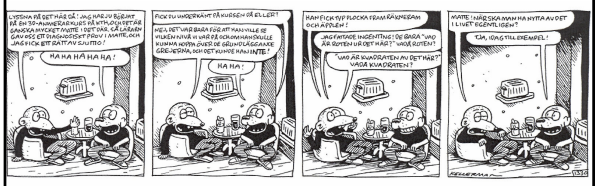


Vectors and Transforms

In

3D Graphics

Reading instructions

MUST

- Read the following slides and
- OH 27-44 by Magnus Bondesson
- Article ”Från Värld till Skärm” by Magnus Bondesson.
  - Except section 10 which is postponed until lecture on Texturing
  - Article can be downloaded from course’s web page
- ”Intoduktion till OpenGL” by Magnus Bondesson
  - Read section 1-9.

You may also read:

- Angel
  - chapter 4, pages 157-228

or

- Real-Time Rendering,
  - chapter 3 - Transforms

Why transforms?

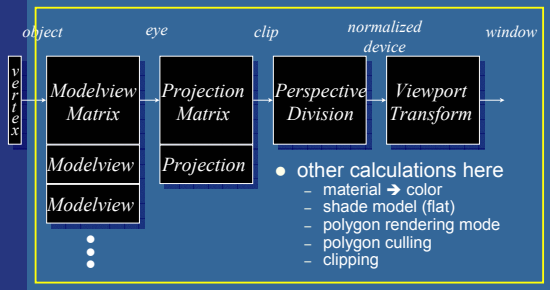
- We want to be able to **animate** objects and the camera
  - Translations
  - Rotations
  - Shears
  - And more…
- We want to be able to use **projection** transforms

How implement transforms?

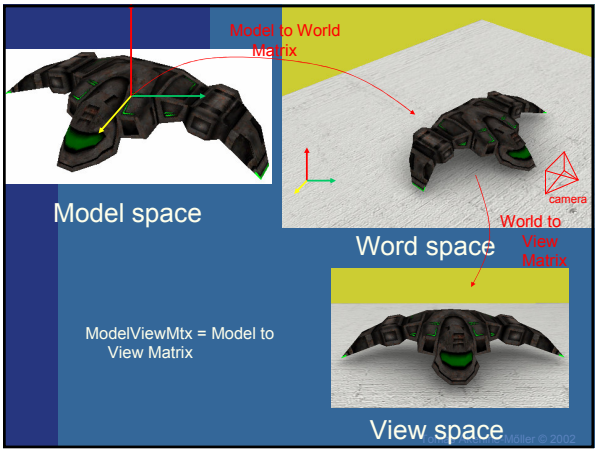
- Matrices!
- Can you really do everything with a matrix?
- Not everything, but a lot!
- We use 3x3 and 4x4 matrices

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \qquad \mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$$

Transformation Pipeline



OpenGL | Geometry stage | Mainly on GPU

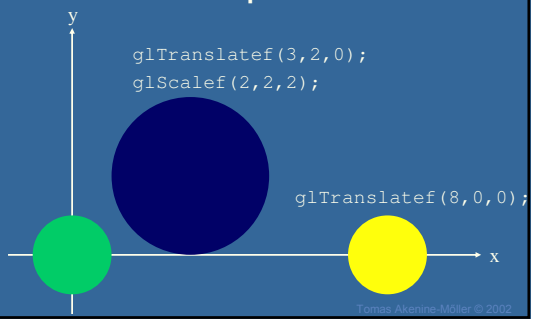


How do I use transforms practically?

- Say you have a circle with origin at (0,0,0) and with radius 1 – unit circle
- `glTranslatef(8,0,0);`
- `RenderCircle();`
- `glTranslatef(3,2,0);`
- `glScalef(2,2,2);`
- `RenderCircle();`

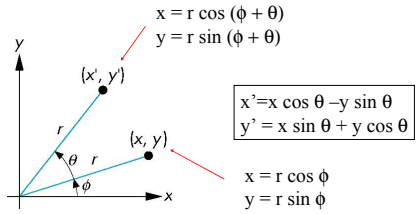
Cont'd from previous slide A simple 2D example

- A circle in **model space**



Rotation (2D)

Consider rotation about the origin by θ degrees
–radius stays the same, angle increases by θ



Derivation of rotation matrix in 2D

$\mathbf{n} = \mathbf{R}_z\mathbf{p}$?

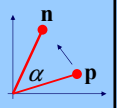
$\mathbf{p} = re^{i\phi} = r(\cos\phi + i\sin\phi)$ [rotation is mult by $e^{i\alpha}$]

$\mathbf{n} = e^{i\alpha}p = re^{i\alpha}e^{i\phi} =$

$= r[(\cos\alpha + i\sin\alpha)(\cos\phi + i\sin\phi)] =$

$= r(\cos\alpha\cos\phi - \sin\alpha\sin\phi) +$

$ir(\cos\alpha\sin\phi + \sin\alpha\cos\phi)$



$\mathbf{p} = (p_x, p_y)^T = (r\cos\phi, r\sin\phi)^T$

$\mathbf{n} = (n_x, n_y)^T = (r(\cos\alpha\cos\phi - \sin\alpha\sin\phi),$

$r(\cos\alpha\sin\phi + \sin\alpha\cos\phi))^T$

Derivation 2D rotation, cont'd

$\mathbf{p} = (p_x, p_y)^T = (r\cos\phi, r\sin\phi)^T$

$\mathbf{n} = (n_x, n_y)^T = (r(\cos\alpha\cos\phi - \sin\alpha\sin\phi),$

$r(\cos\alpha\sin\phi + \sin\alpha\cos\phi))^T$

$\mathbf{n} = \mathbf{R}_z\mathbf{p}$ what is $\mathbf{R}_z$?

$$\begin{pmatrix} n_x \\ n_y \end{pmatrix} = \underbrace{\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}}_{\mathbf{R}_z} \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

Rotations in 3D

- Same as in 2D for Z-rotations, but with a 3x3 matrix

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \Rightarrow \mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- For X

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

- For Y

$$\mathbf{R}_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

Translations must be simple?

