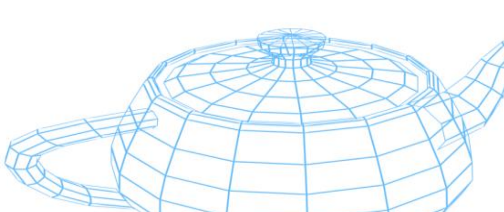
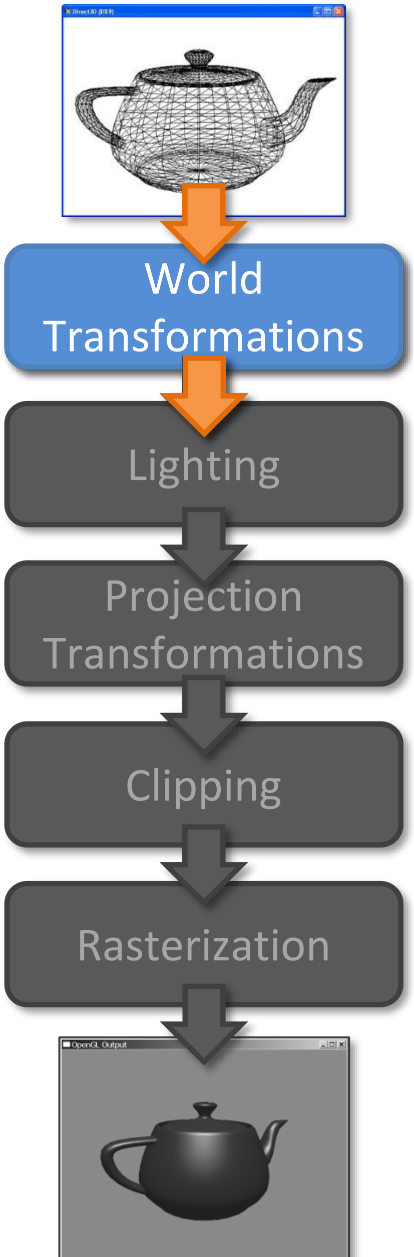
SUPSI

Computer Graphics

Mathematics for Computer Graphics (2)

Achille Peternier, adjunct professor



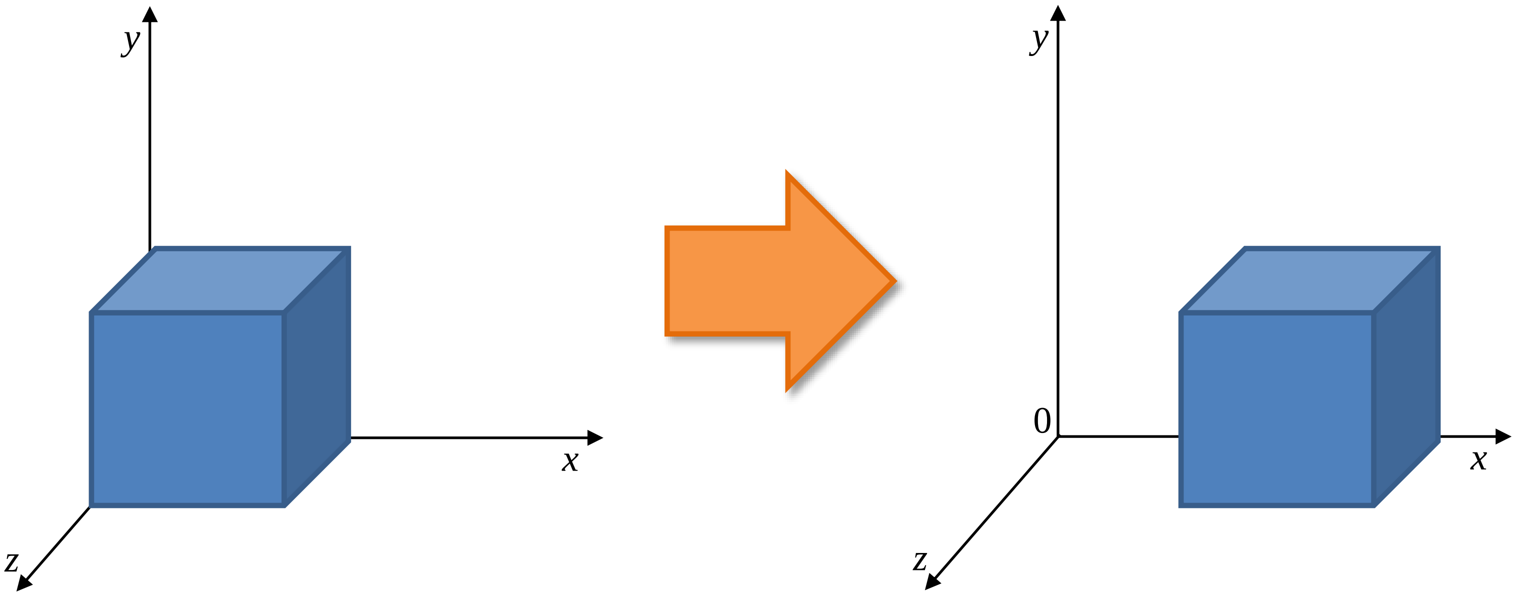

World Transformations
Lighting
Projection Transformations
Clipping
Rasterization

Translation

$$\mathbf{v}_n = \mathbf{v}_p + \mathbf{t}$$

$$\text{e.g.:}\quad \begin{bmatrix} 0.5 \\ 1 \\ 1.5 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 2.5 \\ 5 \\ 7.5 \end{bmatrix}$$

Translation

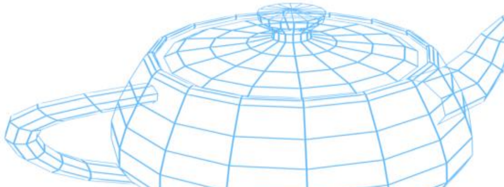


Rotation

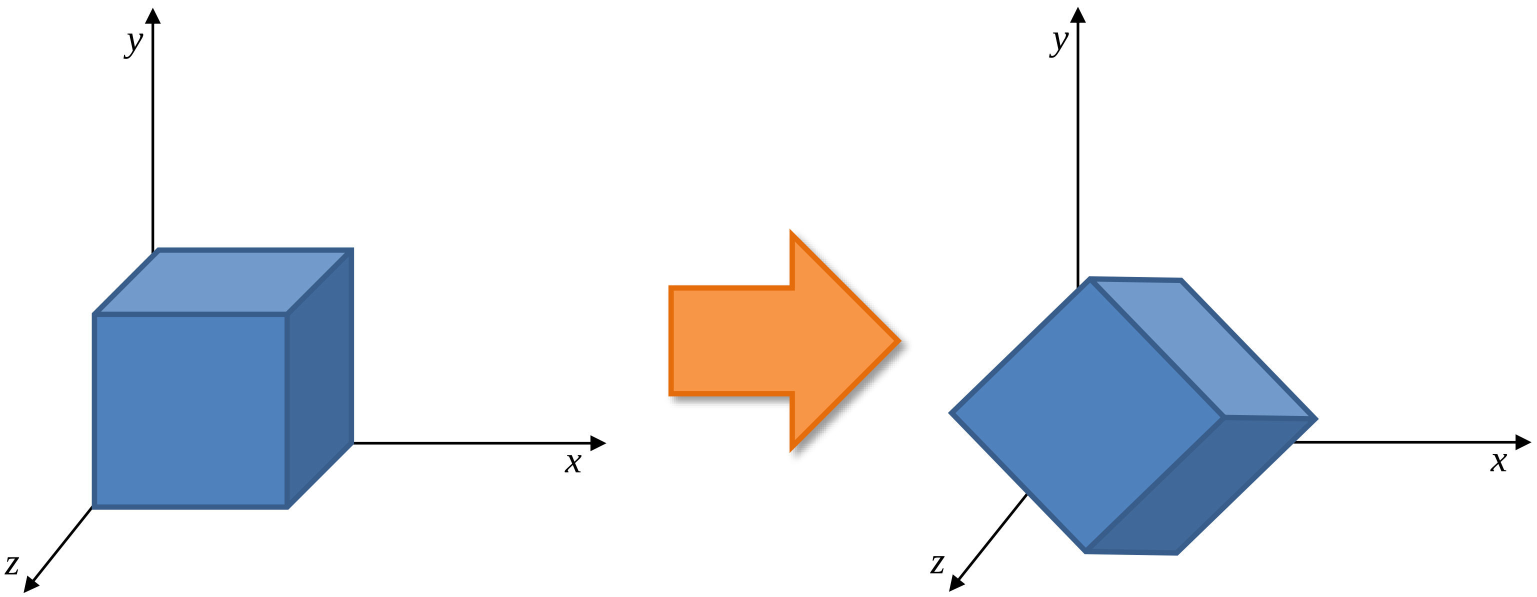
$$\mathbf{v}_n = \mathbf{R}\mathbf{v}_p$$

