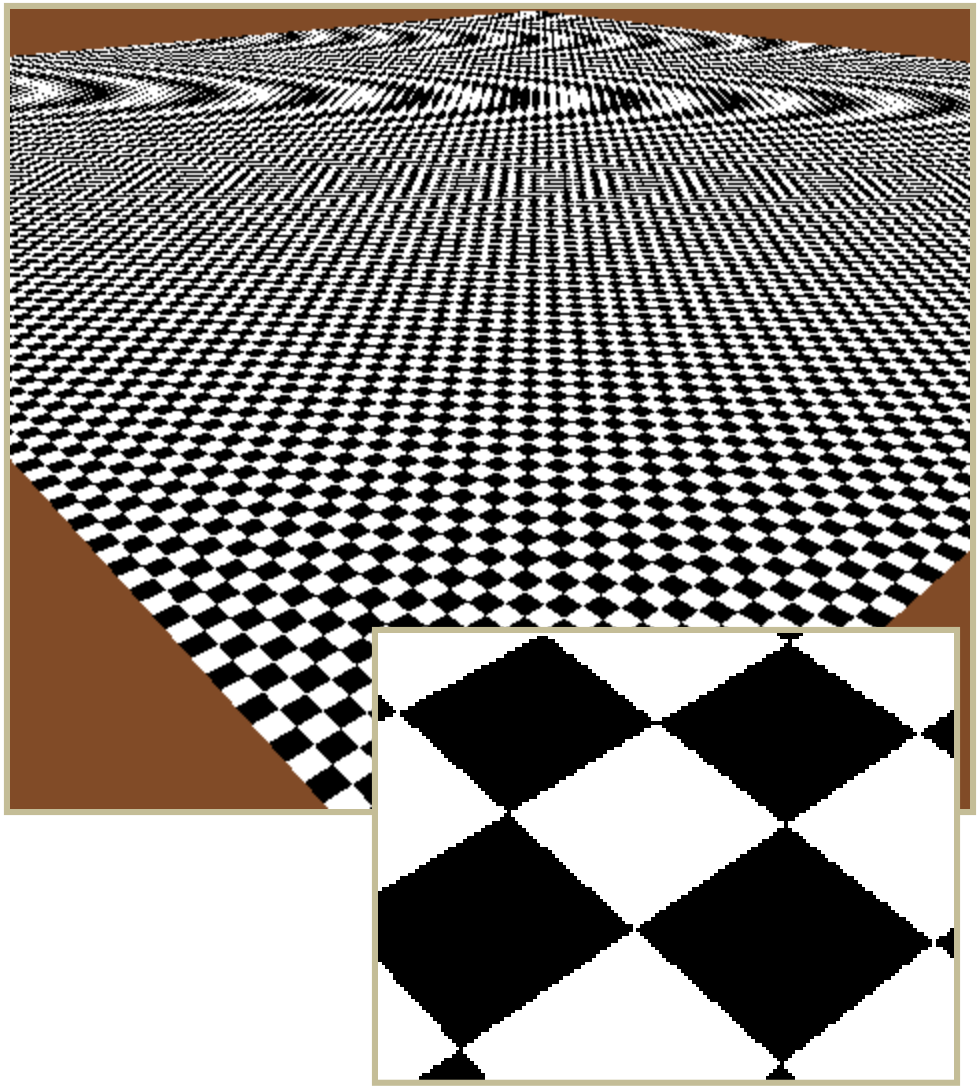
```
glNormal3fv(&nml)
glTexCoord3fv(&tc)
glColor3fv(&col)
glVertex3fv(&pos)
```



Mipmap-Hierarchie für Filterung

Texturierung
Anti-Aliasing

- **Minification:** perspective shortening can increase the spatial frequencies of projected textures significantly. Sampling the texture over the pixels leads to aliasing artefacts manifesting as ghost frequencies
- **Magnification:** zooming onto a textured surface can result in staircase artefacts
- These artefacts can be alleviated by appropriate filtering techniques