Translation	Rotation
$\begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix} \mathbf{p} = \mathbf{p} + \mathbf{t}$	$\mathbf{Rp} = \mathbf{n}$

- Rotation is matrix mult, translation is add
- Would be nice if we could only use matrix multiplications…
- Turn to **homogeneous coordinates**
- Add a new component to each vector

Homogeneous notation

- A point: $\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & 1 \end{pmatrix}^T$
- Translation becomes:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{T(t)}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{Td} = \mathbf{d}$
- Also allows for projections (later)

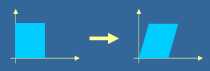
Rotations in 4x4 form

- Just add a row at the bottom, and a column at the right:

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Similarly for X and Y
- det(R)=1 (for 3x3 matrices)
- Trace(R)=1+2cos(alpha) (for any axis,3x3)

More basic transforms

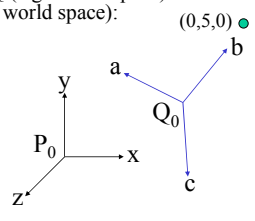
- Scaling 
- Shear 
- Rigid-body: rotation then translation

$$\mathbf{X} = \mathbf{TR}$$

- Concatenation of matrices
  - Not commutative, i.e., $\mathbf{RT} \neq \mathbf{TR}$
  - In $\mathbf{X} = \mathbf{TR}$, the rotation is done first
- Inverses and rotation about arbitrary axis:
  - Rigid body: $X^{-1} = X^T$

Change of Frames

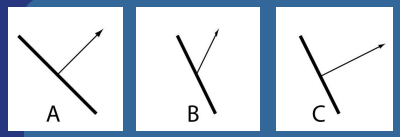
Computing the matrix $M_{Q\to P}$ that transforms a vertex from coordinate system Q (e.g. model space) into coordinate system P (e.g. world space):

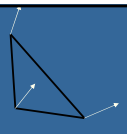


$$M_{Q\to P} = \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix}$$

Ex: $\mathbf{p}_P = M_{Q\to P}\,\mathbf{p}_Q = M_{Q\to P}(0,5,0)^T = 5\,\mathbf{b}$

Normal transforms Not so normal…



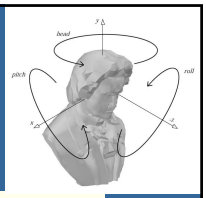


- Cannot use same matrix to transform normals

$$\text{Use}: \mathbf{N} = \left(\mathbf{M}^{-1}\right)^T \quad \text{instead of } \mathbf{M}$$

- **M** works for rotations and translations, though

The Euler Transform



- Assume the view looks down the negative z axis, with up in the y direction, x to the right

$$\mathbf{E}(h,p,r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)$$

- *h*=head
- *p*=pitch
- *r*=roll
- Gimbal lock can occur – looses one degree of freedom
- Example: h=0,p=π/2, then the z- rotation is the same as doing a previous rot around *y*-axis

Quaternions

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = (q_x, q_y, q_z, q_w)$$
$$= iq_x + jq_y + kq_z + q_w$$

- Extension of imaginary numbers
- Avoids *gimbal lock* that the Euler could produce
- Focus on unit quaternions:

$$n(\hat{\mathbf{q}}) = q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$$

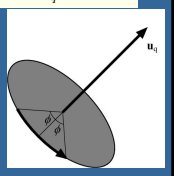
- A unit quaternion is:

$$\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi) \quad \text{where} \| \mathbf{u}_q \| = 1$$

Unit quaternions are perfect for rotations!

$$\hat{\mathbf{q}} = (\sin\phi\mathbf{u}_q, \cos\phi)$$

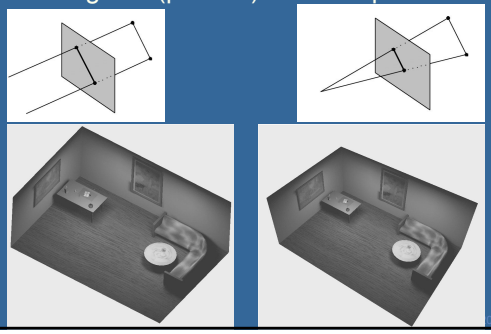
- Compact (4 components)
- Can show that $\hat{\mathbf{q}}\,\hat{\mathbf{p}}\,\hat{\mathbf{q}}^{-1}$
- …represents a rotation of 2ϕ radians around $\mathbf{u}_q$ of $\mathbf{p}$