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \propto & -\sin \propto \\ 0 & \sin \propto & \cos \propto \end{bmatrix} \qquad \mathbf{R}_y = \begin{bmatrix} \cos \propto & 0 & \sin \propto \\ 0 & 1 & 0 \\ -\sin \propto & 0 & \cos \propto \end{bmatrix}$$

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



Rotation



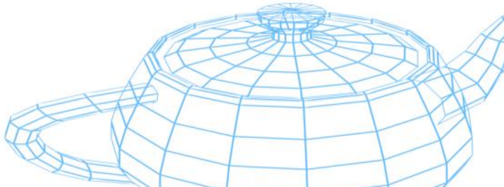
Scaling

$$\mathbf{v}_n = \mathbf{S}\mathbf{v}_p$$

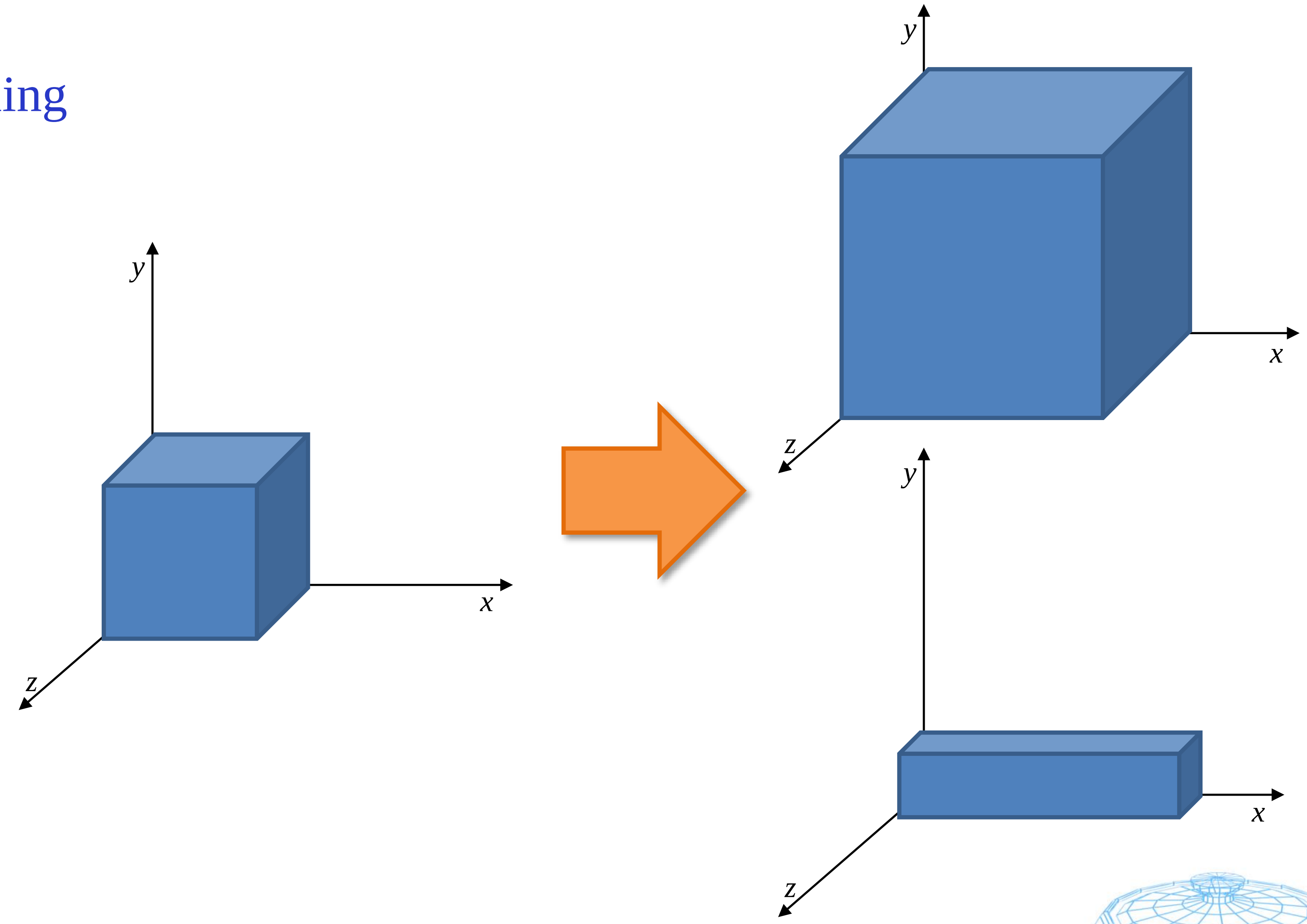
$$\mathbf{S} = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix}$$

When *x=y=z* → **uniform/isotropic** scaling

Otherwise → **non-uniform/anisotropic** scaling

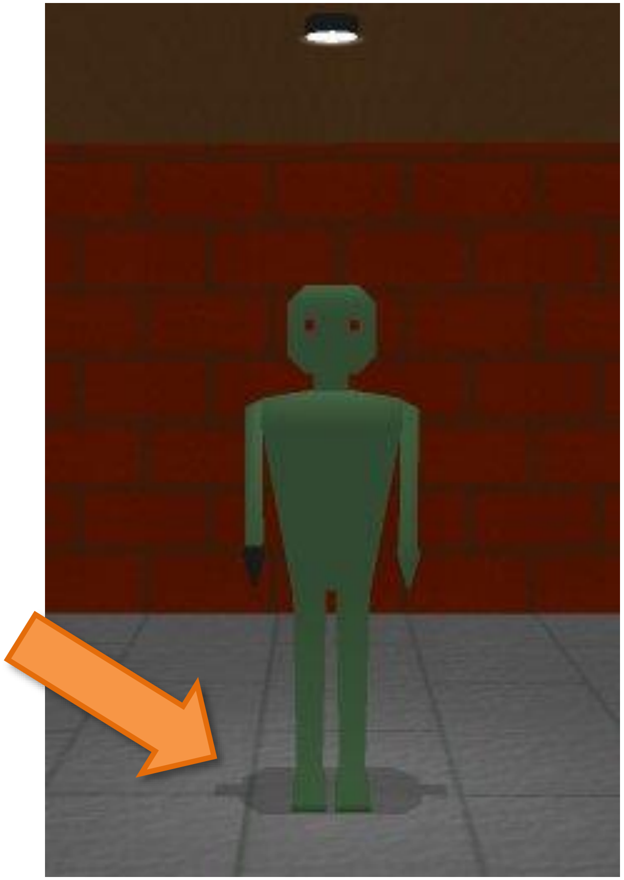


Scaling



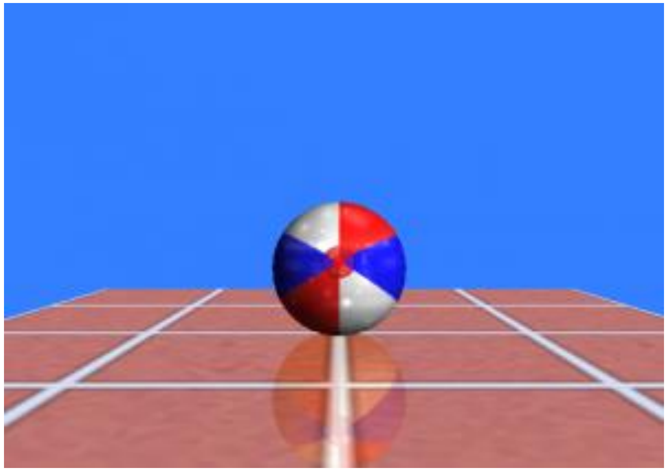
Scaling

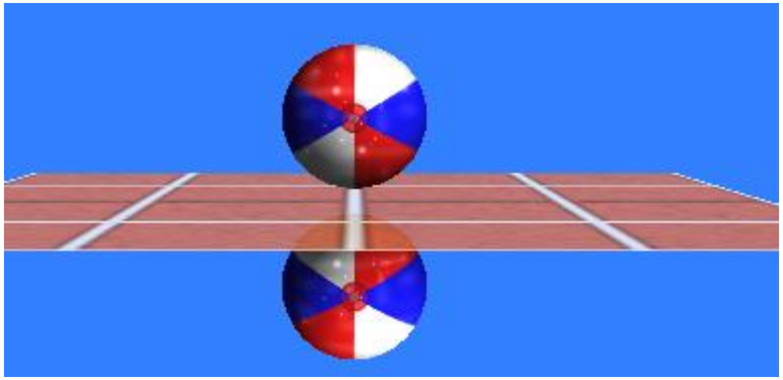
$$\boldsymbol{S} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



Scaling

$$\boldsymbol{S} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$





Homogeneous coordinates

- Without homogeneous coordinates:
  - Translation = vector by vector addition.
  - Rotation = matrix by vector multiplication.
  - Scaling = matrix by vector multiplication.

