Lecture 8-10: Transformations View, Projections and Instancing

CGRA 354 : Computer Graphics Programming

Instructor: Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

With slides from: Holly Rushmeier, Yale; Steve Marschner, Cornell; Taehyun Rhee, CMIC; Zohar Levi, ECS

Next six lectures

- Lighting continued and linear algebra recap
- *Transformations*
- Projections
- Instancing
- Textures
- Animation started

Recap: Vectors

Basic operations:

$$\bar{a} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x = \begin{pmatrix} 1+x \\ 2+x \\ 3+x \end{pmatrix} \qquad -\bar{a} = -\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -x \\ -y \\ -z \end{pmatrix}$$

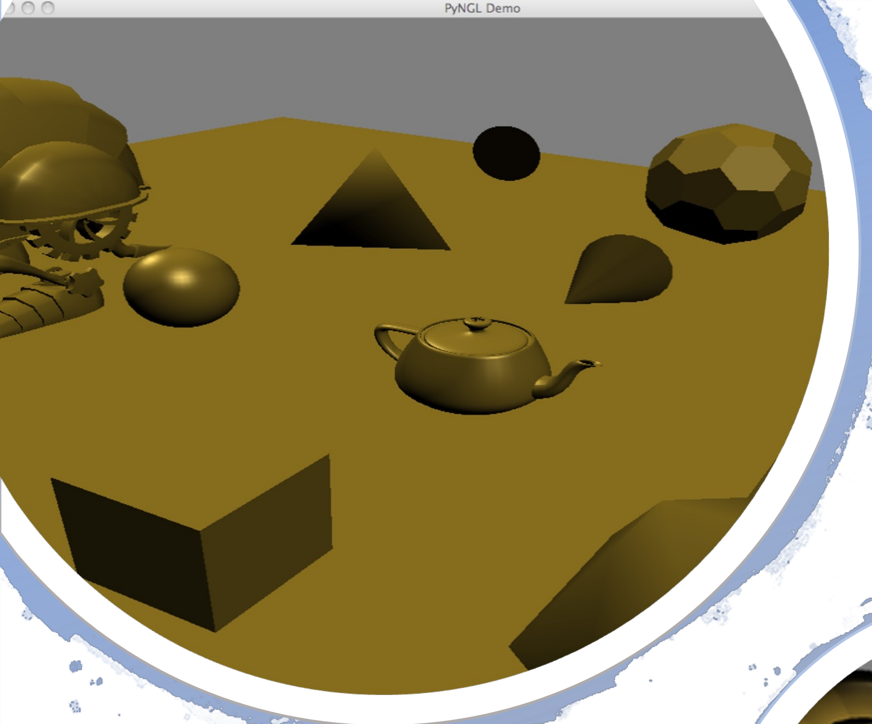
$$\bar{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \bar{b} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \rightarrow \bar{a} + \bar{b} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

$$||\bar{a}|| = \sqrt{x^2 + y^2 + z^2} \qquad \mathbf{b} = \frac{\mathbf{a}}{||\mathbf{a}||}$$

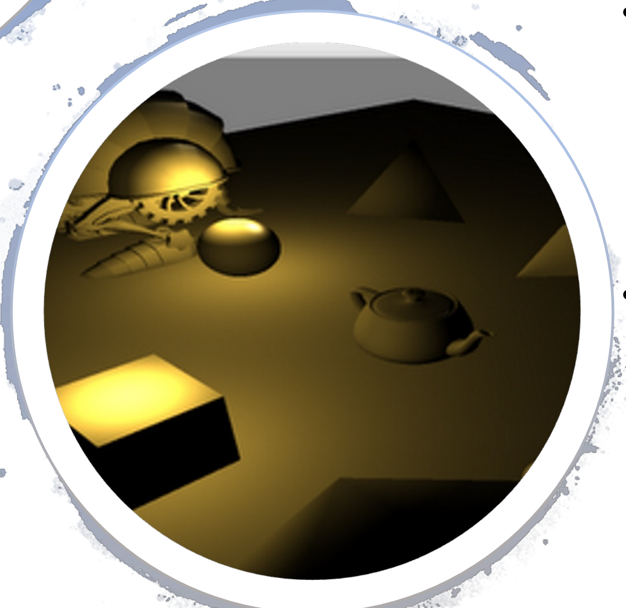
Dot and Cross product:

$$\mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}||\ ||\mathbf{b}|| \cos\theta \qquad \begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 * 0) + (-0.8 * 1) + (0 * 0) = -0.8 \qquad \mathbf{b} \rightarrow \mathbf{a} = ||\mathbf{b}|| \cos\theta = \frac{\mathbf{b} \cdot \mathbf{a}}{||\mathbf{a}||}$$

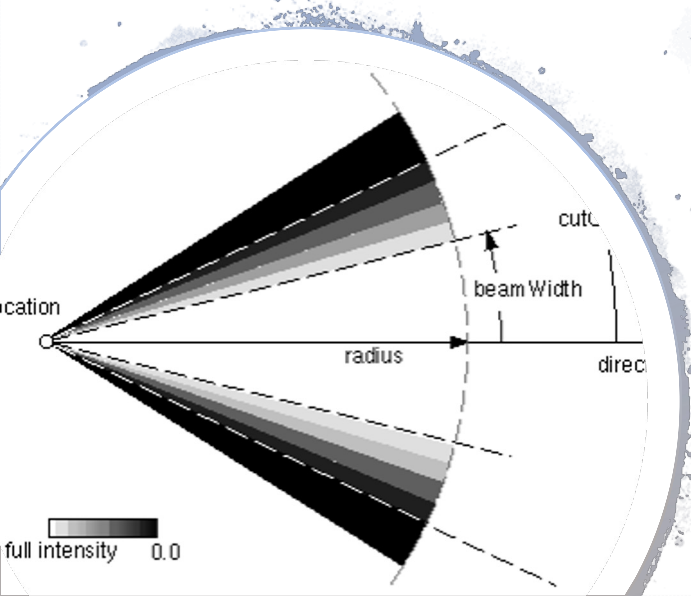
$$||\mathbf{a} \times \mathbf{b}|| = ||\mathbf{a}||\ ||\mathbf{b}|| \sin\theta \qquad \begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix}$$



Recap: Light Source Models



- Simple mathematical models:
  - Point Light
  - Directional Light
  - Spot Light
- Two other light properties
  - Ambient Light
  - Emission

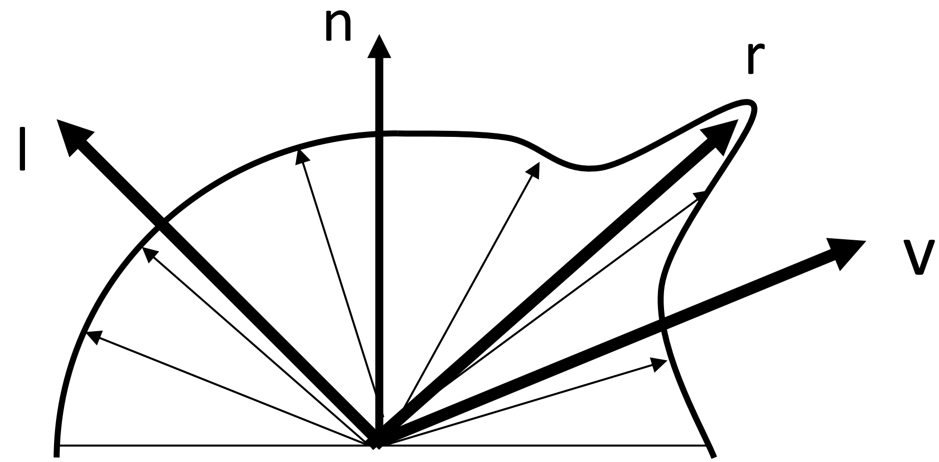


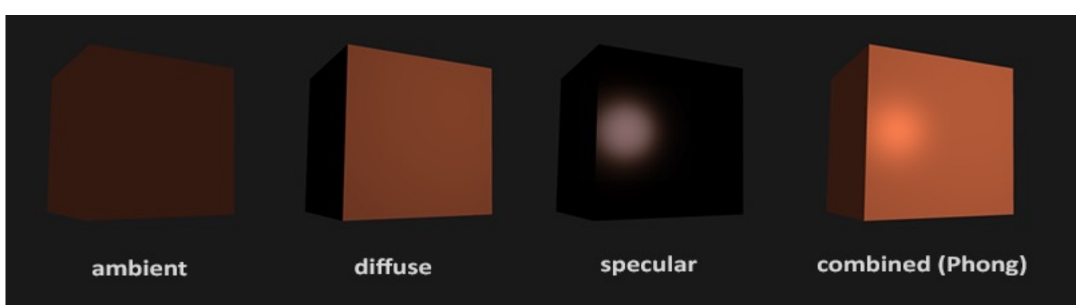
Phong Model in OpenGL

- Phong illumination model is combination of
  - Ambient i_{amb}+ Diffuse i_{diff} + Specular terms i_{sepc}
  - Developed by Bui Tuong Phong at Univ. Utah 1973

$$\mathbf{I} = k_a i_a + k_d i_d (\mathbf{n} \bullet \mathbf{l}) + k_s i_s (\mathbf{r} \bullet \mathbf{v})^{m_{shi}}$$

  - k_a k_d k_s are material properties having RGB components



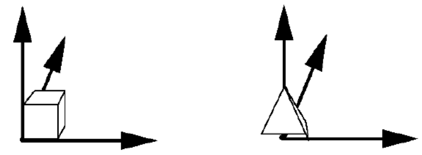


Transformations outline

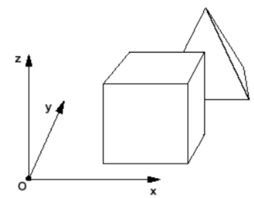
- *Intro to Transformations*
- Classes of Transformations
- Representing Transformations
- Combining Transformations

Common Coordinate systems

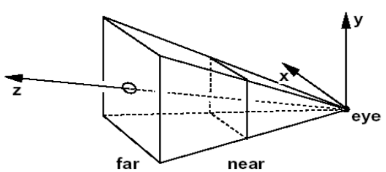
- Object space
  - local to each object



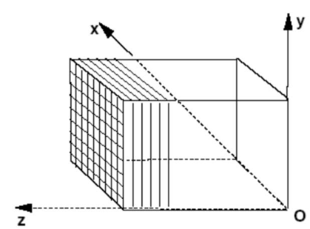
- World space
  - common to all objects



- Eye space / Camera space
  - derived from view frustum

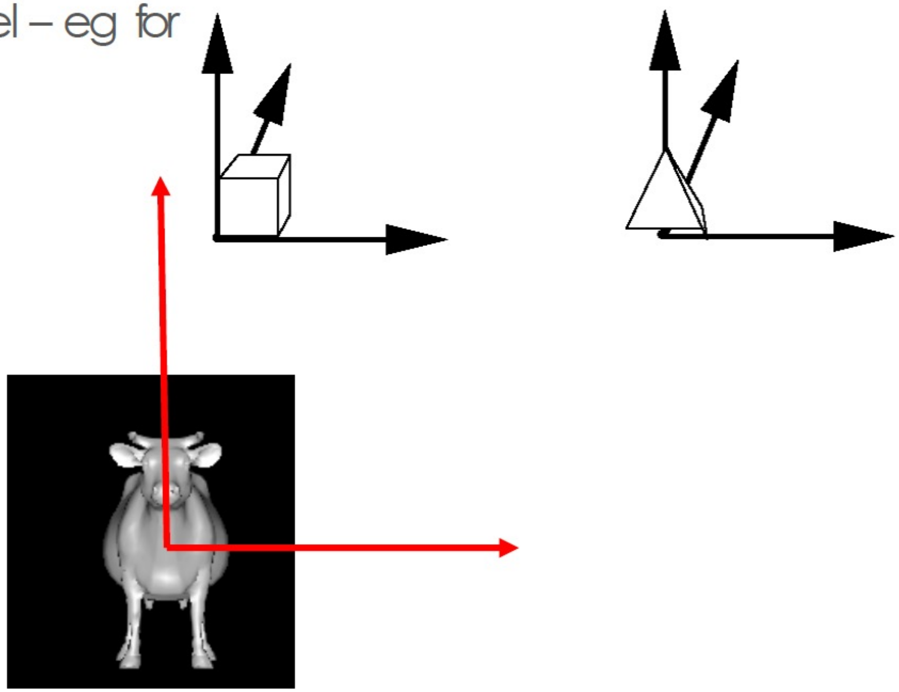


- Screen space
  - indexed according to hardware attributes



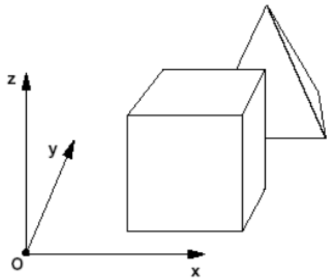
Object space

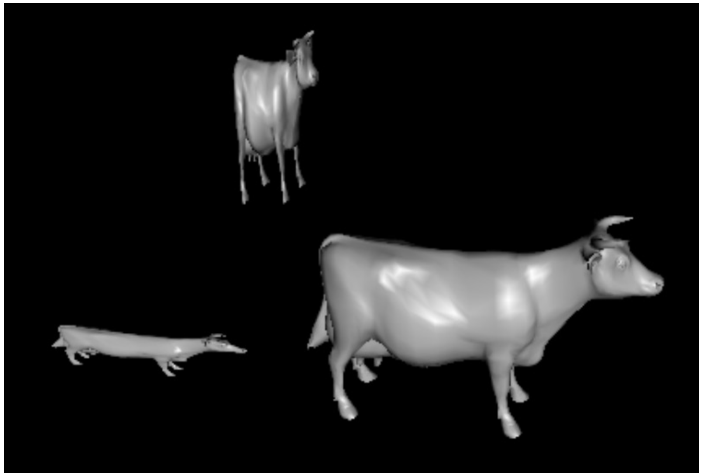
- Coordinate system convenient for model – eg for symmetry