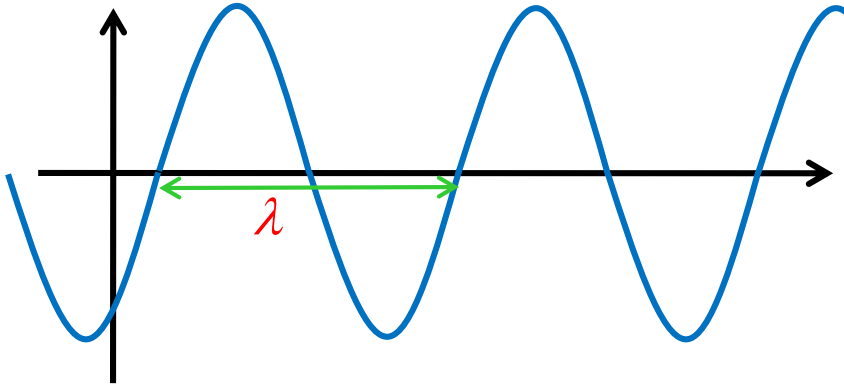


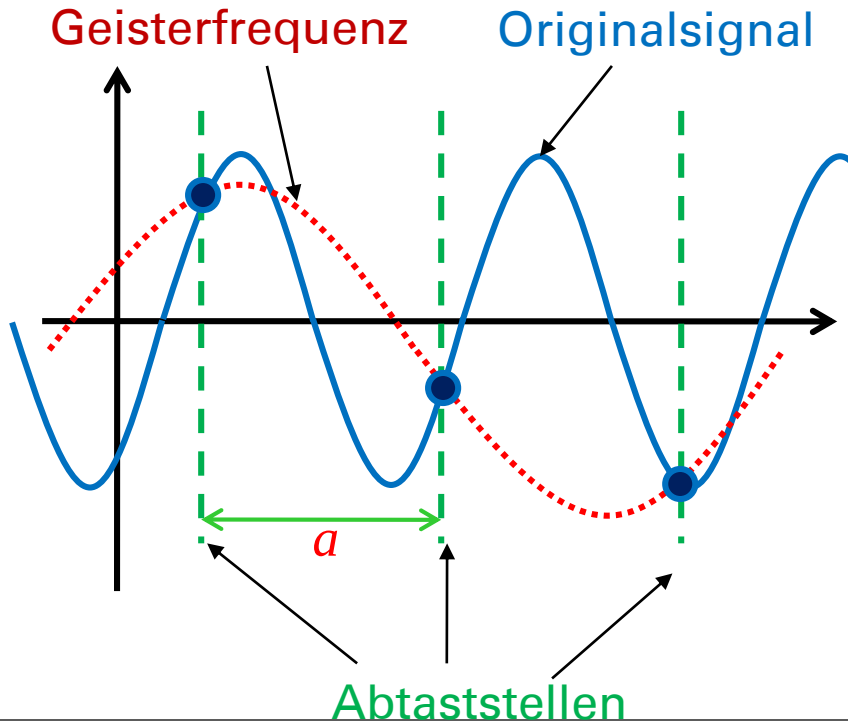
Texturierung
Abtastartefakte

- Aliasing im Zusammenhang mit Abtasten einer Funktion (Signal) ist aus der Signalverarbeitung bekannt.
- Jedes Signal (Funktion) kann als Überlagerung von Schwingungen unterschiedlicher Frequenzen interpretiert werden.
- Die Frequenz einer Schwingung ist eins durch die Wellenlänge λ
- Genauso spricht man beim Abtasten im Abstand von a von der Abtastfrequenz $f_a=1/a$
- Abtasttheorem: Tastet man ein Signal, das Schwingungen bis zu einer maximalen Frequenz f_{orig} enthält, mit einer Abtastfrequenz von

$$f_a > f_{\text{Nyquist}}=2f_{\text{orig}}$$

ab, so kann das Signal exakt (ohne Artefakte) rekonstruiert werden.





Texturierung Konfiguration

- Wirkung der Textur

```
glTexEnv(GL_TEXTURE_ENV,
   GL_TEXTURE_ENV_MODE,
   GL_MODULATE/DECAL/BLEND/REPLACE)
glTexEnv(GL_TEXTURE_ENV,
   GL_TEXTURE_ENV_COLOR, GLfloat*)
```

- Filterung

```
glTexParameterf(
   GL_TEXTURE_MAG_FILTER,
   GL_NEAREST/LINEAR);
glTexParameterf(
   GL_TEXTURE_MIN_FILTER,
   GL_NEAREST/LINEAR/
   NEAREST_MIPMAP_NEAREST/
   NEAREST_MIPMAP_LINEAR/
   LINEAR_MIPMAP_NEAREST/
   LINEAR_MIPMAP_LINEAR);
```

- Randbehandlung

```
glTexParameterf(
   GL_TEXTURE_WRAP_S/T,
   GL_CLAMP/REPEAT/CLAMP_TO_EDGE
```

Base internal format	GL_MODULATE	GL_DECAL	GL_BLEND	GL_REPLACE
GL_ALPHA	$C_v=C_f$ $A_v=A_fA_t$	undefined	$C_v=C_f$ $A_v=A_f$	$C_v=C_f$ $A_v=A_t$
GL_LUMINANCE 1	$C_v=L_tC_f$ $A_v=A_f$	undefined	$C_v=(1-L_t)C_f+L_tC_c$ $A_v=A_f$	$C_v=L_t$ $A_v=A_f$
GL_LUMINANCE_ALPHA 2	$C_v=L_tC_f$ $A_v=A_fA_t$	undefined	$C_v=(1-L_t)C_f+L_tC_c$ $A_v=A_fA_t$	$C_v=L_t$ $A_v=A_t$
GL_INTENSITY	$C_v=C_fI_t$ $A_v=A_fI_t$	undefined	$C_v=(1-I_t)C_f+I_tC_c$ $A_v=(1-I_t)A_f+I_tA_c$	$C_v=I_t$ $A_v=I_t$
GL_RGB 3	$C_v=C_tC_f$ $A_v=A_f$	$C_v=C_t$ $A_v=A_f$	$C_v=(1-C_t)C_f+C_tC_c$ $A_v=A_f$	$C_v=C_t$ $A_v=A_f$
GL_RGBA 4	$C_v=C_tC_f$ $A_v=A_fA_t$	$C_v=(1-A_t)C_f+A_tC_t$ $A_v=A_f$	$C_v=(1-C_t)C_f+C_tC_c$ $A_v=A_fA_t$	$C_v=C_t$