- Goals:
  - Reducing the number of operations required.
  - Using one same method for implementing all the base transformations of the rendering pipeline (including projections).

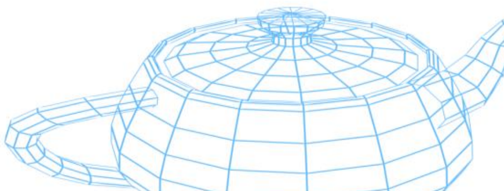
- With homogeneous coordinates:
  - Translation, rotation, scaling = matrix by vector multiplication.

Homogeneous coordinates

2D: $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

3D: $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ *w*

(`glVertex3f()` implicitly sets *w = 1*)



Homogeneous coordinates

$$\mathbf{v}_n = \mathbf{T}\mathbf{v}_p$$

where **T** can be any transformation among translation, rotation, and scaling

Homogeneous coordinates - translation

$$\mathbf{v}_n = \mathbf{T}\mathbf{v}_p$$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Homogeneous coordinates - translation

$$\mathbf{v}_n = \mathbf{T}\mathbf{v}_p$$

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ w_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

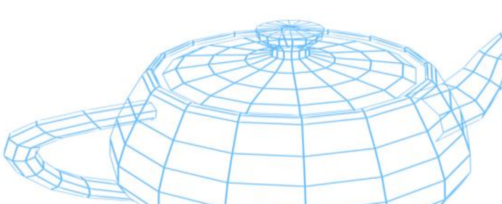
$$\begin{aligned} x_n &= x + 2 \times 1 \\ y_n &= y + (-3) \times 1 \\ z_n &= z + 4 \times 1 \\ w_n &= 1 \times 1 \end{aligned}$$

Homogeneous coordinates - rotation

$$\mathbf{v}_n = \mathbf{R}\mathbf{v}_p$$

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\propto & -\sin\propto & 0 \\ 0 & \sin\propto & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_y = \begin{bmatrix} \cos\propto & 0 & \sin\propto & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\propto & 0 & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



Homogeneous coordinates - scaling

$$\mathbf{v}_n = \mathbf{S}\mathbf{v}_p$$

$$\mathbf{S} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenation

- Given three transformations $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$:

$$\mathbf{v}_n = \mathbf{T}_3\mathbf{T}_2\mathbf{T}_1\mathbf{v}_p$$

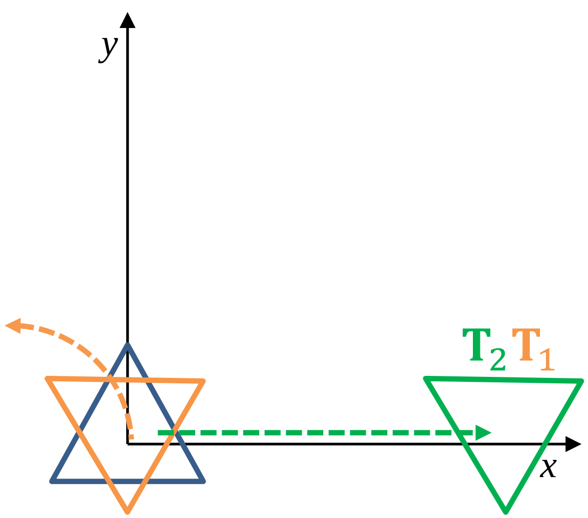
using post-multiplication and column vectors

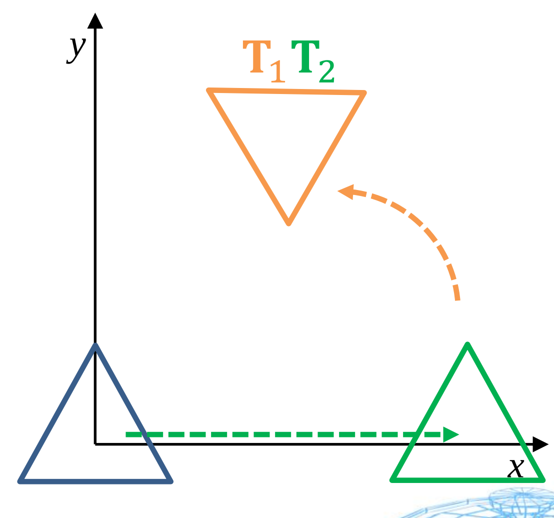
Concatenation

- Let: $\mathbf{T}_1 = \text{rotation of } 60°$

  $\mathbf{T}_2 = \text{translation}(10, 0)$

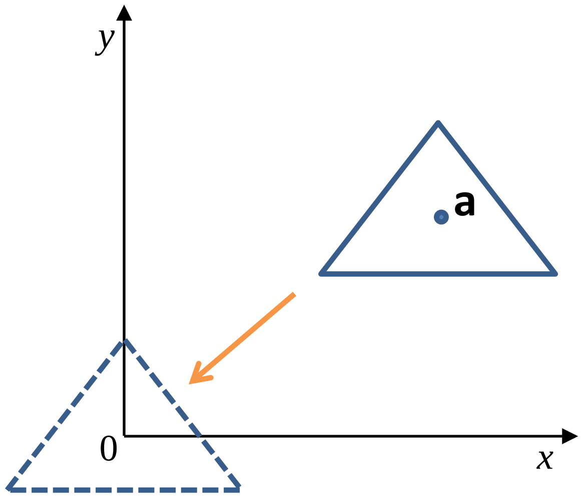
$$\mathbf{T}_2\,\mathbf{T}_1 \neq \mathbf{T}_1\,\mathbf{T}_2$$





Concatenation

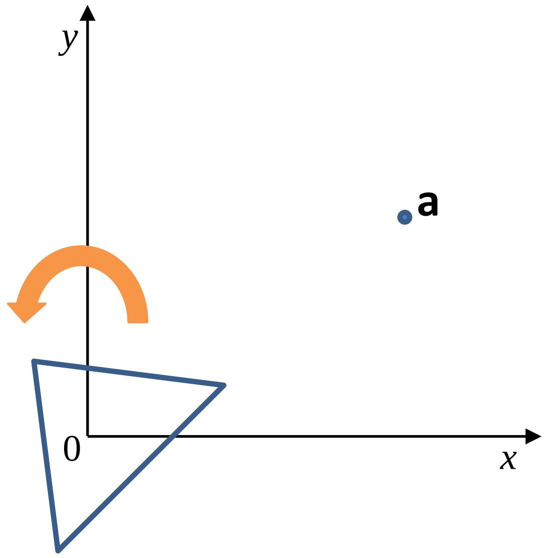
- Rotation of 45° about point **a**:



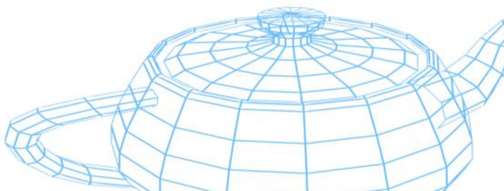
1) Subtract **a** to center the object at the origin

Concatenation

- Rotation of 45° about point **a**:

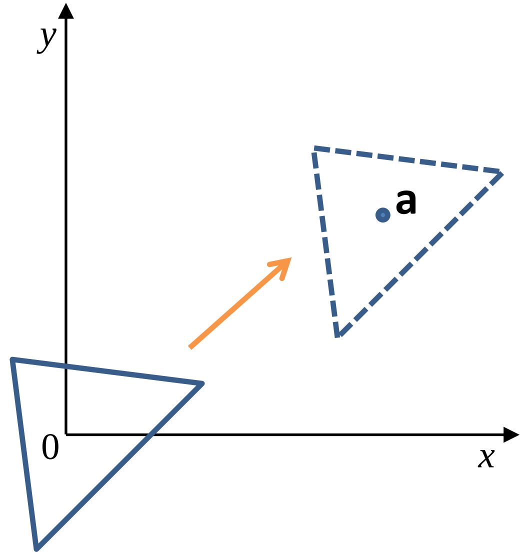


2) Rotate the object around the origin

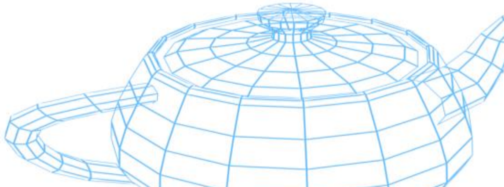


Concatenation

- Rotation of 45° about point **a**:

