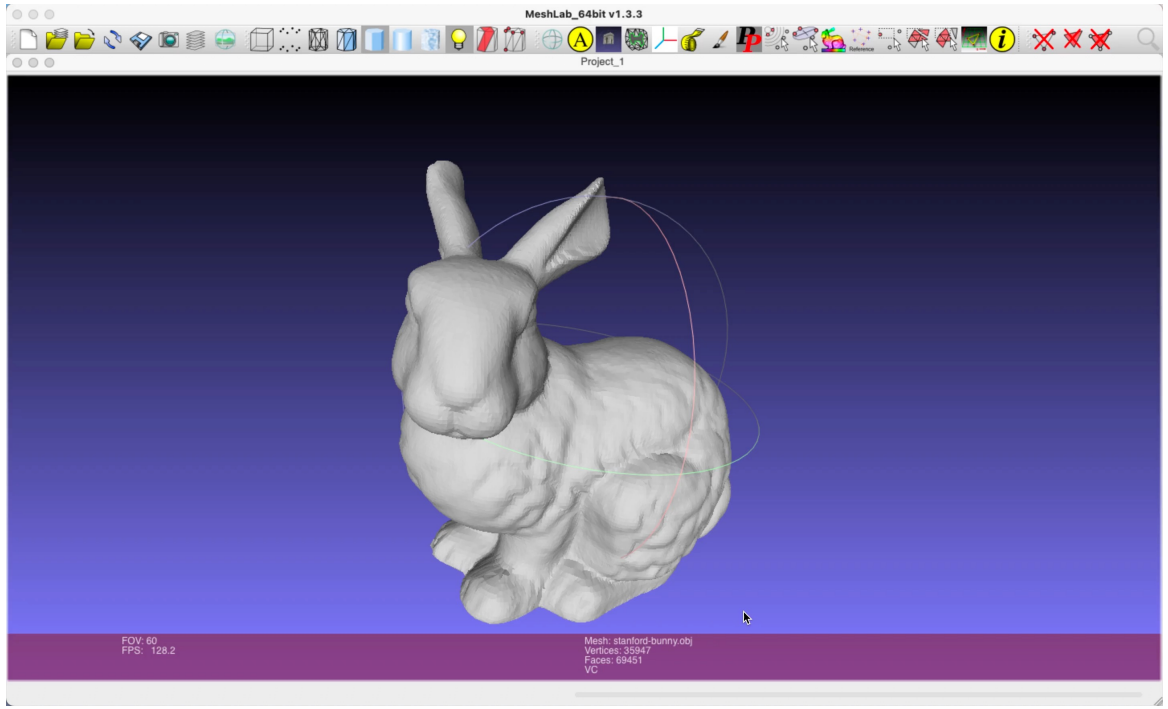
Graphics Pipeline and 3D Drawing

1. Overview of Graphics Pipeline

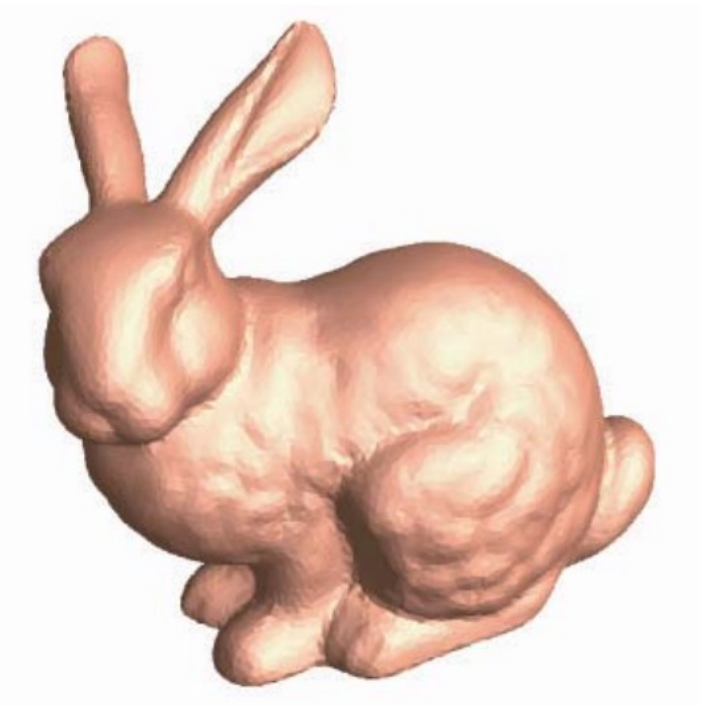
Graphics Pipeline

Input: 3D object / 3D scene

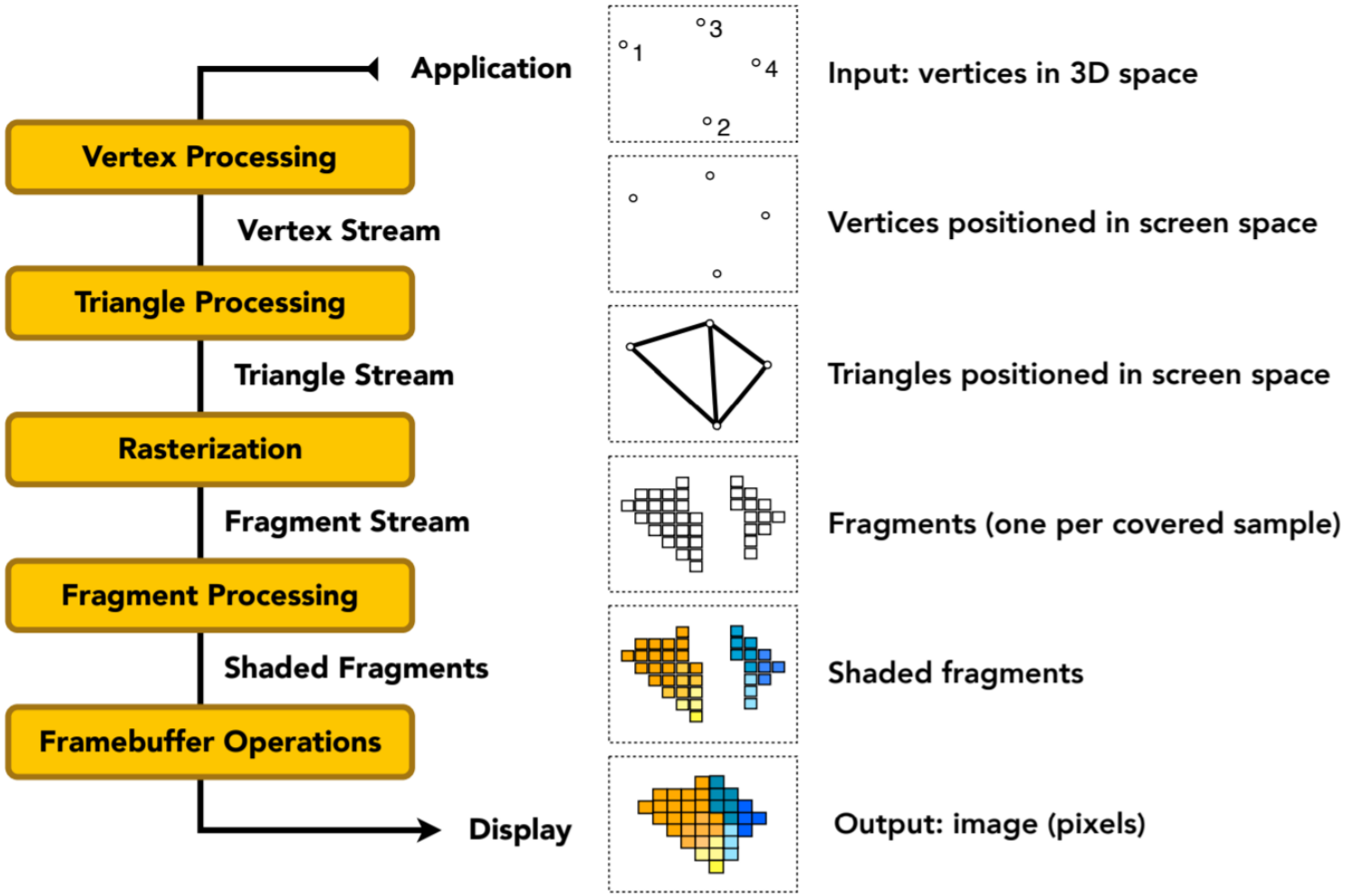


?