World space

Objects placed in scene





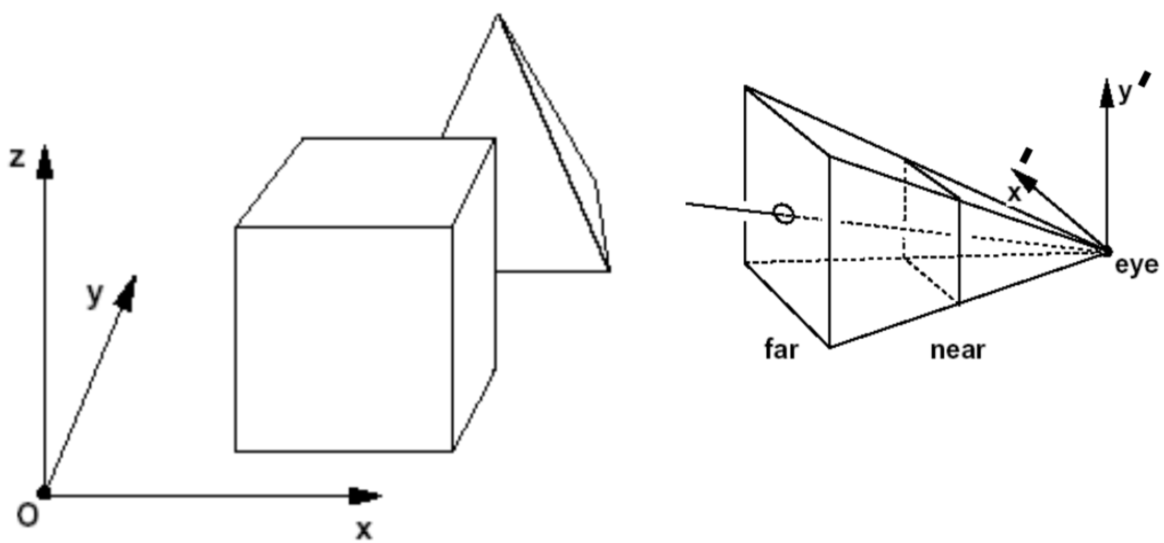
```
#VRML V2.0 utf8
    Transform{
      translation 5 0 0
      scale 2 2 2
      children[Inline { url "cow.wrl"}, ]}

    Transform{
      translation -5 0 0
      scale 1.5 .5 .5
      children[ Inline {url "cow.wrl"}, ]}

Transform{
      translation 0 6 0
      scale .5 1.5 1.5
      children[ Inline {url "cow.wrl"}, ]}
```

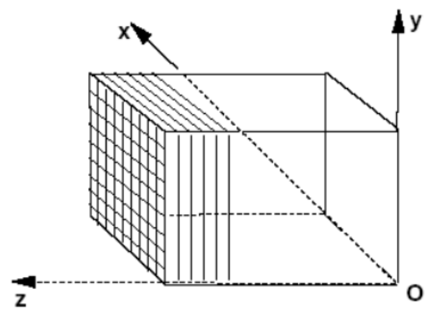
Eye Space

Eye is located inside the world, would be convenient to transform to its coordinate system.



Screen Space

Pixel locations, and a coordinate to sort depth

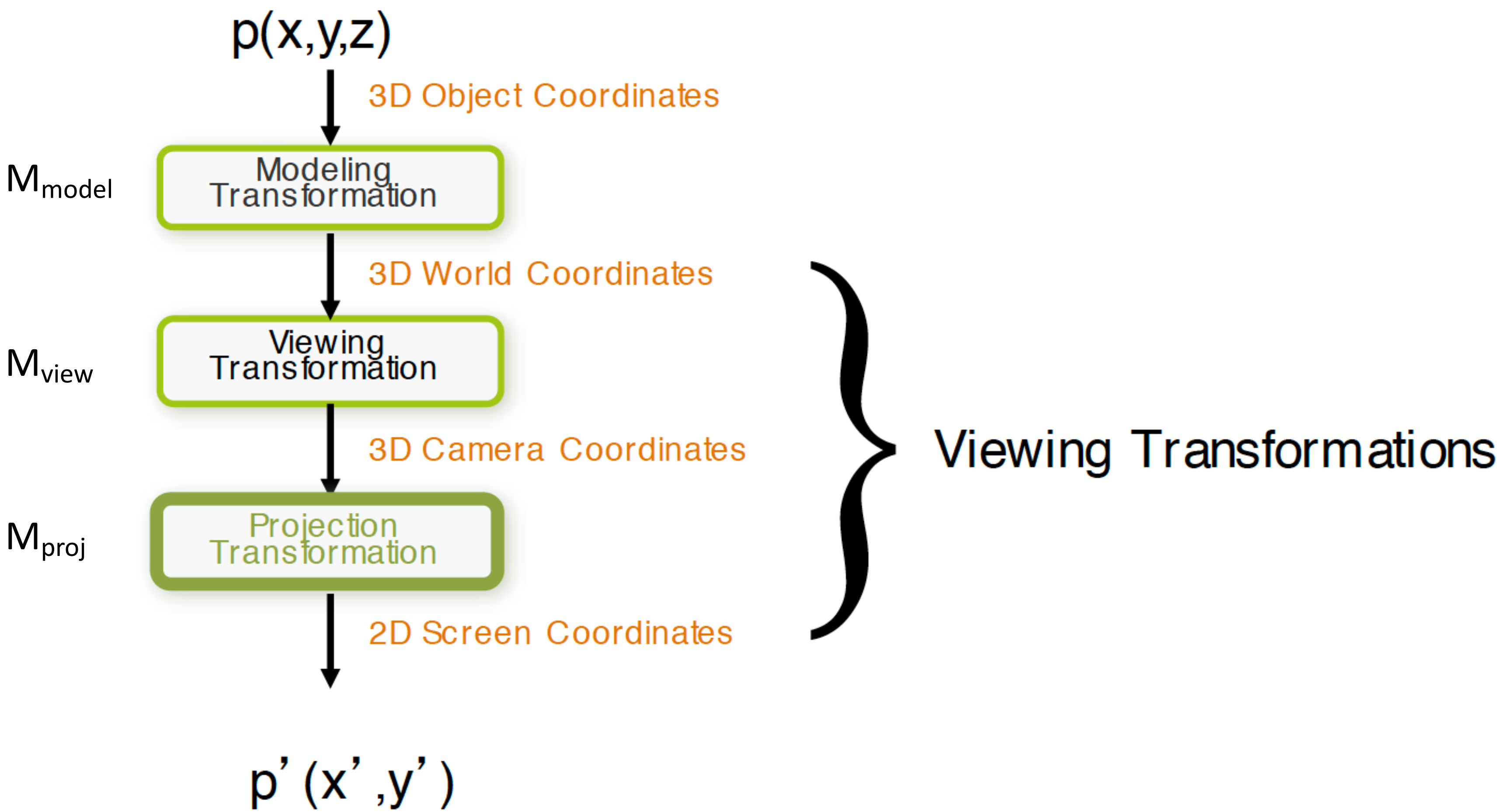


Transformations are used

- Position objects in a scene
- Reuse/change the shape of objects
- Create multiple copies of objects
- Hierarchical modeling
- Kinematics
- Animations
- Projections for virtual cameras/viewing

…

Stages of Vertex Transformations

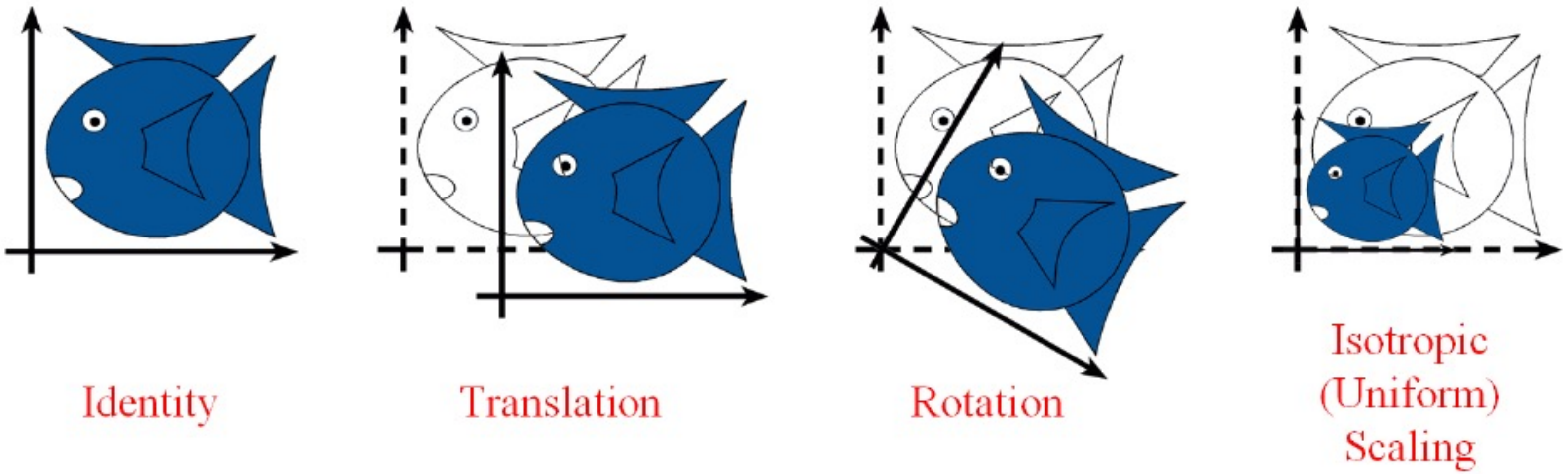


Modeling transformation: scaling FIRST, and THEN the rotation, and THEN the translation.

Classes of Transformations

- *Intro to Transformations*
- *Classes of Transformations*
  - Rigid Body / Euclidean Transforms
  - Similitudes / Similarity Transforms
  - Linear
  - Affine
  - Projective
- Representing Transformations
- Combining Transformations

Common transformations

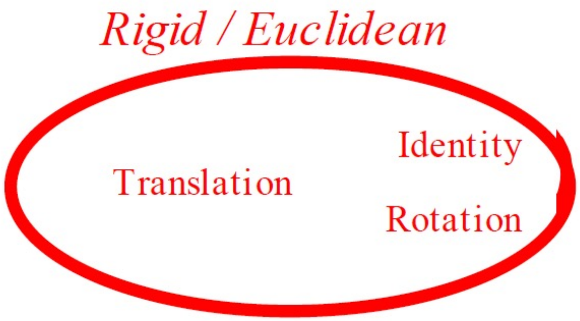


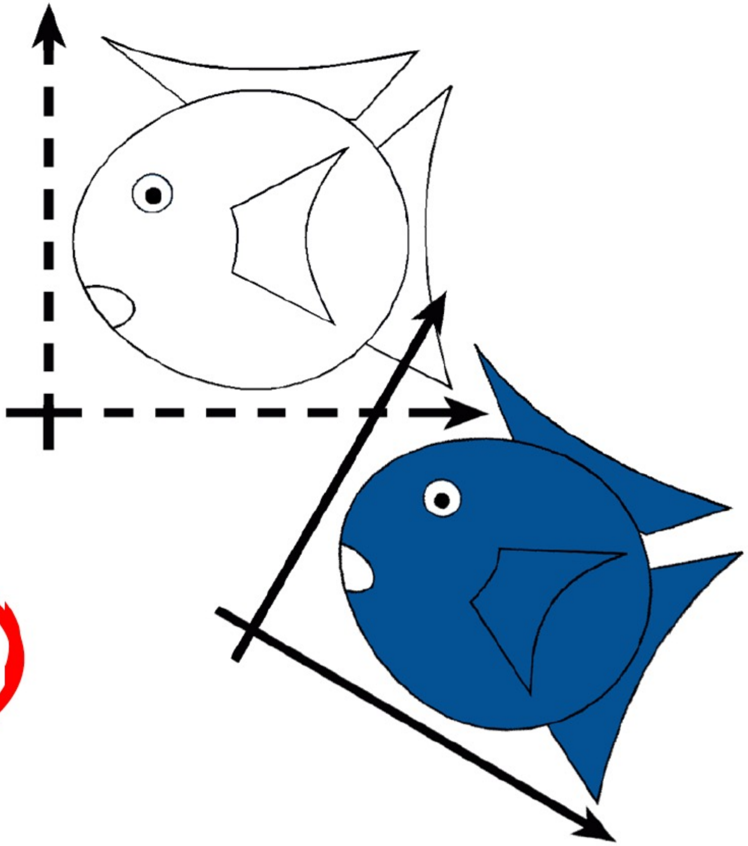
- Can be combined
- Are these operations invertible?

Yes, except scale = 0

Rigid-Body / Euclidean Transforms

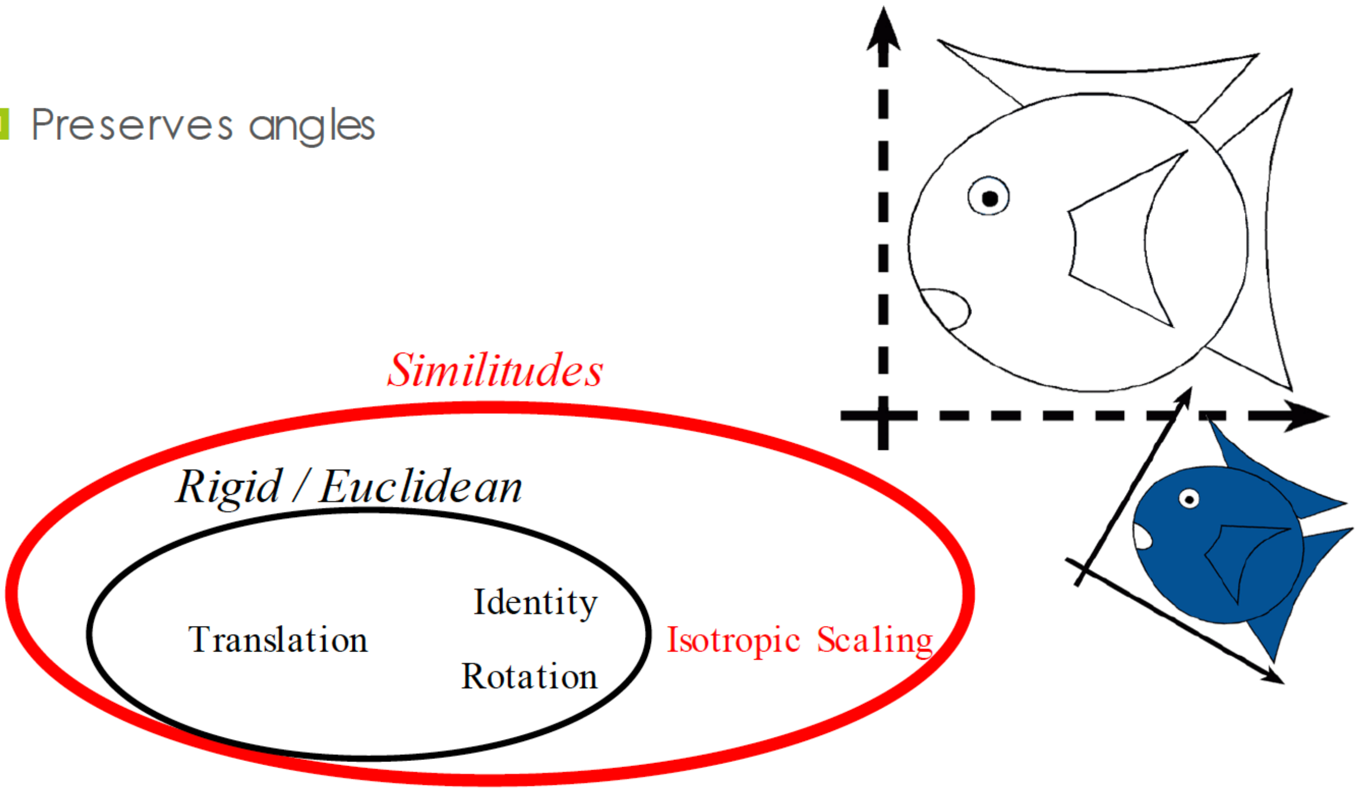
- Preserves distances
- Preserves angles