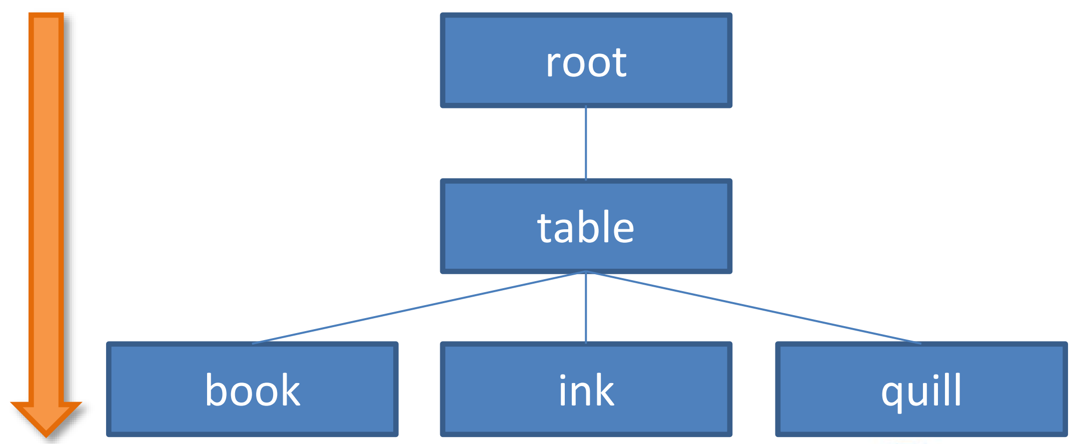


3) Add **a** to translate the object back



Scene graph

- The scene is represented as a hierarchy (tree) of dependencies:
  - Each node has its own matrix.
  - Each node multiplies the previous node’s matrix by its own matrix:
    - The cumulated resulting matrix is used by the next level.
  - Use push/pop to store/restore states as you go deeper in the tree.



Object positioning

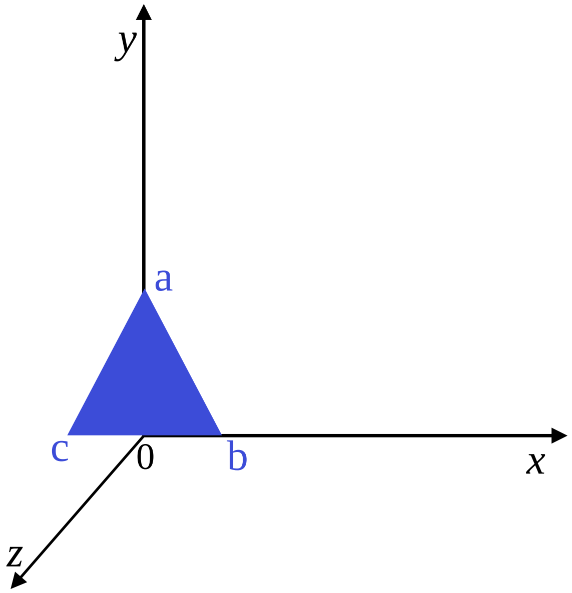
- Let put objects A, B, and C somewhere in the 3D world coordinates:
  - At each frame, we start from scratch and reprocess the scene-graph.
  - Reset the current matrix (set to identity).
  - Apply required transformations to place object A.
  - If object B depends on A’s position, apply the next transformations **without** resetting the current matrix:
    - New transformations stack on top of the previous ones.
  - If object C does not depend on previous objects’ position, reset the current matrix and start again.

Coordinate spaces

- **Object/model coordinates:**
  - The object's 3D vertices are defined as relative to the origin, i.e., the object is centered at (0, 0, 0).
- **World coordinates:**
  - 3D vertices with absolute position:
    - World center is the origin (0, 0, 0).
    - Object matrix * object coordinates = world coordinates.
- **Eye/view/camera coordinates:**
  - 3D vertices relative to the viewer's position:
    - Center is the viewer's position (0, 0, 0).
    - Camera matrix^{-1} * world coordinates = eye coordinates.

Object/model coordinates

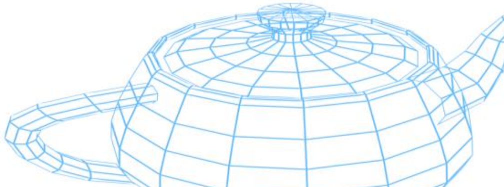
- The vertices of each 3D object are defined as relative to its origin.
- The origin usually refers to the object’s pivot point (center, barycenter or basement).
- Single 3D models are designed in object coordinates, then are moved around and their vertices become world coordinates.



a = (0, 2, 0)

b = (1, 0, 0)

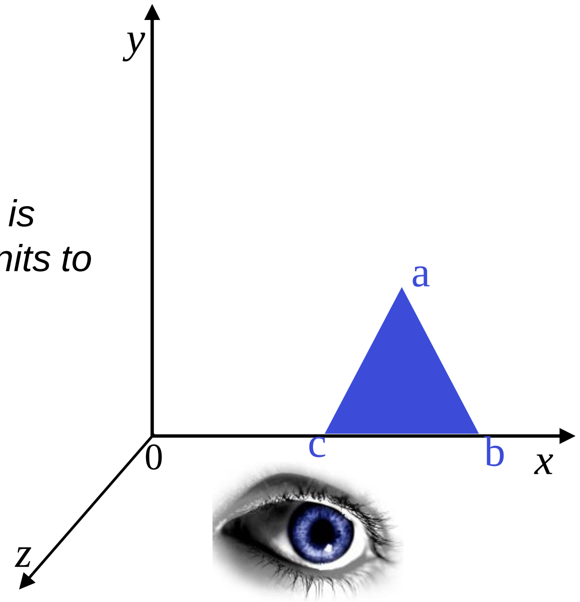
c = (-1, 0, 0)



World coordinates

- Coordinates are relative to the world's origin.
- One same object can be put at different locations in the same scene:
  - Each instance will have its own absolute coordinates.
  - Vertices are relative to the object's center in object coordinates → objects are relative to the world's center in world coordinates.

The object is translated 5 units to the right

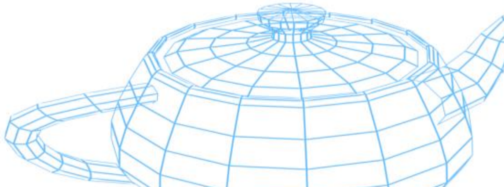


a = (5, 2, 0)

b = (6, 0, 0)

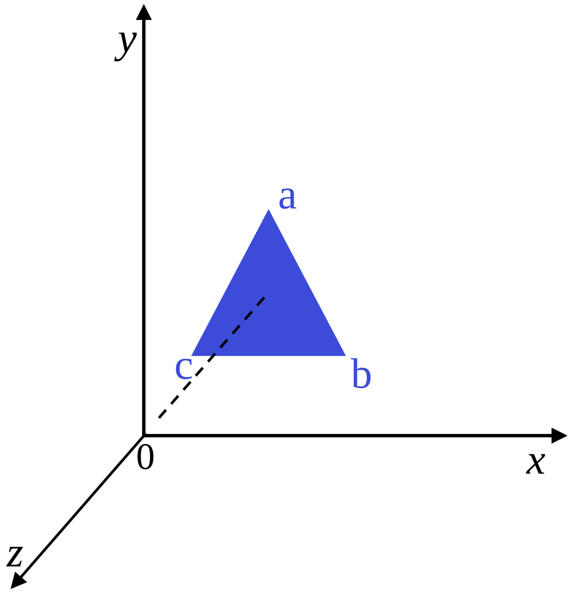
c = (4, 0, 0)

eye = (5, 1, 3)



Eye/view/camera coordinates

- The eye is now at the origin (0, 0, 0).
- Coordinates are relative to the eye's position.

