HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Introduction to Computer Graphics and Animation
Lecture 2 of 5

Prof. Dr. Dennis Allerkamp
December 3, 2024

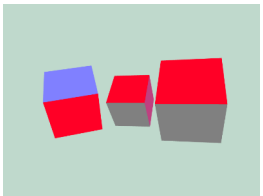
Summary

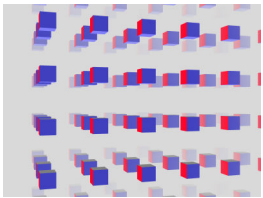
December 3, 2024

The following topics will be covered today:

- Coordinate systems in OpenGL
- Homogeneous coordinates
- Translation, rotation and scaling
- Camera transformation
- Perspective transformation
- Z-buffer algorithm
- Fog

After this day, participants will be able to render a scene with simple 3D objects.





Overview of Coordinate Systems

Euclidean Coordinate System

- OpenGL uses a *3-dimensional Euclidean coordinate system*
- Three *perpendicular axes* (x, y and z)
  - *x-axis* points to the right
  - *y-axis* points upwards
  - *z-axis* points towards the viewer
- It is a "*right-handed*" coordinate system

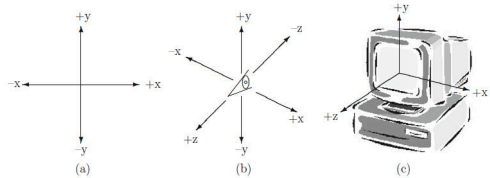
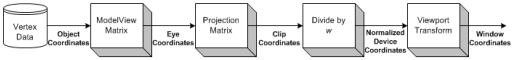


Bild 7.1: Die Definition des Koordinatensystems in OpenGL aus verschiedenen Perspektiven: (a) aus der Perspektive des Augenpunkts, (b) aus der Perspektive eines dritten Beobachters, der nach rechts oben versetzt ist und auf den Ursprung des Koordinatensystems blickt, in dem der Augenpunkt standardmäßig sitzt. (c) aus einer ähnlichen Perspektive wie in der Mitte, aber jetzt mit Blick auf den Bildschirm

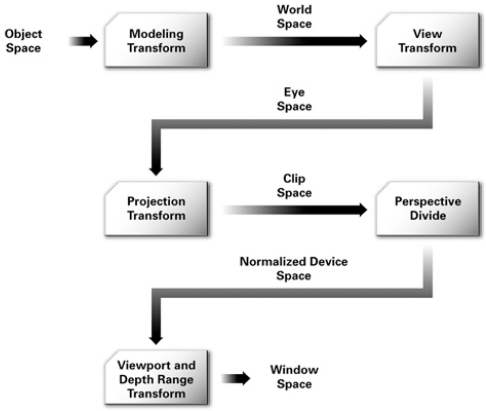
Computergrafik und Bildverarbeitung Nischwitz 2011 S.122

Overview of Coordinate Systems

- During the pipeline, OpenGL goes through several coordinate systems and transformations



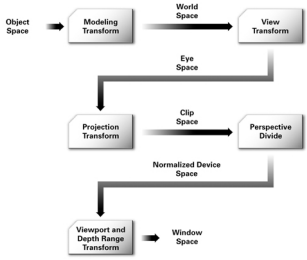
http://www.songho.ca/opengl/gl_transform.html



Computergrafik und Bildverarbeitung Nischwitz 2011 S.123

Quick Overview of Coordinate Systems

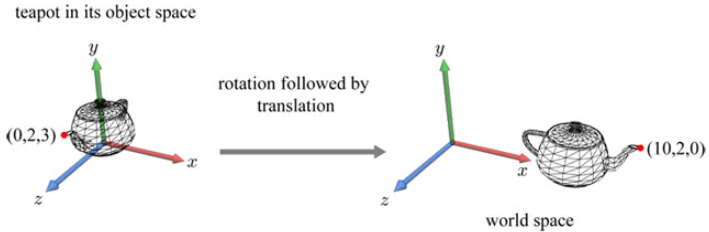
- Object space:
  - Coordinates in which the 3D objects are *locally defined*
- World space:
  - *Common* coordinate system of all objects after the model transformation
- Eye space:
  - Coordinate system after the view/camera transformation
  - contains location and direction of objects *relative to the camera*
- Clip space:
  - Coordinate system after the projection transformation
  - contains *perspective* of the camera
- Normalized device space:
  - Coordinates after the division (perspective divide) of the projection coordinates by w into a value range of -1..1
- Window space:
  - Coordinates that represent the scene after the viewport transformation in the selected *viewport window size*



Computergrafik und Bildverarbeitung Nischwitz 2011 S.123

Model Transformation

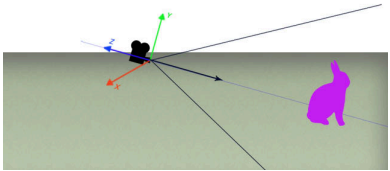
- 3D objects have an *own local coordinate system (Object Space)*
- They are *moved/translated*, *rotated* or *scaled* with model transformations.
- After the transformation, individual objects enter a common *World Space*



https://developer.tizen.org/development/guides/native-application/graphics/opengl-es/vertex-shader

Viewing Transformation

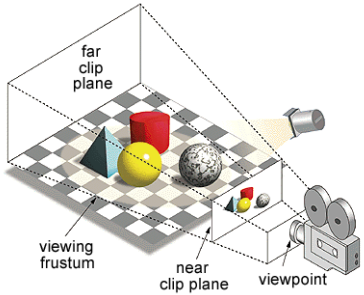
- The "*Camera*" of the scene
- Sets from *which point* the scene should be "photographed"
- For this, the matrix transforms all vertices of the scene into *Eye-/View-/Camera Coordinates*
- Camera at the origin (0,0,0), looking in -z direction
- It doesn't matter if the camera moves or the scene shifts



http://www.siliconjazz.net/computer-graphics/opengl/generic-3d-rendering-part3.html