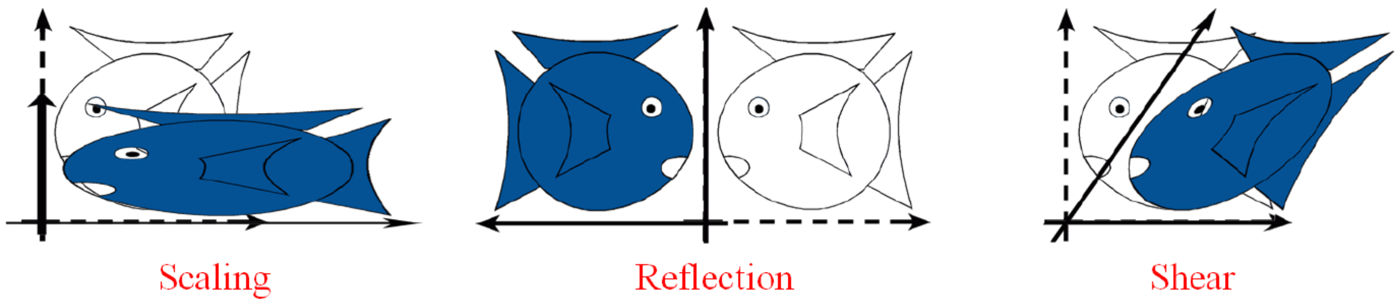


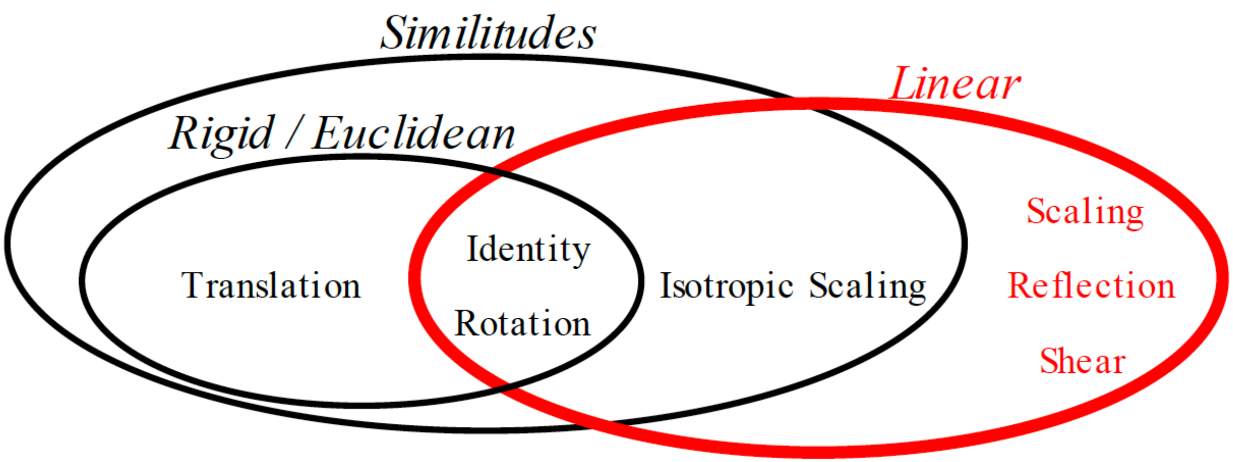
Similitudes / Similarity Transforms

- Preserves angles



Linear Transformations



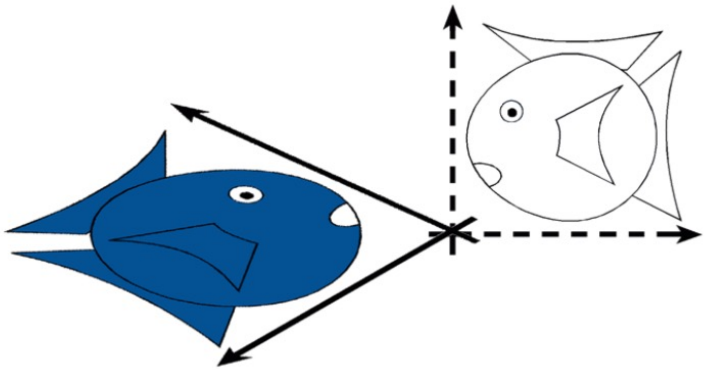


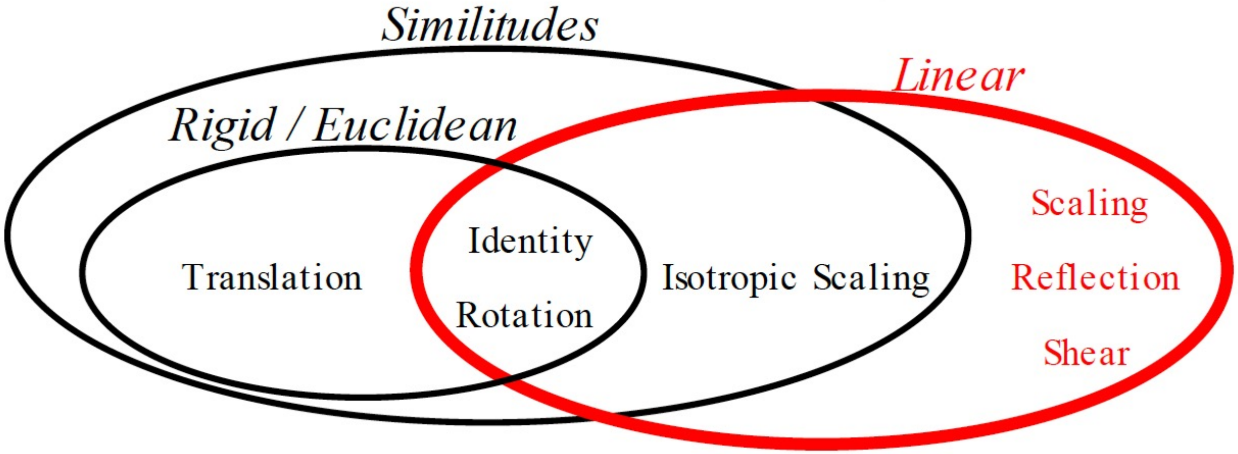
Linear Transformations

- Vectors p, q, scalar a:

  $L(p+q) = L(p) + L(q)$

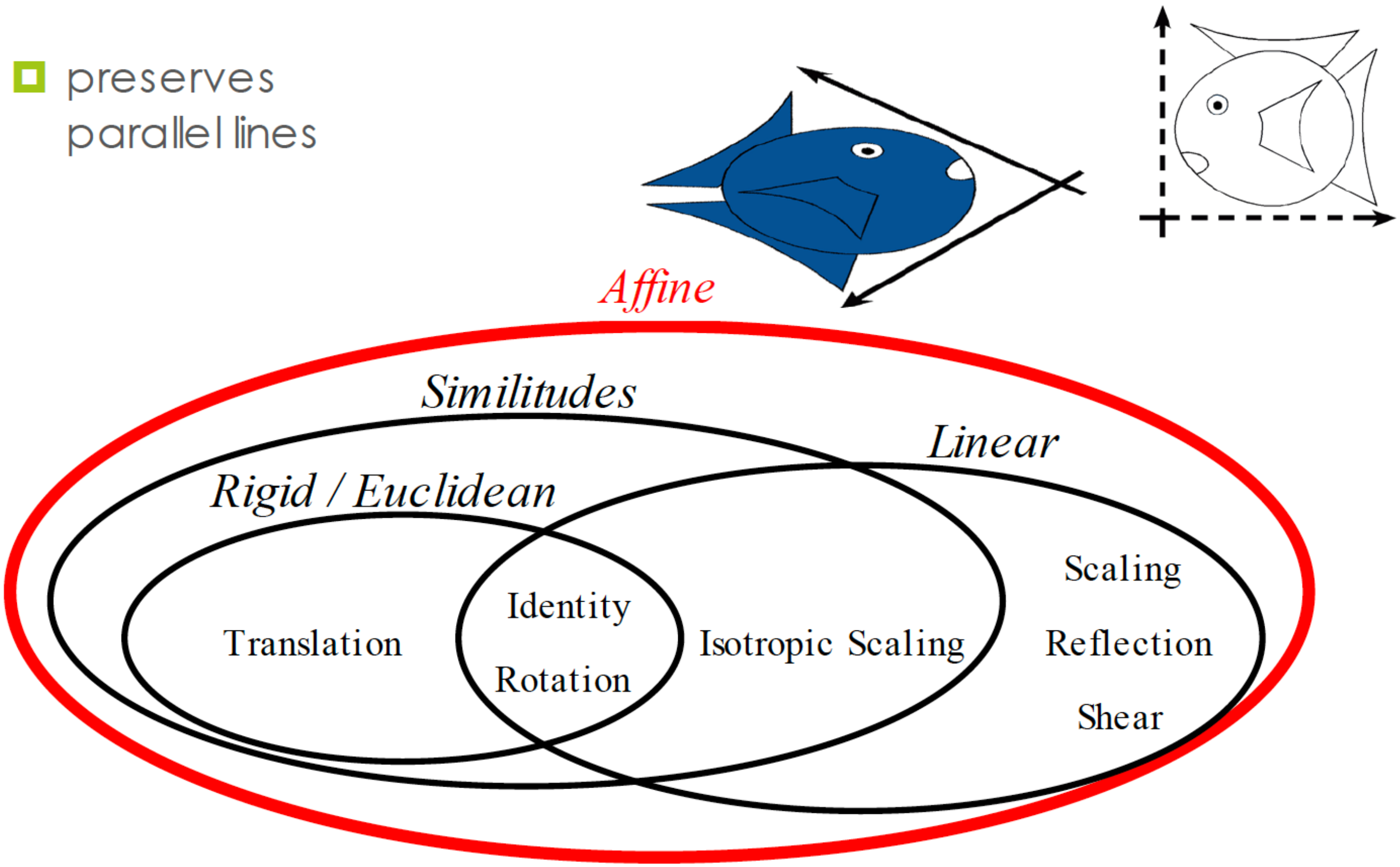
  $L(ap) = aL(p)$





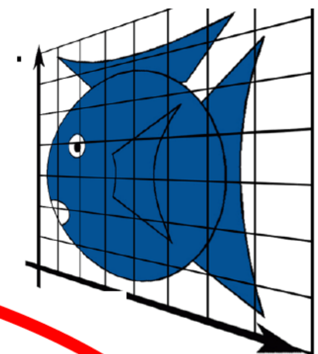
Affine Transformations

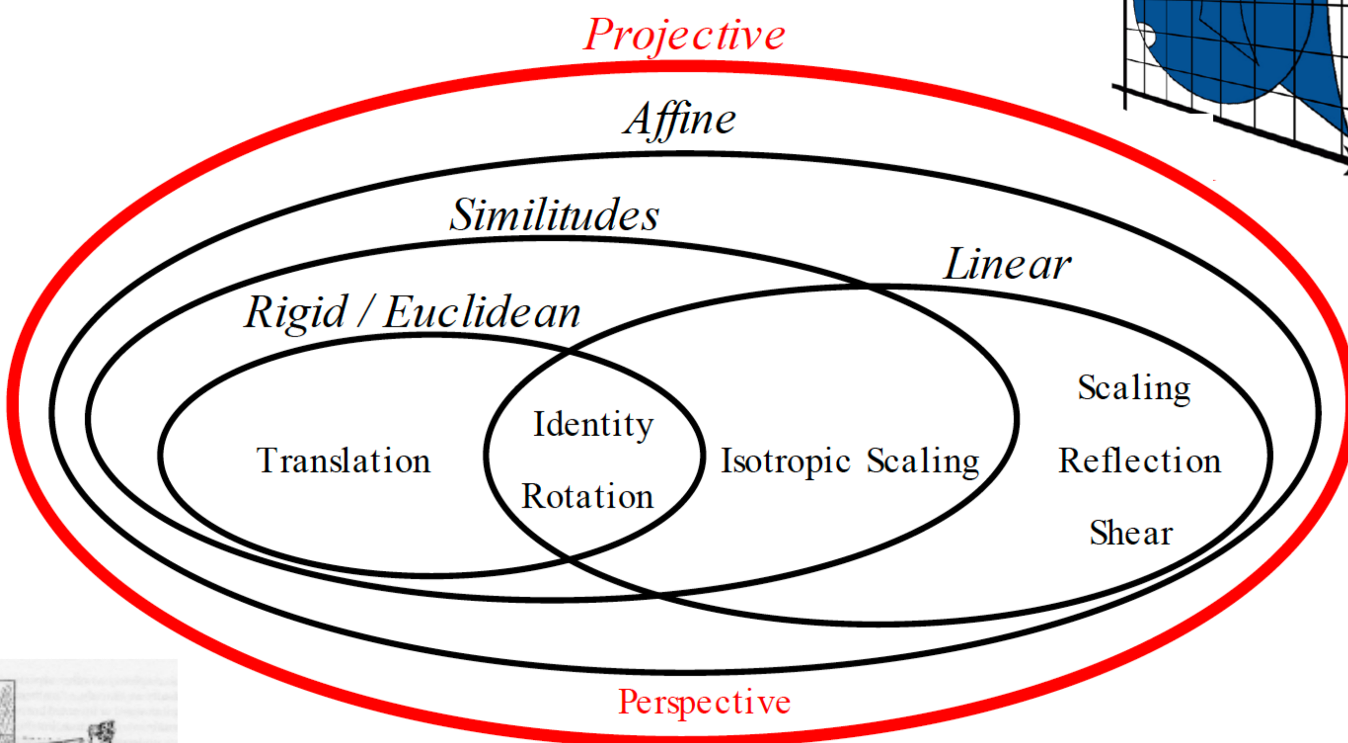
- preserves parallel lines

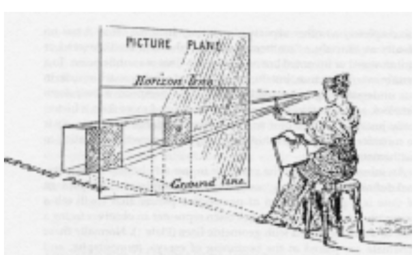


Projective Transformations

- preserves lines







Classes of Transformations

- *Intro to Transformations*
- *Classes of Transformations*
- Representing Transformations
- Combining Transformations

What is a Transformation?