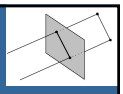
- That is: a unit quaternion represent a rotation as a **rotation axis** and an **angle**
- OpenGL: `glRotatef(ux,uy,uz,angle);`
- Interpolation from one quaternion to another is much simpler, and gives optimal results

Projections

- Orthogonal (parallel) and Perspective

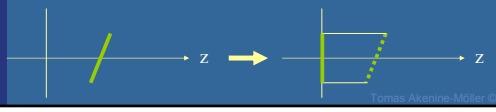


Orthogonal projection

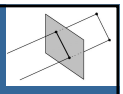


- Simple, just skip one coordinate
  - Say, we're looking along the z-axis
  - Then drop z, and render

$$\mathbf{M}_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \mathbf{M}_{ortho} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix}$$



Orthogonal projection

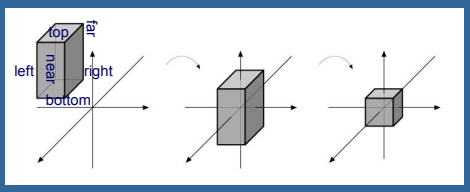


- Not invertible! (determinant is zero)
- For Z-buffering
  - It is not sufficient to project to a plane
  - Rather, we need to "project" to a box

- Unit cube is also used for perspective proj.
- Simplifies clipping

Orthogonal projection

- The ”unitcube projection” is invertible
- Simple to derive
  - Just a translation and scale



What about those homogenenous coordinates?

$$\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & p_w \end{pmatrix}^T$$

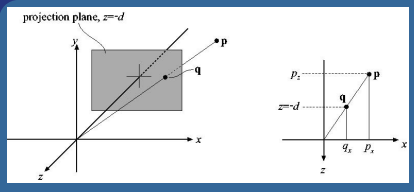
- p_w=0 for vectors, and p_w=1 for points
- What if p_w is **not** 1 or 0?
- Solution is to divide all components by p_w

$$\mathbf{p} = \begin{pmatrix} p_x / p_w & p_y / p_w & p_z / p_w & 1 \end{pmatrix}^T$$

- Gives a point again!
- Can be used for projections, as we will see

Perspective projection

$d>0$

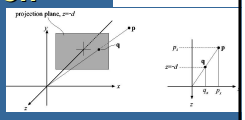


$$\frac{q_x}{p_x} = \frac{-d}{p_z} \Rightarrow q_x = -d\frac{p_x}{p_z} \qquad \text{For y: } q_y = -d\frac{p_y}{p_z}$$

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

Perspective projection

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \qquad \mathbf{P}_p\mathbf{p} = ?$$



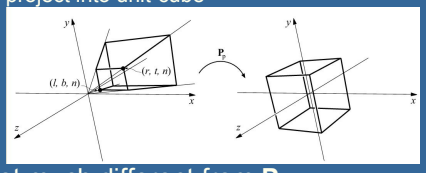
$$\mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \mathbf{q} = \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -dp_z/p_z \\ 1 \end{pmatrix} = \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}$$

$$q_x = -d\frac{p_x}{p_z} \qquad q_y = -d\frac{p_y}{p_z}$$

- The ”arrow” is the homogenization process

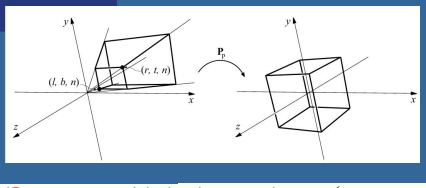
Perspective projection

- Again, the determinant is 0 (not invertible)
- To make the rest of the pipeline the same as for orhogonal projection:
  - project into unit-cube



- Not much different from $\mathbf{P}_p$
- Do not collapse z-coord to a plane

Understanding the projection matrix



$$\mathbf{P}_p\mathbf{p} = \begin{pmatrix} s_x & 0 & a & 0 \\ 0 & s_y & b & 0 \\ 0 & 0 & s_z & c \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x + a p_z \\ s_y p_y + b p_z \\ s_z p_z + c \\ -p_z/d \end{pmatrix} \Rightarrow \mathbf{q} = \begin{pmatrix} -(d_1 p_x/p_z) + a' \\ -(d_2 p_y/p_z) + b' \\ -(d_3 p_z + c)/p_z \\ 1 \end{pmatrix}$$

- Scaling
- Skew
- Keep z info

Perspective projection matrices

- See ”Från Värld till Skärm” secion 4 for more details.
- BREAK...

Följande slides är enbart till för att ge lite mer kött på benen om tidigare slides inte räckte till för att förstå.

De förklarar samma sak, fast på ett lite annat sätt och med mer detaljer. Skall ses som ett valfritt komplement.