Projection Transformation

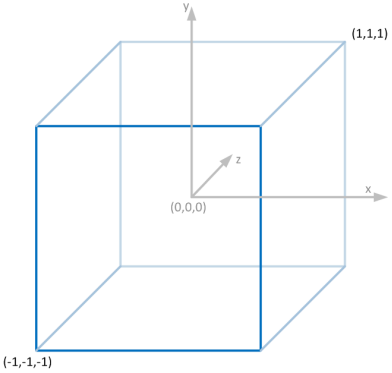
- Properties like *view angle*
  - Virtually corresponds to the selection of a *camera lens*
- Applies projection transformation to Eye coordinates
- Defines the visibility volume (*view frustrum*)
- Vertices outside the volume are removed (*clipping*)



http://encyclopedia2.thefreedictionary.com/View+frustum

Perspective Division and Normalization

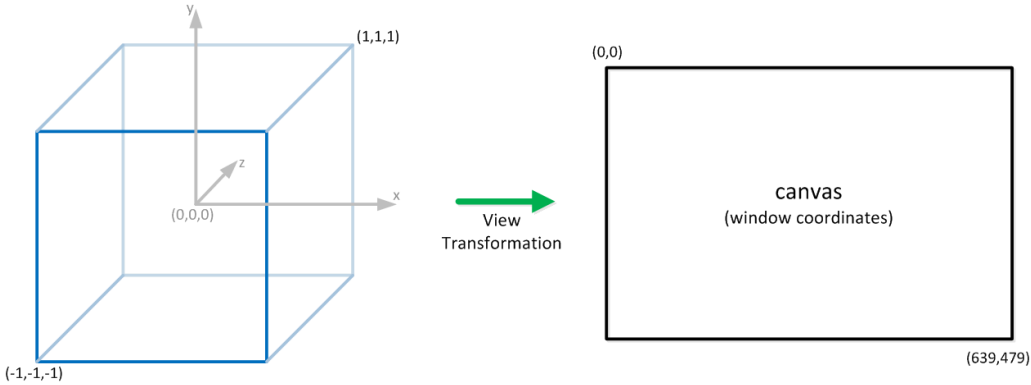
- Consists of *two steps*
  1. x, y and z coordinates are transformed to [-w, +w]
  2. x, y and z coordinates divided by w
- The steps generate *Normalized Device Coordinates* [-1.0,1.0]
  - NDC in a left-handed coordinate system



http://www.martinchristen.ch/webgl/tutorial02

Viewport Transformation

- Depending on the screen window defined in pixels
- Transforms *NDC to screen coordinates*



http://www.martinchristen.ch/webgl/tutorial02

Transformation Matrices

Mathematical Basics – Homogeneous Coordinates

- Point in *Euclidean space* can be defined by $(x, y, z)^T$
- Any transformation of points can be described by a 3x3 matrix
- Problem: Translations cannot be described by a 3x3 matrix

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{mit} \quad \begin{aligned} x' &= m_{11} \cdot x + m_{12} \cdot y + m_{13} \cdot z \\ y' &= m_{21} \cdot x + m_{22} \cdot y + m_{23} \cdot z \\ z' &= m_{31} \cdot x + m_{32} \cdot y + m_{33} \cdot z \end{aligned}$$

Computer Graphics and Image Processing Nischwitz 2011 S.124

Fourth Component w – Homogeneous Coordinates

- The *additive component* is missing for translations
- $\rightarrow$ Representation by four components x_h, y_h, z_h, w
- w is *inverse scaling factor*
- Mapping a position to the Euclidean space:

$$x = \frac{x_h}{w} \qquad y = \frac{y_h}{w} \qquad z = \frac{z_h}{w}$$

Fourth Component w

$$x = \frac{x_h}{w} \qquad y = \frac{y_h}{w} \qquad z = \frac{z_h}{w}$$

- $w = 1$ is *standard*
- At $w = 0.5$, the coordinates are stretched by a factor of 2
  - (2, 4, -3, 1) and (1, 2, -1.5, 0.5) describe the same Euclidean location (2, 4, -3)
- Division by $w = 0$ *is not defined*
  - $=>$ $w = 0$ maps to infinity
- (x, y, z, 0) are therefore considered as *direction vectors*
  - e.g., important in light calculation (parallel light rays, sunlight)

General Transformation Matrices

- Transformation of a location vector $v = (x, y, z, w)$
- Achieved by a 4x4 matrix

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

Computer Graphics and Image Processing Nischwitz 2011 S.125

Optimization – Combining Matrices

- Each vertex v of a scene must go through all transformation stages,

$$v' = (V \cdot N \cdot P \cdot (S \cdot R \cdot T)*) \cdot v$$

  - Model, View up to projection transformation (P),
  - Normalization (N)
  - and viewport transformation (V)
- $(S \cdot R \cdot T)*$ is an *arbitrary combination* of scalings, rotations, and translations
  - (Order of affine transformations is exemplary and varies depending on the use case)

Optimization – Combining Matrices

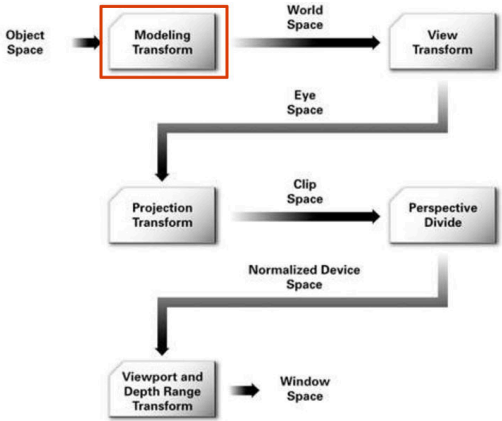
- Efficiency is achieved by *combining the transformation matrices* into a single matrix

$$v' = (V(N(P(S(R(T \cdot v)))))) = (V \cdot N \cdot P \cdot S \cdot R \cdot T) \cdot v$$

- With millions of vertices of an object, *n vertices* are therefore multiplied by only *one transformation matrix*.
- In an *object with a hierarchy* (e.g. a car with wheels), *each node* needs its own transformation matrix.

Model Transformations

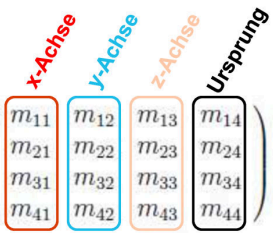
Model Transformations



Model Transformations with Matrices

- Matrix transforms *object/vector* into another *coordinate system*

$$\mathbf{v}' = \mathbf{M}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$



- Possible operations/ affine transformations:
  - Translation,
  - Rotation,
  - Scaling
  - affine transformations = collinearity and parallelism are preserved
- *Positions, orientates and scales* the object *in the scene*

Model Transformations – Translation

- Shifts the object's coordinate system
- As a result, the object is shifted

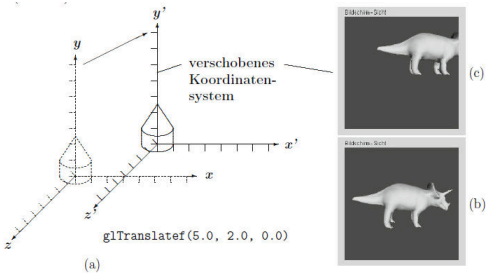


Bild 7.3: Translation des Koordinatensystems in OpenGL: (a) gestrichelt: nicht verschobenes Koordinatensystem (x, y, z), durchgezogen: verschobenes Koordinatensystem (x', y', z'). 3D-Objekt im Ursprung: Zylinder mit aufgestülptem Kegelmantel. (b) Bildschirm-Sicht eines nicht verschobenen Objekts (Triceratops). (c) Bildschirm-Sicht des mit `glTranslatef(5.0, 2.0, 0.0)` verschobenen Objekts.