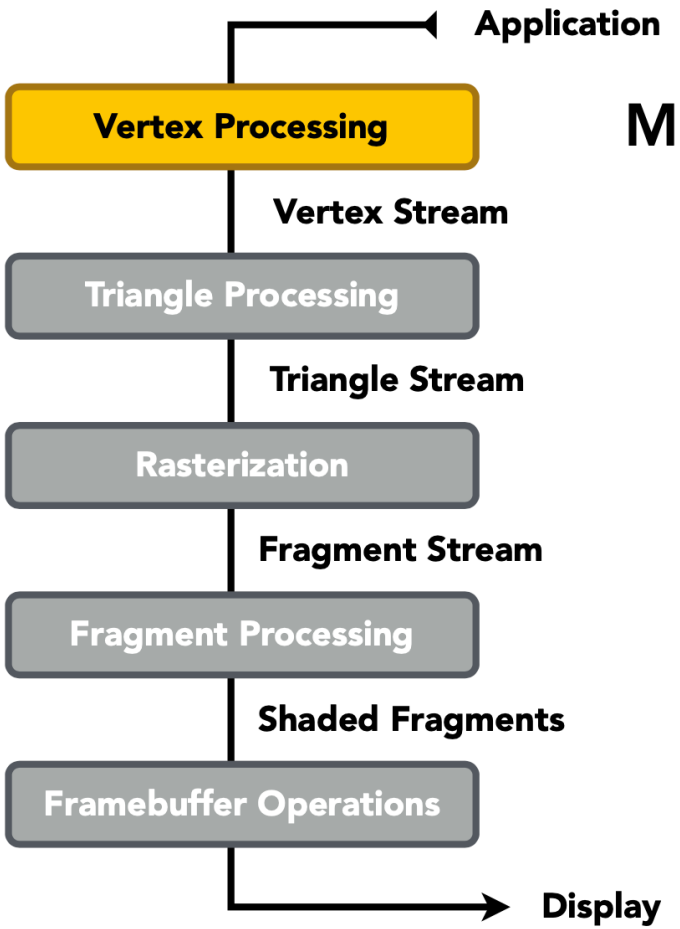
Output: 2D rendering



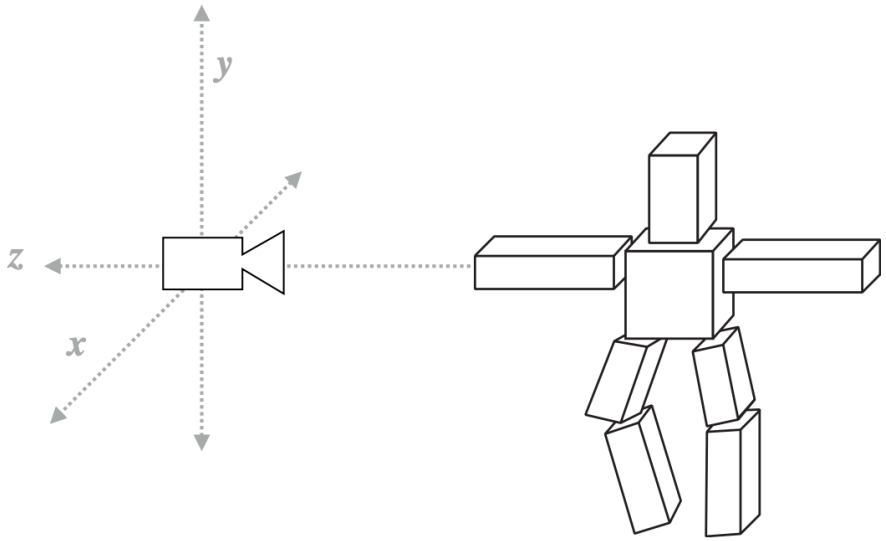
Graphics Pipeline



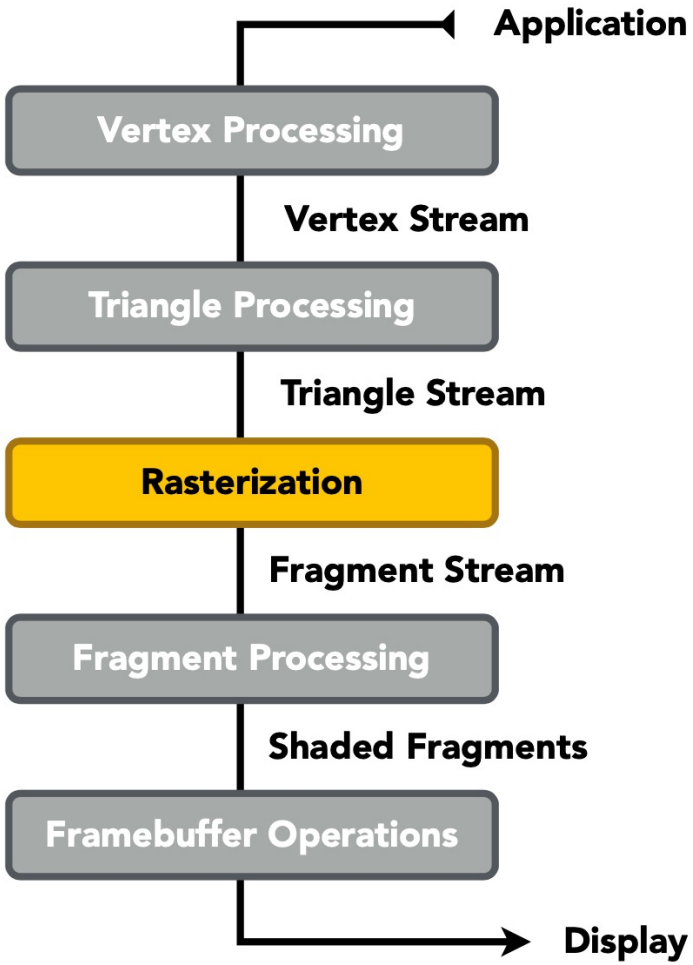
Graphics Pipeline



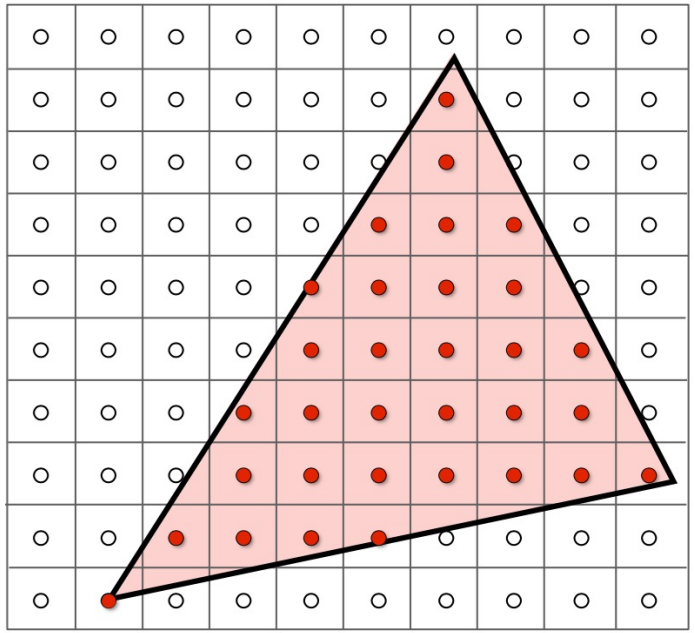
Model, View, Projection transforms



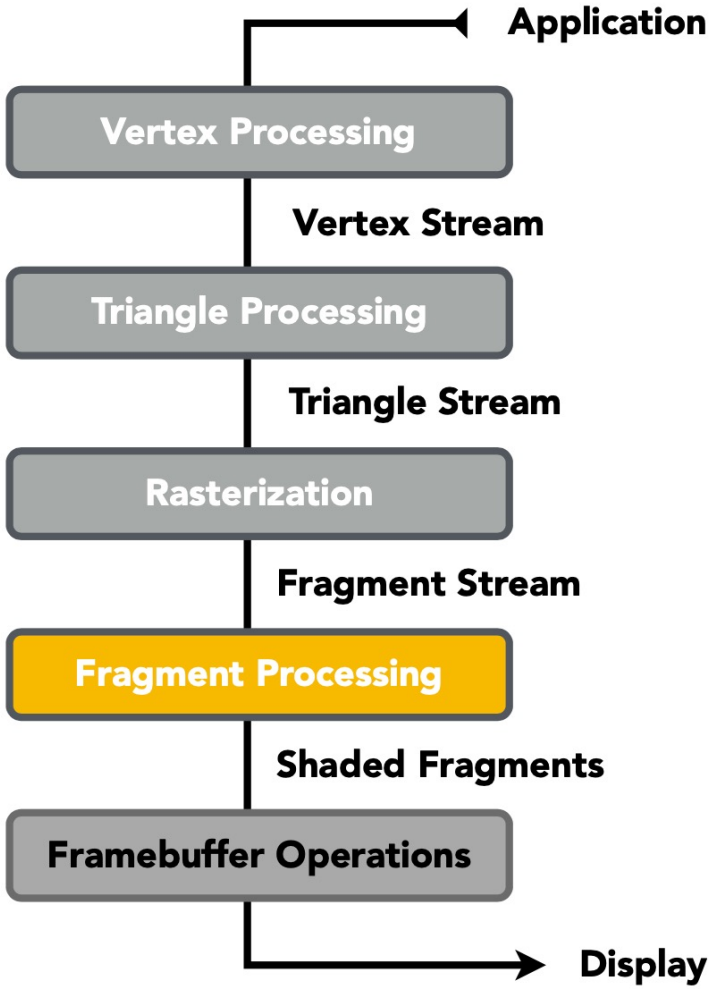
Graphics Pipeline



Sampling triangle coverage

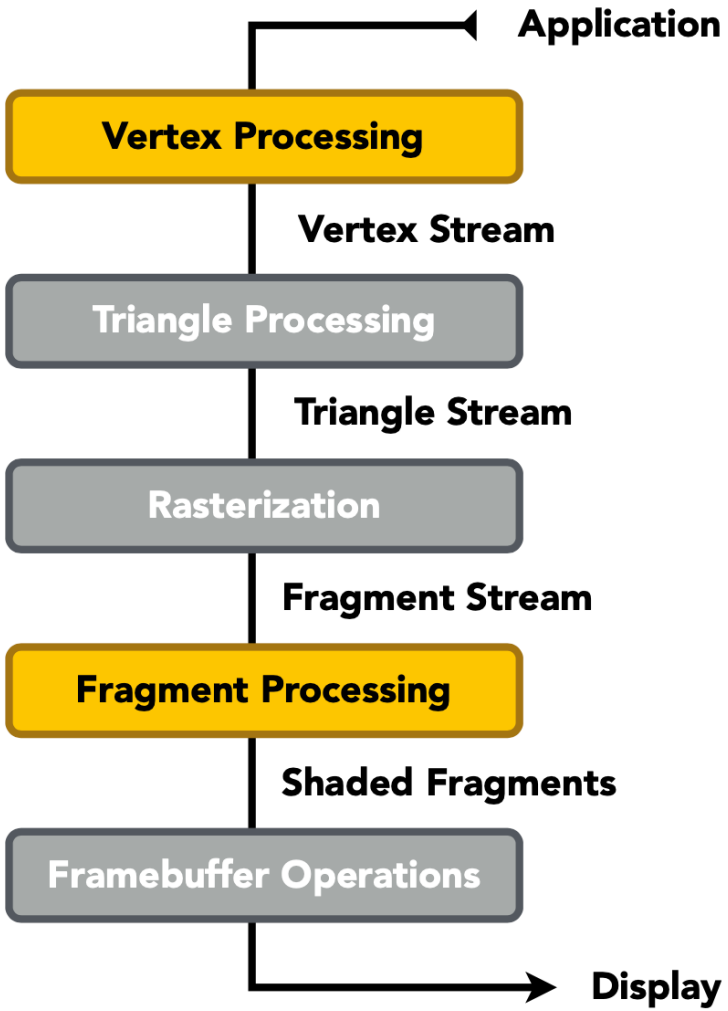


Graphics Pipeline

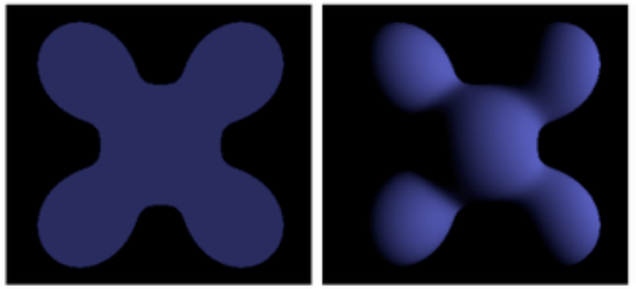


Z-Buffer Visibility Tests

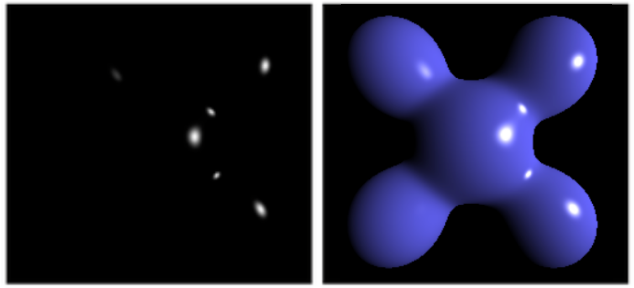
Graphics Pipeline



Shading

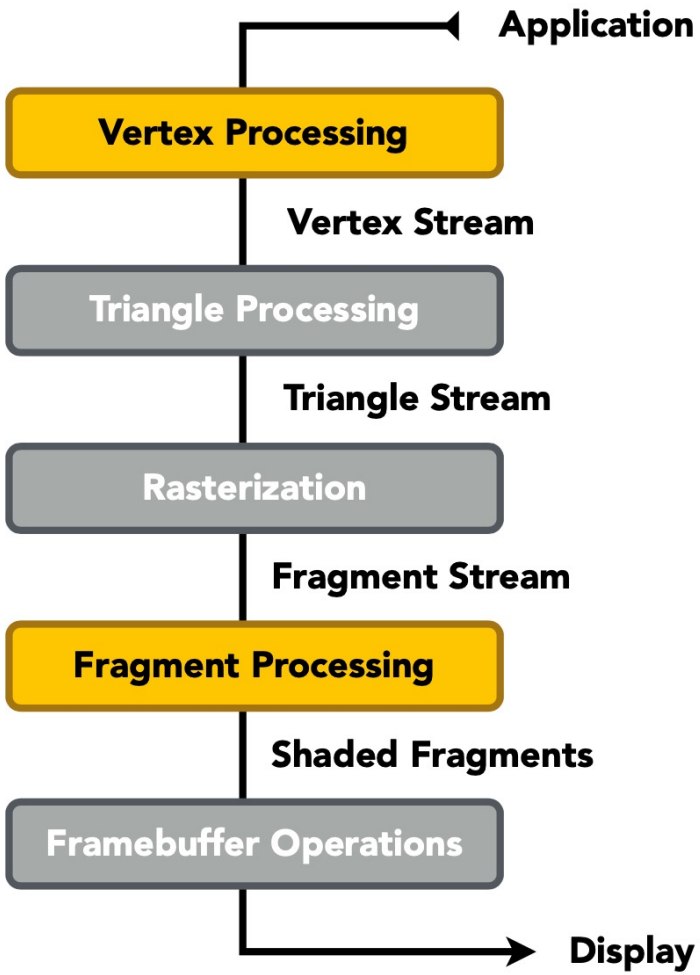


Ambient + Diffuse



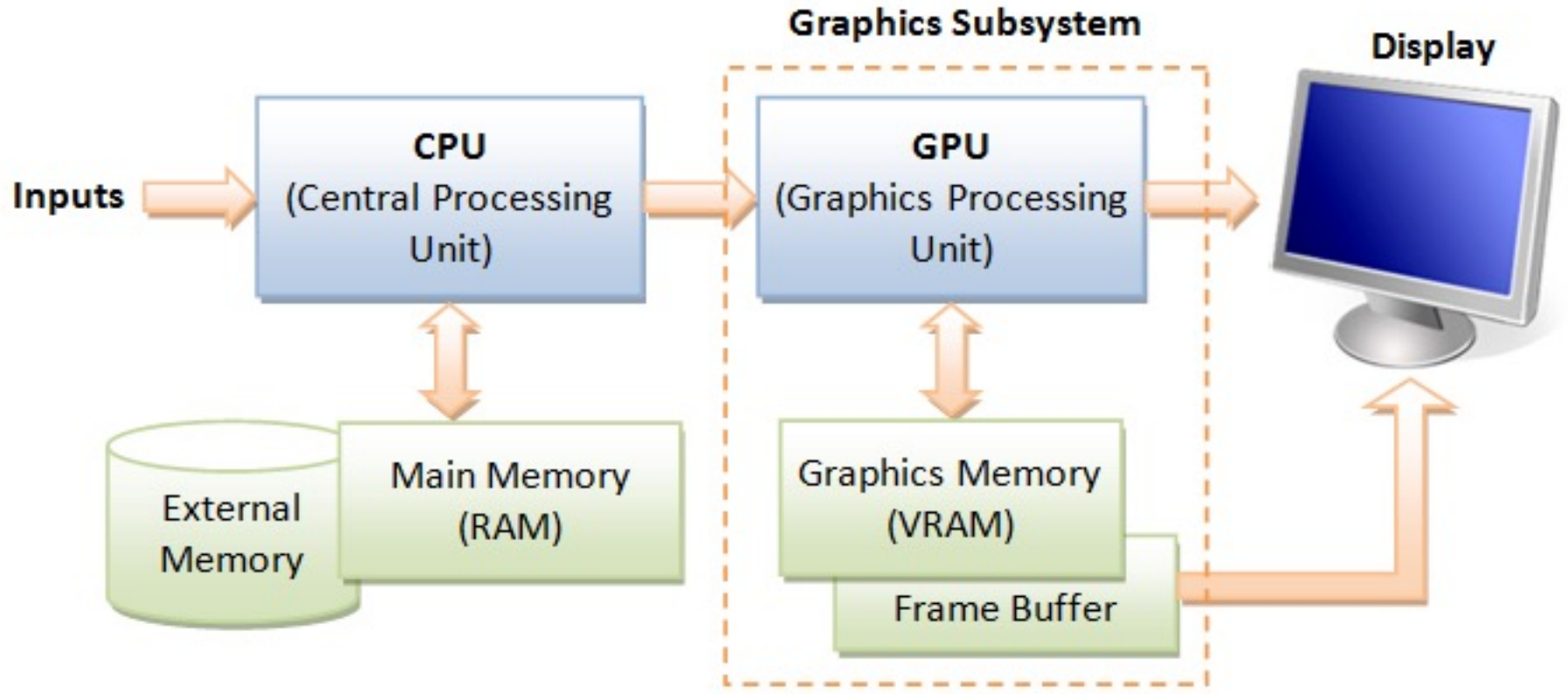
+ Specular = Blinn-Phong Reflectance Model

Graphics Pipeline



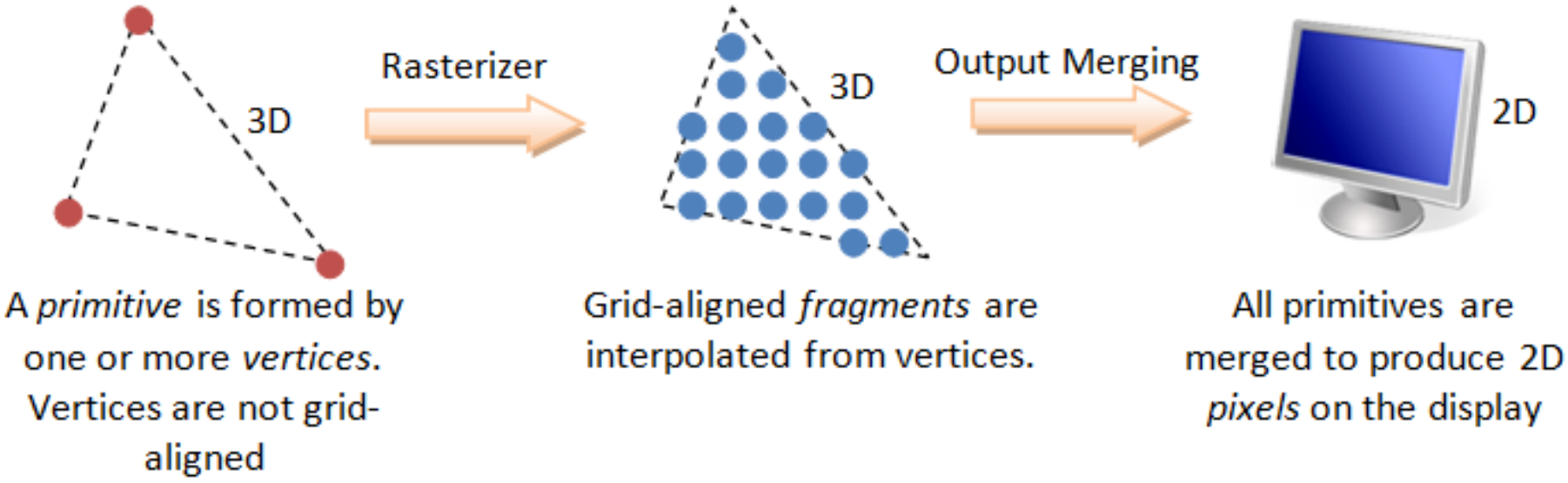
Texture mapping
(introducing soon)

Computer Graphics Hardware



- **GPU (Graphics Processing Unit)** Modern day computer has dedicated Graphics Processing Unit (GPU) to produce images for the display, with its own graphics memory (or Video RAM or VRAM).

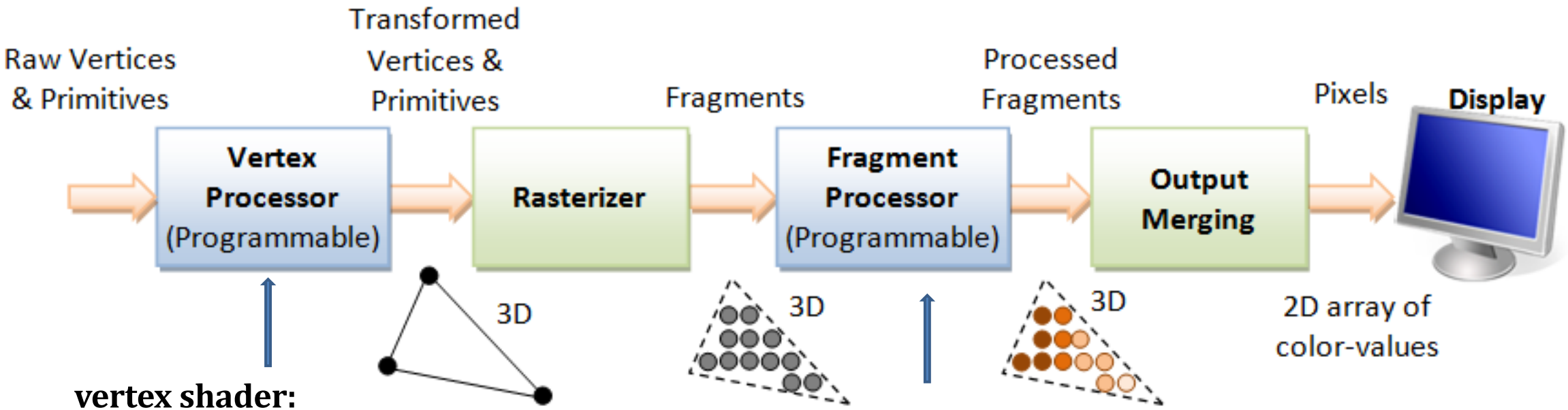
Pixel vs. Fragment



Vertex, Primitives, Fragment and Pixel

- Pixels refers to the dots on the display. A pixel is 2-dimensional, with a (x, y) position and a RGB color value .
- A fragment is 3-dimensional, with a (x, y, z) position. The (x, y) are aligned with the 2D pixel-grid. The z-value (not grid-aligned) denotes its depth.
- Fragments are produced via *interpolation* of the vertices. Hence, a fragment has all the vertex's attributes such as color, fragment-normal and texture coordinates.