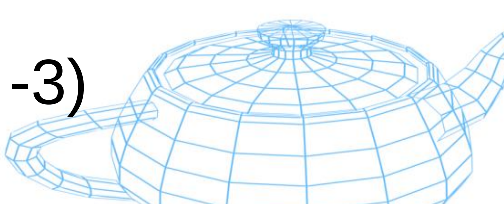


a = (0, 1, -3)

b = (1, -1, -3)

c = (-1, -1, -3)

eye^{-1} = (-5, -1, -3)



Batch transformations

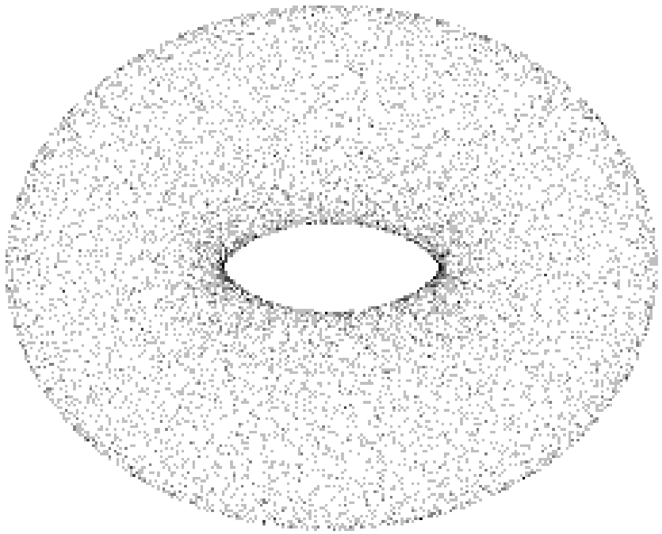
- Given the same three transformations $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_3$ and a list of points $\mathbf{v}_{p(\mathbf{1-1000})}$ it is more efficient to:

  1) compute the final matrix $\mathbf{T}_f = \mathbf{T}_3\mathbf{T}_2\mathbf{T}_1$ just once, then

  2) multiply the 1000 points using:

$$\text{for p=1 to 1000}$$
$$\mathbf{v}_n = \mathbf{T}_f\mathbf{v}_p$$

Point cloud



Matrix interpolation

- There's no easy way to smoothly interpolate between two matrices:
  - Such interpolation is very useful in animation.

- Two options:

1) Use **matrix decomposition** on any arbitrary transformation matrix to obtain its scale, rotation, and translation parameters, which you can linearly interpolate and concatenate back into a matrix.

2) Use **quaternions** (but they only work for rotations).

Quaternions (1843)

William Rowan Hamilton
1805 - 1865

- Generalization of complex numbers.

- A quaternion is a four-tuple:

$$\mathbf{q} = (x, y, z, w) = w + xi + yj + zk$$

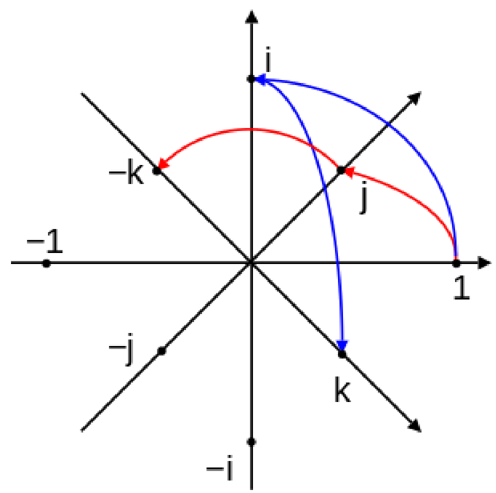
where:

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = k$$

$$ji = -k$$

Not commutative (in general)

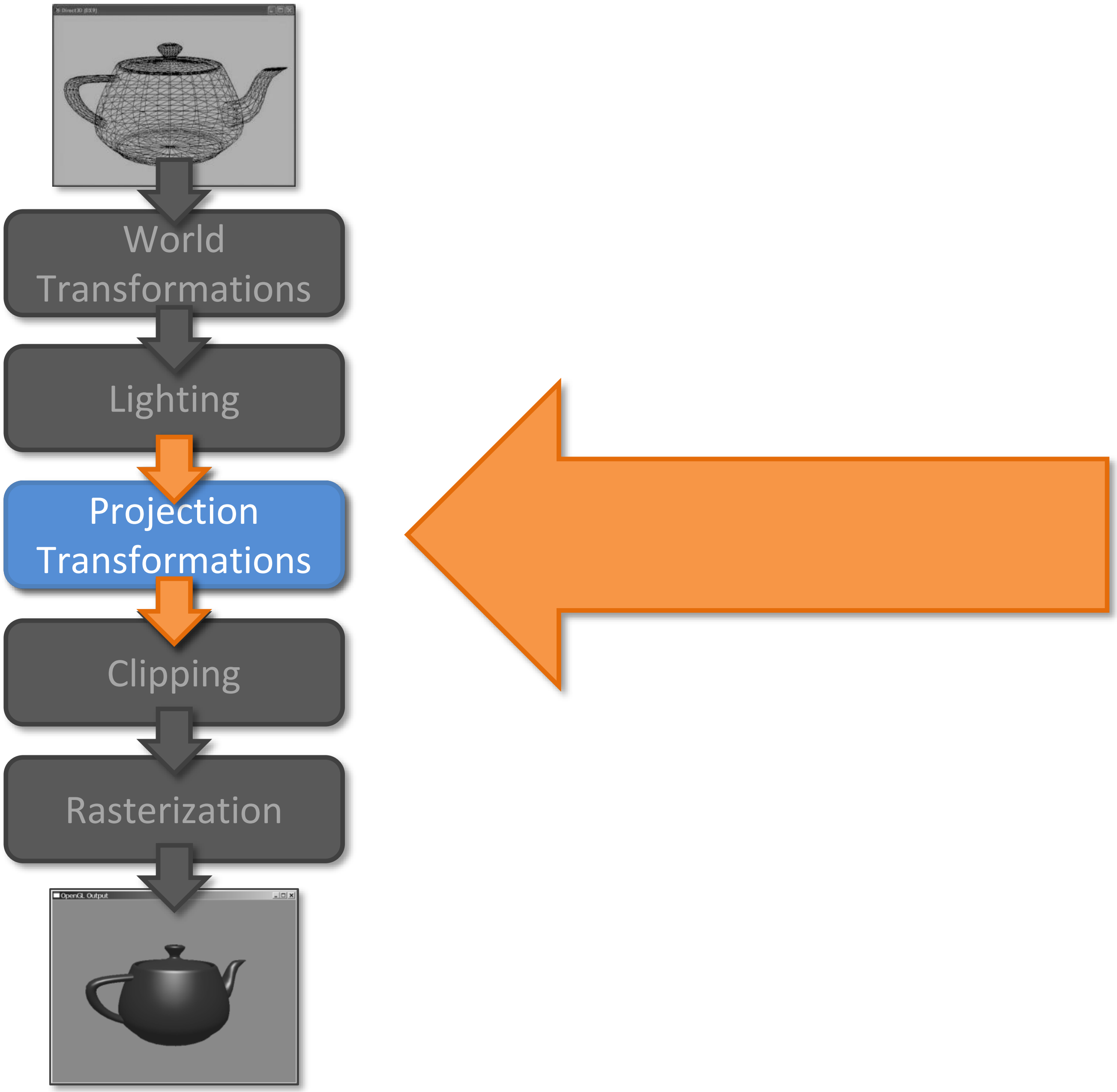


ij = k
ji = −k
ij = −ji

Quaternions

- Quaternions are useful for interpolating rotations between two matrices during animation.

- To do so:
  - Convert the two matrices into quaternions.
  - Use **s**pherical **l**inear int**erp**olation (slerp) to interpolate between the two quaternions.
  - Convert the resulting quaternion back to a matrix.


World Transformations
Lighting
Projection Transformations
Clipping
Rasterization

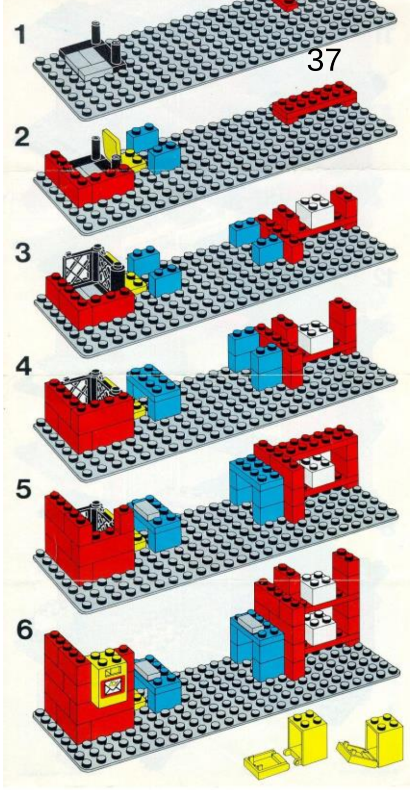
Coordinate spaces

- **Clip coordinates:**
  - Intermediate step before the divide step:
    - Projection matrix * eye coordinates = clip coordinates.
  - The goal of the projection matrix is to setup the *w* component…
    - …for the following division of *x*, *y*, *z* by *w*.
    - …for the normalization of *x*, *y*, *z*.