Computer Graphics and Image Processing Nischwitz 2011 S.128

Model Transformations – Translation

- *Translation* of a vertex $v = (x, y, z)$ by a direction vector (T_x, T_y, T_z) in Cartesian coordinates

$$x' = x + T_x \qquad y' = y + T_y \qquad z' = z + T_z$$

- In Homogeneous Coordinates with a 4x4 Matrix

$$\mathbf{v}' = \mathbf{M}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

x-Achse y-Achse z-Achse Ursprung

Model Transformations – Rotation

- Rotates/turns the coordinate system *around an axis*
- Runs *counterclockwise*
- α in degrees

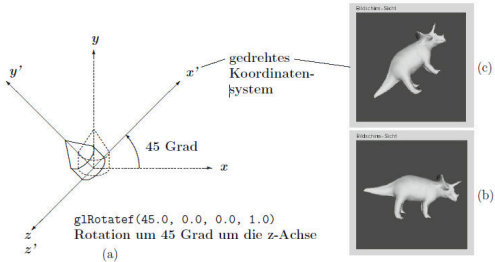


Bild 7.4: Rotation des Koordinatensystems in OpenGL: (a) gestrichelt: nicht gedrehtes Koordinatensystem (x, y, z); durchgezogen: um 45° bzgl. der z–Achse gedrehtes Koordinatensystem (x', y', z'). (b) Bildschirm-Sicht eines nicht gedrehten Objekts (Triceratops). (c) Bildschirm-Sicht des mit `glRotatef(45.0, 0.0, 0.0, 1.0)` gedrehten Objekts.

Computer Graphics and Image Processing Nischwitz 2011 S.130

Model Transformations – Rotation in 3D

- Rotation around the *respective Euclidean axes* by respective rotation matrices:

Rotation um die x–Achse:

$$\mathbf{R^x} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.9}$$

Rotation um die y–Achse:

$$\mathbf{R^y} = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.10}$$

Rotation um die z–Achse:

$$\mathbf{R^z} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.11}$$

Computer Graphics and Image Processing Nischwitz 2011 S.131

Model Transformations – Rotation in 3D

- Concatenation in a common matrix:
  - Rotation around axis (R_x, R_y, R_z) with angle Θ

$$\begin{bmatrix} \cos\theta + {R_x}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\ R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + {R_y}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\ R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + {R_z}^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

https://learnopengl.com/Getting-started/Transformations

Model Transformations – Scaling

- Separate scaling is ensured by *three factors*

$$x' = S_x \cdot x \qquad y' = S_y \cdot y \qquad z' = S_z \cdot z$$

- General scaling in Homogeneous Coordinates:

$$\mathbf{v}' = \mathbf{S}\mathbf{v} \qquad \Leftrightarrow \qquad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Computer Graphics and Image Processing Nischwitz 2011 S.133

Model Transformations – Scaling

- Stretches the coordinate system in the respective axis with s>1
- Or shrinks if s<1
- Negative values *cause mirroring*
- s=0 is not defined

Skalierungsfaktor (s)	Effekt
$\|s\| > 1.0$	Streckung / Dimensionen vergrößern
$\|s\| = 1.0$	Dimensionen unverändert
$0.0 < \|s\| < 1.0$	Stauchung / Dimensionen verkleinern
$s = 0.0$	unzulässiger Wert

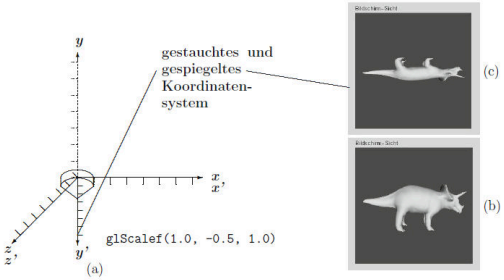


Bild 7.5: Skalierung des Koordinatensystems in OpenGL: (a) gestrichelt: original Koordinatensystem (x, y, z); durchgezogen: bzgl. der y–Achse um den Faktor $|s| = 0.5$ gestauchtes und gespiegeltes Koordinatensystem (x', y', z'). (b) Bildschirm-Sicht eines nicht skalierten Objekts (Triceratops). (c) Bildschirm-Sicht des mit `glScalef(1.0, -0.5, 1.0)` skalierten Objekts.

Order of Transformations – World Coordinates

- Final position heavily depends on the *order of transformations*
- Transformations are *not independent of each other*

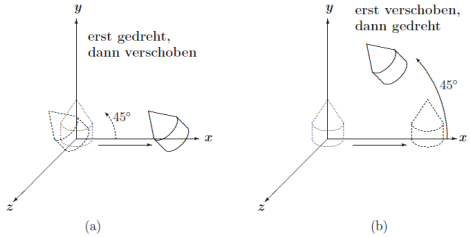


Bild 7.6: Die Reihenfolge der Transformationen in der Denkweise eines festen Weltkoordinatensystems: (a) gepunktet: original Objekt, gestrichelt: erst gedreht bzgl. der z–Achse, durchgezogen: dann verschoben bzgl. der x–Achse. (b) gepunktet: original Objekt, gestrichelt: erst verschoben bzgl. der x–Achse, durchgezogen: dann gedreht bzgl. der z–Achse.

Computer Graphics and Image Processing Nischwitz 2011 S.134

Order of Transformations

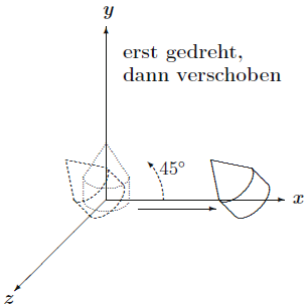
- Concatenation of transformations is defined by *multiplication of the matrices*
- Newly added matrices are multiplied *from the left*: $v' = M_3 \cdot M_2 \cdot M_1 \cdot v$
- Matrix multiplications are *not commutative*
  - *The order of the matrices is decisive*
- Matrix multiplications are *associative*
  - $(M_3 \cdot M_2) \cdot M_1 \cdot v = M_3 \cdot (M_2 \cdot M_1) \cdot v$ (*always multiplied from the right*)

Example: First rotation, then translation

- Initialized with a unit matrix: $M = I$
- Then, the translation matrix is multiplied from the right on the unit matrix: $M = M \cdot T$
- Afterwards, the rotation matrix is multiplied from the right on the transformation matrix: $M = M \cdot R$
- Subsequently, the vertices are multiplied by the total matrix from the right: $Mv = ITRv$
- So first, it is rotated, and afterwards translated: $T(Rv)$

Example with Matrices in OpenGL

- Generate matrices using the GLM Library
- Same example:
  - First rotation, then translation
- Here too: Matrices are multiplied from the right i.e., $I \cdot T \cdot R$