3D Graphics Rendering Pipeline

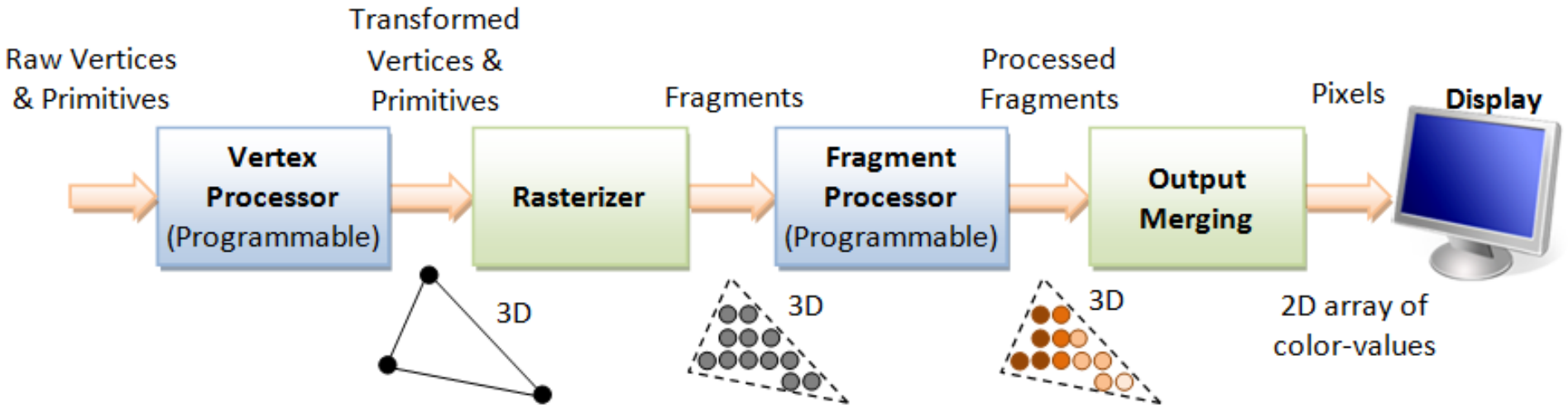


vertex shader:
Compute the position & evaluate a normal and texture coordinates

fragment shader

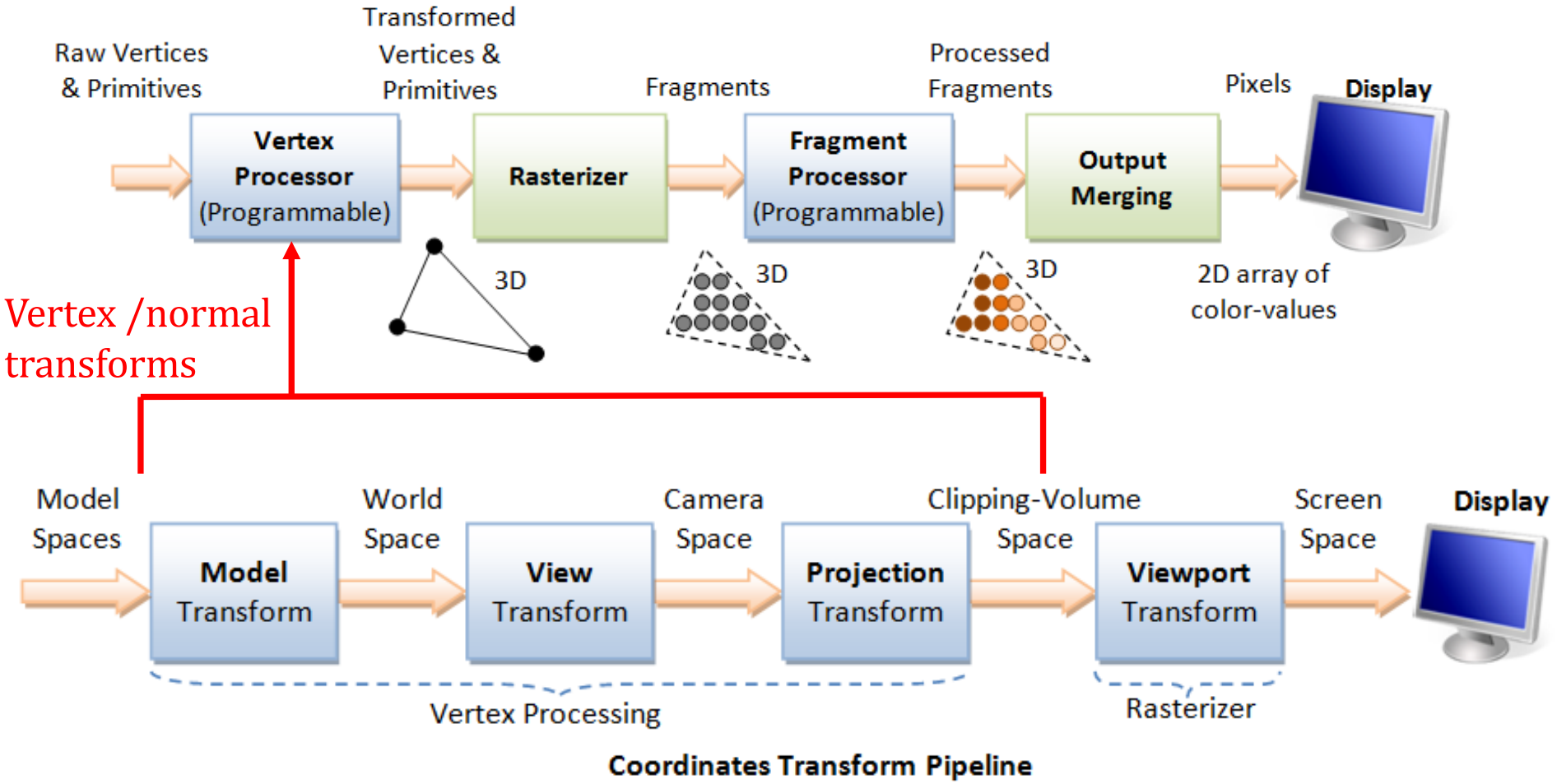
3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

3D Graphics Rendering Pipeline



- The 3D graphics rendering pipeline consists of the following main stages:
  - Vertex Processing: Process and transform individual vertices and normals.
  - Rasterization: Convert each primitive (connected vertices) into a set of fragments. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
  - Fragment Processing: Process individual fragments.
  - Output Merging: Combine the fragments of all primitives (in 3D space) into 2D color-pixel for the display.

Vertex Transforms

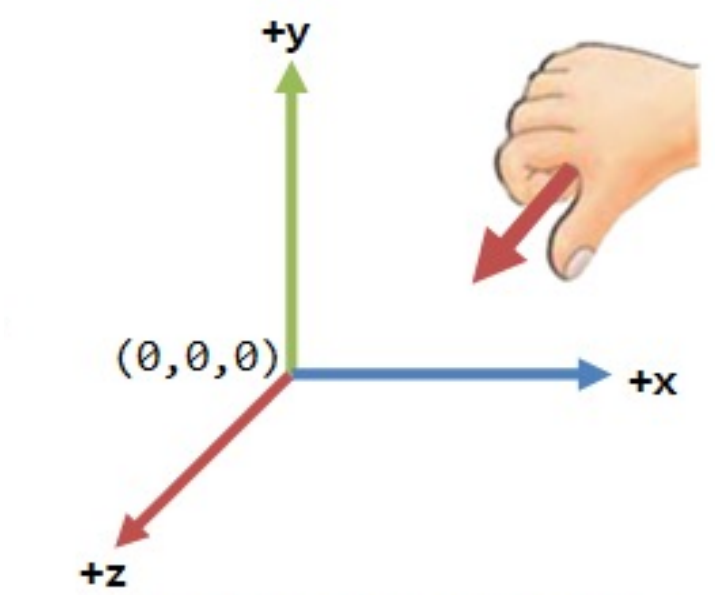


Coordinates Transform Pipeline

Coordinate System

- Right hand coordinate system(RHS)

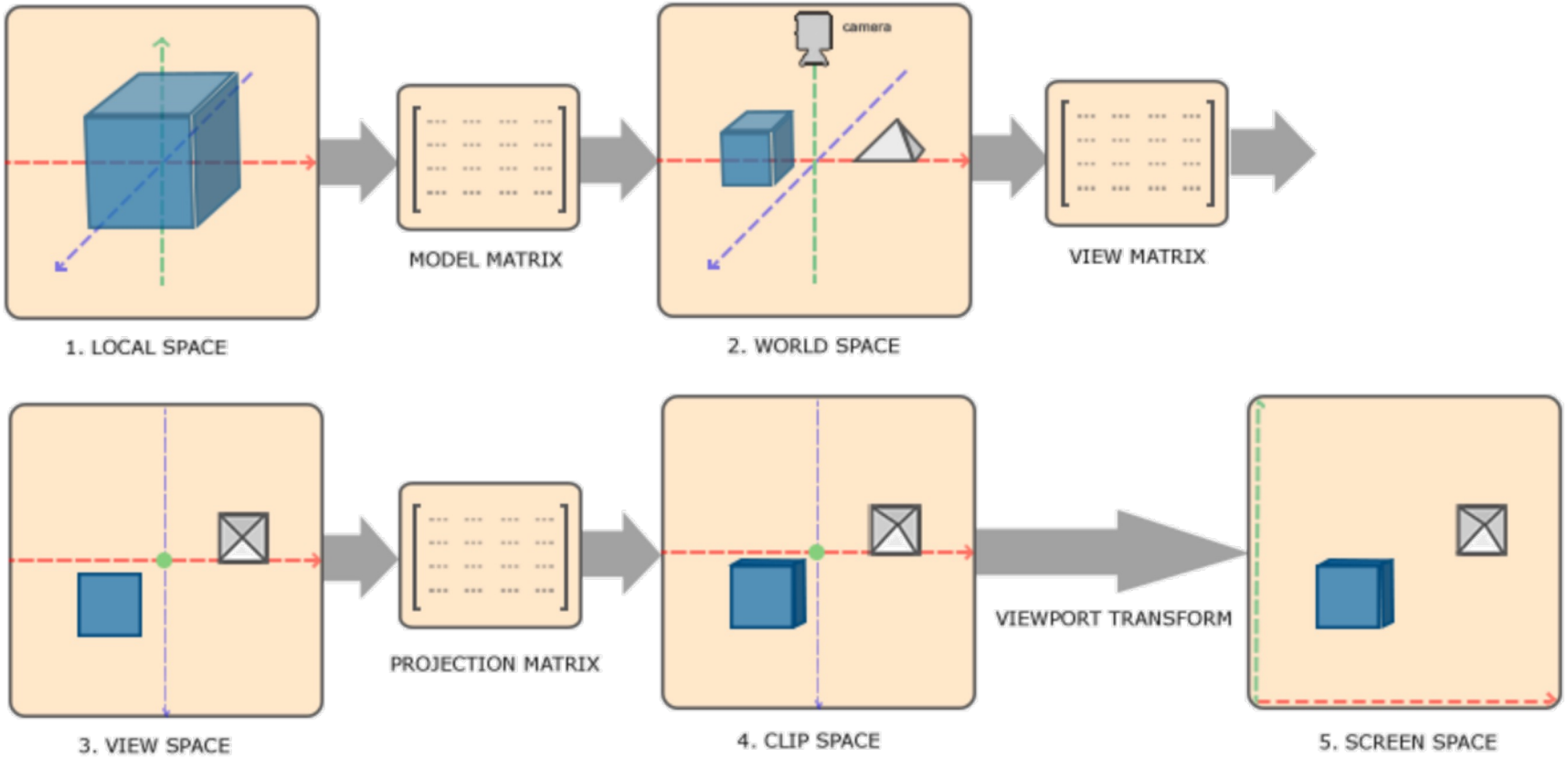
- a few different coordinate systems:
  - object coordinates (local coordinates)
  - world coordinates
  - viewing coordinates
  - also clip, normalized device, and
  - window coordinates



Right-Hand System (RHS) or
Counter-Clockwise (CCW) System

Coordinate System

- To transform the coordinates in one space to the next coordinate space, we will use Model, View and Projection matrix



Coordinate System

1. Local coordinates: are the coordinates of your object relative to its local origin
2. World-space coordinates: are relative to a global origin of the world
3. Viewing coordinates: is as seen from the camera or viewer's point of view.
4. Clip coordinates: are processed to the -1.0 and 1.0 range and determine which vertices will end up on the screen.
5. Screen coordinates: viewport transformation transforms the coordinates from -1.0 to 1.0 to the coordinate range. The resulting coordinates are then sent to the rasterizer to turn them into fragments.

Coordinate System

- 1.Local Space
  - The coordinate space that is local to object
  - The origin of your object is probably at (0,0,0) even though might end up at a different location in your final application
- 2.World space
  - The coordinates of all your vertices relative to a world
  - Accomplished with the Model Matrix
  - (Model Matrix) translates, scales and/or rotates your object to place it in the world at a location/orientation they belong to

Coordinate System

- 3.Viewing space
  - Also known as the camera space or eye space
  - The result of transforming your world-space coordinates to coordinates that are in front of the user's view
  - The space as seen from the camera's point of view
  - Accomplished with the View Matrix (translate/rotate)
  - (View Matrix) Store the combination of translations and rotations to translate/rotate the scene

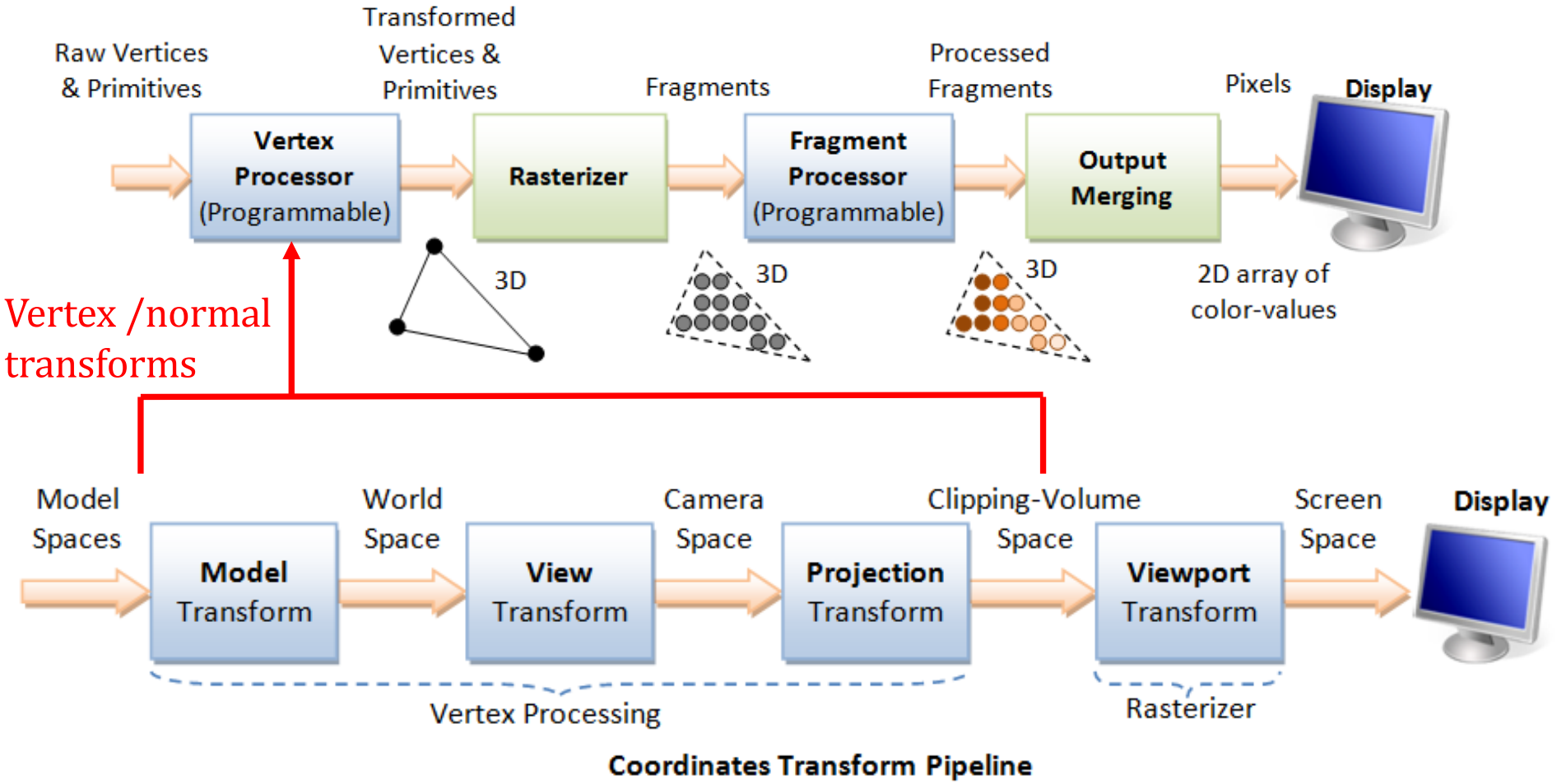
Coordinate System

- 4.Clip space
  - Expects the coordinates to be within a specific range and any coordinate that falls outside this range is clipped
  - To transform vertex coordinates from view to clip-space, we use the Projection Matrix
  - (Projection Matrix) transforms coordinates within specified range (e.g. -1000 to 1000) to normalized device coordinates (-1.0, 1.0)

Coordinate System

- Frustum
  - A projection matrix creates a viewing box, called frustrum
  - Each vertex inside this box will be projected to -1~1 end up on the user's screen, each vertex outside this box will be clipped.
- Projection
  - The total process to convert coordinates within a specified range to NDC that can easily be mapped to 2D view-space coordinates
- The projection matrix has two forms
  - Orthographic projection matrix
  - Perspective projection matrix

