# Introduction to Computer Graphics (CS360A)


## Instructor: Soumya Dutta

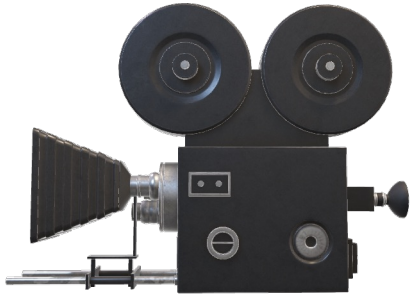
Department of Computer Science and Engineering

Indian Institute of Technology Kanpur (IITK)

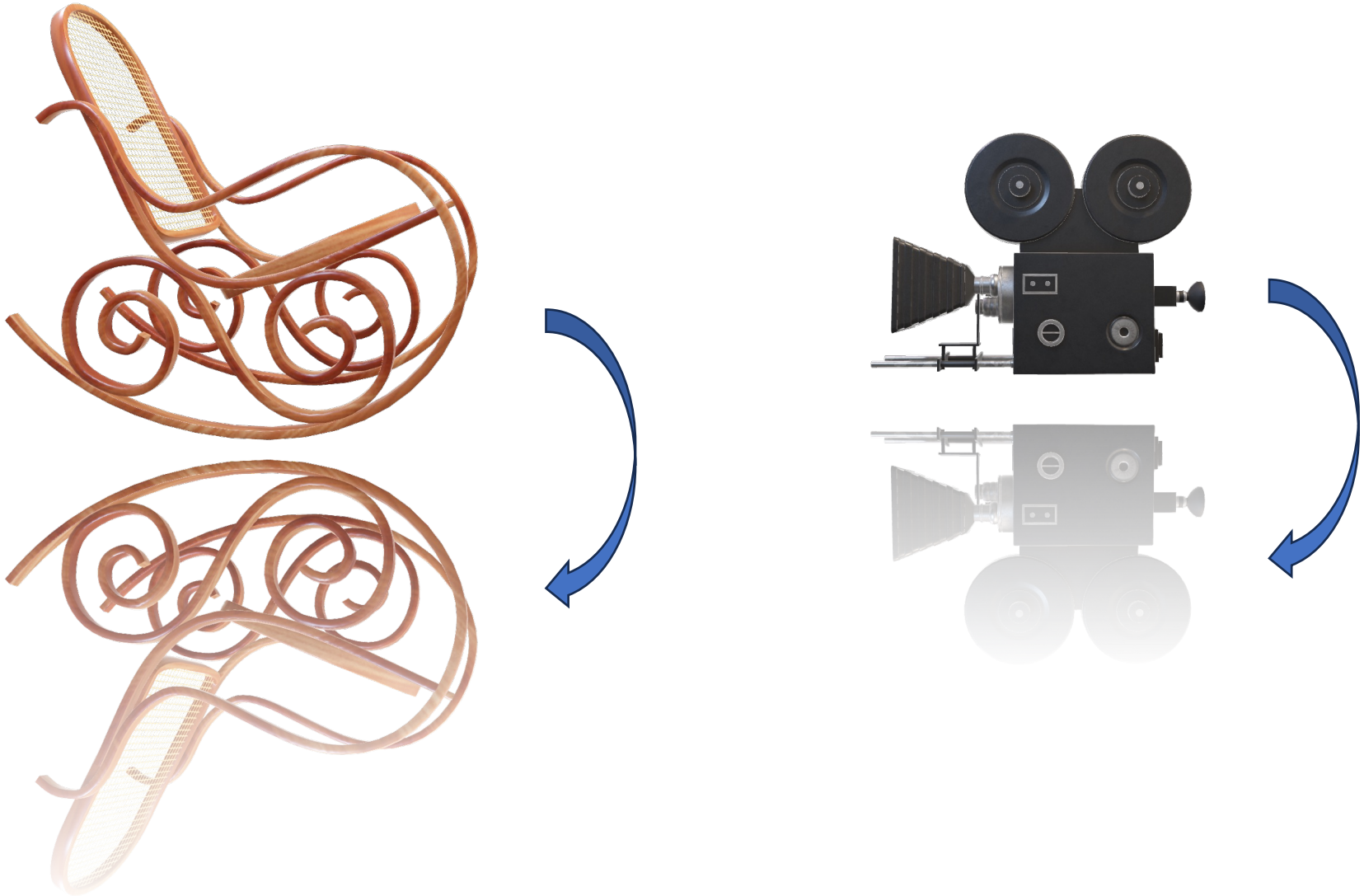
email: soumyad@cse.iitk.ac.in