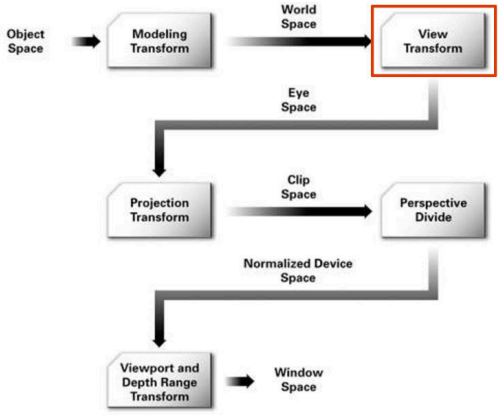


- GLM takes the to-be-multiplied matrix as the first parameter

```
glm::mat4 Model = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f, 0.0f, 0.0f));
Model = glm::rotate(Model, 40.0f, glm::vec3(0.0f, 1.0f, 0.0f));
// Model = Translation * Rotation
```

Viewing Transformation

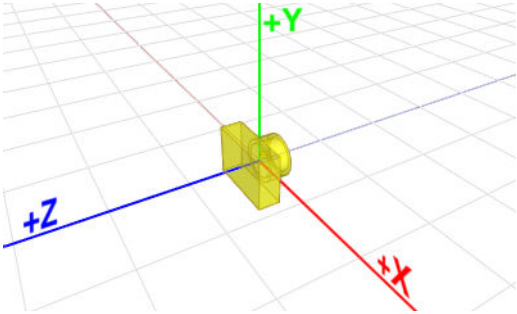
Viewing Transformation



Viewing Transformation

Viewing Transformation

- Changes the *position and viewing direction* of the camera
- Identity matrix represents the camera *at the origin*
- *Viewing direction is towards the negative z-axis* and y-axis points upwards



http://www.songho.ca/opengl/gl_camera.html

Viewing Transformation

- The view matrix *describes the camera*
  - Position,
  - Viewing direction
  - Up vector
- Defines together with the model matrix the *modelview matrix* (modelview = view*model)
- Auxiliary functions for the generation of the matrix for this are usually called *lookAt()*

Viewing Transformation - `glm::lookAt()`

- `glm::lookAt( eye, center, up );`
  - *Eye* = Position of the camera
  - *Center* = Position the camera is looking at (sometimes called *Look*)
  - *Up* = Direction pointing upwards

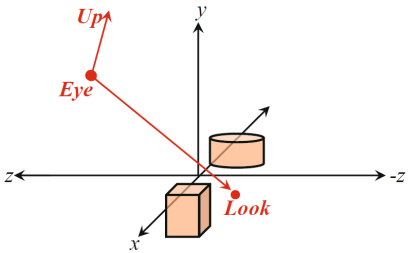


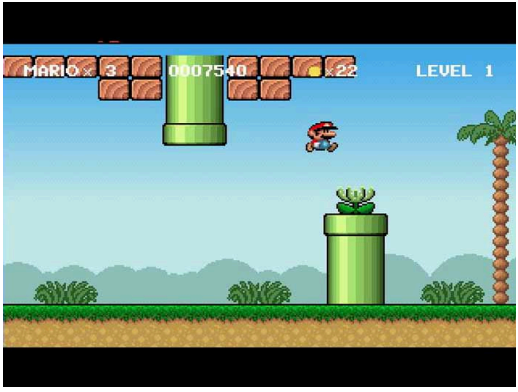
Bild 7.8: Das x, y, z-Koordinatensystem vor der Augenpunkttransformation `gluLookAt()`: Der Augenpunkt befindet sich am Ort **Eye** und der Blick geht von dort in Richtung des Punkts **Look**, in dessen Nähe sich die darzustellenden Objekte der Szene befinden. Die vertikale Ausrichtung des Augenpunkts (bzw. der Kamera) wird durch den Vektor **Up** angegeben.

Computer graphics and image processing Nischwitz 2011 S.136

Viewing Transformation – `glm::lookAt()`

- The `glm::lookAt()` matrix transforms the entire scene,
  - so that the *camera remains at the origin*
- Technically, the camera is not moved
  - but the *objects in the scene*
- Optically *indistinguishable*

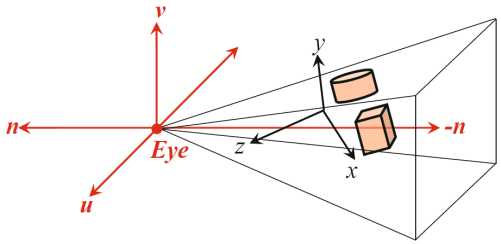
The world moves to the left and *not* the camera to the right.



https://mozzastryl.wordpress.com/2013/01/20/types-of-game-perspectives/

Viewing Transformation

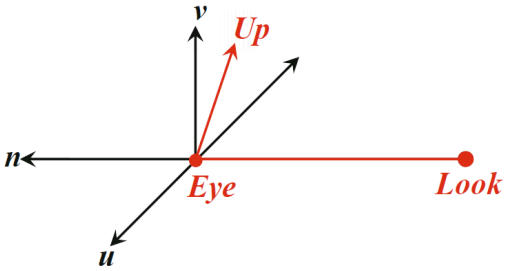
- Axes of the camera are denoted by u, v, n
- Origin of the coordinate system is at location *Eye*
- Transformation consists of 2 steps:
  1. Rotation in space, so that the viewing axis n is rotated into z and the vector v is parallel to y
  2. Shifting the Eye into the origin (0,0,0)
- As a result, the scene transforms automatically because the view matrix is multiplied with the model matrix



Computer graphics and image processing Nischwitz 2011 S.137

Viewing Transformation

- n, u and v can be obtained by simple operations
  $n = \text{Eye} - \text{Look}$
  $u = \text{Up} \times n$
  $v = n \times u$



- Finally, *normalization of the vectors*

$$\mathbf{n}' = \frac{\mathbf{n}}{|\mathbf{n}|} = \frac{1}{\sqrt{\mathbf{n_x^2} + \mathbf{n_y^2} + \mathbf{n_z^2}}} \cdot \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

$$\mathbf{u}' = \frac{\mathbf{u}}{|\mathbf{u}|} = \frac{1}{\sqrt{\mathbf{u_x^2} + \mathbf{u_y^2} + \mathbf{u_z^2}}} \cdot \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}$$

$$\mathbf{v}' = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{\sqrt{\mathbf{v_x^2} + \mathbf{v_y^2} + \mathbf{v_z^2}}} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

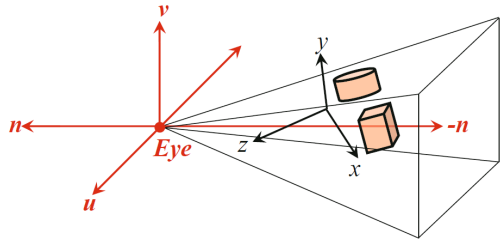
Viewing Transformation - Rotation Matrix