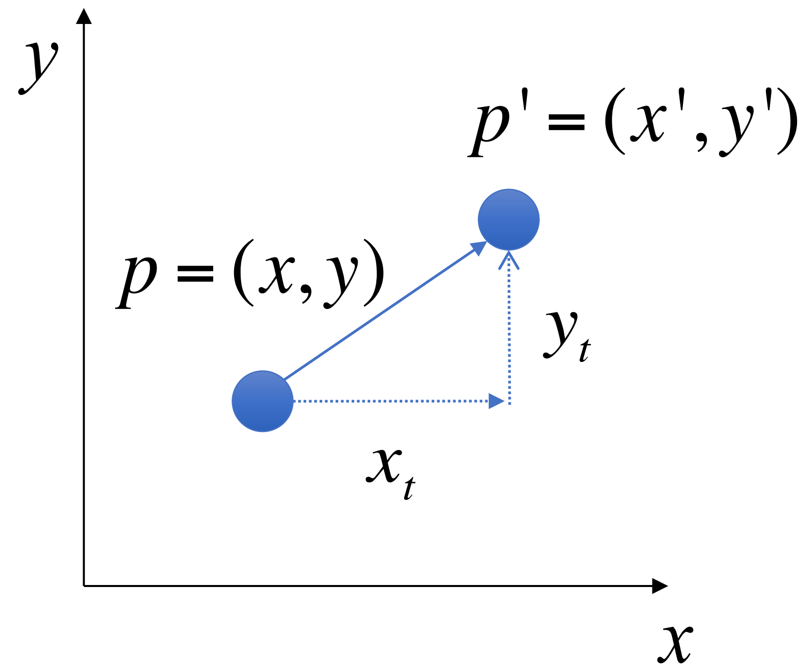
- Maps points (*x*, *y*) in one coordinate system to points (*x*', *y*') in another coordinate system

$$x' = ax + by + c$$
$$y' = dx + ey + f$$

Translations

2D:

- **p' = p + t**
  - **P** = (x, y)
  - **t** = (x_t, y_t)
  - **p'** = ($x+x_t$, $y+y_t$)



3D:

- **p' = p + t**
  - **p** = (x, y, z)
  - **t** = (x_t, y_t, z_t)
  - **p'** = ($x+x_t$, $y+y_t$, $z+z_t$)

Properties of Translations

- Zero identity

$$T(0,0,0)\mathbf{v} = \mathbf{v}$$

- Additive

$$T(s_x, s_y, s_z)\, T(t_x, t_y, t_z)\mathbf{v} = T(s_x + t_x, s_y + t_y, s_z + t_z)\, \mathbf{v}$$

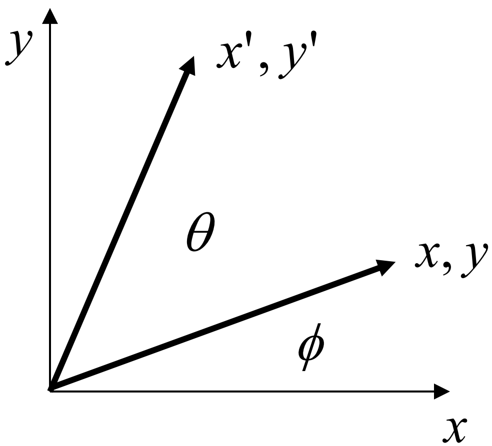
- Commutative

$$T(s_x, s_y, s_z)\, T(t_x, t_y, t_z)\mathbf{v} = T(t_x, t_y, t_z)\, T(s_x, s_y, s_z)\mathbf{v}$$

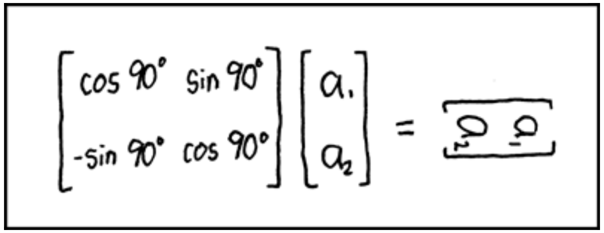
- Inverse

$$T^{-1}(t_x, t_y, t_z)\, \mathbf{v} = T(-t_x, -t_y, -t_z)\, \mathbf{v}$$

Rotations 2D



$$x = r\cos\phi$$
$$y = r\sin\phi$$



$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

$$\cos(\phi + \theta) = \cos\phi\cos\theta - \sin\phi\sin\theta$$
$$\sin(\phi + \theta) = \cos\phi\sin\theta - \sin\phi\cos\theta$$

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

$$x' = (r\cos\phi)\cos\theta - (r\sin\phi)\sin\theta$$
$$y' = (r\cos\phi)\sin\theta + (r\sin\phi)\cos\theta$$

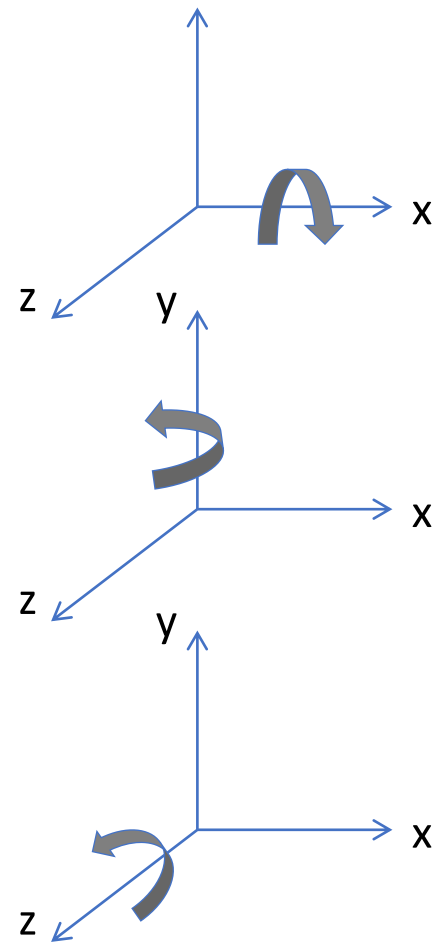
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotations 3D

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

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

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



How are Transforms Represented?

$$x' = ax + by + c$$
$$y' = dx + ey + f$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}$$

$$p' = M p + t$$

Matrices
(please refer to the Supplement A)

Recap: Basic operations

Sum
$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{11}+y_{11} & x_{12}+y_{12} \\ x_{21}+y_{21} & x_{22}+y_{22} \end{bmatrix}$$

Scalar Product
$$y * \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} yx_{11} & yx_{12} \\ yx_{21} & yx_{22} \end{bmatrix}$$

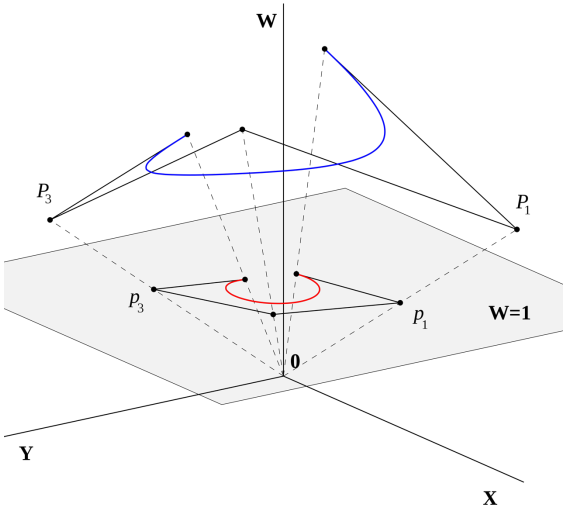
Identity:
$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplication (commutative property does not hold): $\mathbf{AB} \neq \mathbf{BA}$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 5+2\cdot 7 & 1\cdot 6+2\cdot 8 \\ 3\cdot 5+4\cdot 7 & 3\cdot 6+4\cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Homogeneous Coordinates

- Homogeneous coordinates represents N-dimensional coordinates with N+1 number
  [August Ferdinand Mobius]
  - $(x', y')_{Euclidean} \rightarrow (x,y,w)_{homogeneous}$
  - $(x,y,w)_{homogeneous} \rightarrow (x/w, y/w)$, if w=0 it goes to infinity



[Bézier curve, wikipedia]

Why use Homogeneous Coordinates?

- An Euclidean point can be converted into many different points in homogeneous coordinates
  - (1,2,3) = (2,4,6) = (4,8,12) = … = (1a, 2a, 3a)

  →(1/3, 2/3) in Euclidean space

- Advantages
  - Allows perspective transformation to be expressed as a matrix equation
  - Allows rigid transformations to be combined with perspective transformation
  - Allow translation to be expressed as a matrix equation

Translation Revisited

- **p' = p + t**
  - ***T***(x_t, y_t, z_t)

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

- Translation Matrix (4x4)

$$T(x_t, y_t, z_t), \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix} \qquad S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Uniform scaling ***iff*** $s_x = s_y = s_z$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

S

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

$\mathbf{S}^{-1}$

Rotations Revisited

About x axis:

$$R_x(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

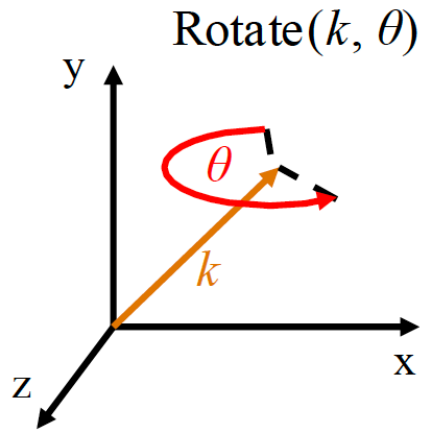
About y axis:

$$R_y(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

About z axis:

$$R_z(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation: arbitrary axis



- About (k_x, k_y, k_z), a unit vector on an arbitrary axis (Rodrigues Formula)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} k_x k_x(1-c)+c & k_z k_x(1-c)-k_z s & k_x k_z(1-c)+k_y s & 0 \\ k_y k_x(1-c)+k_z s & k_z k_x(1-c)+c & k_y k_z(1-c)-k_x s & 0 \\ k_z k_x(1-c)-k_y s & k_z k_x(1-c)-k_x s & k_z k_z(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

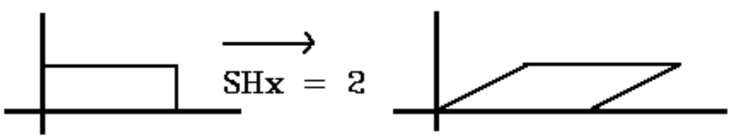
where $c = \cos\theta$ & $s = \sin\theta$

Shearing

- The effect looks like "pushing" an object in a direction parallel to a coordinate axis (2D) or plane
  - How far to push is determined by a sharing factor

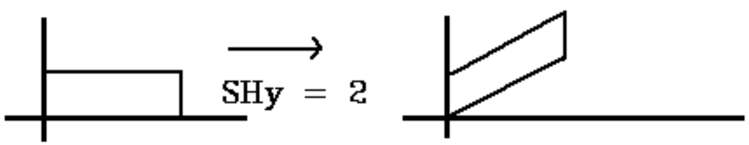
sheep　　sheared sheep

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \begin{aligned} x' &= x + ay \\ y' &= y \end{aligned}$$



x shear with shearing factor a

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \begin{aligned} x' &= x \\ y' &= bx + y \end{aligned}$$



y shear with shearing factor b

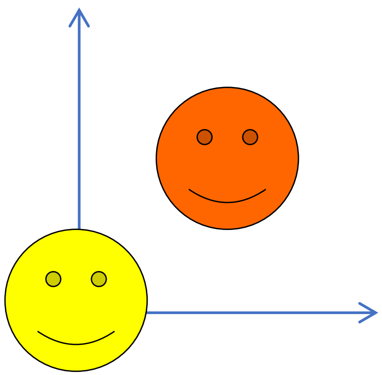
Properties of Affine transform

- A composition of affine transformation is an affine transformation
- Given any two triangles, there exists an affine transformation mapping one to the other

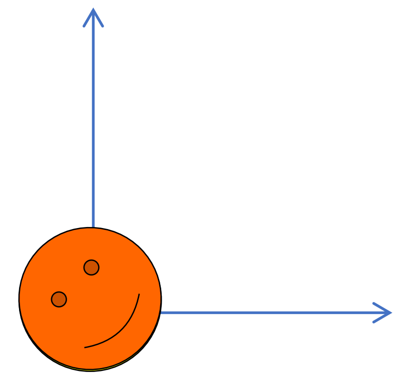
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$m_{30} = m_{31} = m_{31} = 0, \; m_{33} = 1$$

Composition of Transformations



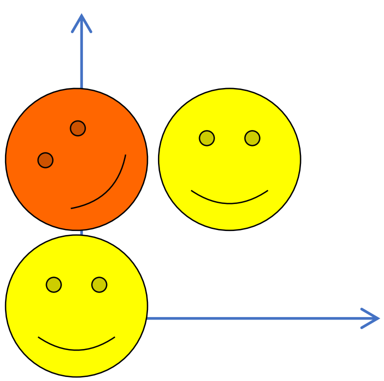
T(1,1)



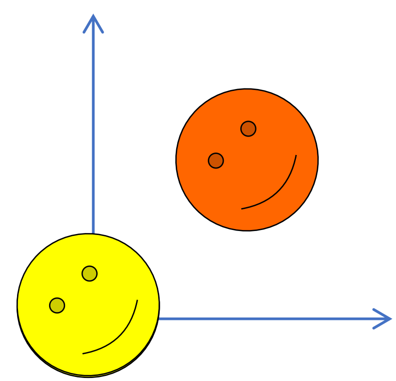
R(45)

Order Matters

R(45)T(1,1) ≠ T(1,1)R(45)



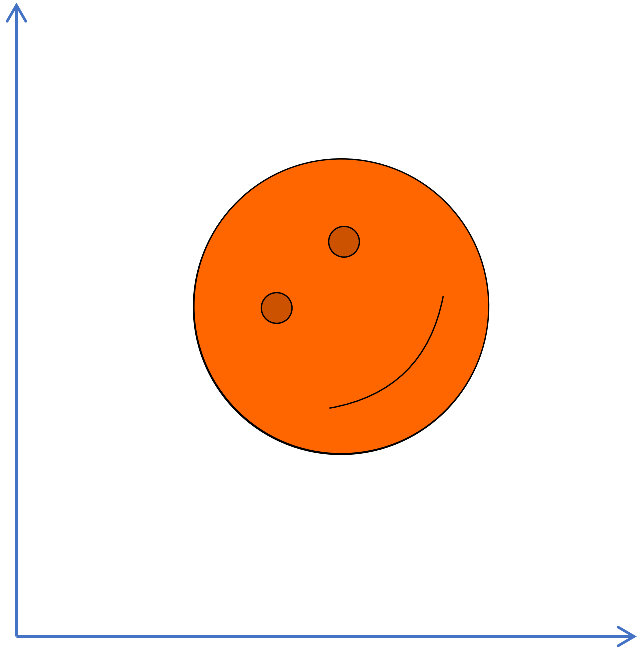
R(45) T(1,1)