/Ulf

Most of the following slides are from

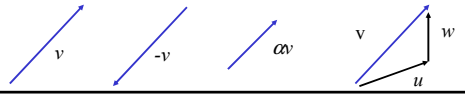
Ed Angel
Professor of Computer Science, Electrical and Computer Engineering, and Media Arts
University of New Mexico

Scalars

- Need three basic elements in geometry
  - –Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses)
- Examples include the real and complex number systems under the ordinary rules with which we are familiar
- Scalars alone have no geometric properties

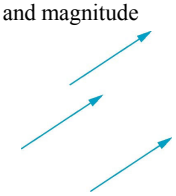
Vector Operations

- Physical definition: a vector is a quantity with two attributes
  - – Direction
  - – Magnitude
- Examples include
  - – Force
  - – Velocity
  - – Directed line segments
    - Most important example for graphics
    - Can map to other types Every vector can be multiplied by a scalar
- There is a zero vector
  - –Zero magnitude, undefined orientation
- The sum of any two vectors is a vector



Vectors Lack Position

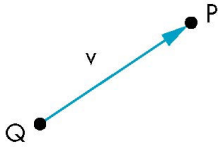
- These vectors are identical
  - –Same length and magnitude



- Vectors insufficient for geometry
  - –Need points

Points

- Location in space
- Operations allowed between points and vectors
  - –Point - point subtraction yields a vector
  - –Equivalent to point + vector addition



$v=P-Q$

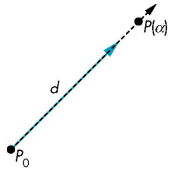
$P=v+Q$

Affine Spaces

- Point + a vector space
- Operations
  - –Vector + vector addition
  - –Scalar • vector multiplication
  - –Point + vector addition
  - –Scalar • scalar operations
- For any point define
  - –1 • P = P
  - –0 • P = **0** (zero vector)

Lines

- Consider all points of the form
  - –$P(\alpha)=P_0 + \alpha\,\mathbf{d}$
  - –Set of all points that pass through P_0 in the direction of the vector **d**



Parametric Form

- This form is known as the parametric form of the line
  - –More robust and general than other forms
  - –Extends to curves and surfaces
- Two-dimensional forms
  - –Explicit: y = kx + m
  - –Implicit: ax + by +c =0
  - –Parametric:

$x(\alpha) = \alpha x_0 + (1-\alpha)x_1$

$y(\alpha) = \alpha y_0 + (1-\alpha)y_1$

Rays and Line Segments

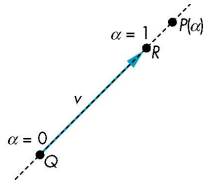
- If $\alpha >= 0$, then $P(\alpha)$ is the *ray* leaving P_0 in the direction **d**

If we use two points to define v, then

$P(\alpha) = Q + \alpha (R-Q) = Q+\alpha v$

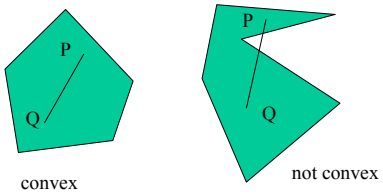
$=\alpha R + (1-\alpha)Q$

For $0<=\alpha<=1$ we get all the points on the *line segment* joining R and Q



Convexity

- An object is *convex* iff for any two points in the object all points on the line segment between these points are also in the object



Affine Sums

- Consider the "sum"

$P=\alpha_1 P_1+\alpha_2 P_2+\ldots..+\alpha_n P_n$

Can show by induction that this sum makes sense iff

$\alpha_1+\alpha_2+\ldots..\alpha_n=1$

in which case we have the *affine sum* of the points $P_1, P_2, \ldots..P_n$

- If, in addition, $\alpha_i >= 0$, we have the *convex hull* of $P_1, P_2, \ldots..P_n$

Convex Hull

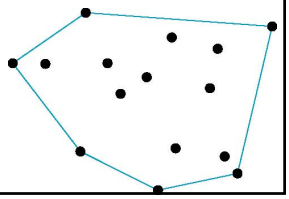
Consider the linear combination

$P=\alpha_1P_1+\alpha_2P_2+\ldots..+\alpha_nP_n$

- If $\alpha_1+\alpha_2+\ldots..\alpha_n=1$
  - (in which case we have the *affine sum* of the points $P_1,P_2,\ldots..P_n$)

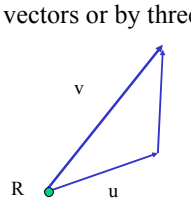
and if $\alpha_i>=0$, we have the *convex hull* of $P_1,P_2,\ldots..P_n$

- Smallest convex object containing $P_1,P_2,\ldots..P_n$

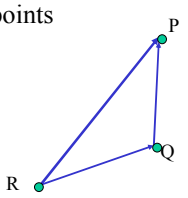


Planes

- A plane can be defined by a point and two vectors or by three points

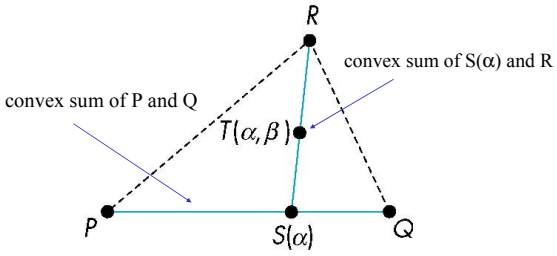


$P(\alpha,\beta)=R+\alpha u+\beta v$



$P(\alpha,\beta)=R+\alpha(Q-R)+\beta(P-Q)$

Triangles

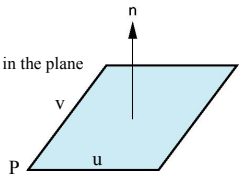


for $0<=\alpha,\beta<=1$, we get all points in triangle

Normals

- Every plane has a vector n normal (perpendicular, orthogonal) to it
- From point/vector form
  - $P(\alpha,\beta)=R+\alpha u+\beta v$

  we know we can use the cross product to find
  - $n = u \times v$
- Plane equation:
  - $n \cdot x - d = 0$,
  - where $d = -n \cdot p$ and p is any point in the plane