- The desired *rotation matrix* is as follows:
  (The rows define the axes)

$$\mathbf{M_R} = \begin{pmatrix} u'^T \\ v'^T \\ n'^T \end{pmatrix} = \begin{pmatrix} u'_x & u'_y & u'_z \\ v'_x & v'_y & v'_z \\ n'_x & n'_y & n'_z \end{pmatrix}$$

- With *Up = (0,1,0)* this applies:

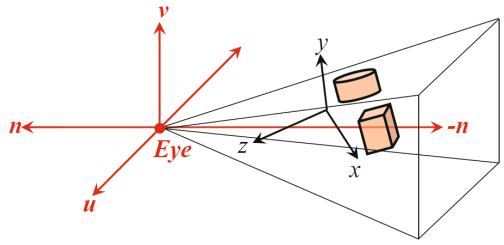
$$\begin{aligned} \mathbf{u}' &= (n'_z, 0, -n'_x) \\ \mathbf{v}' &= (-n'_x \cdot n'_y, n'^2_x + n'^2_y, -n'_y \cdot n'_z) \end{aligned}$$



Computer graphics and image processing Nischwitz 2011 S.137

Viewing Transformation – Shifting to the Origin

- Translation to origin with -eye *not possible*, because the coordinate system has been rotated
- *Solution*: Set up equation with Eye coordinates and set result equal to 0



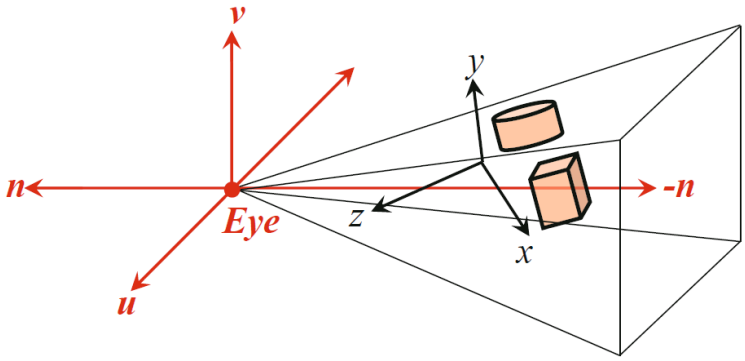
Computer graphics and image processing Nischwitz 2011 S.137

$$\mathbf{M}_{\mathrm{RT}} \cdot \begin{pmatrix} Eye_x \\ Eye_y \\ Eye_z \\ 1 \end{pmatrix} = \begin{pmatrix} u'^T & t_x \\ v'^T & t_y \\ n'^T & t_z \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Eye_x \\ Eye_y \\ Eye_z \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} \mathbf{u}' \cdot \mathbf{Eye} + t_x = 0 \\ \mathbf{v}' \cdot \mathbf{Eye} + t_y = 0 \\ \mathbf{n}' \cdot \mathbf{Eye} + t_z = 0 \end{pmatrix}$$

$$= \begin{pmatrix} u'_x & u'_y & u'_z & t_x \\ v'_x & v'_y & v'_z & t_y \\ n'_x & n'_y & n'_z & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} Eye_x \\ Eye_y \\ Eye_z \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} t_x = -\mathbf{u}' \cdot \mathbf{Eye} \\ t_y = -\mathbf{v}' \cdot \mathbf{Eye} \\ t_z = -\mathbf{n}' \cdot \mathbf{Eye} \end{pmatrix}$$

Viewing Transformation – Hints

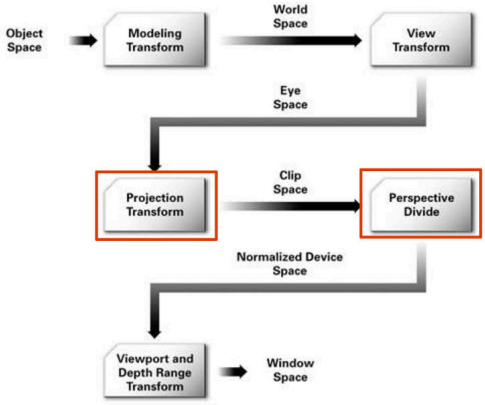
- Eye and Look *must not be identical*, because n = Eye – Look disappears
- The Up-vector must not be parallel to the direction vector n, because Up × n = u is 0



Computer graphics and image processing Nischwitz 2011 S.137

Projection Transformation

Projection Transformation



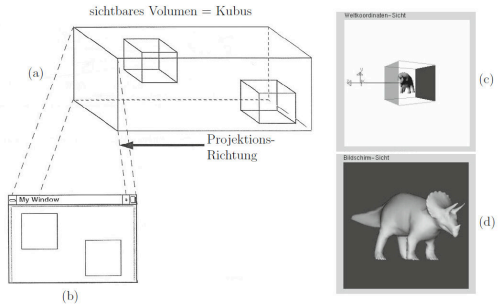
Projection Transformation

Projection Transformation

- Model transformations and Viewing Transformation are summarized:
  - *ModelView Matrix*
  - After the Modelview transformation, all objects are already at the desired position in 3D space.
  - Seen from the eye point / the camera.
  - Viewing Transformation = position and orientation of the camera.
- *Goal of projection transformation*: Projection of the 3D scene onto a 2D surface (x-y plane).
  - However, the z-values are kept (for covering calculations in later chapter).
  - Projection transformation = lens of the camera.
- Many projections possible, two relevant in practice.
  - Orthographic projection.
  - Perspective projection.

Projection Transformation: Orthographic Projection

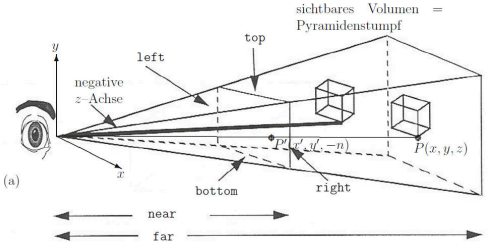
- Maps onto the projection surface via *parallel rays*.
- View frustrum is a *cube*.
  - All vertices outside the cube are clipped off (*clipping*).
- Angles and size of all objects *remain*.
  - Regardless of distance / depth value.
- Popular in CAD.
  - Side view, front view and top view.
  - Optimal for technical drawings, because *dimensions remain*.

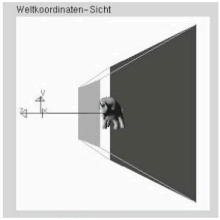


Computer graphics and image processing Nischwitz 2011 S.143

Projection Transformation: Perspective Projection

- *Converging rays* that converge in the Eye point.
- Objects that are closer to the Eye point appear *larger* than far objects.