Projections

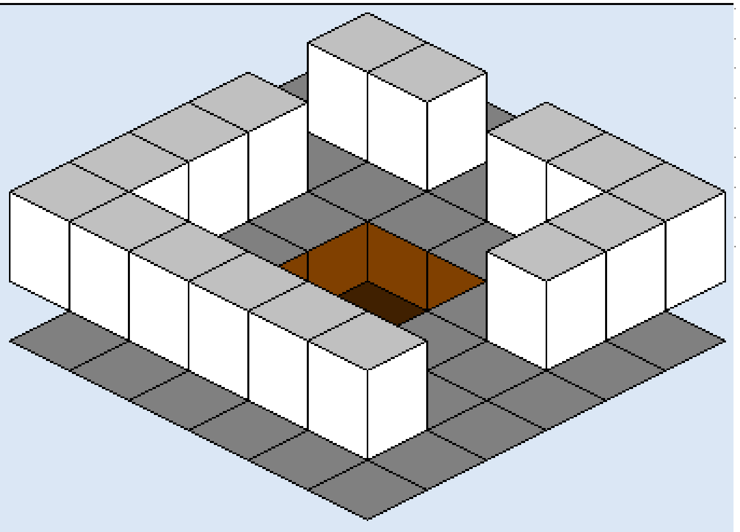
- Two main types of projection (matrices):
  - Orthographic.
  - Perspective.

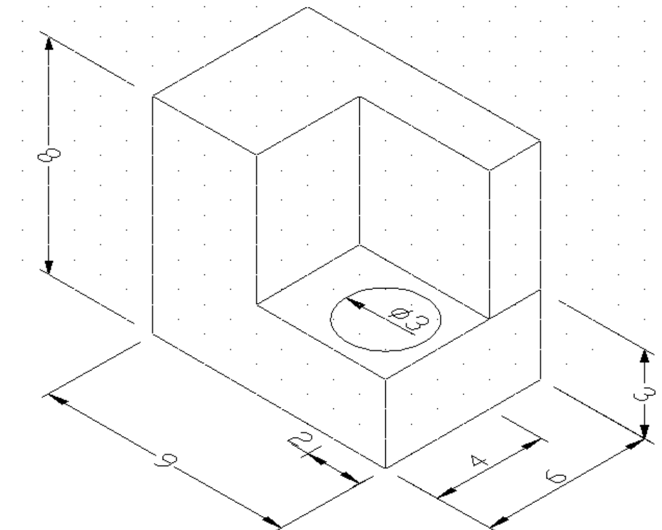
- Other kinds of projection are difficult to implement (e.g., fish-eye).

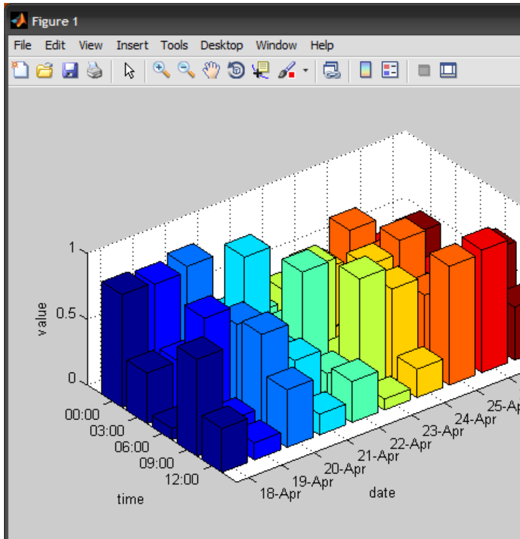


Orthographic projection

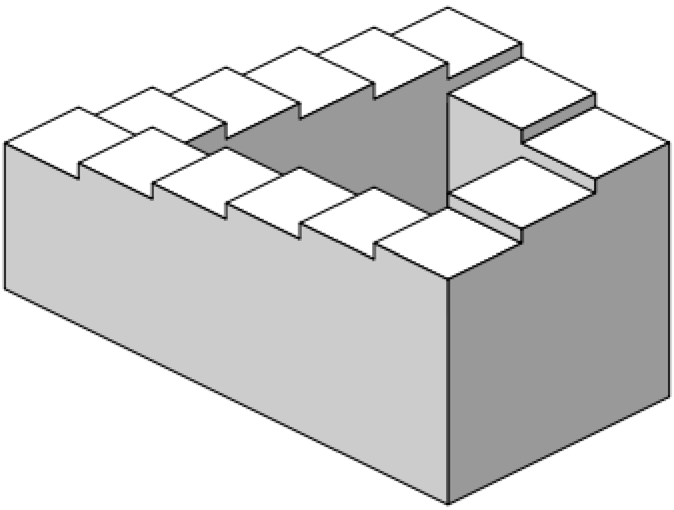
- Distant objects appear with the same size, no perspective:
  - The clipping space is a cube (and not a truncated pyramid).
  - Useful for drawing 2D graphics, diagrams, blueprints, CAD tools, etc.

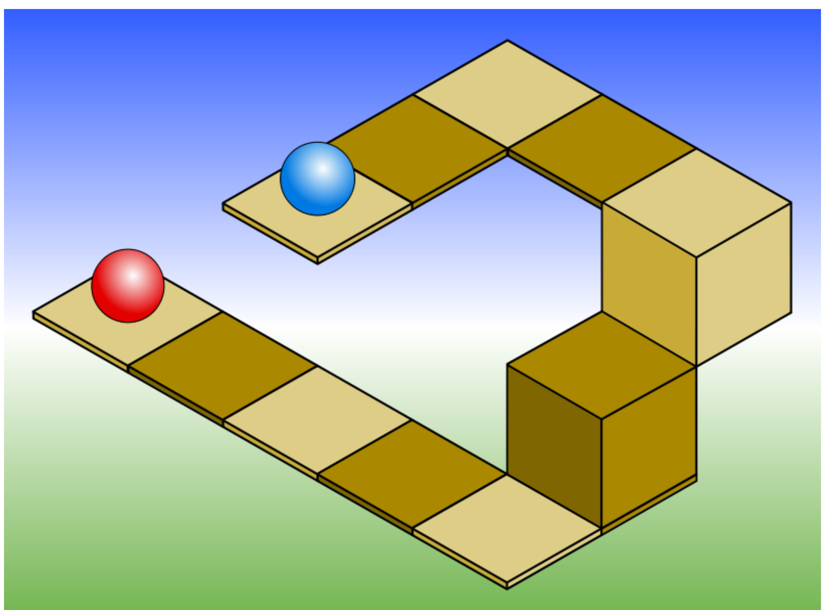






Orthographic projection limitations






SIMCITY SOCIAL
FOR SALE
FOR SALE
FOR SALE
FOR SALE
FOR SALE
HOTEL
HOTEL
SUPERMARKET
SIM CORP
ATM


Rene Guerin
61
66
42
64


USING
HOLDING
POWER
021940
00:03:33

https://youtu.be/Me4ymG_vnOE

Orthographic projection

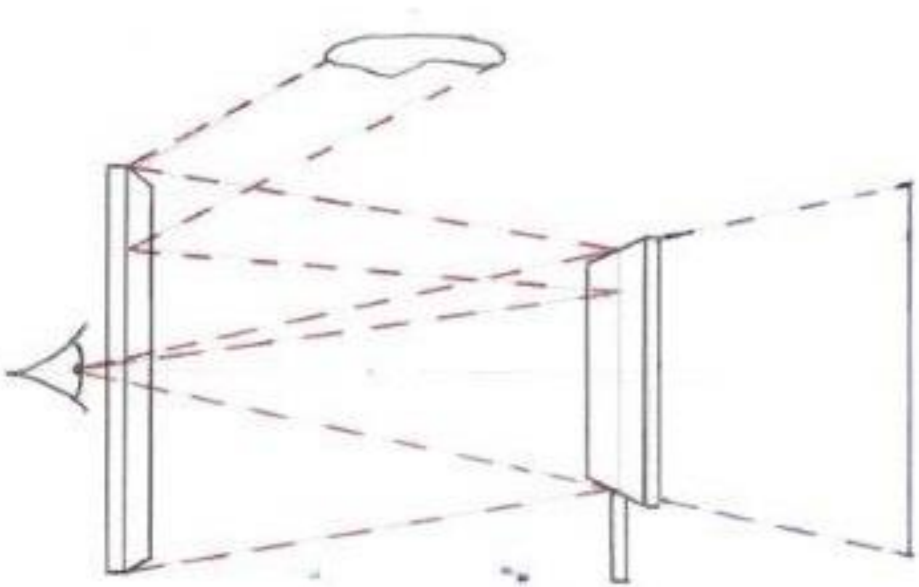
z is inverted!

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

(as defined in *glOrtho* and *gluOrtho2D*)

- The orthographic projection is basically a scaling of the scene into the clipping space.