Vertex Transforms

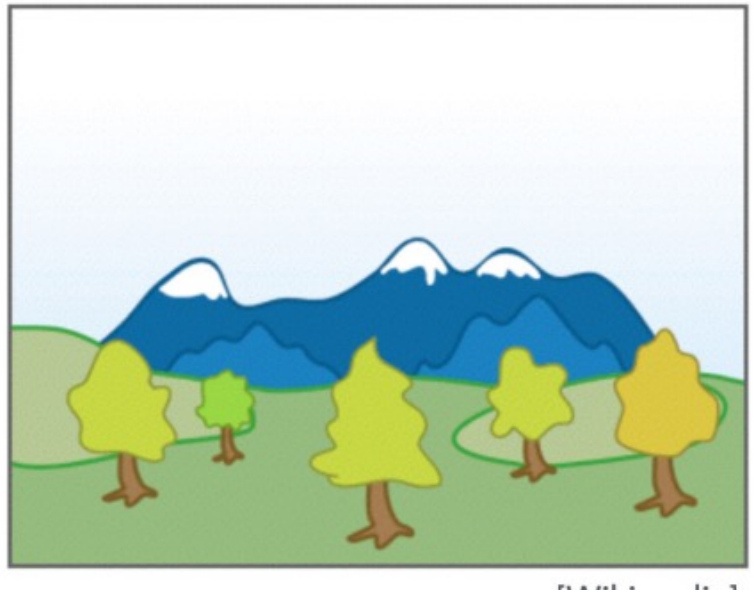


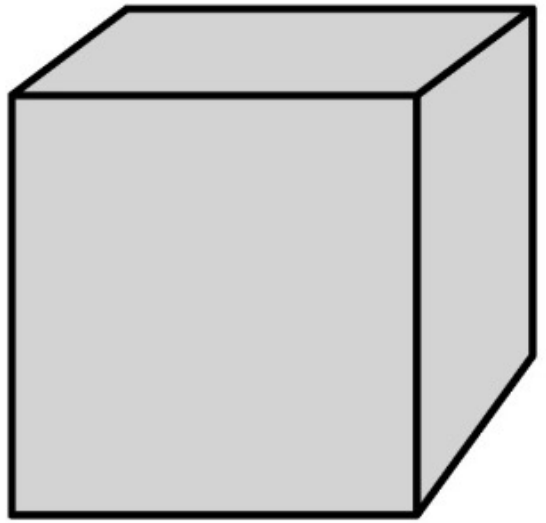
2. Visibility

Painter's Algorithm

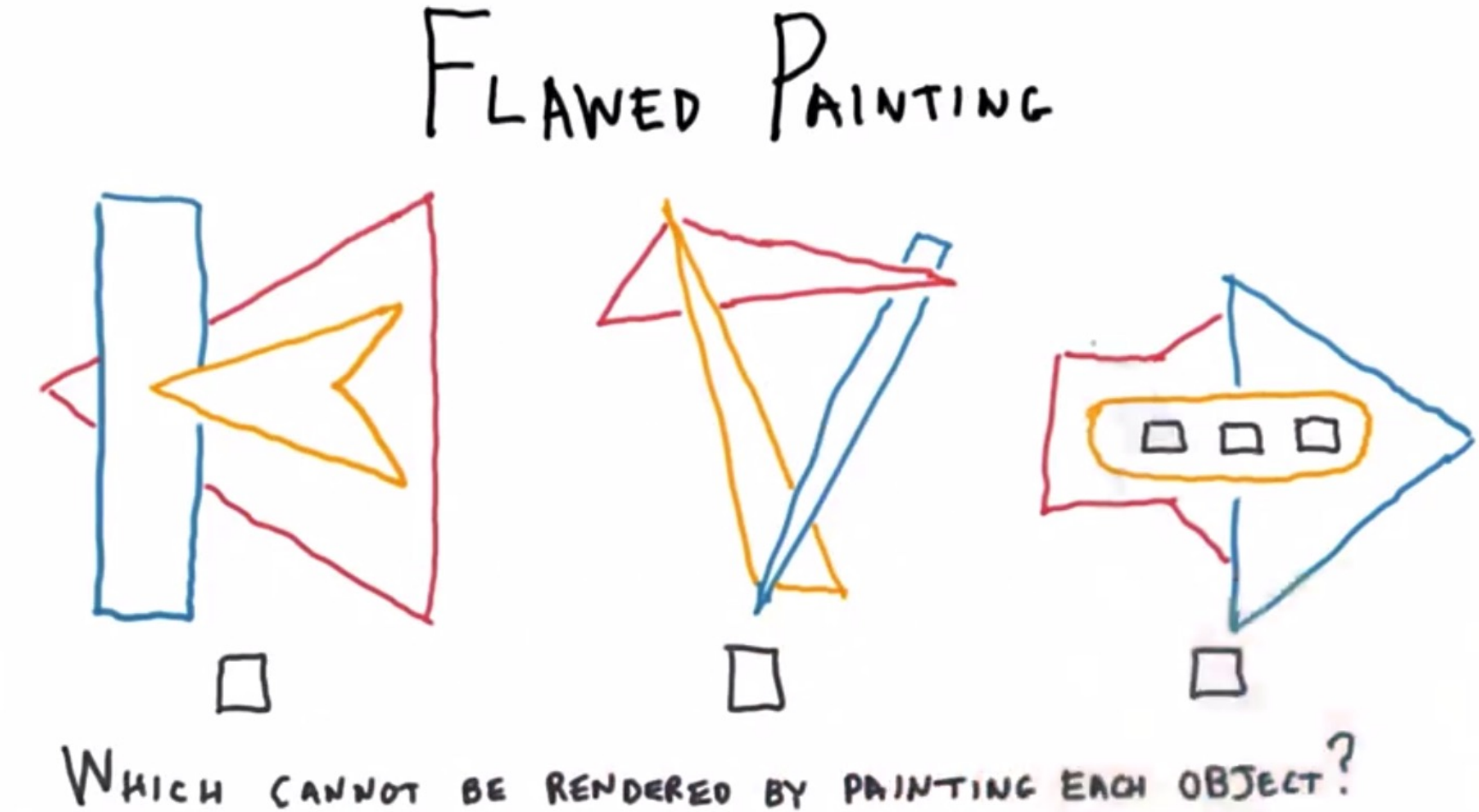
- Inspired by how painters paint

  Paint from back to front, overwrite in the framebuffer





Flawed Painting Exercise

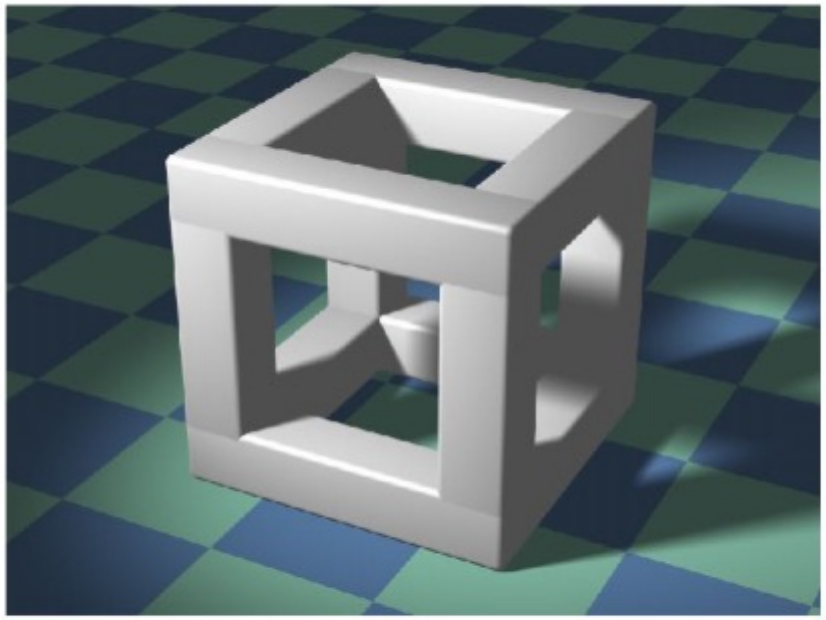


Z-Buffer

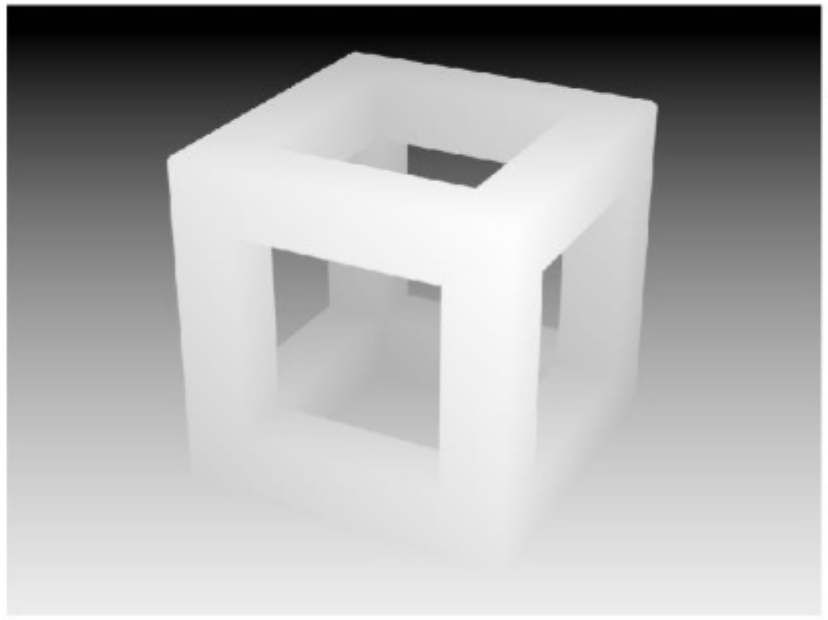
- This is the hidden-surface-removal algorithm that eventually won
- **Idea**
  - Store current min. z-value for each sample position
  - Needs an additional buffer for depth values
    - framebuffer stores RGB color values
    - depth buffer (z-buffer) store depth

Z-Buffer(Video)

Z-Buffer Example



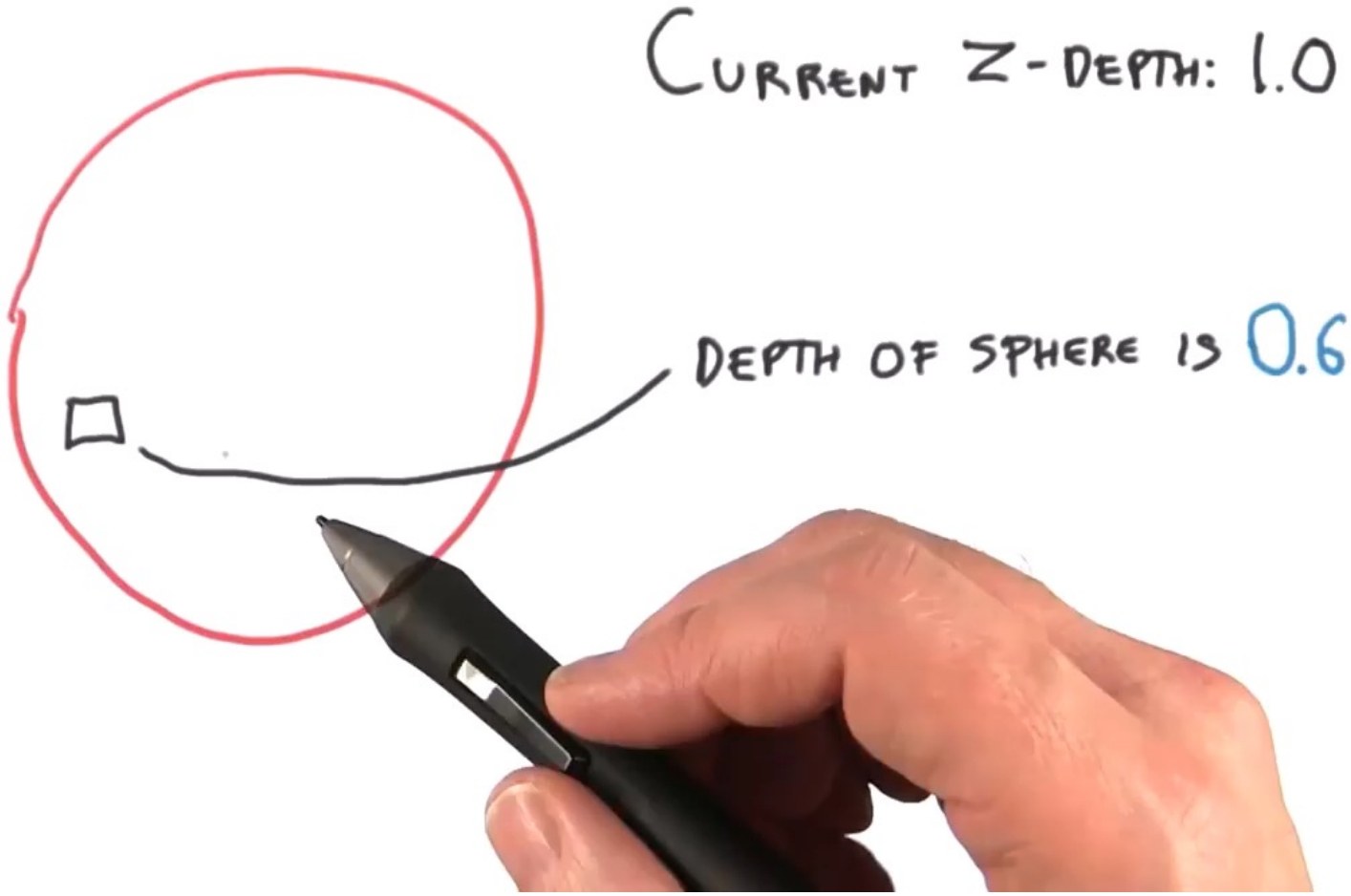
Rendering



Depth buffer

Image credit: Dominic Alves, flickr.

Z-Buffer Algorithm(Video)



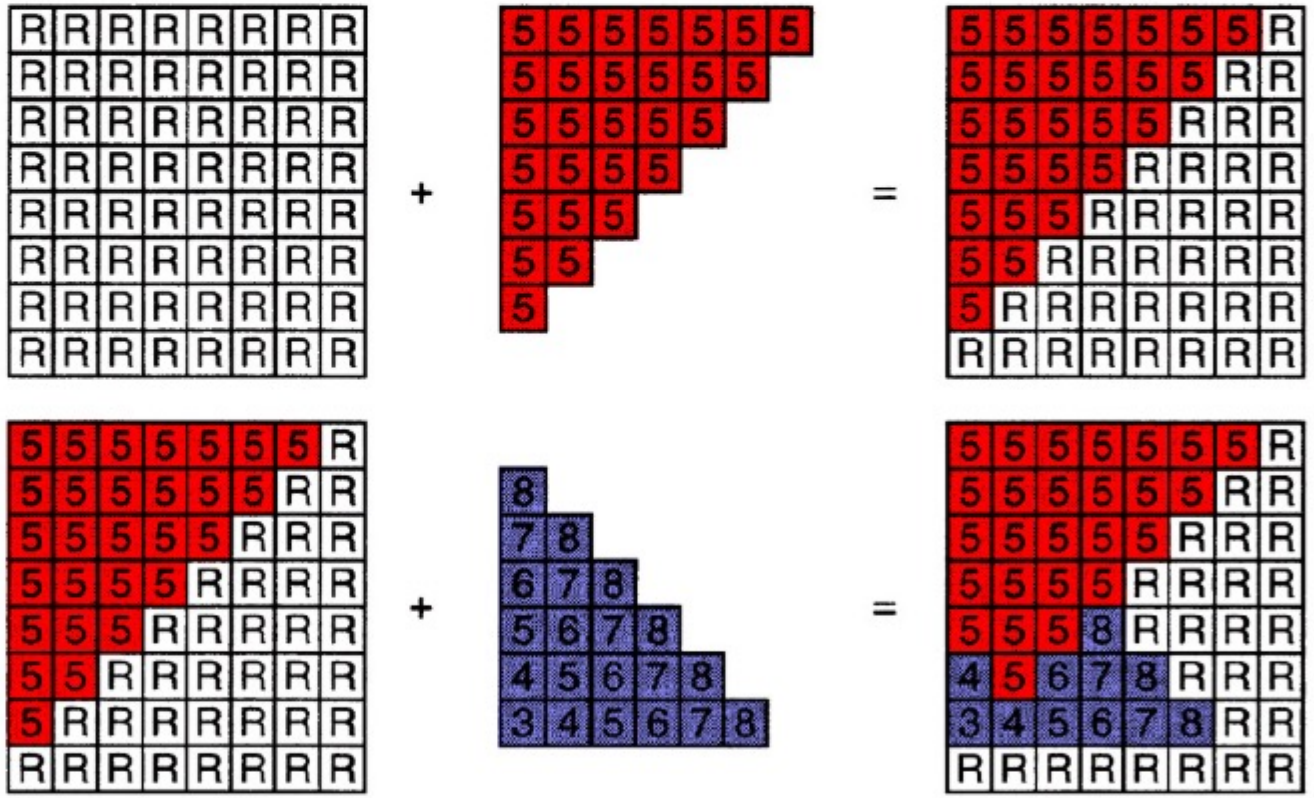
Z-Buffer Algorithm

Initialize depth buffer to ∞

During rasterization:

```
for (each triangle T)
  for (each sample (x,y,z) in T)
    if (z < zbuffer[x,y])          // closest sample so far
      framebuffer[x,y] = rgb;      // update color
      zbuffer[x,y]     = z;        // update z
    else
      ;     // do nothing, this sample is not closest
```

Z-Buffer Algorithm

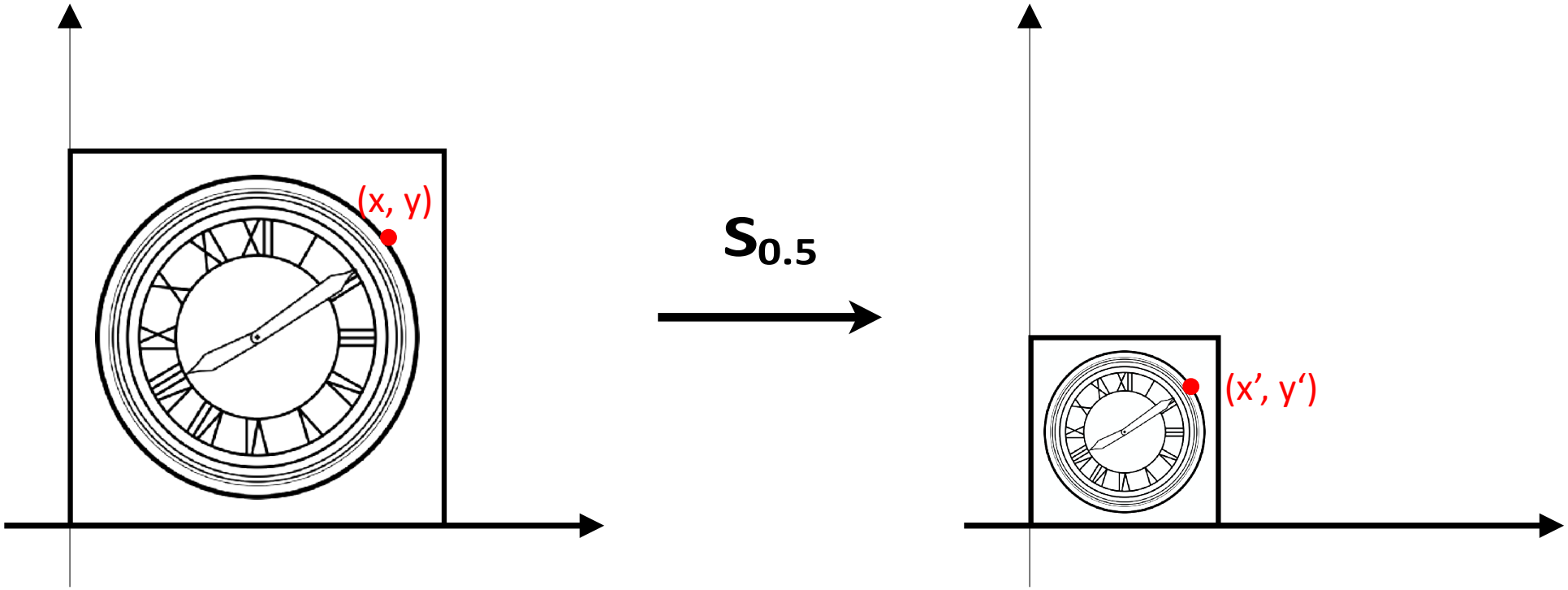


3. Basic Transform

Why study Transforms?