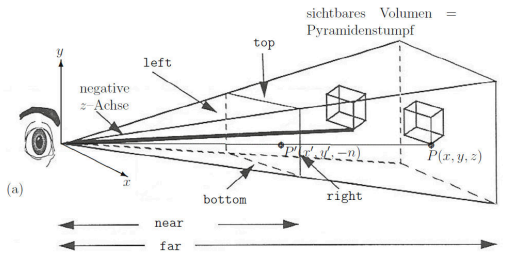
(b)

(c)

Computer graphics and image processing Nischwitz 2011 S.144

Projection Transformation: Perspective Projection

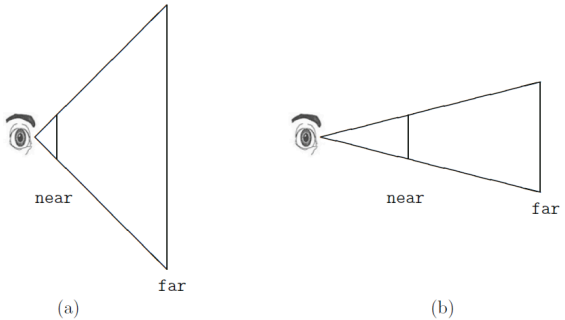
- Central perspective = approximates our natural perception
  - Light falls through the lens onto the retina of the eyes.
- Most common *use in 3D computer graphics*.
- View frustrum is a *pyramid trunk*.
  - All vertices outside the truncated pyramid are clipped off (*clipping*).



Computer graphics and image processing Nischwitz 2011 S.144

Projection Transformation: Perspective Projection

- Has six limiting planes (*clipping-planes*).
- The bottom of the truncated pyramid is the *far clipping plane*.
- At the tip near the eye lies the *near clipping plane*.



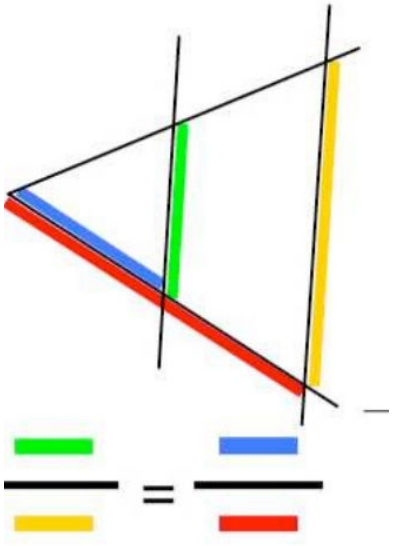
Computer graphics and image processing Nischwitz 2011 S.146

Projection Transformation: Perspective Projection

- Derivation by the intercept theorem.
  - Near clipping plane is moved from eye point along the negative z-axis by *n=near* (analogously *f = far*).
  - Both planes are *parallel to the x-y plane*.
- The point $P = (x, y, z)$ is mapped to the point $P' = (x', y', -n)$ as follows:

$$\frac{x'}{-n} = \frac{x}{z} \quad \Leftrightarrow \quad x' = -\frac{n}{z} \cdot x$$

$$\frac{y'}{-n} = \frac{y}{z} \quad \Leftrightarrow \quad y' = -\frac{n}{z} \cdot y$$



Computer graphics and image processing Nischwitz 2011 S.146

Perspective Projection - Mapping the Points

- The transformation matrix P *fulfills the theorem of intersecting chords equations*

$$\mathbf{v}' = \mathbf{P}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1+\frac{f}{n} & f \\ 0 & 0 & -\frac{1}{n} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \\ y \\ \left(1+\frac{f}{n}\right) z + fw \\ -\frac{z}{n} \end{pmatrix}$$

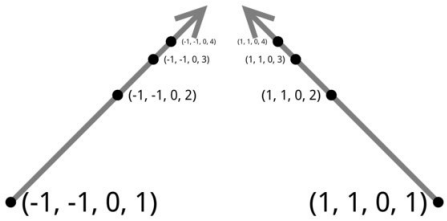
- The reason for this is the *subsequent division with the w component*
  (Conversion of homogeneous to Euclidean coordinates)

$$\begin{pmatrix} x'_E \\ y'_E \\ z'_E \end{pmatrix} = \begin{pmatrix} -\frac{n}{z} \cdot x \\ -\frac{n}{z} \cdot y \\ -\frac{n}{z} \cdot fw - (f+n) \end{pmatrix}$$

Perspective Projection – w and Perspective

- The projection matrix itself does not make objects appear smaller that are further away
- Only the later *perspective division* makes objects appear smaller that are further away.
- Reason: w is dependent on z (x, y approach (0,0) when z is large)
- Visually speaking, one can think of w as the distance of the camera to the object

$$\begin{pmatrix} x'_E \\ y'_E \\ z'_E \end{pmatrix} = \begin{pmatrix} -\frac{n}{z} \cdot x \\ -\frac{n}{z} \cdot y \\ -\frac{n}{z} \cdot fw - (f+n) \end{pmatrix}$$



http://www.learnopengles.com/understanding-opengls-matrices/dividebyw/

Normalization to Normalized Device Coordinates (NDC)

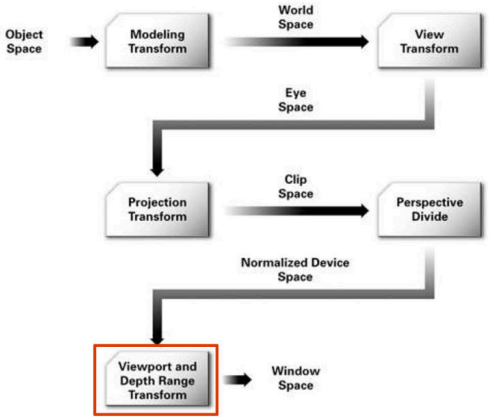
- All vertices in the *view frustrum* now lie on the *near clipping plane*
- *Arbitrarily large* near clipping planes must be brought to a fixed number of pixels.
- Therefore, they need to be *normalized*, i.e., the Euclidean coordinates must lie *between -1 and 1* (Normalized Device Coordinates)
- In OpenGL, the normalization matrix is merged with the projection matrix to achieve the value range [-1,1] *in one step*

$$\mathbf{M} = \mathbf{N} \cdot \mathbf{P} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1+\frac{f}{n} & f \\ 0 & 0 & -\frac{1}{n} & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2}{r-l} & 0 & \frac{1}{n}\frac{r+l}{r-l} & 0 \\ 0 & \frac{2}{t-b} & \frac{1}{n}\frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{1}{n}\frac{f+n}{f-n} & -\frac{2f}{f-n} \\ 0 & 0 & -\frac{1}{n} & 0 \end{pmatrix}$$

Viewport Transformation

Viewport Transformation



Viewport Transformation

- X-Y plane *defined by pixels*
- Conversion of the NDC into a chosen viewport size

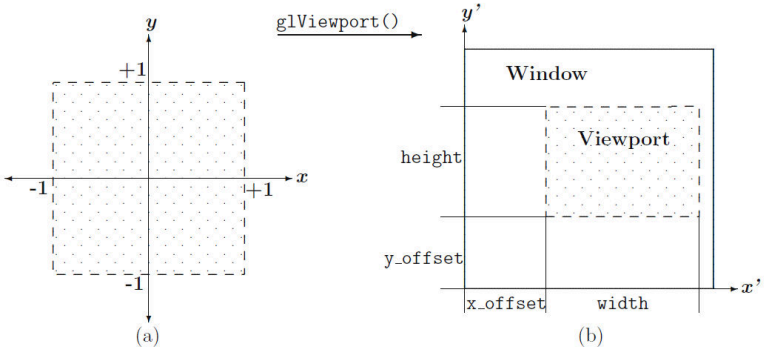


Bild 7.14: Abbildung des sichtbaren Volumens in einen *Viewport*: (a) Wertebereich der normierten Koordinaten des sichtbaren Volumens. (b) Wertebereich der Bildschirm–Koordinaten des Viewports nach der Viewport-Transformation. Der Ursprung der Bildschirm–Koordinaten liegt in der linken unteren Ecke des Windows.