Perspective (~1413)



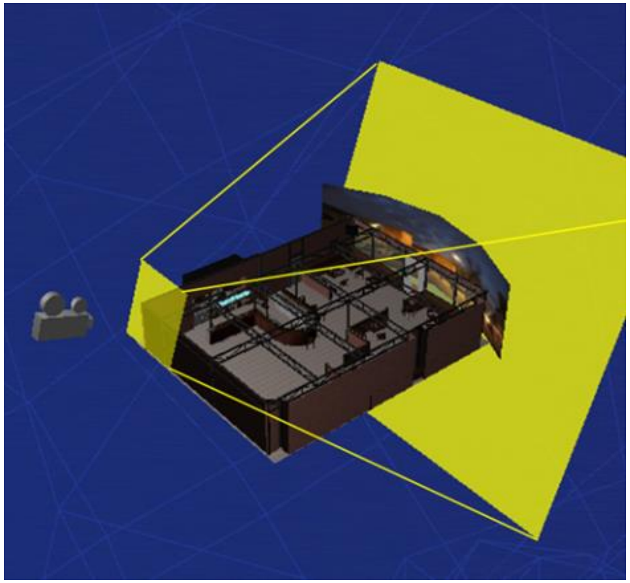
Filippo Brunelleschi
1377 - 1446

Città ideale, anon., ~1480-90

Andrea Pozzo, church of S. Ignazio, Rome (1691-1694)

Perspective projection

- The *w* component of each vertex increases with its distance from the near plane:
  - Division of *x*, *y*, *z* by *w* is done just after.
- Points converge to the center according to their distance.
- The clipping space is a truncated pyramid.



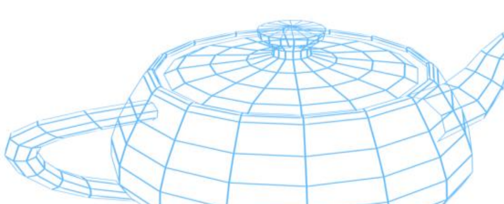
Perspective projection

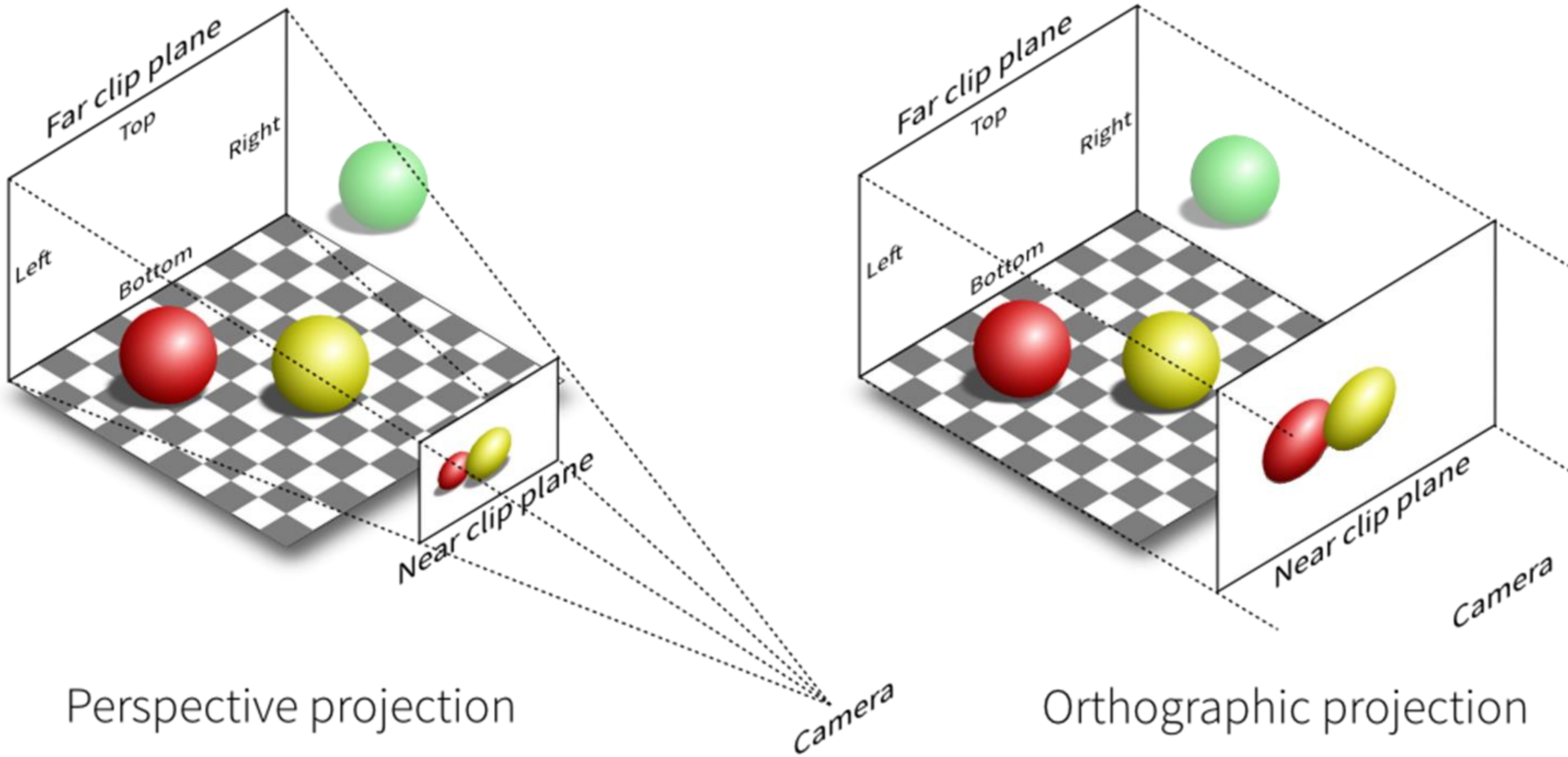
z is inverted and copied to w!

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{far + near}{near - far} & \frac{2 \times far \times near}{near - far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

fieldOfView	vertical (y) view angle
f	cotangent(*fieldOfView*/2)
aspect	aspect ratio (4:3, 16:9, etc.)

(as defined in *gluPerspective*)



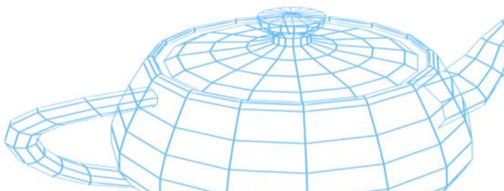


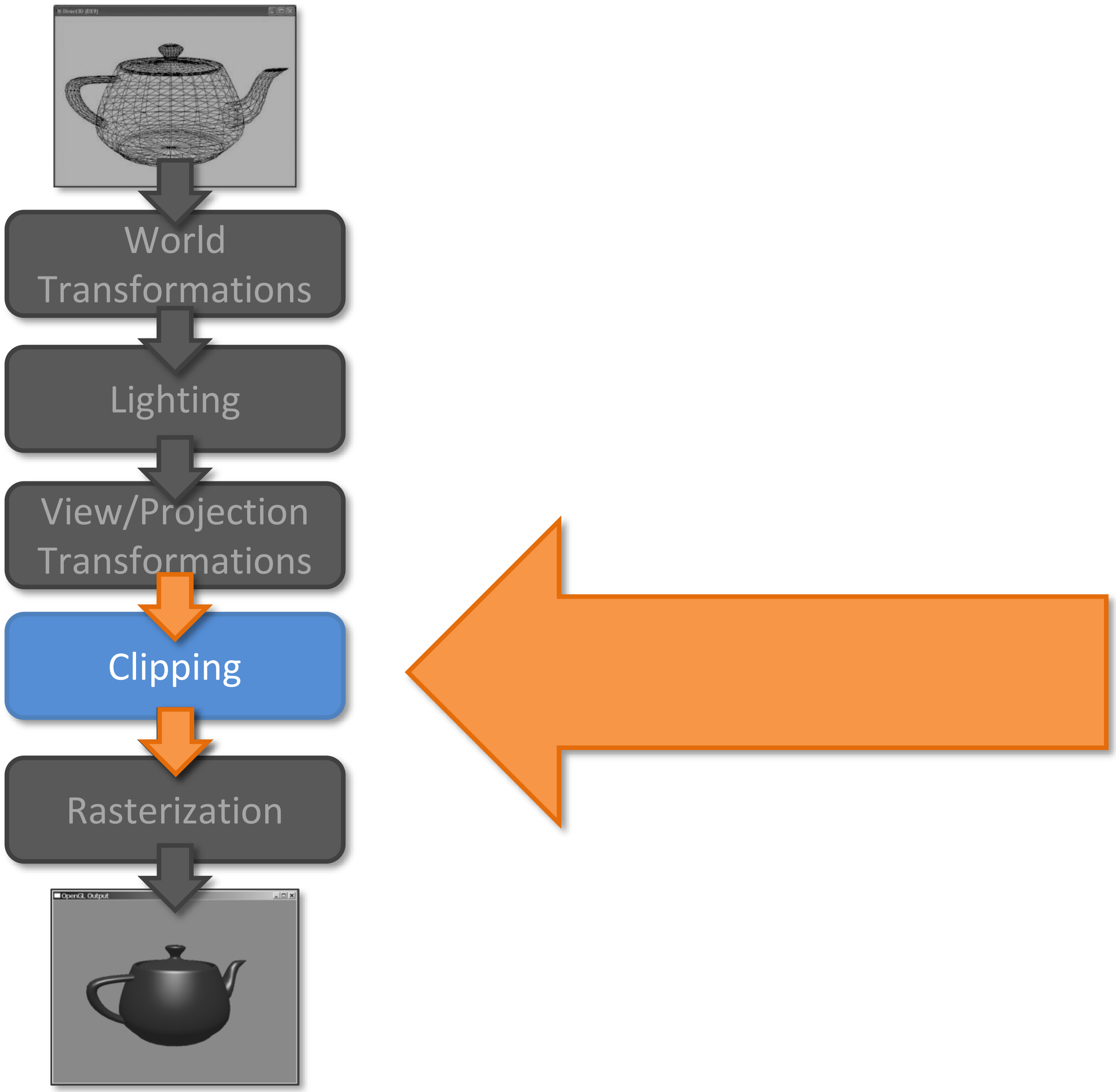
(source: Nicolas Rougier [modified])