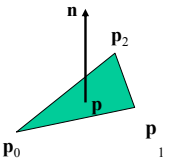


Normal for Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

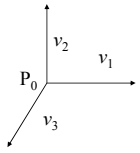
normalize $\mathbf{n} \leftarrow \mathbf{n}/|\mathbf{n}|$



Note that right-hand rule determines outward face

Frames

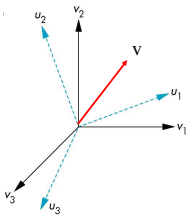
- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*



Representing one basis in terms of another

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms

$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$
$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$
$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$



Matrix Form

The coefficients define a 3 x 3 matrix

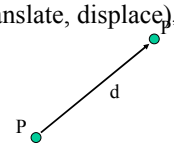
$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

$$\mathbf{a}=\mathbf{M}^T\mathbf{b}$$

Translation

- Move (translate, displace) a point to a new location

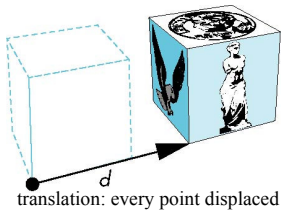


- Displacement determined by a vector d
  - Three degrees of freedom
  - P'=P+d

How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually only one way

object



translation: every point displaced by same vector

Translation Using Representations

Using the homogeneous coordinate representation in some frame

$\mathbf{p}=[x\ y\ z\ 1]^T$
$\mathbf{p}'=[x'\ y'\ z'\ 1]^T$
$\mathbf{d}=[dx\ dy\ dz\ 0]^T$

Hence $\mathbf{p}' = \mathbf{p} + \mathbf{d}$ or

$x'=x+d_x$
$y'=y+d_y$
$z'=z+d_z$

note that this expression is in four dimensions and expresses point = vector + point

Translation Matrix

We can also express translation using a 4 x 4 matrix **T** in homogeneous coordinates

$\mathbf{p}'=\mathbf{T}\mathbf{p}$ where

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together

Homogeneous Coordinates

The homogeneous coordinates form for a three dimensional point [x y z] is given as

$\mathbf{p}=[x'\ y'\ z'\ w]^T=[wx\ wy\ wz\ w]^T$

We return to a three dimensional point (for $w\neq 0$) by

$x \leftarrow x'/w$

$y \leftarrow y'/w$

$z \leftarrow z'/w$

If w=0, the representation is that of a vector

Note that homogeneous coordinates replaces points in three dimensions by lines through the origin in four dimensions

For w=1, the representation of a point is [x y z 1]

Homogeneous Coordinates and Computer Graphics

- Homogeneous coordinates are key to all computer graphics systems
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices
  - Hardware pipeline works with 4 dimensional representations
  - For orthographic viewing, we can maintain w=0 for vectors and w=1 for points
  - For perspective we need a *perspective division*

Rotation about the z axis

- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x'=x\cos\theta - y\sin\theta$$
$$y'=x\sin\theta + y\cos\theta$$
$$z'=z$$

  - or in homogeneous coordinates

$$\mathbf{p}'=\mathbf{R}_z(\theta)\mathbf{p}$$

Rotation Matrix

$$\mathbf{R}=\mathbf{R}_z(\theta)=\begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Rotation about x and y axes

- Same argument as for rotation about *z* axis
  - For rotation about *x* axis, *x* is unchanged
  - For rotation about *y* axis, *y* is unchanged

$$\mathbf{R}=\mathbf{R}_x(\theta)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$\mathbf{R}=\mathbf{R}_y(\theta)=\begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

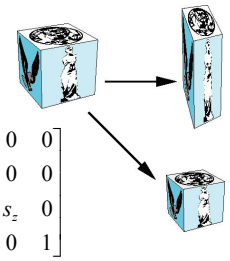
Scaling

Expand or contract along each axis (fixed point of origin)

$$x'=s_x x$$
$$y'=s_y x$$
$$z'=s_z x$$

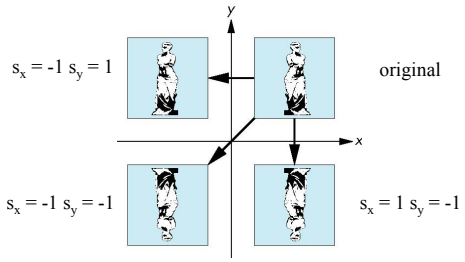
$$\mathbf{p}'=\mathbf{S}\mathbf{p}$$

$$\mathbf{S}=\mathbf{S}(s_x, s_y, s_z)=\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$



Reflection

corresponds to negative scale factors



Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations
  - –Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - –Rotation: $\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$
    - Holds for any rotation matrix
    - Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

      $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{T}(\theta)$
  - –Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices
- Because the same transformation is applied to many vertices, the cost of forming a matrix **M**=**ABCD** is not significant compared to the cost of computing **Mp** for many vertices **p**
- The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

  $\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$
- Note many references use column matrices to represent points. In terms of column matrices

  $\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$

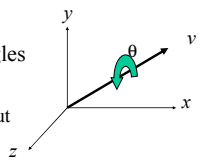
General Rotation About the Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\,\mathbf{R}_y(\theta_y)\,\mathbf{R}_x(\theta_x)$$