Computer Graphics and Image Processing Nischwitz 2011 S.146

Viewport Transformation

- Mapping to the window is defined as follows:
  - Set X to the value range [-width/2, width/2]
  - Set Y to the value range [-height/2, height/2]
  - Move the viewport by half the window width/height
  - Further movable with *offset*

$$x' = \frac{width}{2} \cdot x + \left(x\_offset + \frac{width}{2} \right)$$

$$y' = \frac{height}{2} \cdot y + \left(y\_offset + \frac{height}{2} \right)$$

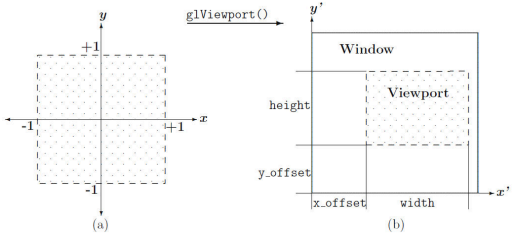


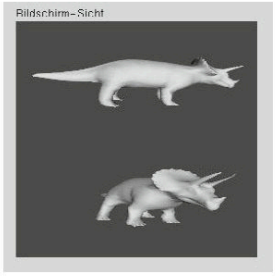
Bild 7.14: Abbildung des sichtbaren Volumens in einen *Viewport*: (a) Wertebereich der normierten Koordinaten des sichtbaren Volumens. (b) Wertebereich der Bildschirm–Koordinaten des Viewports nach der Viewport-Transformation. Der Ursprung der Bildschirm–Koordinaten liegt in der linken unteren Ecke des Windows.

Computer Graphics and Image Processing Nischwitz 2011 S.146

Viewport Transformation

- In OpenGL: `glViewport(xoffset, yoffset, width, height)`

`glViewport(0,0,256,256)`
(a)



`glViewport(0,128,256,128)`
`glViewport(0,0,256,128)`
(b)

Computer Graphics and Image Processing Nischwitz 2011 S.151

Occlusion

Occlusion

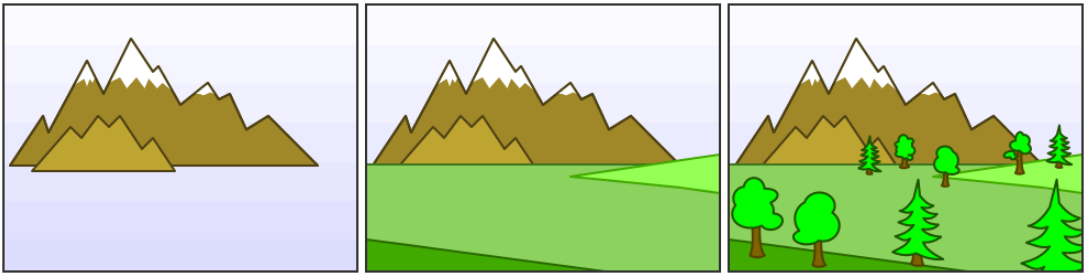
Rene Magritte, The Blank Cheque, 1965

Occlusion

- Occlusion of objects in the background by objects in the foreground
- Important aspect in *spatial perception*
- Reliable *indication of the distance of objects* from the camera
- Example: Object A, which is further away than Object B, can never occlude Object B
- Without an algorithm, all objects are drawn in the order of render calls

Painter Algorithm

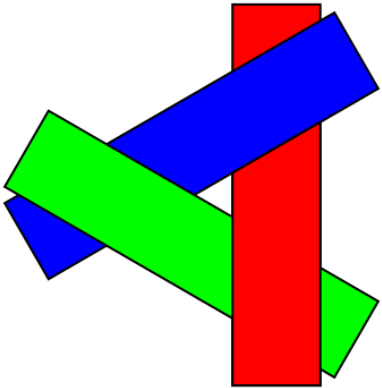
- Approach from painting
  - Step 1: Sort all objects *in relation to their distance to the camera*
  - Step 2: Draw all objects, *starting with the most distant*



https://en.wikipedia.org/wiki/Painter%27s_algorithm

Painter Algorithm

- Approach from painting
  - Step 1: Sort all objects *in relation to their distance to the camera*
  - Step 2: Draw all objects, *starting with the most distant*
- Has *not* established itself in 3D computer graphics
- Disadvantages:
  - The computational effort increases nonlinearly with the number of objects
  - Fails when objects *overlap or penetrate each other*
  - Circular references possible (A>B>C>A)
- Exceptions prove the rule



https://en.wikipedia.org/wiki/Painter%27s_algorithm

Occlusion: Failure of the Painter Algorithm

- Reference point: Majority of vertices? Middle vertex? Foremost vertex?

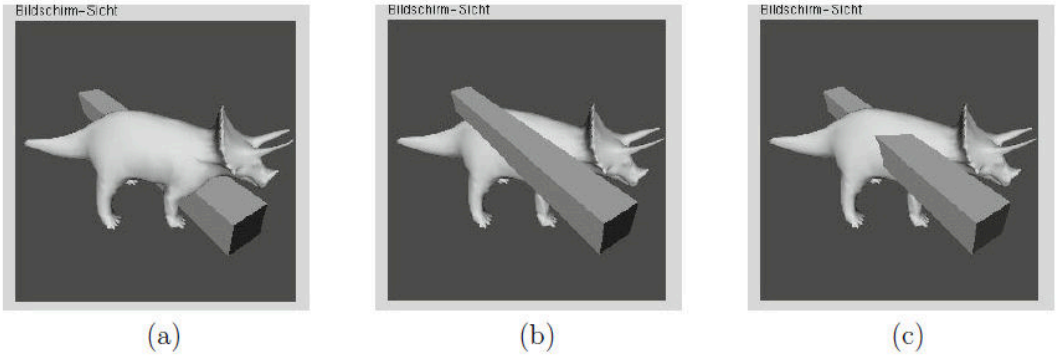
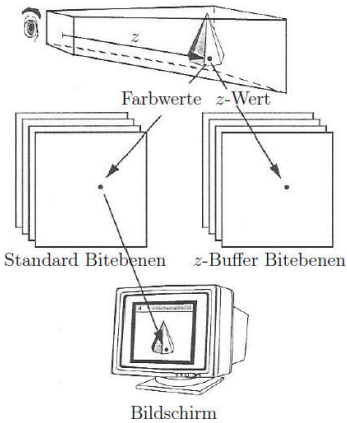


Bild 8.1: Probleme des Maler–Algorithmus' bei sich durchdringenden Objekten: (a) der Triceratops wird zuletzt gezeichnet und überdeckt daher eigentlich sichtbare Teile des Quaders. (b) der Quader wird zuletzt gezeichnet und überdeckt daher eigentlich sichtbare Teile des Triceratops. (c) korrekte Darstellung mit Hilfe des z-Buffer Algorithmus.