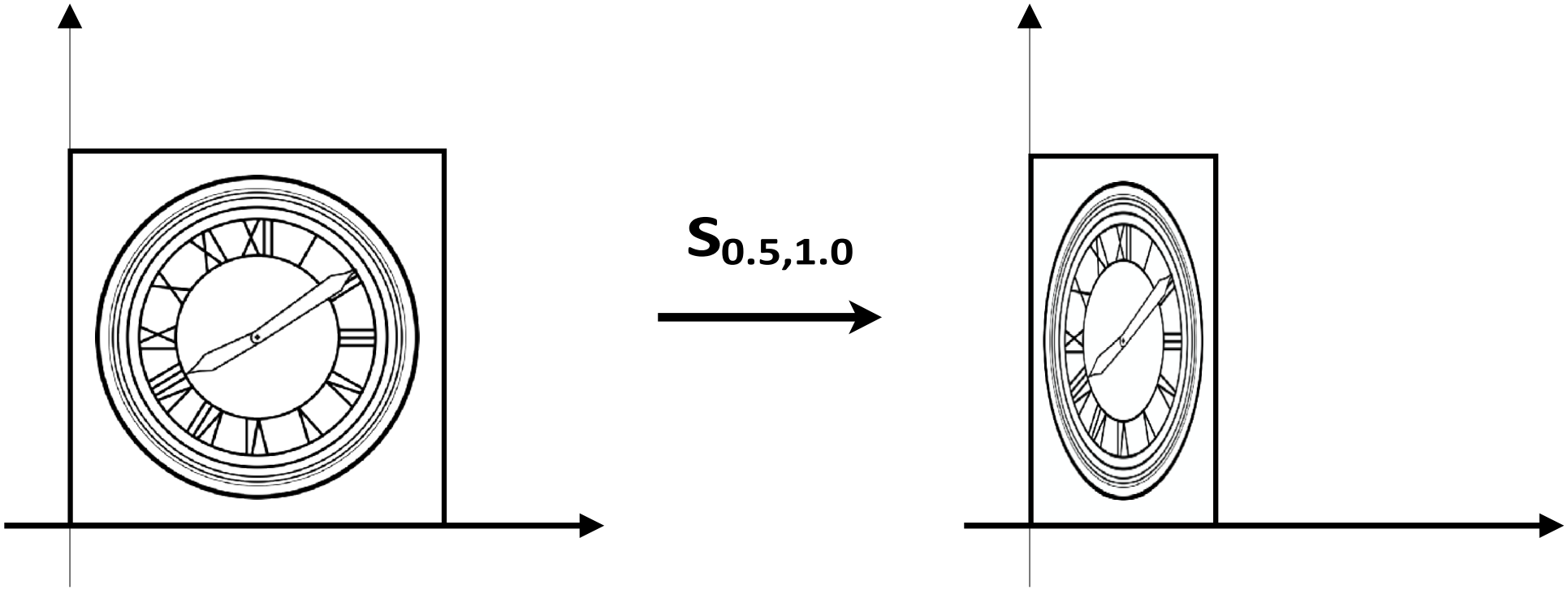
- Modeling
  - Define shapes in convenient coordinates
  - Enable multiple copies of the same object
  - Efficiently represent hierarchical scenes
- Viewing
  - World coordinates to camera coordinates
  - Parallel / perspective projections from 3D to 2D

Scale Transform



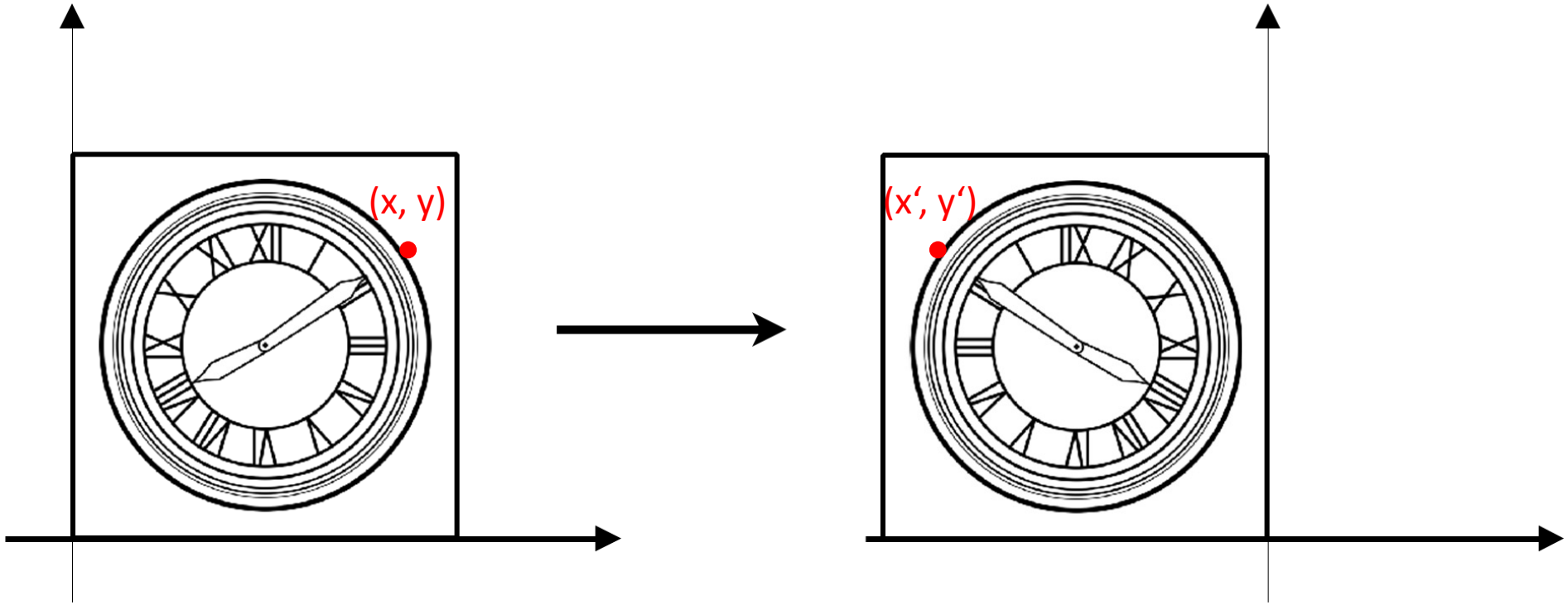
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Scale Matrix (Non-Uniform)



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

Reflection Transform

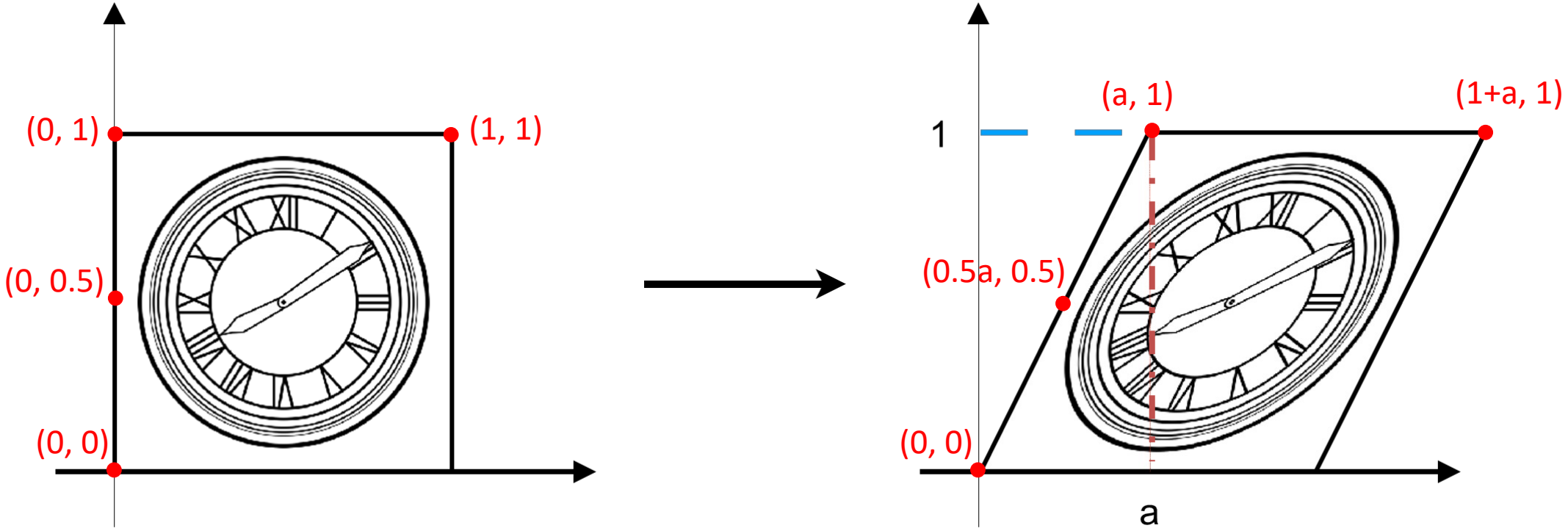


$$x' = -x$$

$$y' = \quad y$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

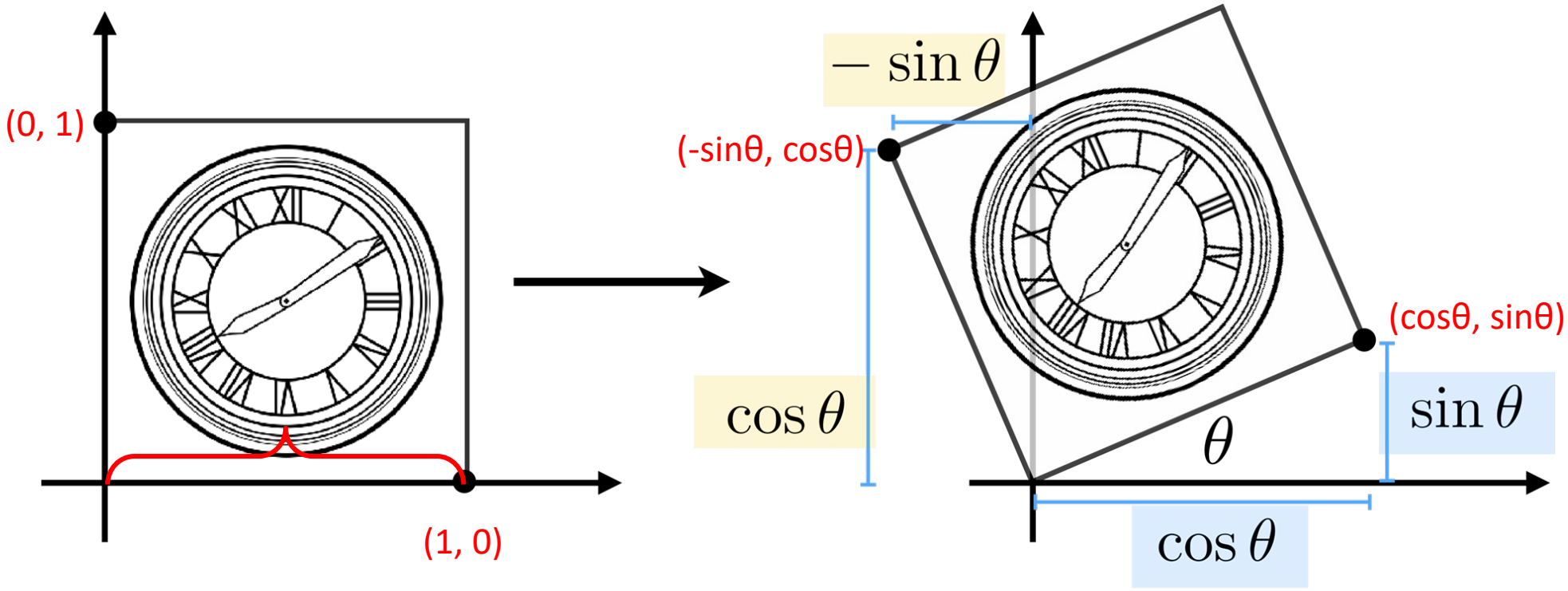
Shear Transform



Horizontal shift is 0 at y=0
Horizontal shift is a at y=1

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotate Transform



$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

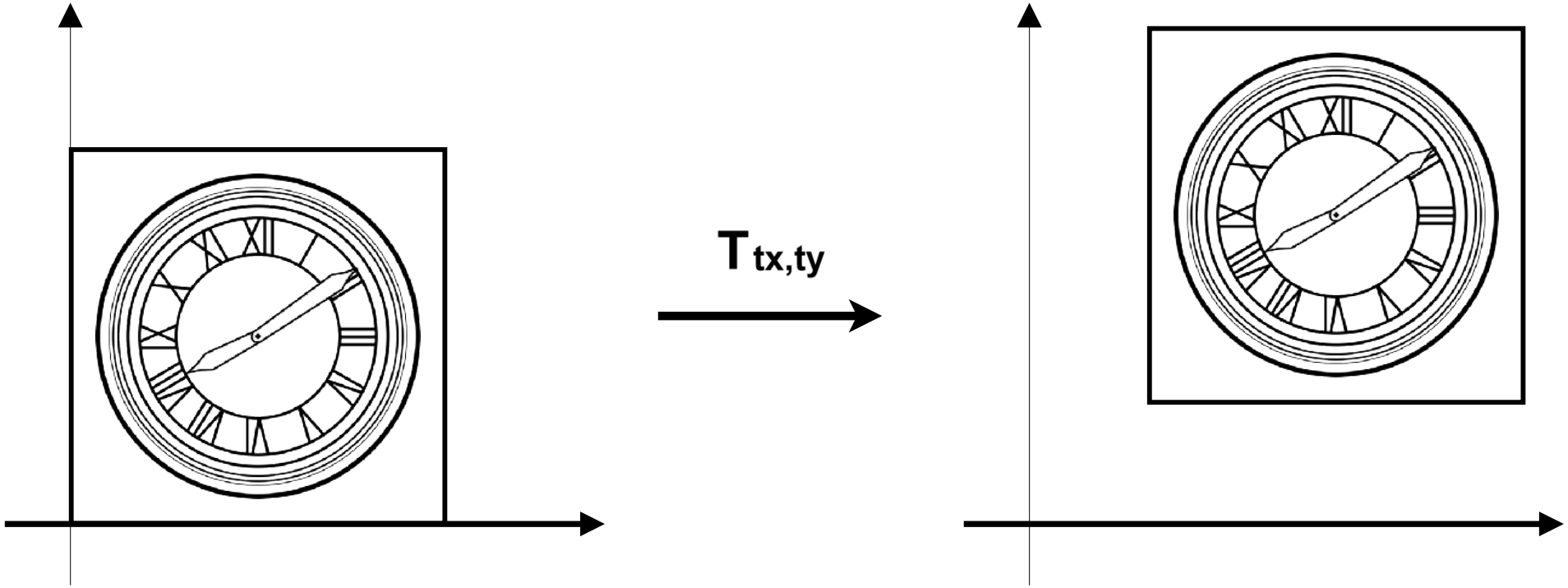
Linear Transforms = Matrices

$$x' = a\,x + b\,y$$
$$y' = c\,x + d\,y$$

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

$$\mathbf{x}' = \mathbf{M}\,\mathbf{x}$$

Translation??



$$x' = x + t_x$$
$$y' = y + t_y$$

Why Homogenous Coordinates

- Translation cannot be represented in matrix form

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

(So, translation is NOT linear transform!)

- But we don't want translation to be a special case

- Is there a unified way to represent all transformations?

(and what's the cost?)

Solution: Homogenous Coordinates

- Add a third coordinate (w-coordinate)
  - 2D point $= (x, y, 1)^T$
  - 2D vector $= (x, y, 0)^T$
- Matrix representation of translations

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

- What if you translate a vector?