# Frame Buffer Objects

# Framebuffer Objects (FBO)

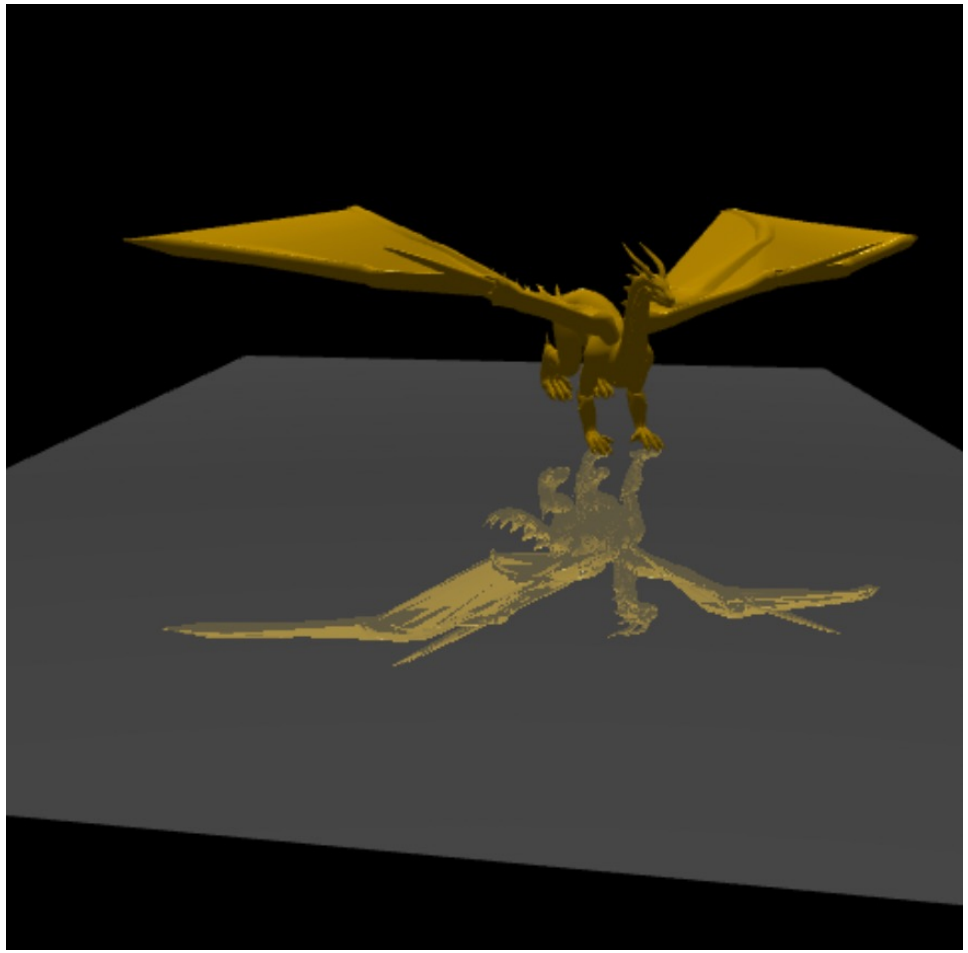


# Framebuffer Objects (FBO)



# Framebuffer Objects (FBO) Uses

- Post-processing of rendered images
  - Many computer graphics effects are implemented using this strategy
    - Planar reflection