Computergrafik und Bildverarbeitung Nischwitz 2011 S.158

Z-Buffer Algorithm

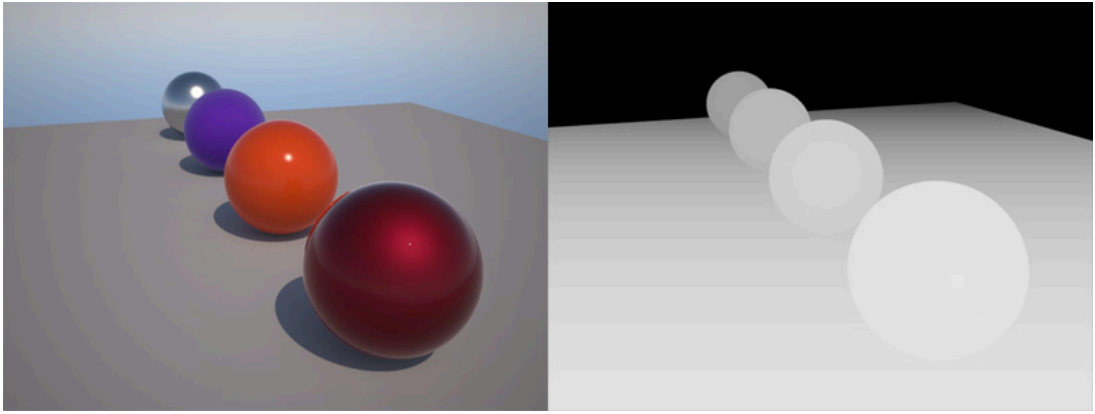
- Algorithm developed by Straßer/Catmull (1974)
- Also Depth Buffer
- Today implemented *in hardware*
- Basic idea: Store a z-value (depth information) for each *pixel*
- Objects are only drawn,
  - when the depth value shows a smaller z-value than previously stored z-values from other objects



Computer Graphics and Image Processing Nischwitz 2011 S.160

Z-Buffer Algorithm

- Visualization of the Z-Buffer as Z-(Buffer)-Channel in *grayscale*



http://support.nextlimit.com/display/maxwelldocs/Z-buffer+channel

Z-Buffer Algorithm - Procedure

Initialize Z-Buffer to the maximum value (1.0) for each pixel of the Window

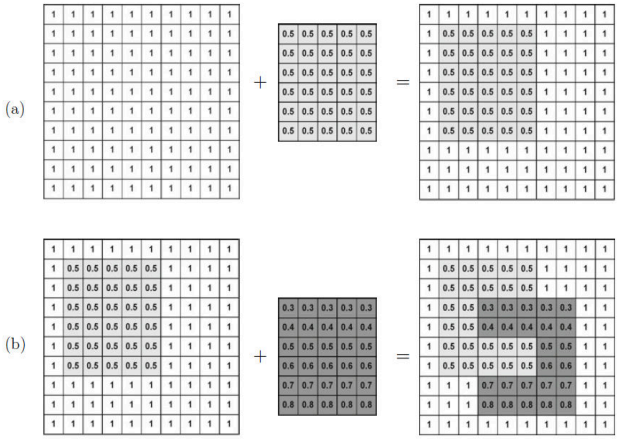
For all Objects (Polygons), that need to be drawn:

For all Pixels of an Object, that need to be drawn:

- Calculate the distance from the eye point to the object for the pixel = Z-Value
- Compare the calculated distance with the saved value in the Z-Buffer
- If (Distance $<$ saved value): *// The object is closer*
  - Enter new color value into the framebuffer and
  - Enter new Z-Value in Z-Buffer for this Pixel
- Otherwise, change nothing *// Object obscured*

Z-Buffer Algorithm – Example (Graphics)

- 10x10 pixel image
  - Number in each pixel is the z-Value
  - Gray scale is the color
- '(a)
  - Addition of polygon with z-value 0.5
  - Overwrites image storage, as $0.5 < 1$
- '(b)
  - Addition of an inclined polygon
  - it cuts the polygon from (a)



Computergrafik und Bildverarbeitung Nischwitz 2011 S.161

Z-Buffer – Implementation - Prerequisite

- `glEnable(GL_DEPTH_TEST)` activates the z-Buffer
  - disabled by default
- `glDisable(GL_DEPTH_TEST)` deactivates the z-Buffer
- `glDepthMask(GL_DISABLE)` deactivates writing in the z-Buffer
- `glDepthMask(GL_ENABLE)` activates writing in the z-Buffer

Z-Buffer – Implementation - Initialization

- Initialization of the image memory is set to 1.0 by default and can be changed with `glClearDepth(GLdouble depth)`
- Depth value ranges from 0.0 (near clipping plane) to 1.0 (far clipping plane)
- Rarely should the range be changed `glDepthRange(near,far)`
- For each new image, the depth buffer must be cleared.
- The `glClearDepth` specified value is transferred to the image memory with `glClear(GL_DEPTH_BUFFER_BIT)`
  - Optimally together with color buffer: `glClear(GL COLOR BUFFER BIT | GL DEPTH BUFFER BIT)`

Z-Buffer – Implementation - Comparison

- Set a *comparison operator* (between new and old z-Value)
- Default set to *GL_LESS* (i.e. ´<´)
- Other comparison operators possible with command `glDepthFunc(GLenum operator)`

operator	Funktion
GL_LESS	$<$, kleiner (Standardwert)
GL_NEVER	0, liefert immer den Wahrheitswert „FALSE“
GL_EQUAL	$=$, gleich
GL_LEQUAL	$\leq$, kleiner gleich
GL_GREATER	$>$, größer
GL_GEQUAL	$\geq$, größer gleich
GL_NOTEQUAL	$\neq$, ungleich
GL_ALWAYS	1, liefert immer den Wahrheitswert „TRUE“

Computergrafik und Bildverarbeitung Nischwitz 2011 S.163