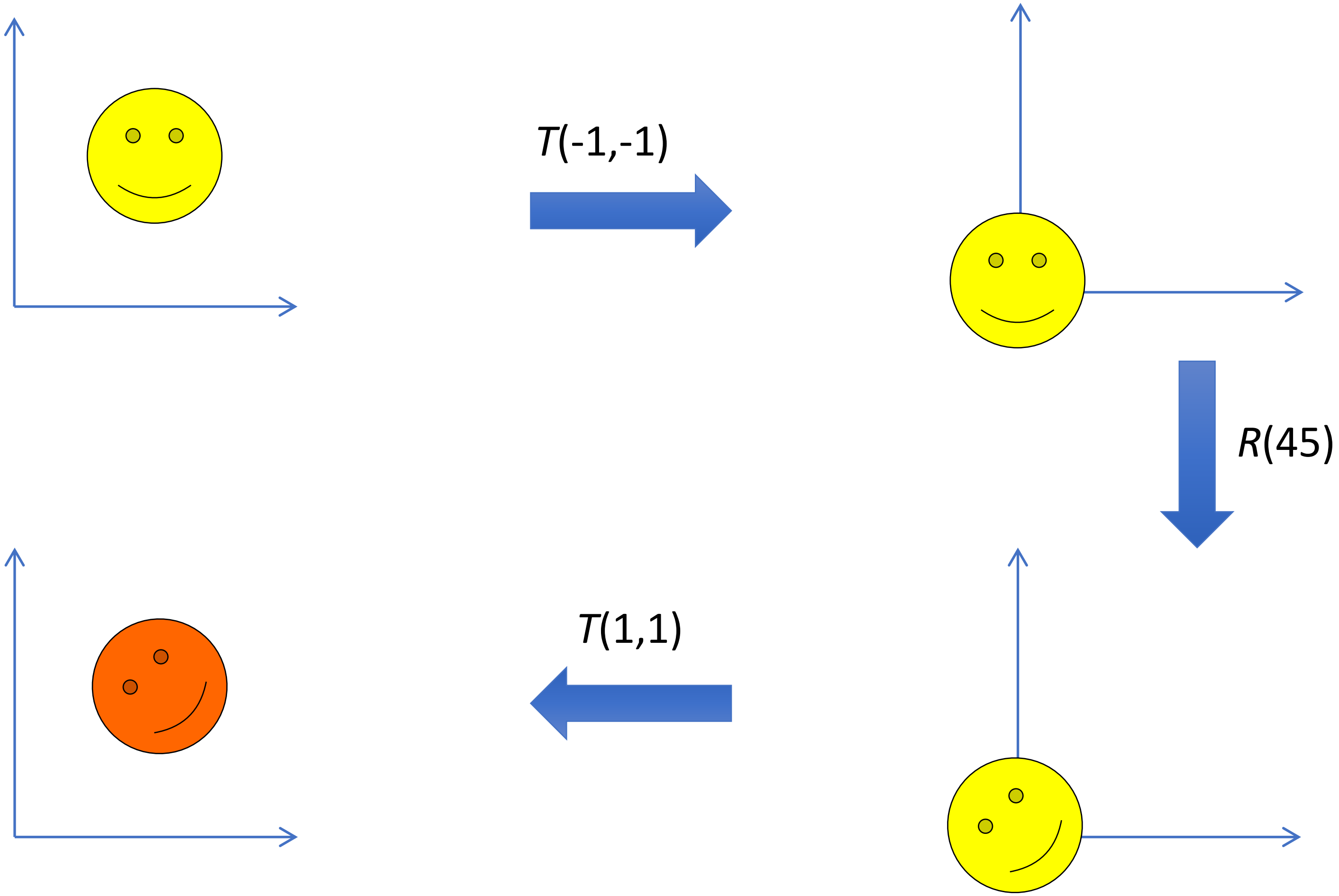
T(1,1) R(45)

Rotation at a point

- Rotate 45 at the center of Object (1,1) ?



Rot(45) at (1,1)



Display multiple instances

- transformations allow you to define an object at one location and then place multiple instances in your scene

OpenGL GLM

Using the GLM library

- OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the GLSL specifications.
- GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.
- This project isn't limited to GLSL features. An extension system, based on the GLSL extension conventions, provides extended capabilities: matrix transformations, quaternions, data packing, random numbers, noise, etc...

GLM: Translation Revisited

$$T(x_t, y_t, z_t), \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

GLM: Rotations Revisited

```
glm::mat4 trans_X = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(45.0f),
                    glm::vec3(1.0, 0.0, 0.0));

glm::mat4 trans_Y = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(45.0f),
                    glm::vec3(0.0, 1.0, 0.0));

glm::mat4 trans_Z = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(45.0f),
                    glm::vec3(0.0, 0.0, 1.0));
```

$$R_x(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$R_y(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$R_z(\theta)\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

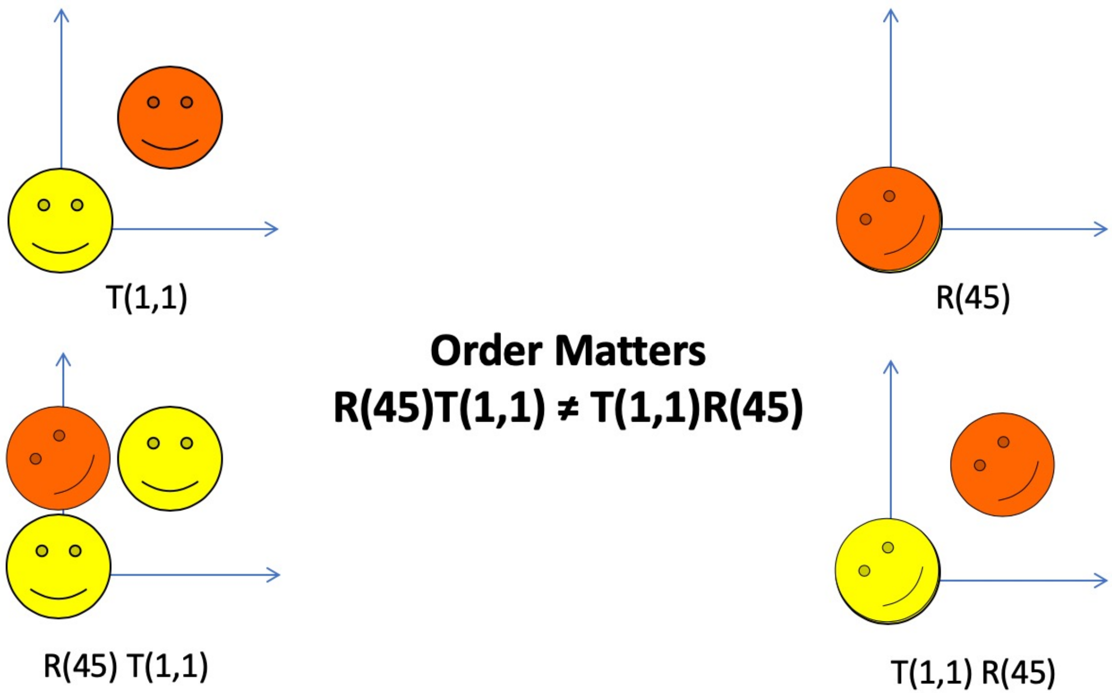
Scaling

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix}$$

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

```
glm::mat4 scale = glm::mat4(1.0f);
scale = glm::scale(scale,  glm::vec3(2.0f, 2.0f ,2.0f));
```

Composition of Transformations




```
glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;
glm::vec4 myTransformedVector = myModelMatrix * myOriginalVector;
std::cout << myTransformedVector.x << myTransformedVector.y << myTransformedVector.z << std::endl;
```

Generally: scaling FIRST, and THEN the rotation, and THEN the translation.

In the Shaders

In basic_model.hpp :

```
void draw(const glm::mat4 &view, const glm::mat4 proj) {
    using namespace glm;

    // cacluate the modelview transform
    mat4 modelview = view * modelTransform;

    // load shader and variables
    glUseProgram(shader);
    glUniformMatrix4fv(glGetUniformLocation(shader, "uProjectionMatrix"), 1, false, value_ptr(proj));
    glUniformMatrix4fv(glGetUniformLocation(shader, "uModelViewMatrix"), 1, false, value_ptr(modelview));
    glUniform3fv(glGetUniformLocation(shader, "uColor"), 1, value_ptr(color));

    // draw the mesh
    mesh.draw();
}
```

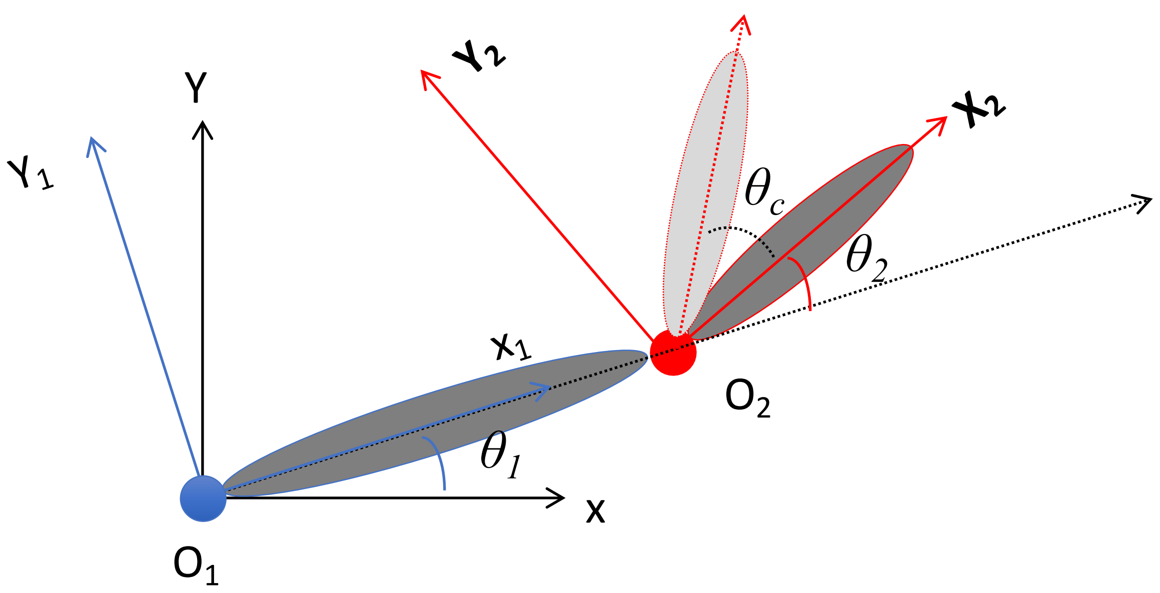
In default_vert.glsl:

```
void main() {
    // transform vertex data to viewspace
    v_out.position = (uModelViewMatrix * vec4(aPosition, 1)).xyz;
    v_out.normal = normalize((uModelViewMatrix * vec4(aNormal, 0)).xyz);
    v_out.textureCoord = aTexCoord;

    // set the screenspace position (needed for converting to fragment data)
    gl_Position = uProjectionMatrix * uModelViewMatrix * vec4(aPosition, 1);
}
```

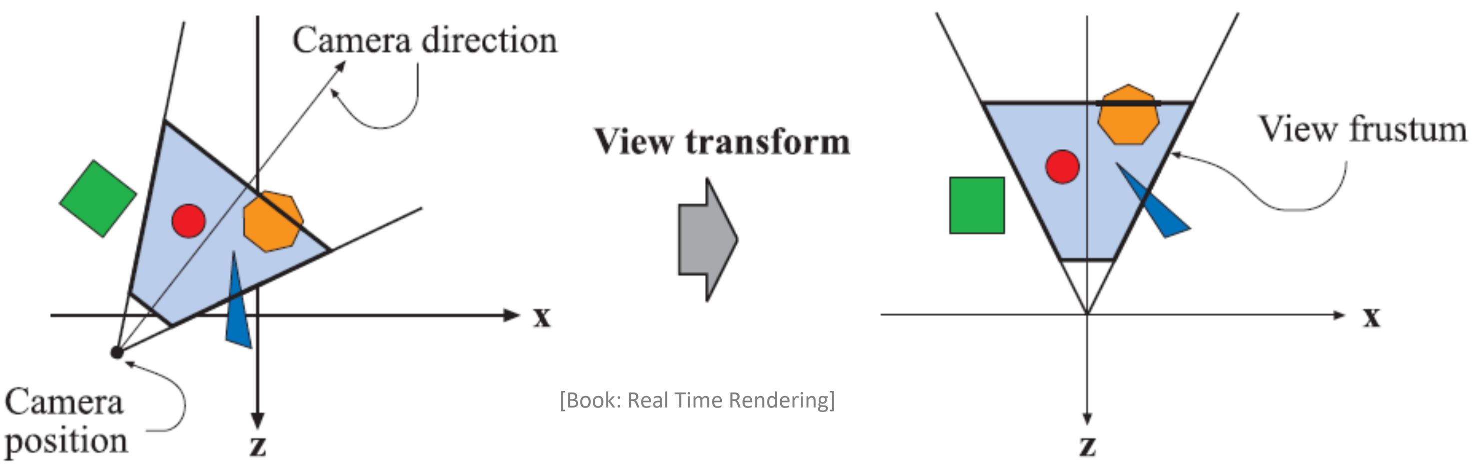
Object Coordinates

- An origin and basis define a frame of reference
- Object is defined in its local coordinates to easy control. Then, it is transferred to the world coordinates using model matrix $\mathbf{M}_{model}$



Eye(camera) coordinates

- Objects are transformed from object space to eye space using a "model" matrix
  - Combination of Model matrix $\mathbf{M}_{model}$ and View matrix $\mathbf{M}_{view}$
  - $\mathbf{M}_{model}$: from object coordinates to world coordinates
  - $\mathbf{M}_{view}$: from world coordinates to eye coordinates
  - In eye coordinates, camera is located at (0,0,0) facing –z axis



[Book: Real Time Rendering]

Move the mountains (world) or move the camera?