Homogenous Coordinates

- Valid operation if w-coordinate of result is 1 or 0
  - vector + vector = vector
  - point – point = vector
  - point + vector = point
  - point + point = ??
- In homogeneous coordinates
- $\begin{pmatrix} x \\ y \\ w \end{pmatrix}$ is the 2D point $\begin{pmatrix} x/w \\ y/w \\ 1 \end{pmatrix}$, w$\neq$ 0

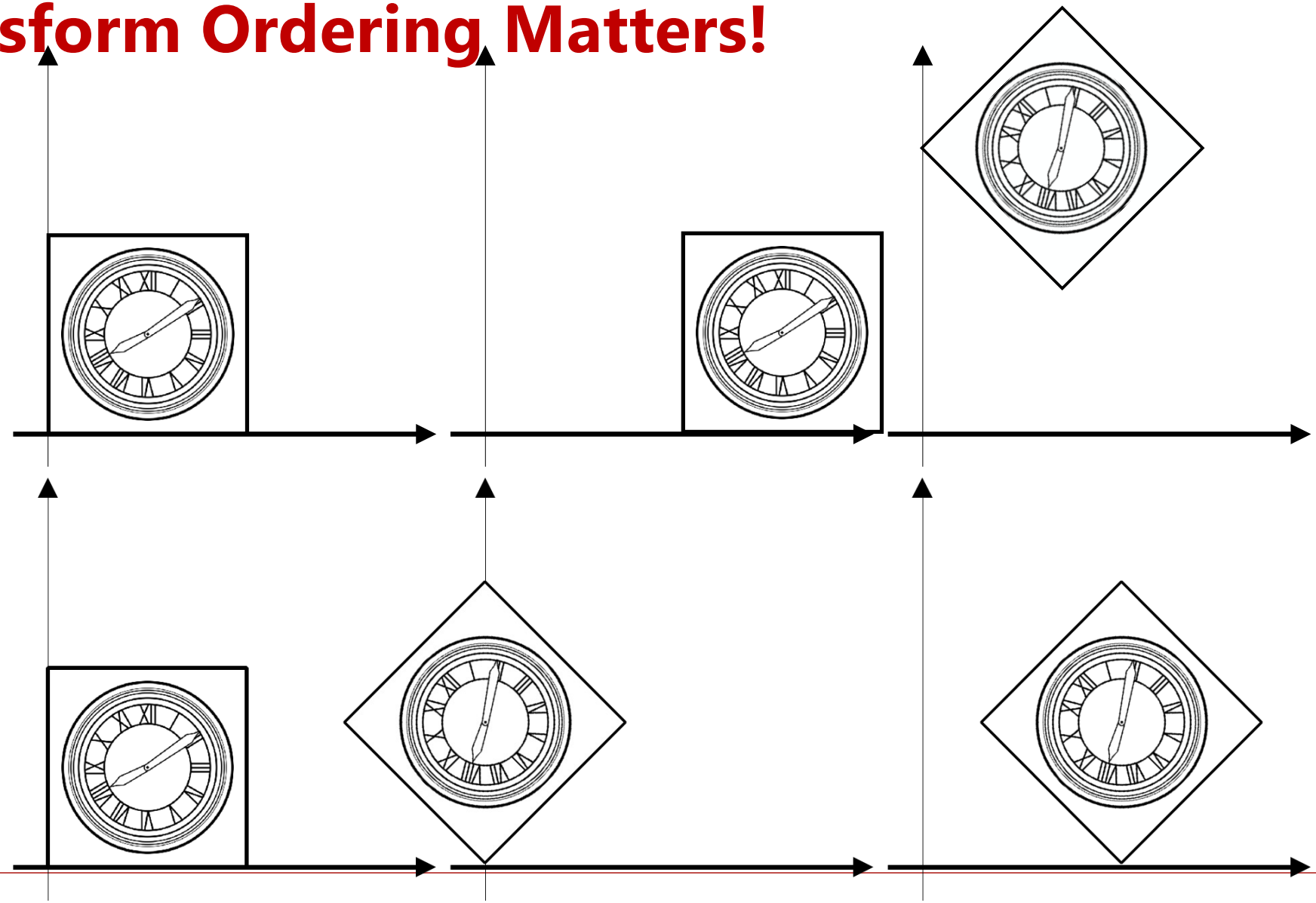
Affine Transformations

- Affine map = linear map + translation
  - $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$
- Using homogeneous coordinates
  - $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

2D Transformations

- Scale $S(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$
- Rotation $\mathrm{R}(\alpha) = \begin{pmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$
- Translation $\mathrm{T}(t_x, t_y) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$

Transform Ordering Matters!



$R_{45} \cdot T_{(1,0)}$

$T_{(1,0)} \cdot R_{45}$

Transform Ordering Matters!

- Multiplication is not commutative

$$\boldsymbol{R_{45} \cdot T_{(1,0)} \neq T_{(1,0)} \cdot R_{45}}$$

- Note that matrices are applied right to left:

$$T_{(1,0)} \cdot R_{45} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Composing Transforms

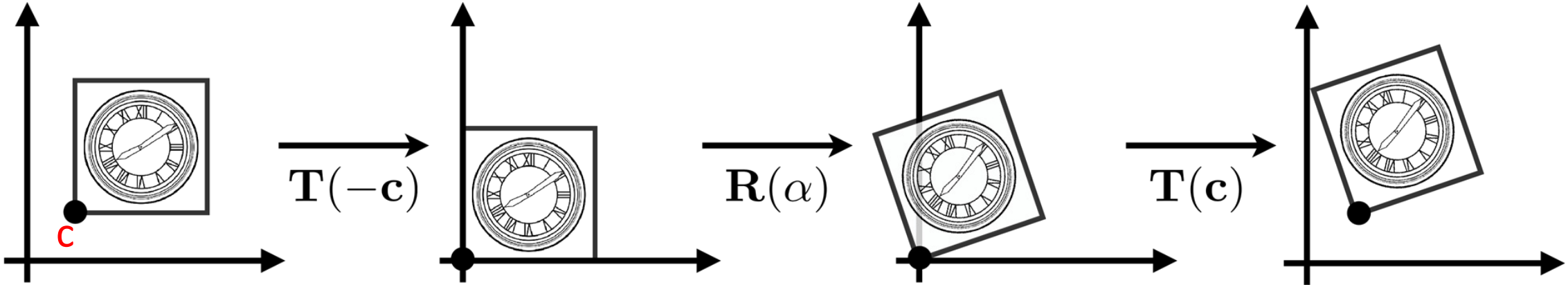
- Sequence of affine transforms A1, A2, A3, ...
  - Compose by matrix multiplication
  - Very important for performance!

$$A_n(\ldots A_2(A_1(\mathbf{x}))) = \mathbf{A}_n \cdots \mathbf{A}_2 \cdot \mathbf{A}_1 \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Pre-multiply *n* matrices to obtain a single matrix representing combined transform

Decomposing Complex Transforms

- How to rotate around a given point c?
  - Translate center to origin
  - Rotate
  - Translate back



- Matrix representation?

3D Transformations

- Using homogeneous coordinates again:
  - 3D point $= (x, y, z, 1)^T$
  - 3D vector $= (x, y, z, 0)^T$

- In general, $(x, y, z, w)(w! = 0)$ is the 3D point:

$$(x/w, y/w, z/w)$$

3D Transformations

- Using 4*4 matrices for affine transformations

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- What's the order?
- Linear Transform first or Translation first?

3D Transformations

- Scale

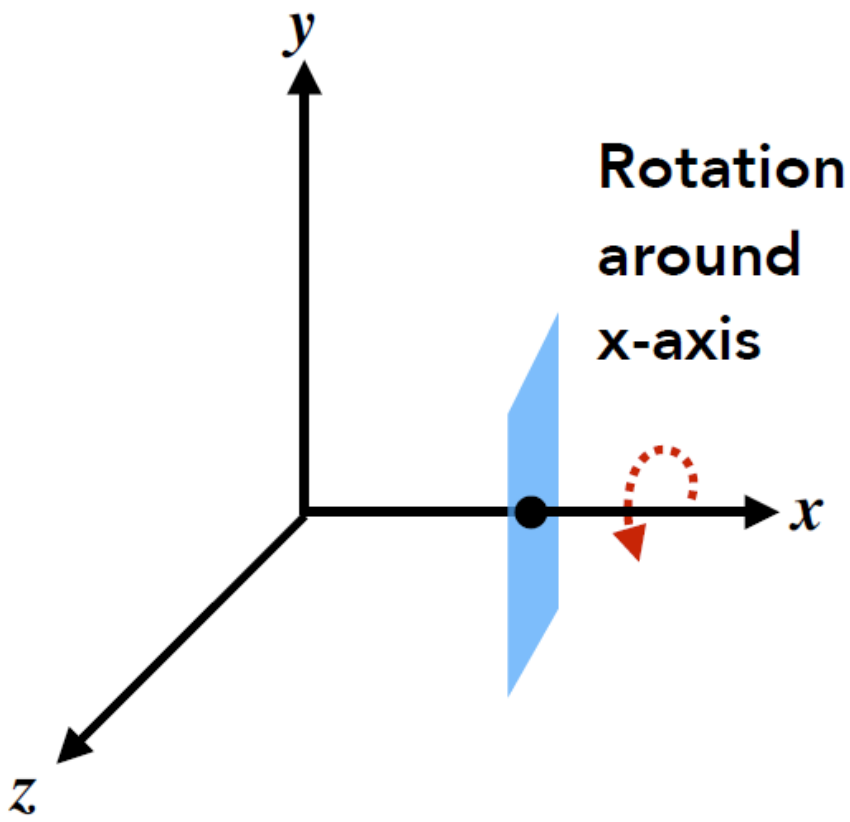
$$\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Translation

$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Transformations

- Rotation around x, y, z-axis



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

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

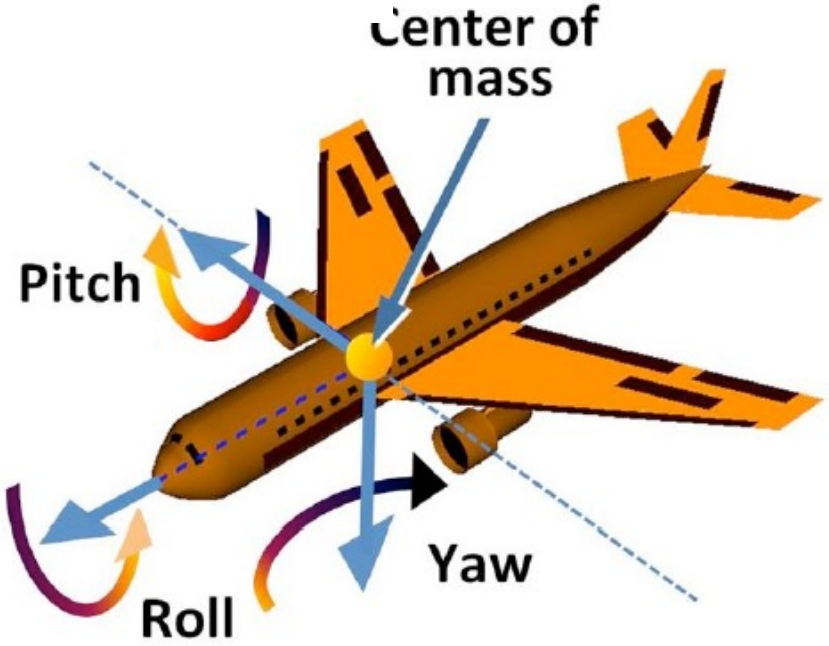
$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Rotations

- Compose any 3D rotation from Rx, Ry, Rz?

$$\mathbf{R}_{xyz}(\alpha, \beta, \gamma) = \mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)$$

- So-called Euler angles
- Often used in flight simulators:
  - roll, pitch, yaw



3D Rotations——Rodrigues' Rotation Formula

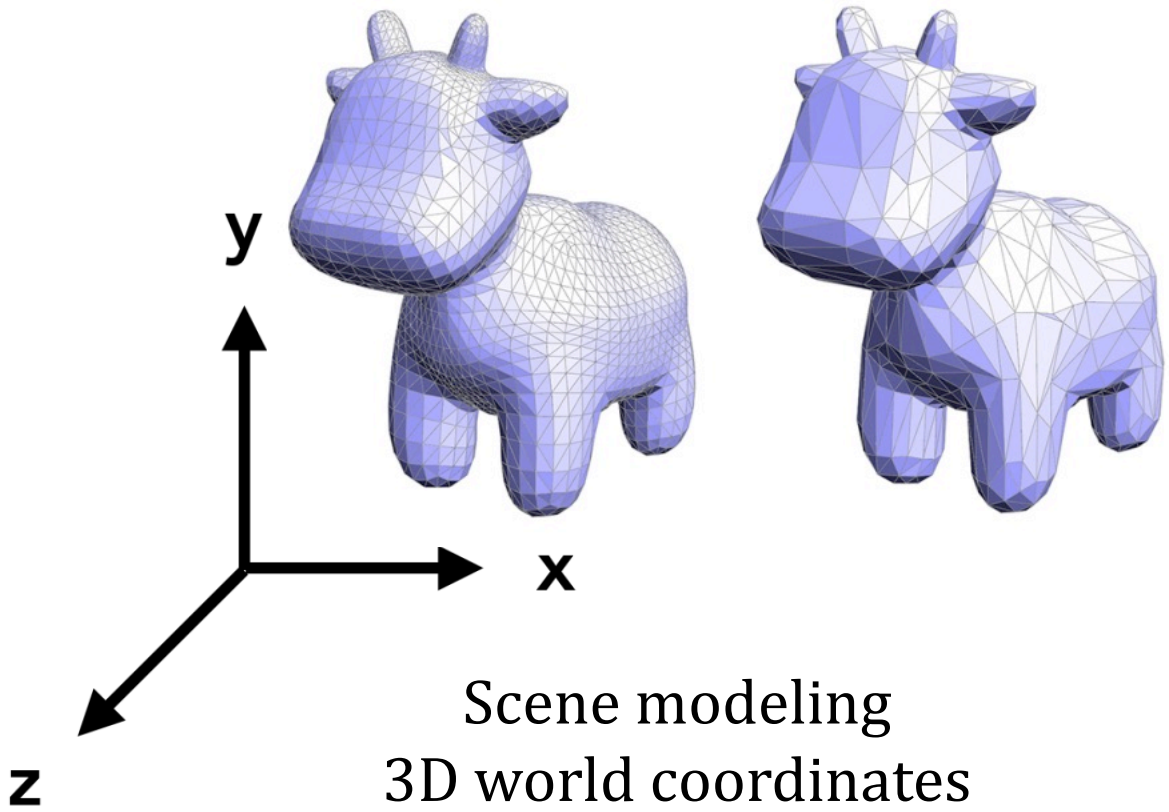
- Rotation by angle α around axis n

$$\mathbf{R}(\mathbf{n},\alpha) = \cos(\alpha)\,\mathbf{I} + (1-\cos(\alpha))\,\mathbf{n}\mathbf{n}^T + \sin(\alpha)\underbrace{\begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}}_{\mathbf{N}}$$

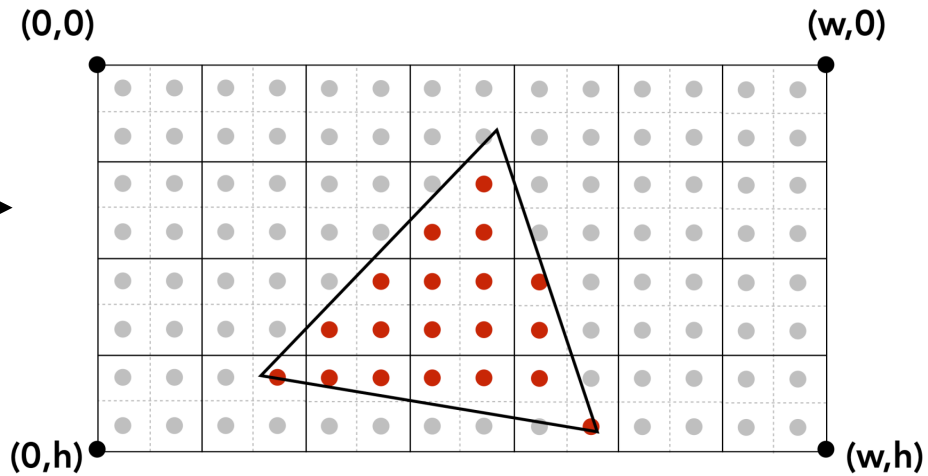
- Try to prove this magic formula?