So far…

clip coords — eye coords — world coords — object coords

$$\begin{bmatrix} clip_x \\ clip_y \\ clip_z \\ clip_w \end{bmatrix} = \text{projMat} * \text{cameraMat}^{-1} * \overbrace{\text{transMat} * \text{rotMat} * \text{scaleMat}}^{\textit{at will and in any order…}} * \begin{bmatrix} obj_x \\ obj_y \\ obj_z \\ 1 \end{bmatrix}$$




World Transformations
Lighting
View/Projection Transformations
Clipping
Rasterization

Coordinate spaces

- **Normalized device coordinates:**
  - 4D → 3D:
    - Clip coordinates *x*, *y*, *z* divided by *w*.
  - In the range (-1, -1, -1) to (1, 1, 1): vertices not within this range are clipped.
  - *z* coordinate is still present.

Coordinate spaces

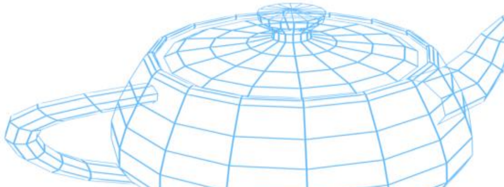
- **Screen/window coordinates:**
  - Final XY(Z) pixel coordinates:
    - Viewport transformation * normalized device coordinates = screen pixels
    - Z used for z-buffer and perspective-correct texture mapping.

$$x_{sc} = (x_{ndc}+1) \times \frac{screenWidth}{2}$$

$$y_{sc} = (y_{ndc}+1) \times \frac{screenHeight}{2}$$

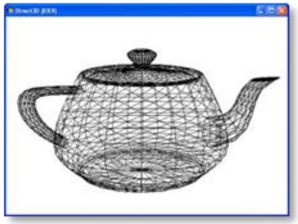
$$z_{sc} = \frac{z_{ndc} + 1}{2}$$

ndc = normalized device coordinates
sc = screen coordinates



Full pipeline

$$\begin{bmatrix} clip_x \\ clip_y \\ clip_z \\ clip_w \end{bmatrix} = \text{projMat} * \text{cameraMat}^{-1} * \text{transMat} * \text{rotMat} * \text{scaleMat} * \begin{bmatrix} obj_x \\ obj_y \\ obj_z \\ 1 \end{bmatrix}$$



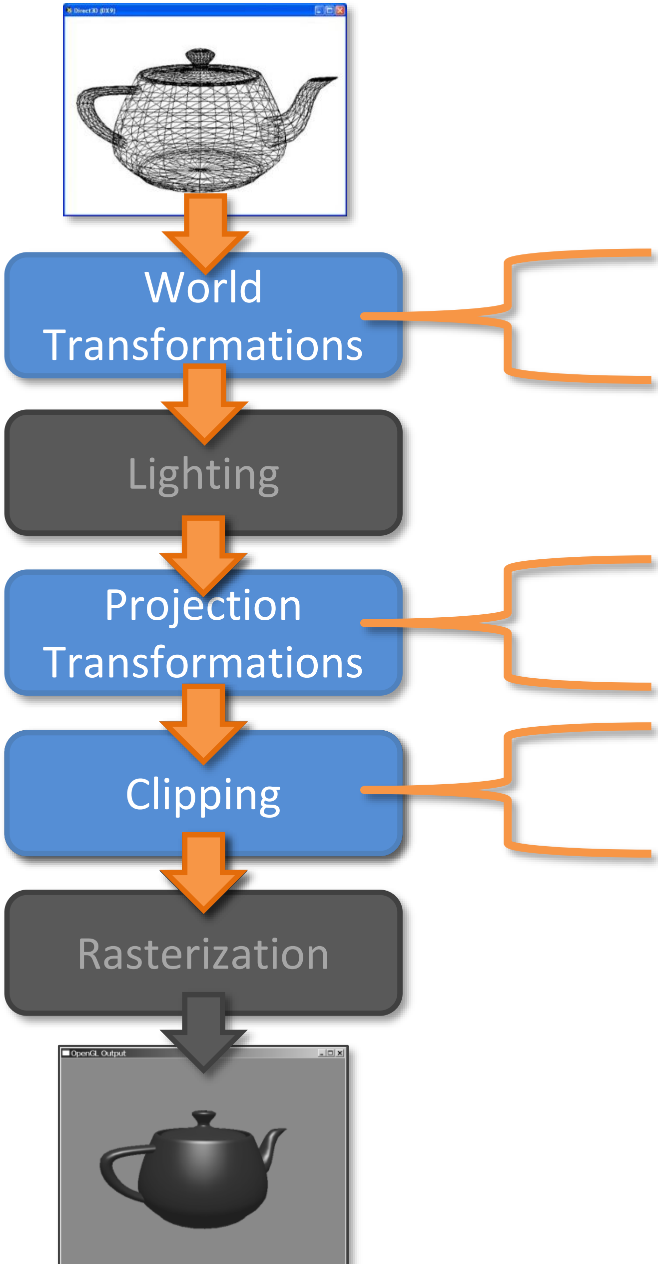
$$division\ by\ W$$

$$x_{sc} = (x_{ndc}+1) \times \frac{screenWidth}{2}$$

$$y_{sc} = (y_{ndc}+1) \times \frac{screenHeight}{2}$$

$$z_{sc} = \frac{z_{ndc}+1}{2}$$

Coordinate spaces (summary)



Object/model coordinates
World coordinates
Eye/view/camera coordinates

Clip coordinates
(Division by w)

Normalized device coordinates
Screen/window coordinates

An in-depth summary on OpenGL coordinate spaces:
http://unspecified.wordpress.com/2012/06/21/calculating-the-gluperspective-matrix-and-other-opengl-matrix-maths/

Camera demo

$$\textbf{Perspective} = \begin{bmatrix} \dfrac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \dfrac{zFar + zNear}{zNear - zFar} & \dfrac{2 \times zFar \times zNear}{zNear - zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

GLM

- Transformations:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

int foo()
{
  glm::vec4 v_obj = glm::vec4(glm::vec3(0.0f), 1.0f);
  glm::mat4 M_rot = glm::rotate(glm::mat4(1.0f),
                                glm::radians(90.0f),
                                glm::vec3(0.0f, 1.0f, 0.0f));
  glm::mat4 M_trans = glm::translate(glm::mat4(1.0f),
                                     glm::vec3(10.0f, 0.0f, 0.0f));
  glm::vec4 v_world = M_trans * M_rot * v_obj;

  return 0;
}
```

GLM

- Constants:

```
#include <glm/glm.hpp>
#include <glm/gtc/constants.hpp>

double squarePi()
{
   return glm::pi<double>() * glm::pi<double>();
}
```

GLM

- Quaternions:

```
#include <glm/glm.hpp>

int foo2()
{
   glm::mat4 M1 = …
   glm::mat4 M2 = …
   glm::quat qM1 = glm::quat(M1);
   glm::quat qM2 = glm::quat(M2);
   glm::quat qR = glm::slerp(qM1, qM2, 0.5f);
   glm::mat4 R = glm::mat4(quat_R);

   return 0;
}
```

GLM

- OpenGL (thus GLM) accesses matrices in column-major order, e.g.:

$$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

← *in the documentation*

- But C arrays are stored in row-major order:

```
glm::mat4 mat( a, b, c, d,
               e, f, g, h,
               i, j, k, l,
               m, n, o, p );
```

← *in the code*