- Moving camera is reverse movement of objects
  - Rotate/Move Camera R_y(theta) is same as rotate object R_y(-theta)

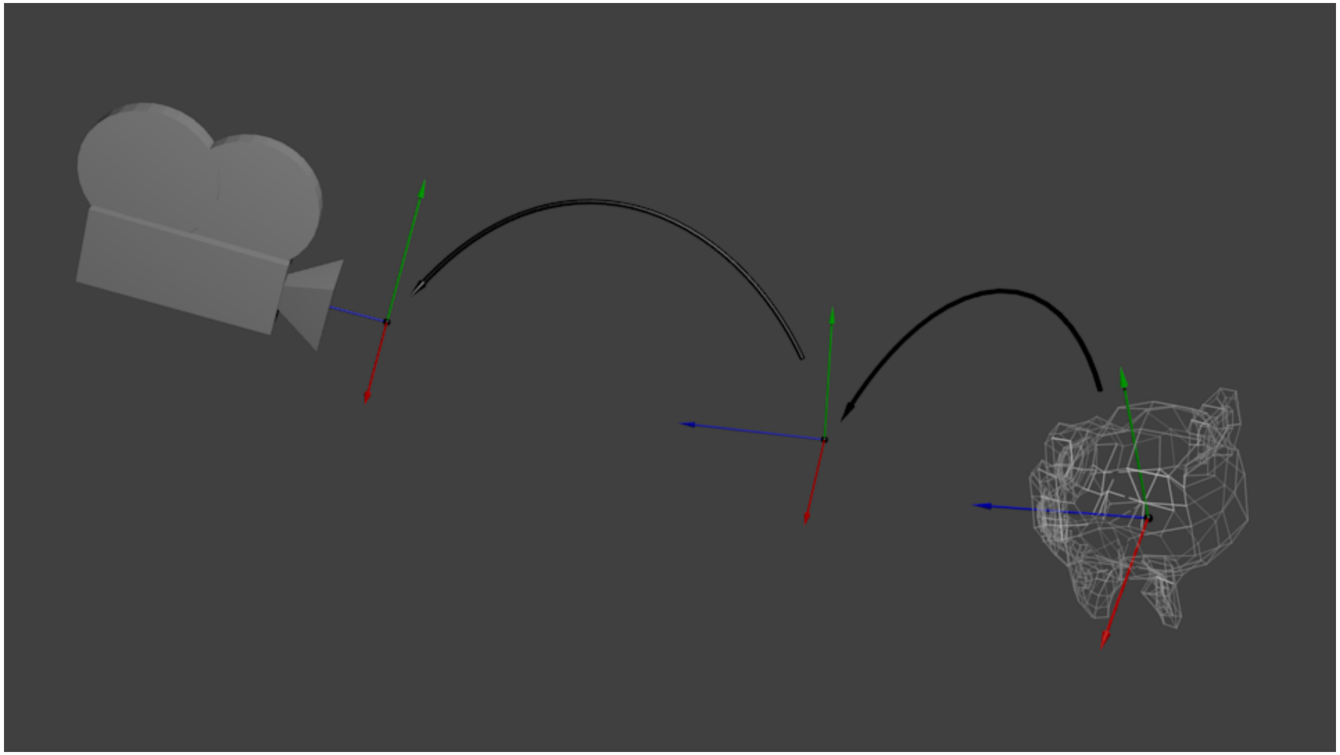


Image credit: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

View Transformation

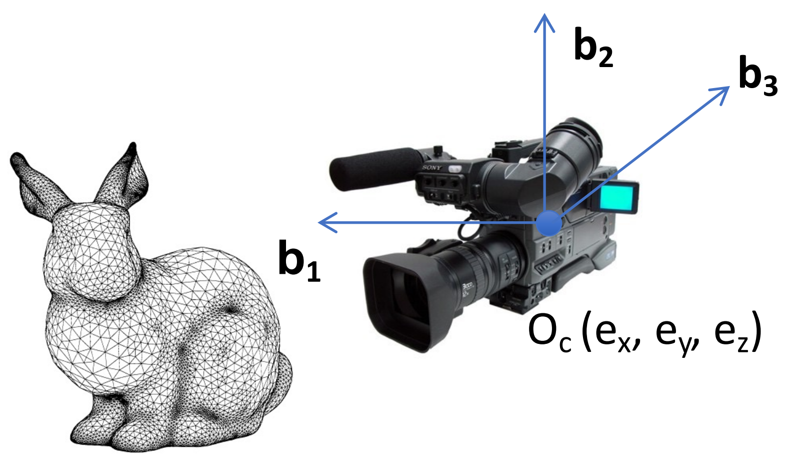
- The basis are all normalized and orthogonal
  - We can make a world coordinates transformation matrix which can move camera (position and orientation)
    in world coordinates
  - E.g. define a function LookAt(e_x, e_y, e_z, c_x, c_y, c_z, up_x, up_y, up_z), where

$\mathbf{b}_3 = -(\mathbf{c} - \mathbf{e})$

$\mathbf{b}_1 = \mathbf{up} \times \mathbf{b}_3$

$\mathbf{b}_2 = \mathbf{b}_3 \times \mathbf{b}_1$

$$O_c = \begin{bmatrix} b_{1x} & b_{2x} & b_{3x} & e_x \\ b_{1y} & b_{2y} & b_{3y} & e_y \\ b_{1z} & b_{2z} & b_{3z} & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} O_w$$



Parameters

eye	Position of the camera
center	Position where the camera is looking at
up	Normalized up vector, how the camera is oriented. Typically (0, 0, 1)

https://glm.g-truc.net/0.9.5/api/a00176.html

In the Code/Shaders

Application.cpp :

```
mat4 view = translate(mat4(1), vec3(0, -5, -m_distance)); // TODO replace view matrix with the camera transform
```

```
// display current camera parameters
ImGui::Text("Application %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
ImGui::SliderFloat("Distance", &m_distance, 0, 100, "%.1f");
ImGui::SliderFloat3("Model Color", value_ptr(m_model.color), 0, 1, "%.2f");
```

```
// cacluate the modelview transform
mat4 modelview = view * modelTransform;
```

GLM's LookAt:

```
glm::mat4 CameraMatrix = glm::lookAt(
                                    cameraPosition, // the position of your camera, in world space
                                    cameraTarget,   // where you want to look at, in world space
                                    upVector        // glm::vec3(0,1,0), but (0,-1,0) would make you looking upside-down
                                    );
```

```
for(unsigned int i = 0; i < nModels; i++)
{
    DoSomePreparations(); // bind VAO, bind textures, set uniforms etc.
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

Instancing (*hint*)

- transformations allow you to define an object at one location and then place multiple instances in your scene

Instancing hint: Code/GLSL

glDrawArraysInstanced

Drawing

The function `glDrawArraysInstanced` draws multiple **instances** of the same object which allows for much greater efficiency than drawing these objects individually using calls like `glDrawArrays`. Via GLSL's built in `gl_InstanceID` or *instanced arrays* it is then possible to manipulate the vertices per instance.

The parameters of `glDrawArraysInstanced (GLenum mode, GLint first, GLsizei count, GLsizei primcount)` are as follows:

- `mode`: specifies the kind of primitive to render. Can take the following values: `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON`.
- `first`: specifies the starting index in the enabled arrays.
- `count`: specifies the number of vertices required to render a single instance.
- `primcount`: specifies the number of instances to render.

Example usage

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
glBindVertexArray(0);
```

Credit: https://learnopengl.com/Advanced-OpenGL/Instancing

Instancing: Code/GLSL

In the program:

```
glm::vec2 translations[100];
int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)
{
    for(int x = -10; x < 10; x += 2)
    {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

In the shader:

```
uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

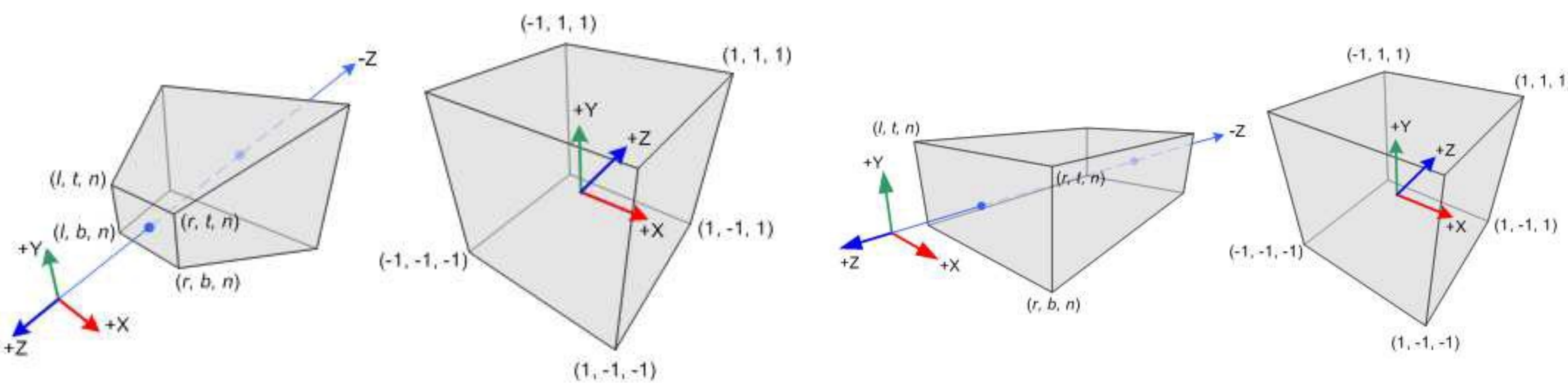
Credit: https://learnopengl.com/Advanced-OpenGL/Instancing

Projection

- In eye coordinates, the objects are still in 3D space
- The 3D scene in eye coordinates needs to be transferred to the 2D image on screen
- The projection matrix transfer objects in eye coordinates into clip coordinates.
- Then, perspective division (dividing with w component) of the clip coordinates transfer them to the normalized device coordinates (NDC)

Projection Matrix

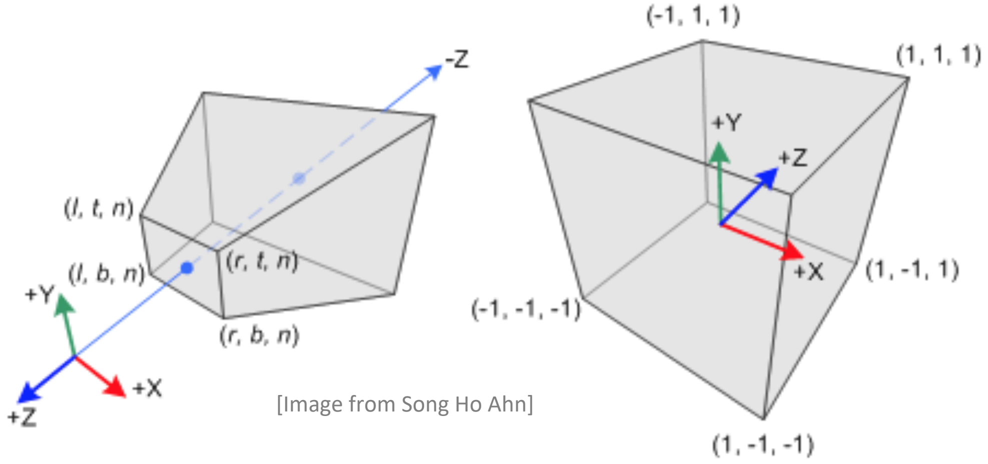
- The projection matrix defines a view frustum determining objects to be drawn or clipped out
  - Frustum culling (clipping) is performed in the clip coordinates, before dividing points by w_c
  - Perspective Projection, Orthographic Projection



Perspective Projection [Image from Song Ho Ahn] Orthographic Projection

Perspective Projection

- 3D objects in eye coordinates are mapped into a canonical view volume
  - The view volume is specified by
    [left, right, bottom, top, near, far]
  - The view volume is transformed into a canonical view volume which is a cube from (-1,-1,-1) to (1,1,1)
    - X: [l, r] → [-1, 1]
    - Y: [b, t] → [-1, 1]
    - Z: [n, f] → [-1, 1]

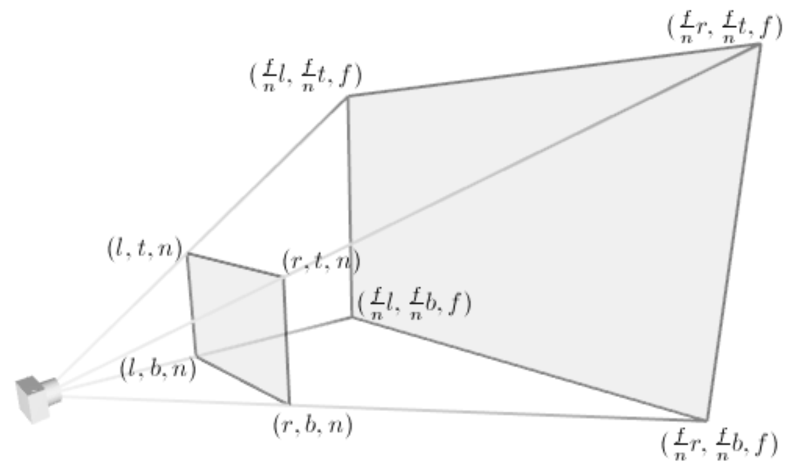


[Image from Song Ho Ahn]

Perspective Projection in OpenGL

- The perspective Projection matrix of a frustum [l ,r,b,t,n,f] is:

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

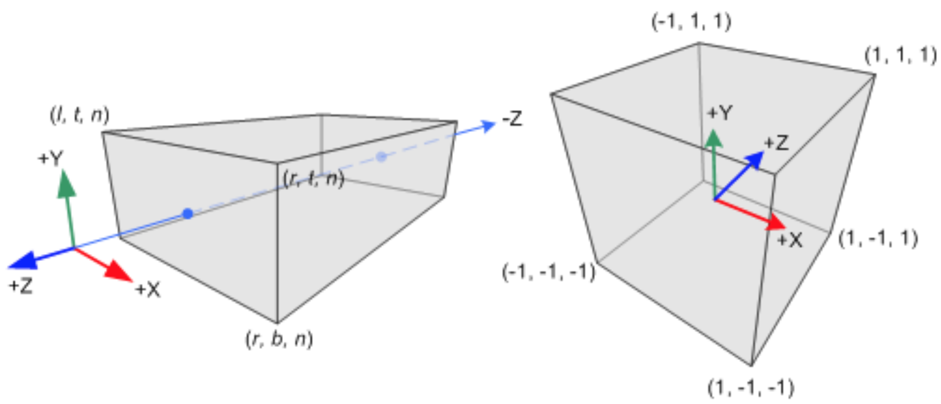


[Image from Song Ho Ahn]

```
// calculate the projection and view matrix
mat4 proj = perspective(1.f, float(width) / height, 0.1f, 1000.f);
```