4. Viewing and Perspective Transform

Viewing and Perspective Transforms



Scene modeling
3D world coordinates



Rasterization
2D screen coordinates

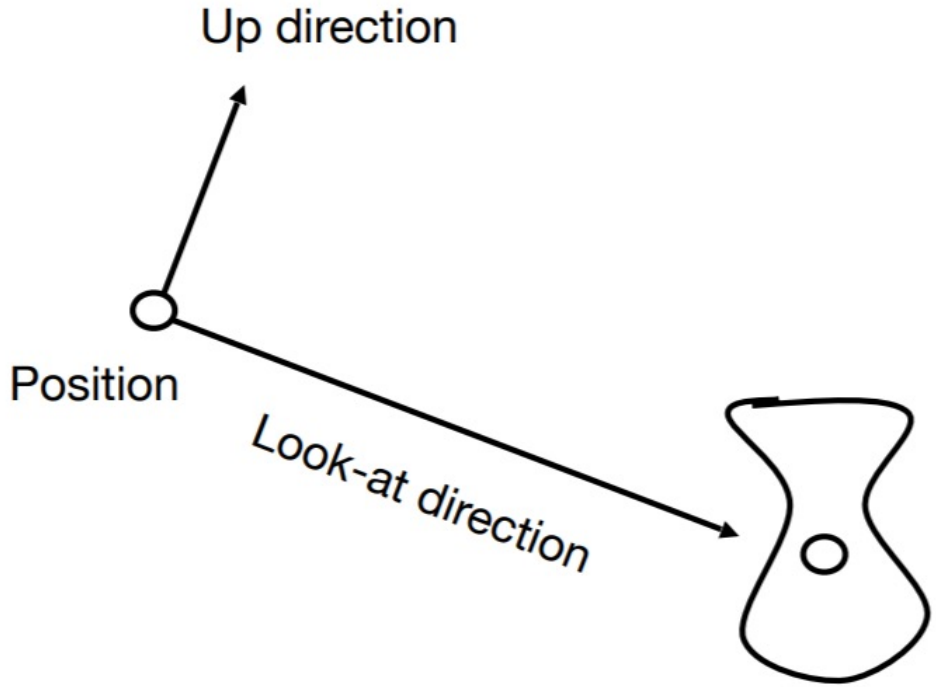
Viewing Transformation

- What is a view transformation?
- Think about how to take a photo
  - Find a good place and arrange people (model transformation)
  - Find a good “angle” to put the camera (view transformation)
  - Cheese!/茄子(projection transformation)

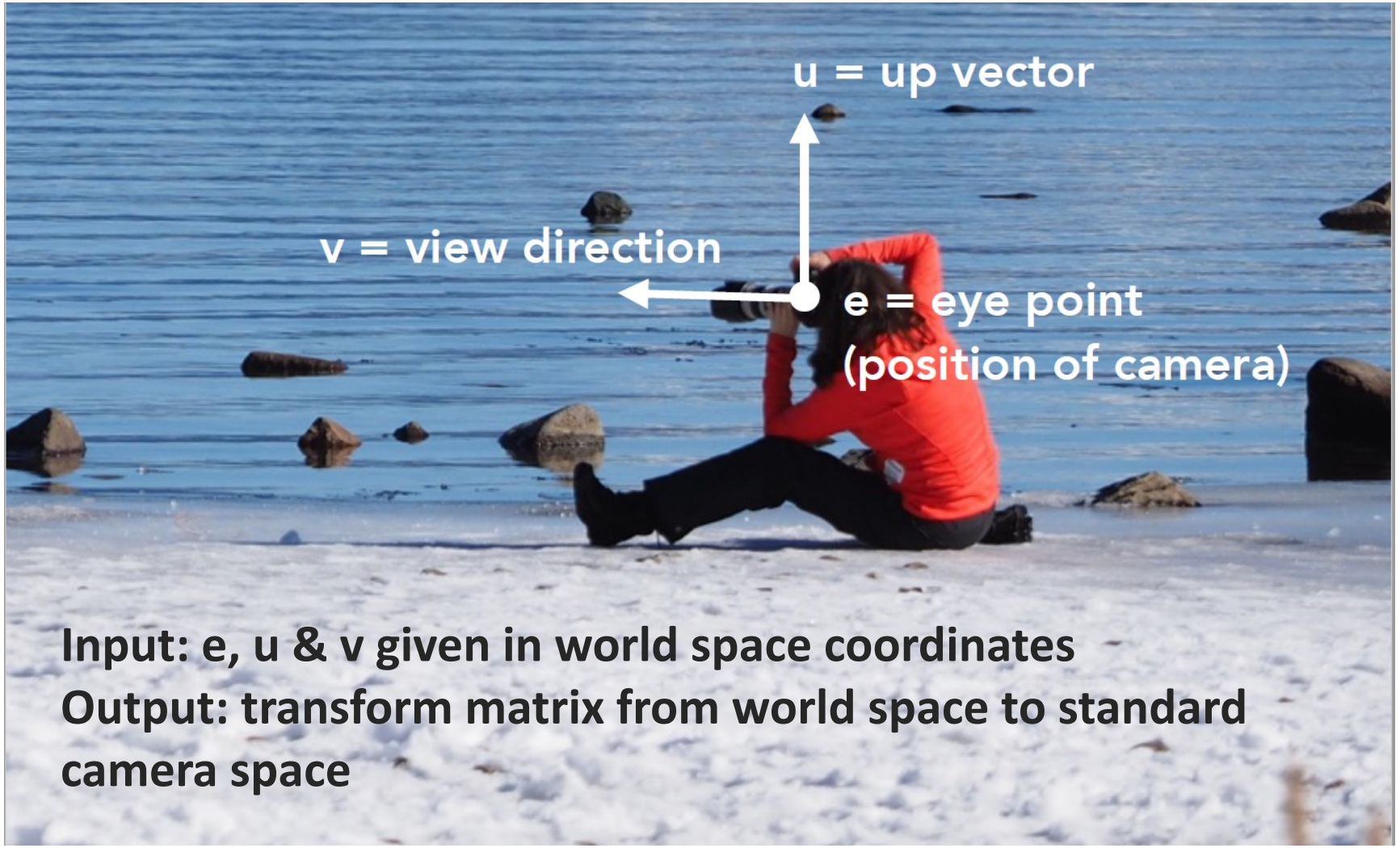
View / Camera Transformation

- How to perform view transformation?
- Define the camera first
  - Position $\vec{e}$
  - Look-at / gaze direction $\vec{g}$
  - Up direction $\vec{t}$

  (assuming perp. to look-at)



Considering A Camera Pointing in The World

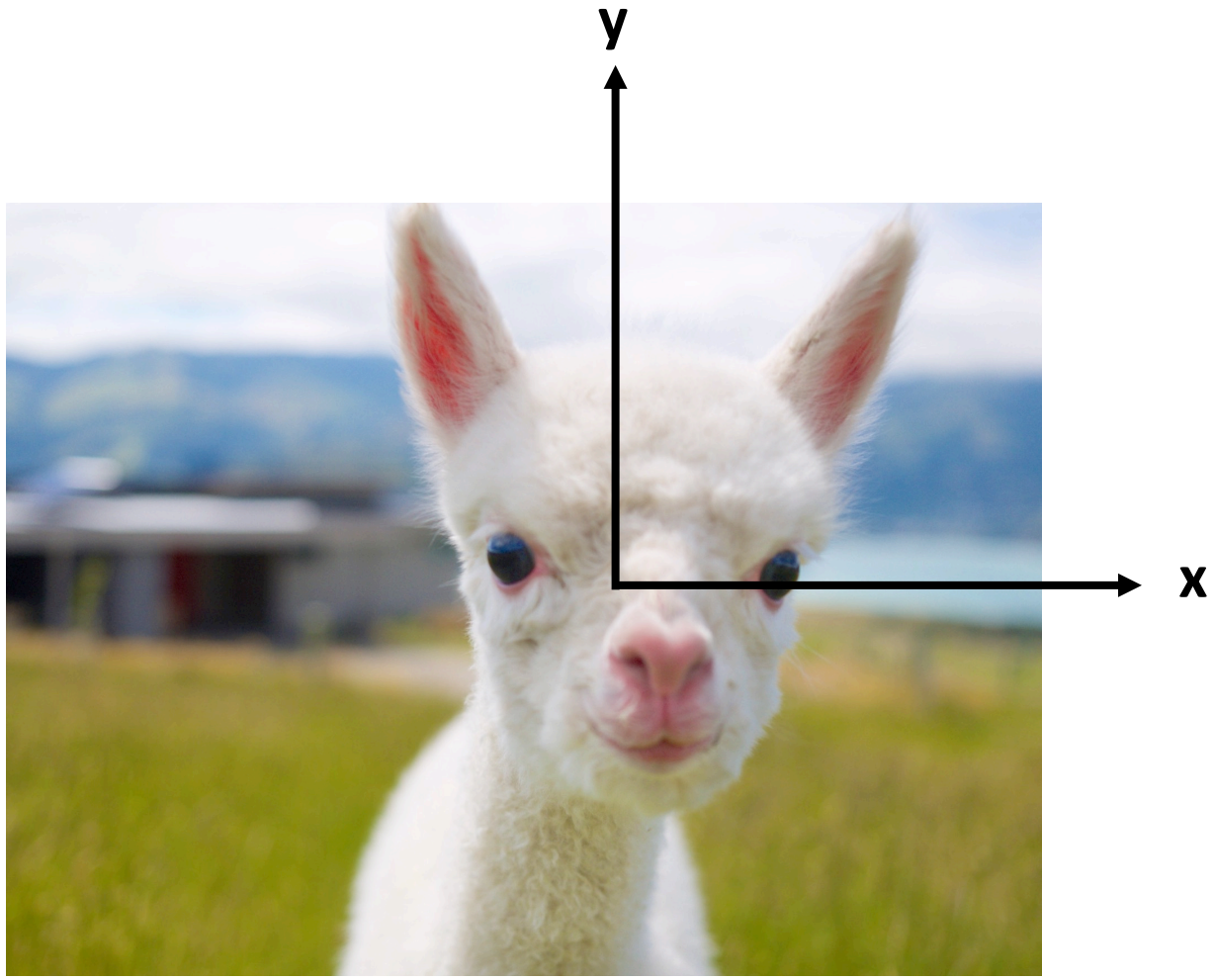


"Standard" Camera Space

- We will use this convention for "standard" camera coordinates:

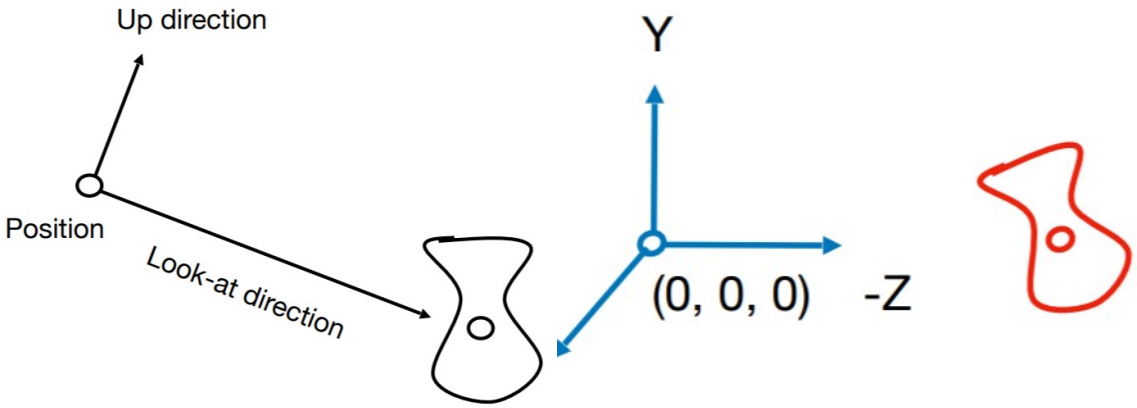
  - camera located at the origin
  - looking down negative z-axis
  - vertical vector is y-axis
  - (x-axis) orthogonal to y & z

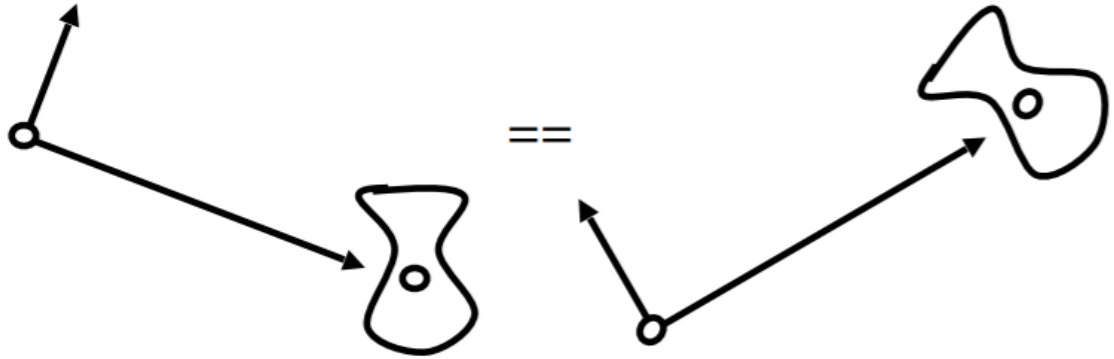
“Standard” Camera Space



Resulting image
(z-axis pointing away from scene)

Considering A Camera Pointing in The World





- Key observation
  - If the camera and all objects move together, the "photo" will be the same
- How about that we always transform the camera to
  - The origin, up at $\vec{Y}$, look at $-\vec{Z}$
  - And transform the objects along with the camera

View/Camera Transformation

- *Mview in math?*
  - Let's write *Mview = RviewTview*
  - 1）Translate $\vec{e}$ to origin (0,0,0)

$$T_{view} = \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

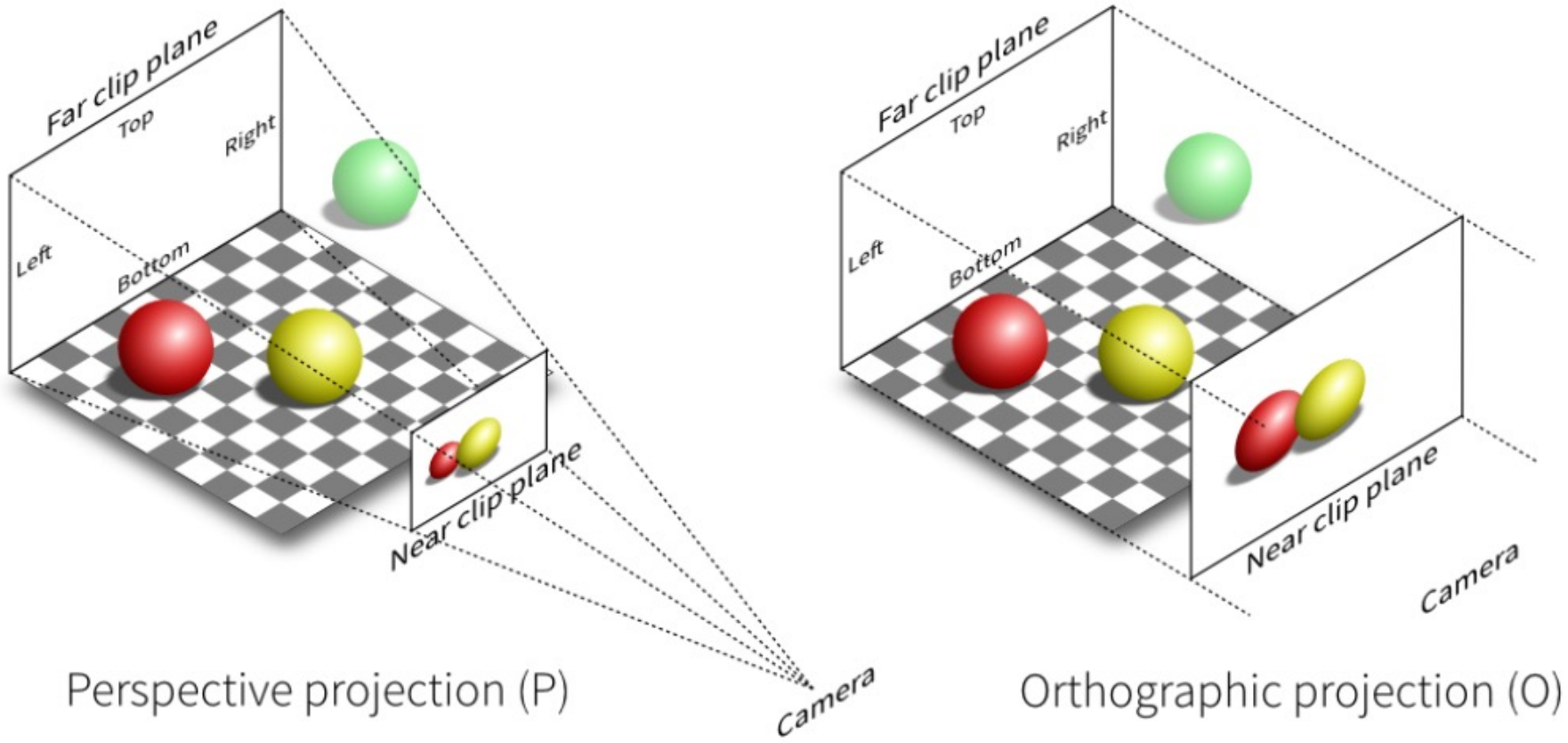
  - 2）Rotate $\vec{g}$ to -$\vec{Z}$, $\vec{t}$ to $\vec{Y}$, ($\vec{g}$ x $\vec{t}$) to $\vec{X}$
  - Consider its inverse rotation: $\vec{X}$ to ($\vec{g}$ x $\vec{t}$), $\vec{Y}$ to $\vec{t}$, $\vec{Z}$ to - $\vec{g}$

$$R_{view}^{-1} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & x_t & x_{-g} & 0 \\ y_{\hat{g}\times\hat{t}} & y_t & y_{-g} & 0 \\ z_{\hat{g}\times\hat{t}} & z_t & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