θ_x θ_y θ_z are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but with different angles



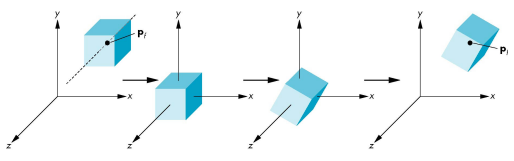
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

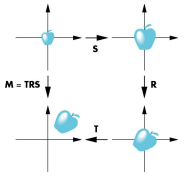
Move fixed point back

$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$



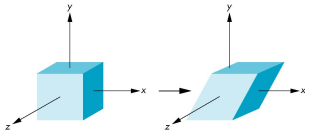
Instancing

- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size
- We apply an *instance transformation* to its vertices to
  - Scale
  - Orient
  - Locate



Shear

- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions

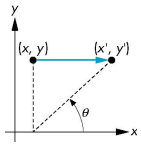


Shear Matrix

Consider simple shear along *x* axis

$$x' = x + y \cot\theta$$
$$y' = y$$
$$z' = z$$

$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



OpenGL Transformations

Objectives

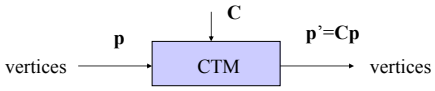
- Learn how to carry out transformations in OpenGL
  - –Rotation
  - –Translation
  - –Scaling
- Introduce OpenGL matrix modes
  - –Model-view
  - –Projection

OpenGL Matrices

- In OpenGL matrices are part of the state
- Multiple types
  - –Model-View (**GL_MODELVIEW**)
  - –Projection (**GL_PROJECTION**)
  - –Texture (**GL_TEXTURE**) (ignore for now)
  - –Color(**GL_COLOR**) (ignore for now)
- Single set of functions for manipulation
- Select which to manipulated by
  - –`glMatrixMode(GL_MODELVIEW);`
  - –`glMatrixMode(GL_PROJECTION);`

Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline
- The CTM is defined in the user program and loaded into a transformation unit



CTM operations

- The CTM can be altered either by loading a new CTM or by postmutiplication

Load an identity matrix: **C ← I**
Load an arbitrary matrix: **C ← M**

Load a translation matrix: **C ← T**
Load a rotation matrix: **C ← R**
Load a scaling matrix: **C ← S**

Postmultiply by an arbitrary matrix: **C ← CM**
Postmultiply by a translation matrix: **C ← CT**
Postmultiply by a rotation matrix: **C ← C R**
Postmultiply by a scaling matrix: **C ← C S**

Rotation about a Fixed Point

Start with identity matrix: **C ← I**
Move fixed point to origin: **C ← CT**
Rotate: **C ← CR**
Move fixed point back: **C ← CT** $^{-1}$

Result: **C = TR T** $^{-1}$ which is **backwards**.

This result is a consequence of doing postmultiplications. Let's try again.

Reversing the Order

We want **C = T** $^{-1}$ **R T**
so we must do the operations in the following order

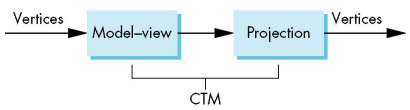
C ← I
C ← CT $^{-1}$
C ← CR
C ← CT

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program

CTM in OpenGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM
- Can manipulate each by first setting the correct matrix mode



Rotation, Translation, Scaling

Load an identity matrix:

```
glLoadIdentity()
```

Multiply on right:

```
glRotatef(theta, vx, vy, vz)
```

theta in degrees, (**vx, vy, vz**) define axis of rotation

```
glTranslatef(dx, dy, dz)
glScalef( sx, sy, sz)
```

Each has a float (f) and double (d) format (**glScaled**)

Example

- Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(1.0, 2.0, 3.0);
glRotatef(30.0, 0.0, 0.0, 1.0);
glTranslatef(-1.0, -2.0, -3.0);
```

- Remember that last matrix specified in the program is the first applied

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program

```
glLoadMatrixf(m)
glMultMatrixf(m)
```

- The matrix **m** is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns
- In **glMultMatrixf**, **m** multiplies the existing matrix on the right

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - Traversing hierarchical data structures (Chapter 10)
  - Avoiding state changes when executing display lists
- OpenGL maintains stacks for each type of matrix
  - Access present type (as set by **glMatrixMode)** by

```
glPushMatrix()
glPopMatrix()
```

Reading Back Matrices

- Can also access matrices (and other parts of the state) by *query* functions

```
glGetIntegerv
glGetFloatv
glGetBooleanv
glGetDoublev
glIsEnabled
```

- For matrices, we use as