Orthographic Projection

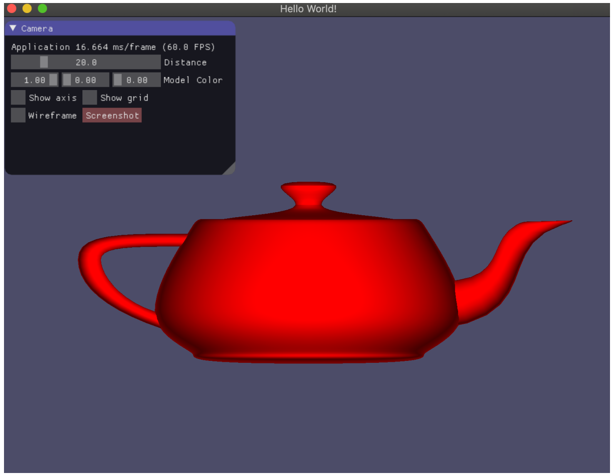
- Constructing a projection matrix for orthographic projection is much simpler
- Linear mapping from (x_e, y_e, z_e) to (x_n, y_n, z_n)



Orthographic Projection Matrix

- The Orthographic Projection matrix of [l,r,b,t,n,f] is

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

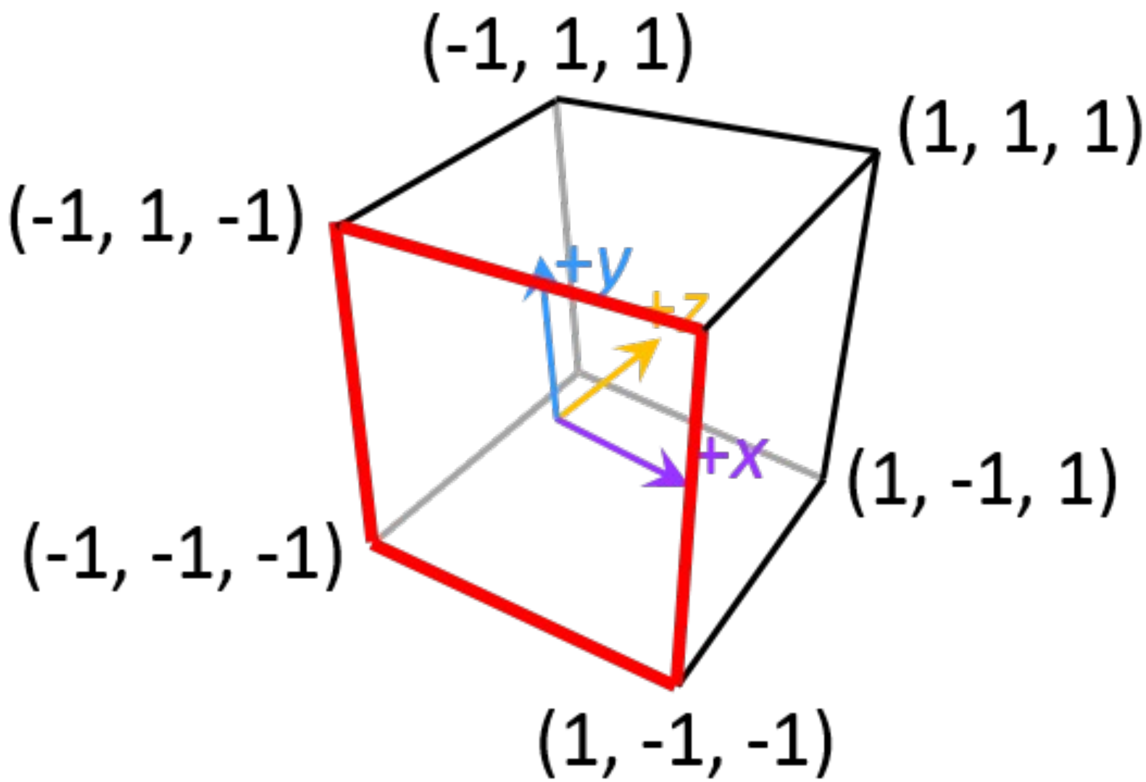


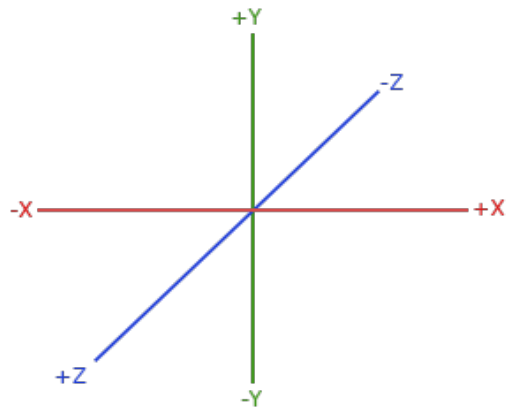
- Since W-component is not necessary, the 4th row of the matrix is remains as (0,0,0,1)

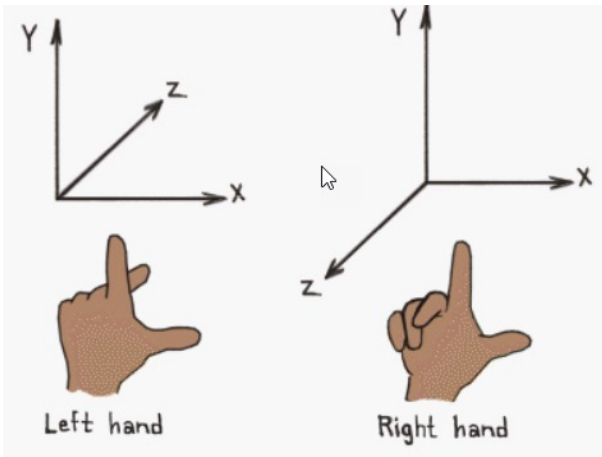
→ Try it at Home !

```
mat4 proj = ortho(-10.0f,10.0f,-10.0f,10.0f,0.0f,100.0f); // In world coordinates
```

Normalized Device Coordinates (NDC)







OpenGL: right-handed; others (e.g. **DirectX**: left-handed)

Normalized Device Coordinates (NDC)

- 3D Homogeneous coordinates

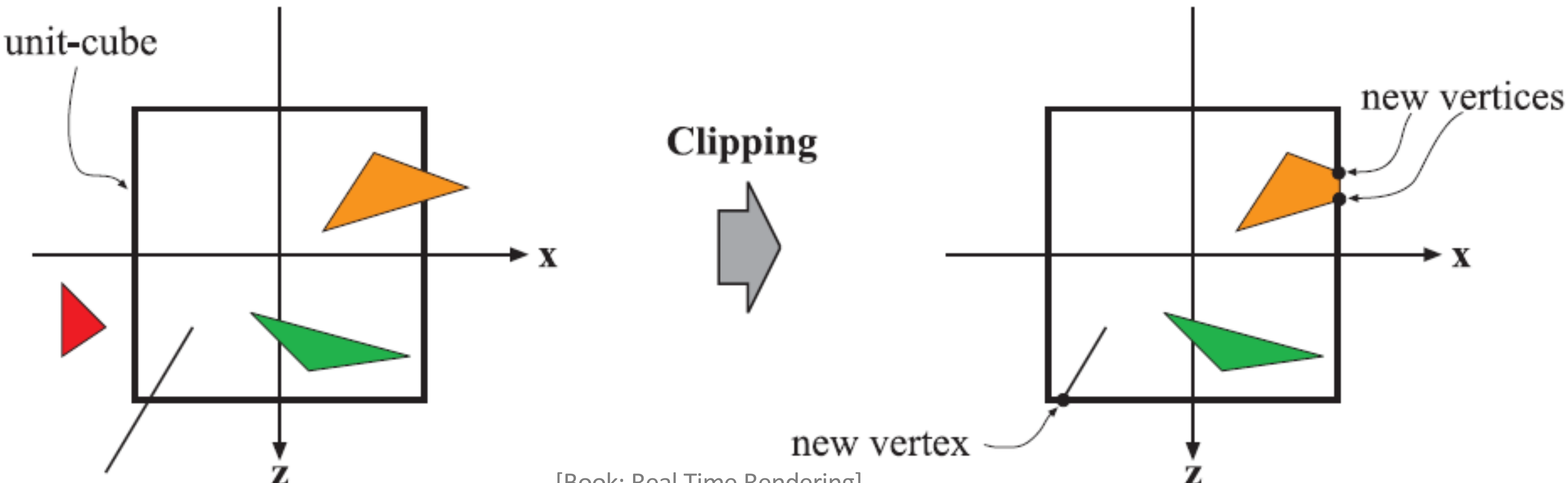
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{ can be represented as } \begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix}$$

where $$x = \frac{X}{w}, \quad y = \frac{Y}{w}, \quad z = \frac{Z}{w}$$

- **Normalized device coordinates (NDC)** is generated by perspective division with w

Clipping

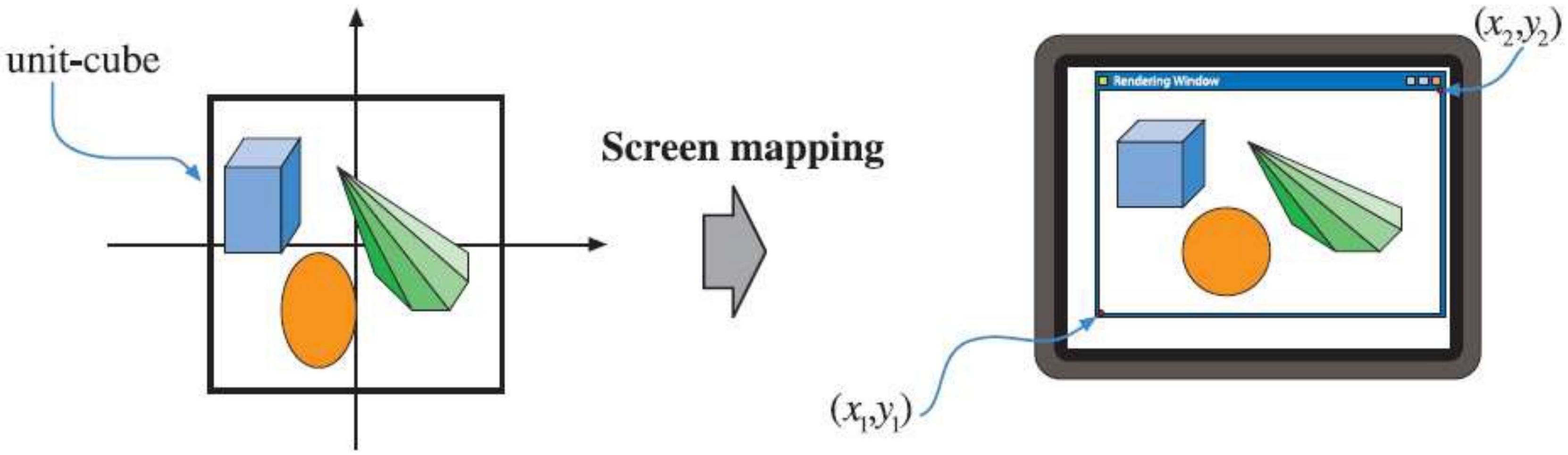
- Canonical view volume clips primitives
  - Primitives inside of the view volume are passed to the next stage
  - Primitives outside of the view volume are clipped
  - Clipping may generate new vertices



[Book: Real Time Rendering]

Viewport Transform

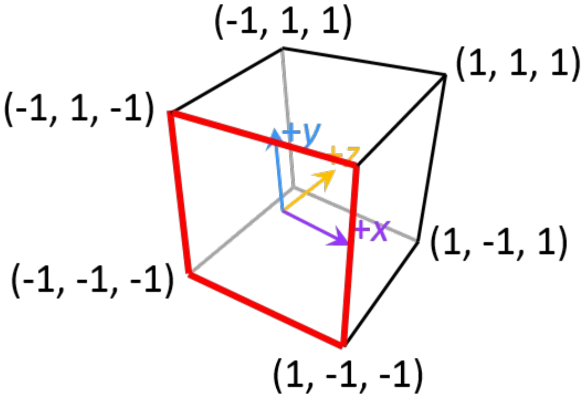
- Clipped primitives of NDC (x_n, y_n, z_n) are transferred to screen coordinates (x_s, y_s)
- Screen coordinates with depth value are window coordinates (x_w, y_w, z_w)



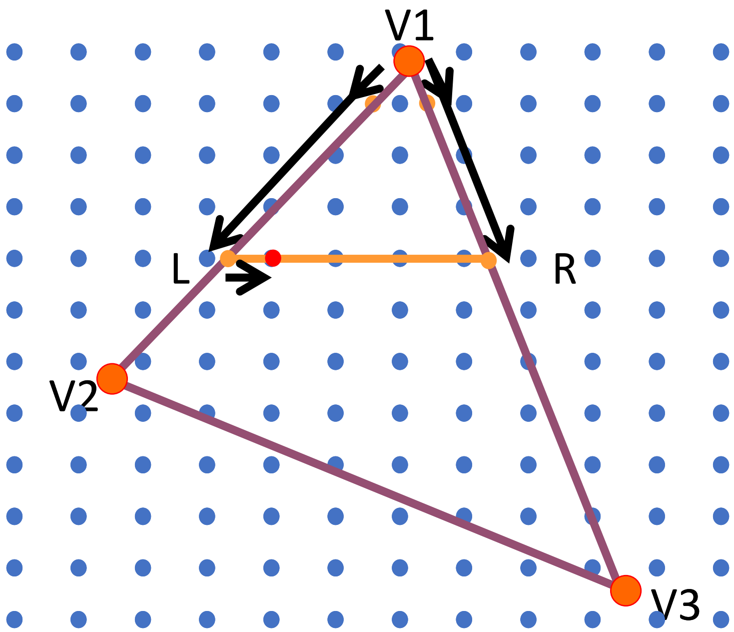
[Book: Real Time Rendering]

Viewport Transform

- (x_n, y_n) in NDC are [-1 1], the value is translated and scaled to the pixel position of screen (x_s, y_s)
- Screen coordinates (x_s, y_s) represent the pixel position of a fragment
- z_n in NDC is [-1 1], the value is translated and scaled on [0 1] for z_w
- z_w is the depth value of the pixel position (x_s, y_s) used for depth test using z-buffering