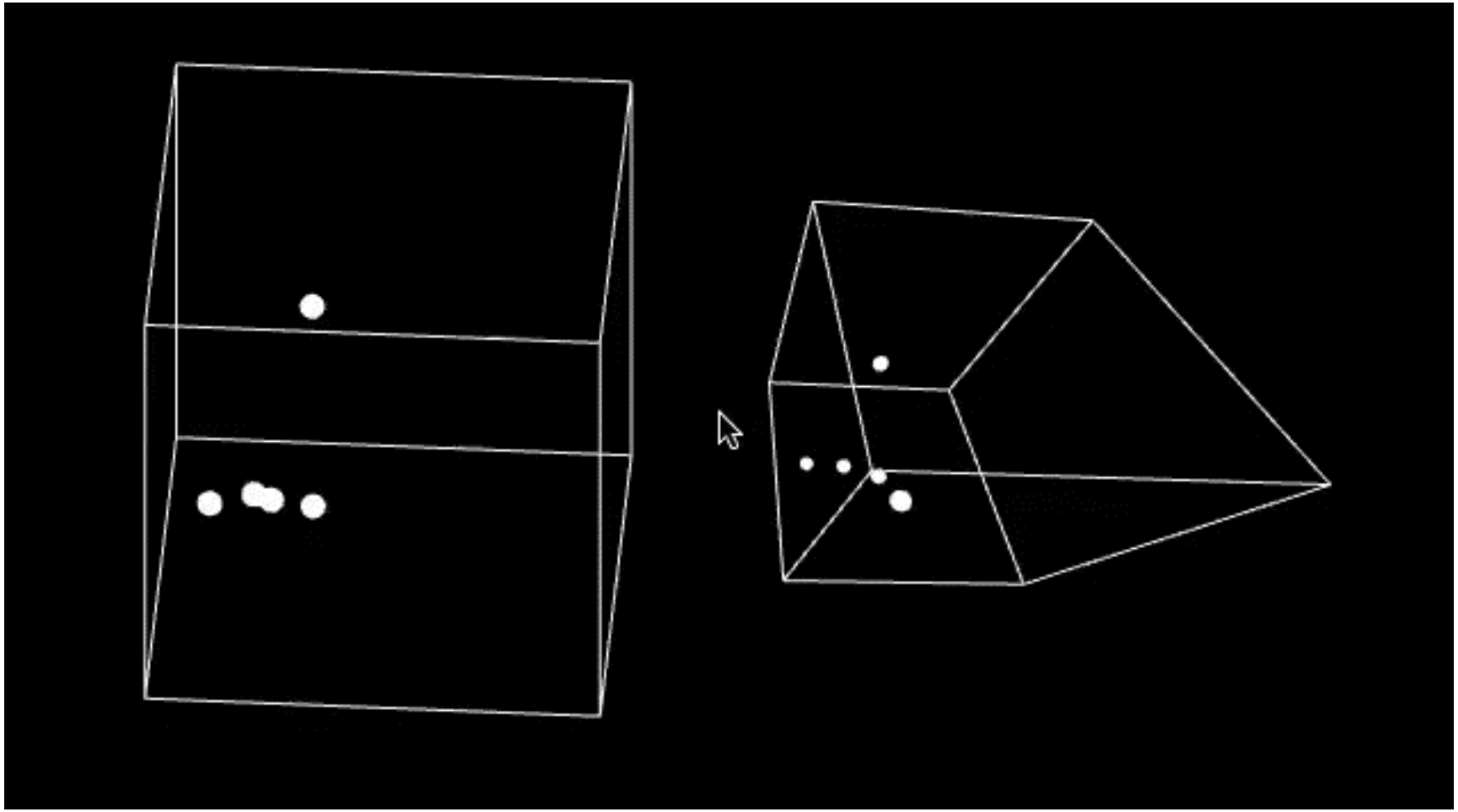
WHY?

$$R_{view} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & y_{\hat{g}\times\hat{t}} & z_{\hat{g}\times\hat{t}} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective vs Orthographic Projection

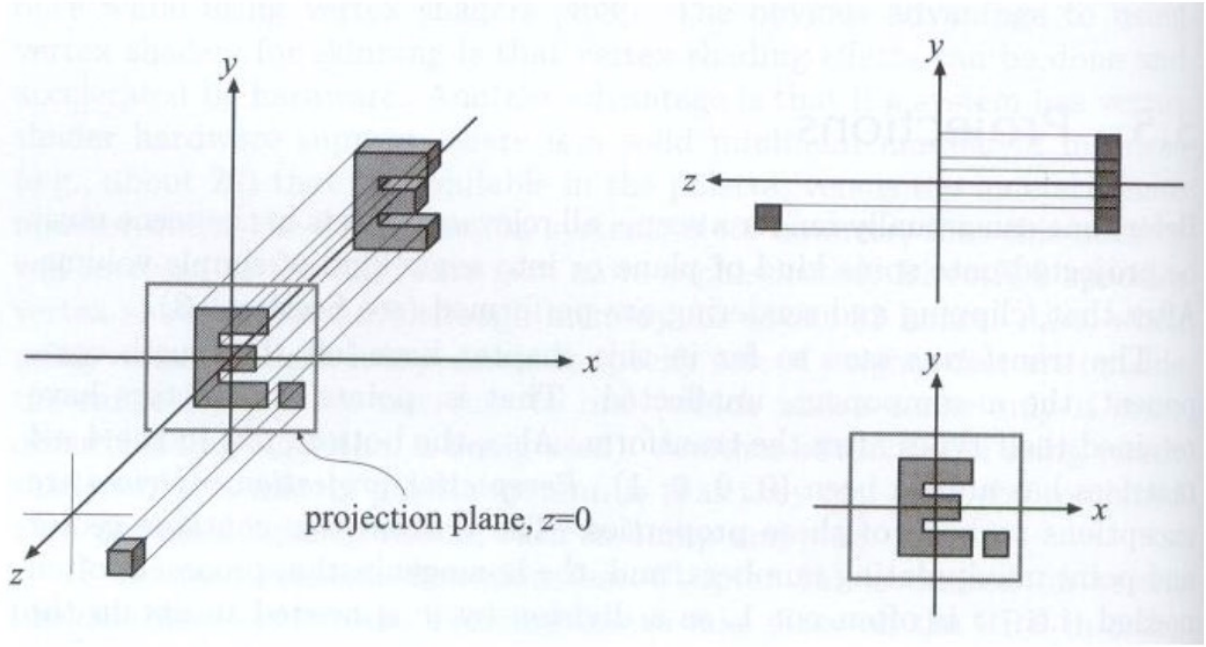


Perspective vs Orthographic Projection(Video)



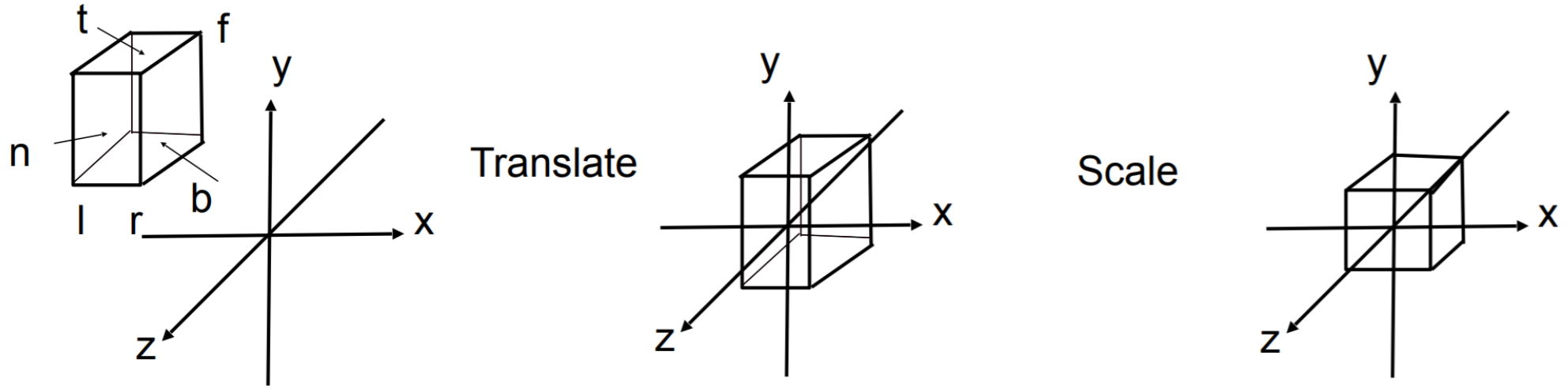
Orthographic Projection

- A simple way of understanding
  - Camera located at origin, looking at -Z, up at Y (looks familiar?)
  - Drop Z coordinate
  - Translate and scale the resulting rectangle to $[-1, 1]^2$



Orthographic Projection

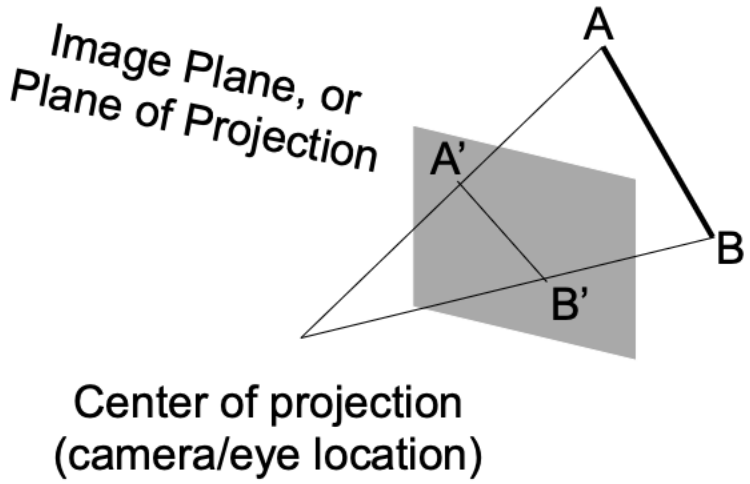
- Camera positioned infinitely far away at z = ∞
- In general
  - We want to map a cuboid [l, r] x [b, t] x **[f, n]** to the "canonical (正则、规范、标准)" cube [-1, 1]^3
  - Center cuboid by translating
  - Scale into "canonical" cube




Perspective

Perspective Projection

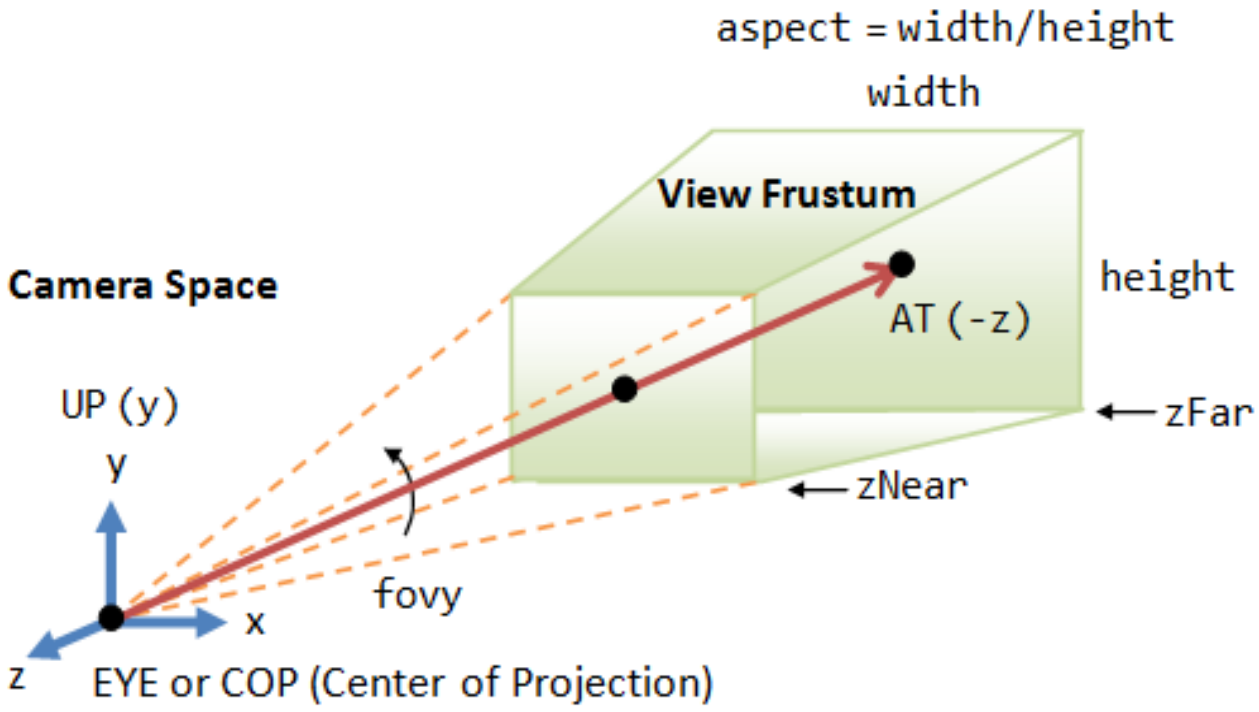
- Most common in Computer Graphics, art, visual system
- Further objects are smaller
- Parallel lines not parallel; converge to single point



Perspective Projection

- The camera has a limited field of view, which exhibits a view frustum (truncated pyramid), and is specified by four parameters: fovy, aspect, zNear and zFar.

- Fovy: specify the total vertical angle of view in degrees.
- Aspect: the ratio of width vs. height
- zNear: near clipping plane (relative from cam)
- zFar: far clipping plane (relative from cam)



Perspective Projection: The camera's view frustum is specified via 4 view parameters: fovy, aspect, zNear and zFar.

Perspective Projection (Math)

- How to do perspective projection
  - First “squish” the frustum into a cuboid (n -> n, f -> f) (Mpersp->ortho)
  - Do orthographic projection (Mortho, already known!)

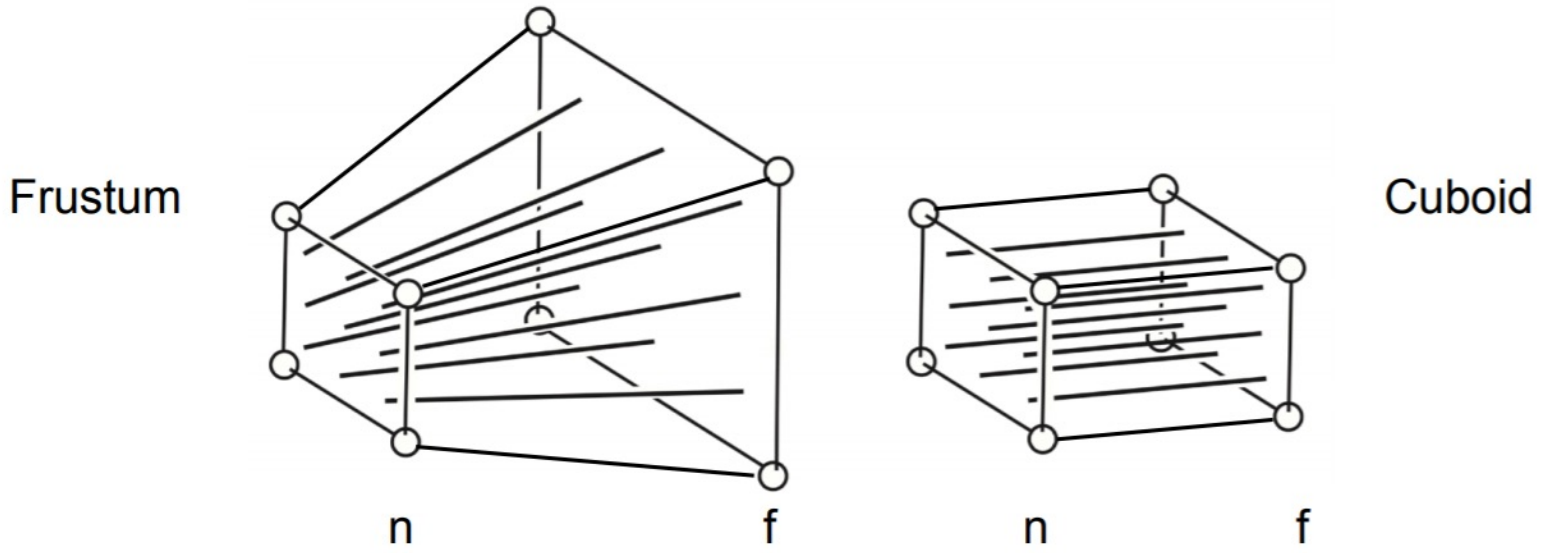
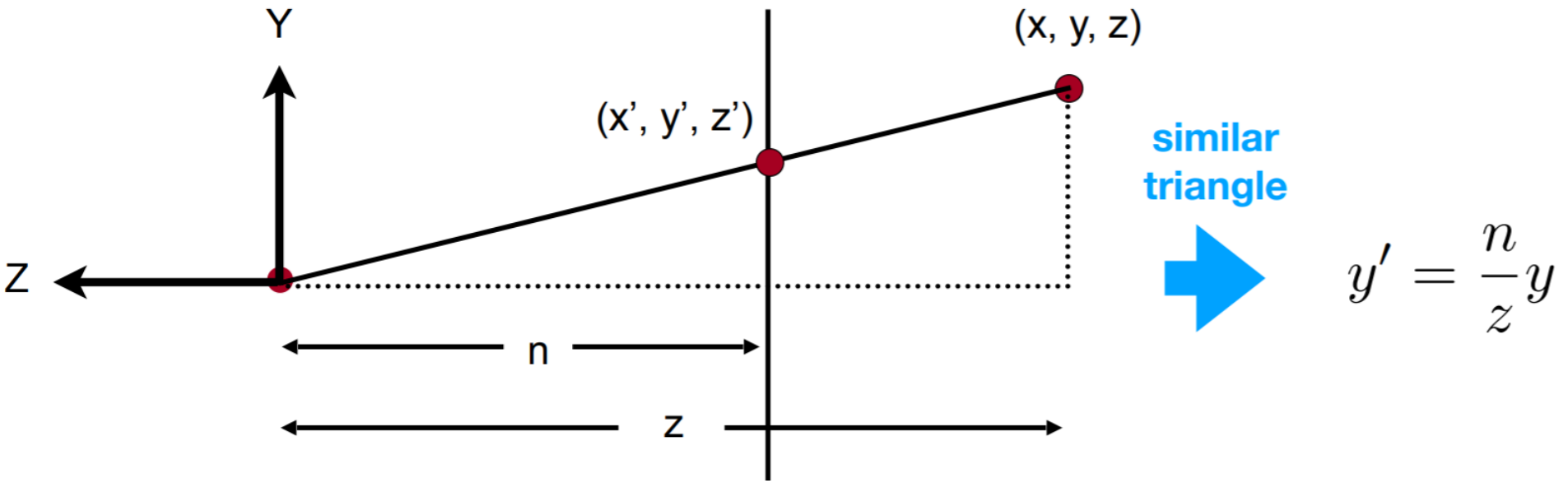


Fig. 7.13 from *Fundamentals of Computer Graphics, 4th Edition*

Perspective Projection (Math)

- In order to find a transformation
  - Recall the key idea: Find the relationship between transformed points (x', y', z') and the original points (x, y, z)



  - y'=ny/z , x' = nx/z

Thank you!