```
double m[16];
glGetFloatv(GL_MODELVIEW, m);
```

Using the Model-view Matrix

- In OpenGL the model-view matrix is used to
  - Position the camera
    - Can be done by rotations and translations but is often easier to use **gluLookAt**
  - Build models of objects
- The projection matrix is used to define the view volume and to select a camera lens

Quaternions

- Extension of imaginary numbers from two to three dimensions
- Requires one real and three imaginary components **i**, **j**, **k** $q=q_0+q_1\mathbf{i}+q_2\mathbf{j}+q_3\mathbf{k}$
- Quaternions can express rotations on sphere smoothly and efficiently. Process:
  - Model-view matrix → quaternion
  - Carry out operations with quaternions
  - Quaternion → Model-view matrix

Computer Viewing

Ed Angel
Professor of Computer Science,
Electrical and Computer Engineering,
and Media Arts
University of New Mexico

Objectives

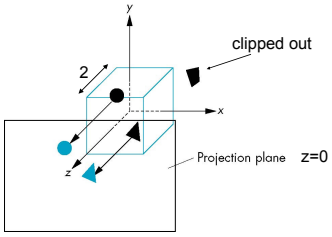
- Introduce the mathematics of projection
- Introduce OpenGL viewing functions
- Look at alternate viewing APIs

Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - –Positioning the camera
    - Setting the model-view matrix
  - –Selecting a lens
    - Setting the projection matrix
  - –Clipping
    - Setting the view volume
    - (default is unit cube, R^3, [-1,1])

Default Projection

Default projection is orthogonal

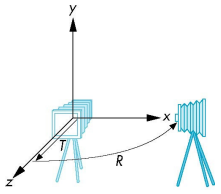


Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - –Move the camera in the positive z direction
    - Translate the camera frame
  - –Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - –Want a translation (`glTranslatef(0.0,0.0,-d);`)
  - –`d > 0`

Moving the Camera

- We can move the camera to any desired position by a sequence of rotations and translations
- Example: side view
  - –Rotate the camera
  - –Move it away from origin
  - –Model-view matrix C = TR



OpenGL code

- Remember that last transformation specified is first to be applied

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
glTranslatef(0.0, 0.0, -d);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

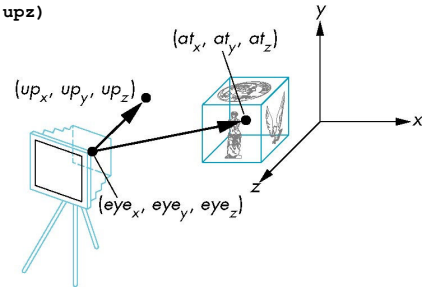
The LookAt Function

- The GLU library contains the function gluLookAt to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Still need to initialize
  - –Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
glMatrixMode(GL_MODELVIEW):
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0., 1.0. 0.0);
```

gluLookAt

```
glLookAt(eyex, eyey, eyez, atx, aty, atz, upx,
 upy, upz)
```

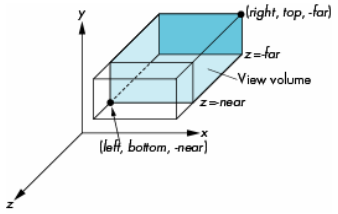


Other Viewing APIs

- The LookAt function is only one possible API for positioning the camera
- Others include
  - –View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - –Yaw, pitch, roll
  - –Elevation, azimuth, twist
  - –Direction angles

OpenGL Orthogonal Viewing

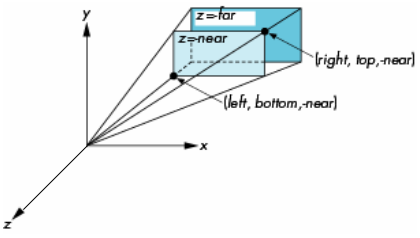
```
glOrtho(left,right,bottom,top,near,far)
```



`near` and `far` measured from camera

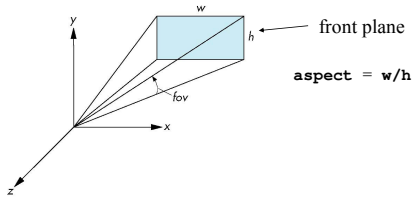
OpenGL Perspective

```
glFrustum(left,right,bottom,top,near,far
)
```



Using Field of View

- With **`glFrustum`** it is often difficult to get the desired view
- **`gluPerpective(fovy, aspect, near, far)`** often provides a better interface



Projections explained differently

- Read the following slides about orthogonal and perspective projections by your selves
- They present the same thing we went through on the lecture, but explained differently

/Ulf

Projections and Normalization

- The default projection in the eye (camera) frame is orthogonal
- For points within the default view volume

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$

- Most graphics systems use *view normalization*
  - –All other views are converted to the default view by transformations that determine the projection matrix
  - –Allows use of the same pipeline for all views

Homogeneous Coordinate Representation

default orthographic projection

$$\mathbf{p}_p = \mathbf{M}\mathbf{p}$$

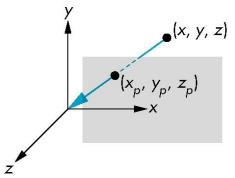
$$x_p = x \quad y_p = y \quad z_p = 0 \quad w_p = 1$$

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In practice, we can let $\mathbf{M} = \mathbf{I}$ and set the z term to zero later

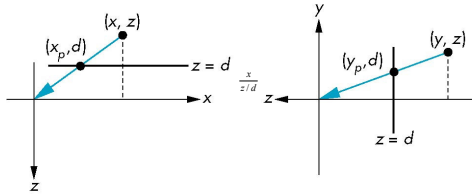
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d$, $d < 0$



Perspective Equations

Consider top and side views



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

Homogeneous Coordinate Form

consider $\mathbf{q} = \mathbf{M}\mathbf{p}$ where

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \Rightarrow \mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Perspective Division