Rasterization



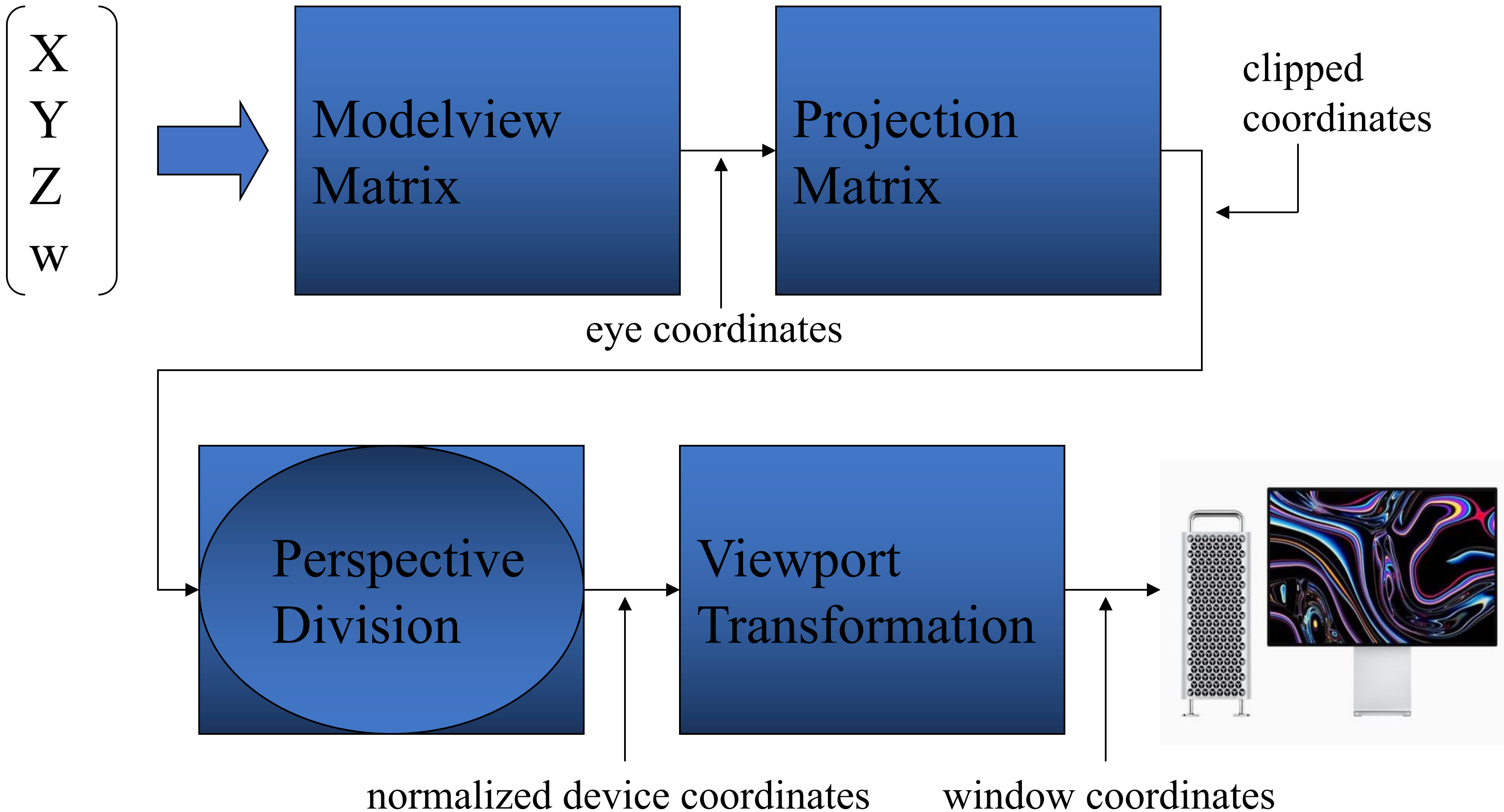
- Convert primitives into fragment
  - Interpolates triangle vertices into fragments
  - Fragments are mapped into frame buffer

Hidden Surface Removal

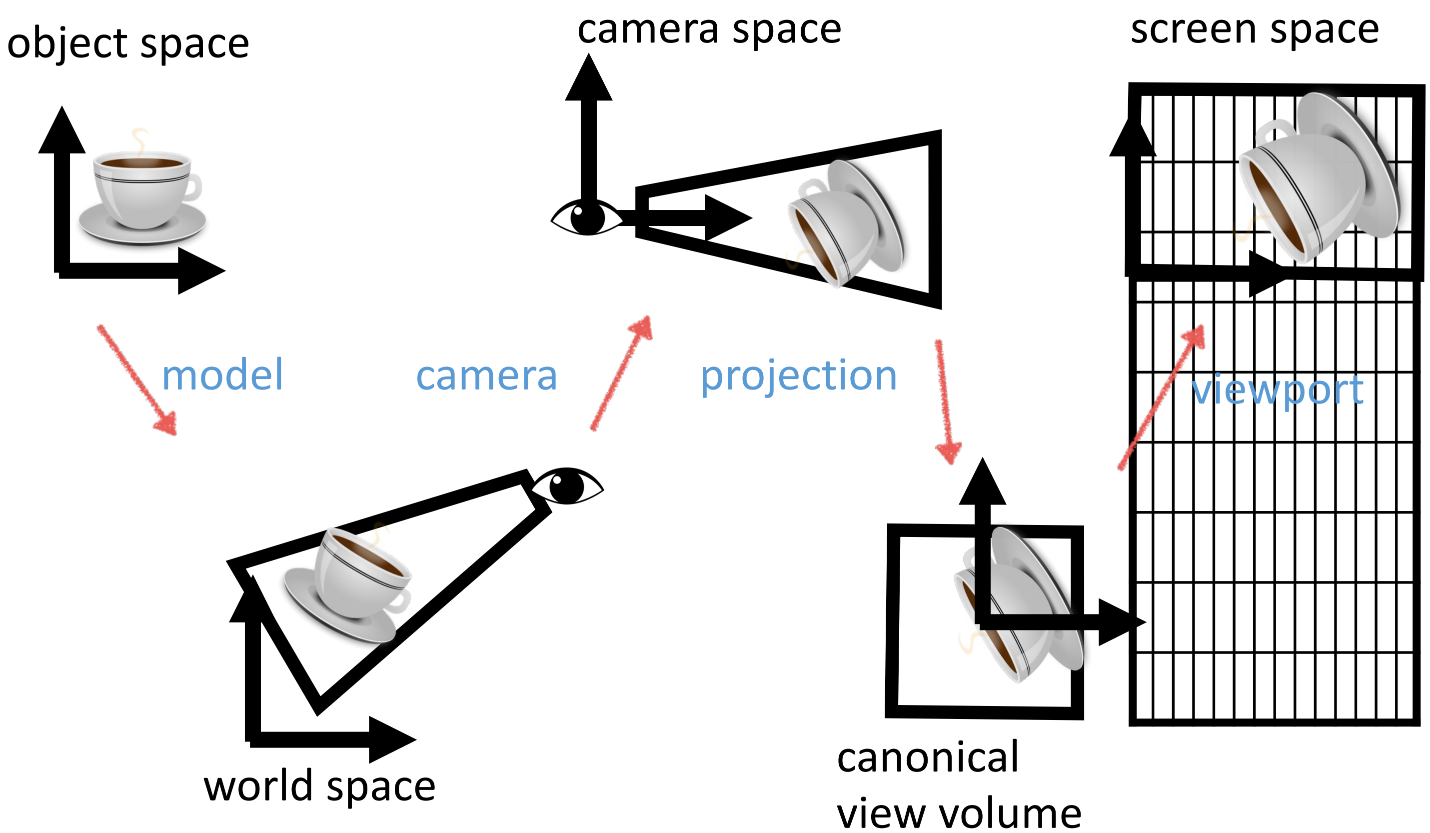
- Eliminate parts that are occluded by others
  - Depth buffer (z-buffer) contains the nearest depth values of each fragment
  - If (current depth < depth buffer),
    update frame buffer and depth buffer
    using the current values (color/depth)

```
glEnable(GL_DEPTH_TEST);
    ...
while (1)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    draw_3d_object_A();
    draw_3d_object_B();
}
```

Stages of Vertex Transformations



Viewing Transformation



Supplement A: Matrices overview

Overview

- Matrices will allow us to conveniently represent and apply transformations to vectors, such as translation, scaling and rotation
- Similarly to what we did for vectors, we will briefly overview their basic operations

Basic operations

- A matrix is an array of numeric elements

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$$

Sum

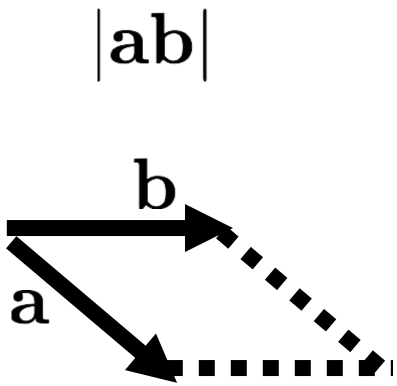
$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{11} + y_{11} & x_{12} + y_{12} \\ x_{21} + y_{21} & x_{22} + y_{22} \end{bmatrix}$$

Scalar Product

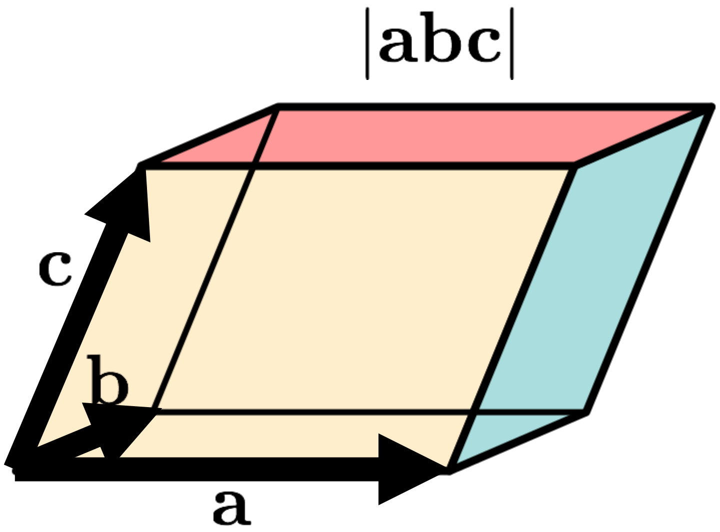
$$y * \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} yx_{11} & yx_{12} \\ yx_{21} & yx_{22} \end{bmatrix}$$

Determinants

- Think of a determinant as an operation between vectors.



Area of the parallelogram



Volume of the parallelepiped
(positive since abc is a right-handed basis)

Transpose

- The transpose of a matrix is a new matrix whose entries are reflected over the diagonal

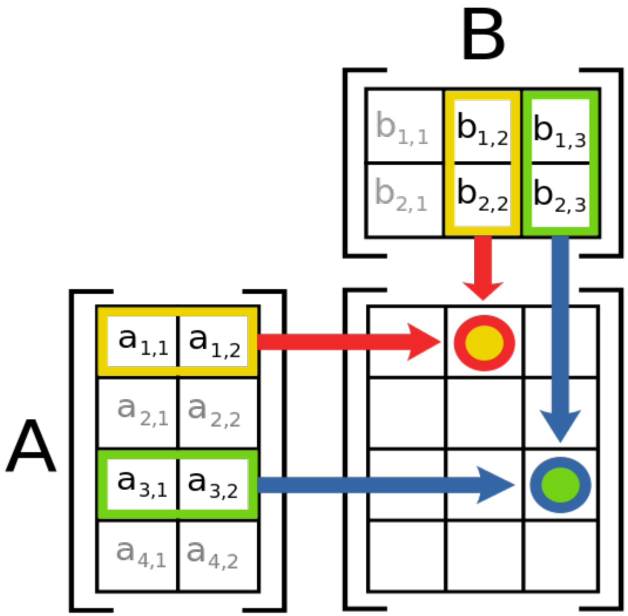
$$\begin{bmatrix} 1 & 2 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

- The transpose of a product is the product of the transposed, in reverse order

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

Matrix multiplication

- The entry i,j is given by multiplying the entries on the i-th row of A with the entries of the j-th column of B and summing up the results
- It is NOT commutative (in general):



$$\mathbf{AB} \neq \mathbf{BA}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\cdot5+2\cdot7 & 1\cdot6+2\cdot8 \\ 3\cdot5+4\cdot7 & 3\cdot6+4\cdot8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Intuition

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} -\mathbf{r_1}- \\ -\mathbf{r_2}- \\ -\mathbf{r_3}- \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix}$$

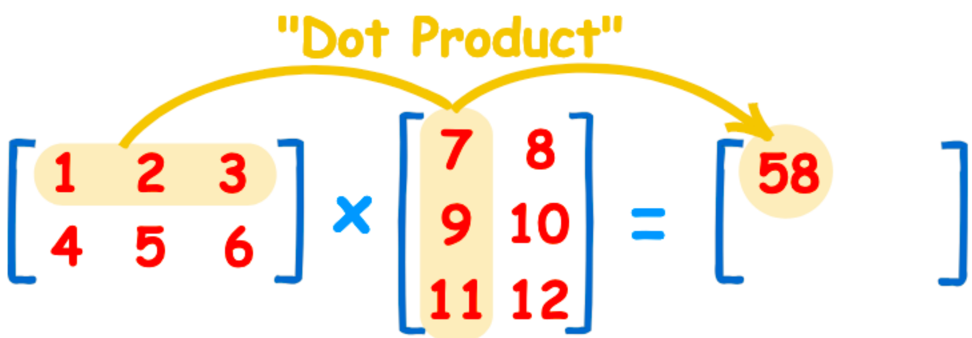
$$y_i = \mathbf{r_i} \cdot \mathbf{x}$$

Dot product on each row

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \mathbf{c_1} & \mathbf{c_2} & \mathbf{c_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{y} = x_1\mathbf{c_1} + x_2\mathbf{c_2} + x_3\mathbf{c_3}$$

Weighted sum of the columns



https://www.mathsisfun.com/algebra/matrix-multiplying.html

Inverse matrix

- The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{A}^{-1}$ such that

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

where **I** is the ***identity matrix***

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

- The inverse of a product is the product of the inverse in opposite order:

$$(\mathbf{A}\mathbf{B})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

Inverse matrix

Need it because, there is no concept of dividing by a matrix: can **multiply by an inverse**, to achieves the same thing.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

determinant ($ad-bc$)

$$\begin{bmatrix} 4 & 7 \\ 2 & 6 \end{bmatrix}^{-1} = \frac{1}{4\times6-7\times2} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \frac{1}{10} \begin{bmatrix} 6 & -7 \\ -2 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 0.6 & -0.7 \\ -0.2 & 0.4 \end{bmatrix}$$

https://www.mathsisfun.com/algebra/matrix-inverse.html

https://www.wikihow.com/Find-the-Inverse-of-a-3x3-Matrix

Diagonal Matrices

- They are zero everywhere except the diagonal:

$$\mathbf{D} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$

- Useful properties:

$$\mathbf{D}^{-1} = \begin{bmatrix} a^{-}1 & 0 & 0 \\ 0 & b^{-}1 & 0 \\ 0 & 0 & c^{-1} \end{bmatrix}$$

$$\mathbf{D} = \mathbf{D}^T$$

Orthogonal Matrices

- An orthogonal matrix is a matrix where
  - each column is a vector of length 1
  - each column is orthogonal to all the others
  - **orthonormal vectors!**
- A useful property of orthogonal matrices that their inverse corresponds to their transpose:

$$Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = I$$

$$Q^{\mathrm{T}} = Q^{-1}$$