- However $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields

$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

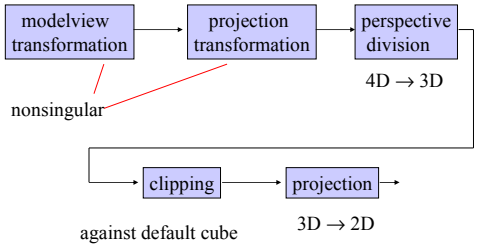
the desired perspective equations

- We will consider the corresponding clipping volume with the OpenGL functions

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Pipeline View



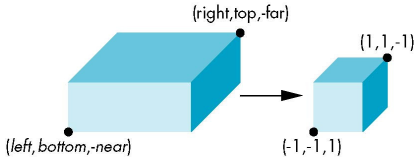
Notes

- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - –Both these transformations are nonsingular
  - –Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection
- Delay final projection until end
  - –Important for hidden surface removal to retain depth information as long as possible

Orthogonal Normalization

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

normalization ⇒ find transformation to convert specified clipping volume to default



Orthogonal Matrix

- Two steps
  - –Move center to origin

    T((left+right)/2, - (bottom+top)/2,(near+far)/2))
  - –Scale to have sides of length 2

    S(2/(left- right),2/(top- bottom),2/(near- far))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Projection

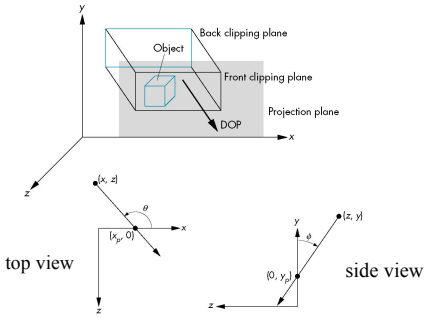
- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

$$\mathbf{M}_{orth} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Hence, general orthogonal projection in 4D is

$$\mathbf{P} = \mathbf{M}_{orth}\mathbf{ST}$$

General Shear



Shear Matrix

xy shear (*z* values unchanged)

$$\mathbf{H}(\theta,\phi) = \begin{bmatrix} 1 & 0 & -\cot\theta & 0 \\ 0 & 1 & -\cot\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

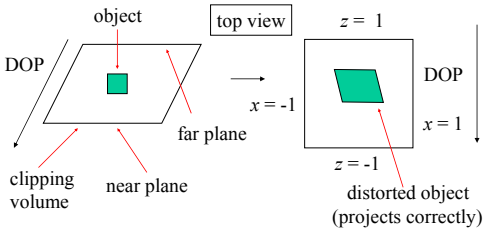
Projection matrix

$$\mathbf{P} = \mathbf{M}_{orth}\,\mathbf{H}(\theta,\phi)$$

General case: $\mathbf{P} = \mathbf{M}_{orth}\,\mathbf{STH}(\theta,\phi)$

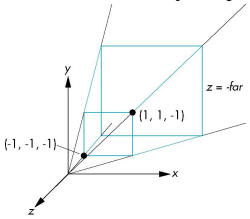
Effect on Clipping

- The projection matrix **P** = **STH** transforms the original clipping volume to the default clipping volume



Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes $x = \pm z$, $y = \pm z$



Perspective Matrices

Simple projection matrix in homogeneous coordinates

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

Note that this matrix is independent of the far clipping plane

Generalization

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

after perspective division, the point $(x, y, z, 1)$ goes to

$x'' = x/z$
$y'' = y/z$
$Z'' = -(\alpha+\beta/z)$

which projects orthogonally to the desired point regardless of α and β

Picking α and β

If we pick

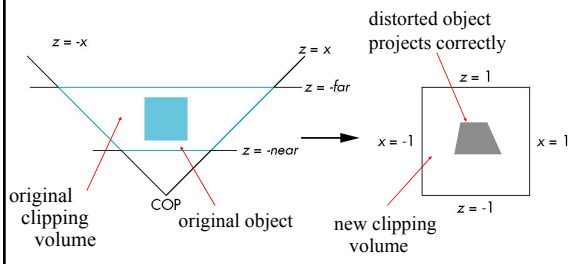
$$\alpha = \frac{\text{near} + \text{far}}{\text{far} - \text{near}}$$

$$\beta = \frac{2\text{near} * \text{far}}{\text{near} - \text{far}}$$

the near plane is mapped to $z = -1$
the far plane is mapped to $z = 1$
and the sides are mapped to $x = \pm 1, y = \pm 1$

Hence the new clipping volume is the default clipping volume

Normalization Transformation

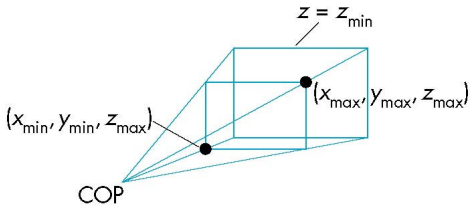


Normalization and Hidden-Surface Removal

- Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$
- Thus hidden surface removal works if we first apply the normalization transformation
- However, the formula $z'' = -(\alpha+\beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small

OpenGL Perspective

- **`glFrustum`** allows for an unsymmetric viewing frustum (although **`gluPerspective`** does not)



OpenGL Perspective Matrix

- The normalization in **`glFrustum`** requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthogonal transformation

P = NSH

our previously defined perspective matrix

shear and scale

Why do we do it this way?

- Normalization allows for a single pipeline for both perspective and orthogonal viewing
- We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
- We simplify clipping