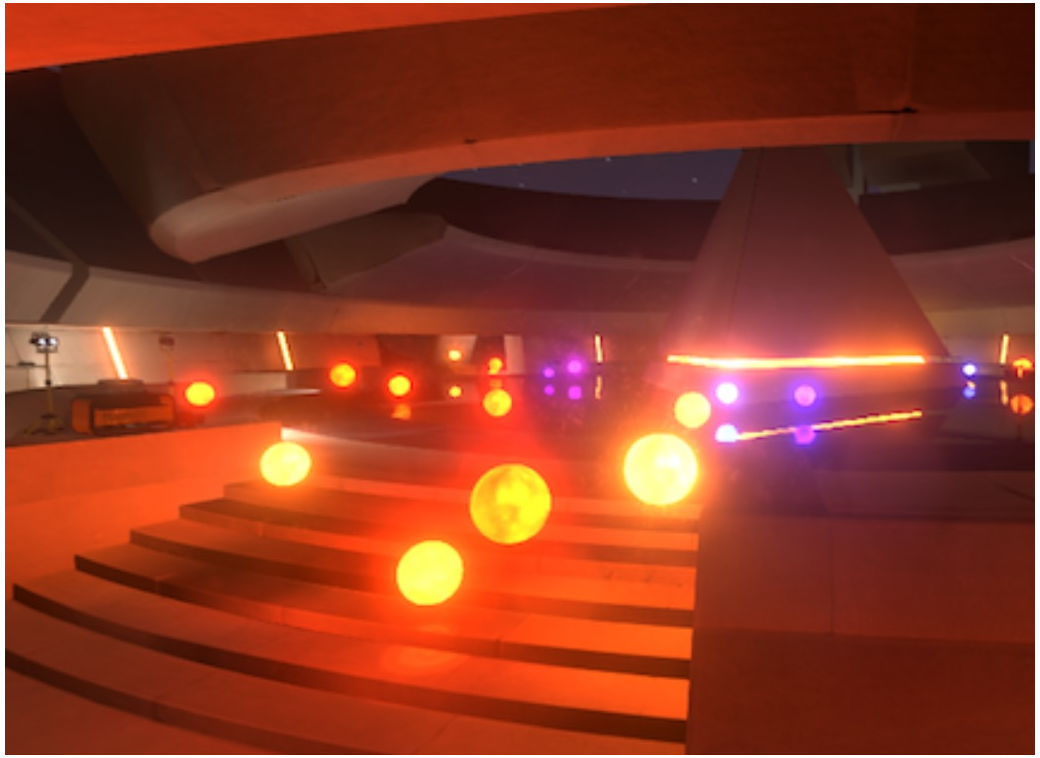


Planer Reflection

# Framebuffer Objects (FBO) Uses



- Post-processing of rendered images
  - Many computer graphics effects are implemented using this strategy
    - Planar reflection
    - Bloom



Bloom

# Framebuffer Objects (FBO) Uses



- Post-processing of rendered images
  - Many computer graphics effects are implemented using this strategy
    - Planar reflection
    - Bloom
    - Blurring


Blurring

# Framebuffer Objects (FBO) Uses



- Post-processing of rendered images
  - Many computer graphics effects are implemented using this strategy
    - Planar reflection
    - Bloom
    - Blurring
    - Transparent/semi-transparent objects


(Semi) Transparent Object

# Framebuffer Objects (FBO) Uses



- Post-processing of rendered images
  - Many computer graphics effects are implemented using this strategy
    - Planar reflection, Bloom, Blurring, modeling transparent/semi-transparent objects etc.
- Composition between different scenes
  - Can be used to create views of other scenes such as simulating a TV in a scene

Simulate TV/monitor in a scene

# Framebuffer



- A Framebuffer is a collection of buffers that can be used as the destination for rendering
- Two kinds of framebuffer
  - Default framebuffer
  - User created framebuffers, called FBO
- The buffers in default framebuffer are part of the primary rendering context and is visible in screen/canvas
- The FBOs are not directly visible by the users
  - They are stored in a render buffer or as a texture for further use
  - Off-screen rendering

# WebGL FBO Set Up



- Create an empty texture where FBO scene will be stored/attached
- Create a frame buffer object and initialize it with proper parameters
- Attach the empty texture with the FBO
- Check if the compatibility and set up is correct and usable

# WebGL FBO Set Up: Create Texture



```
// create a 2D texture in which framrbuffer rendering will be stored
FBOtexture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, FBOtexture);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
gl.texImage2D(
  gl.TEXTURE_2D,
  0,
  gl.RGBA,
  FBO.width,
  FBO.height,
  0,
  gl.RGBA,
  gl.UNSIGNED_BYTE,
  null
);
```

any

# WebGL FBO Set Up: Create FBO



```
// create an FBO
FBO = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, FBO);
FBO.width = gl.viewportWidth;
FBO.height = gl.viewportHeight;
```

# WebGL FBO Set Up: Attach Texture to FBO



```
// attach texture FBOtexture to the framebuffer FBO
gl.framebufferTexture2D(
    gl.FRAMEBUFFER,
    gl.COLOR_ATTACHMENT0,
    gl.TEXTURE_2D,
    FBOtexture,
    0
);
```

# WebGL FBO Set Up: Check FBO Status



```
// check FBO status
var FBOstatus = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
if (FBOstatus != gl.FRAMEBUFFER_COMPLETE)
  console.error("GL_FRAMEBUFFER_COMPLETE failed, CANNOT use FBO");
```

# Two-pass/Multi-pass Rendering



- Two-pass/multi-pass?
  - We render some or all objects of the screen multiple times and then combine such images to produce the final image that has all the special effects
  - How many passes are required depends on the special algorithm or your rendering needs
  - Typically, multiple passes are rendered into FBOs and then the final scene is drawn on the screen by taking information from all the FBO textures
- Example: Planar Reflection
  - It is a two-pass rendering algorithm

# Set Up the First Pass Rendering to FBO



## First Pass Rendering Setup in Draw Function

```
// PASS 1 of Rendering
//// Draw into framebuffer first
gl.bindFramebuffer(gl.FRAMEBUFFER, FBO);
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.enable(gl.DEPTH_TEST);
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

# Switch Back to Default Rendering



## Second (final) Pass Rendering Setup in Draw Function

```
// PASS 2. Now render the object using FBOtexture into normal canvas
// so that we can see it
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.enable(gl.DEPTH_TEST);
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

# Use FBO to Produce Planar Reflection



- Pass 1: Render the object that is supposed to produce reflection inverted and store the result into a FBO texture
  - Also known as ‘render to texture’
  - This inverted image can be generated by flipping your camera and taking an image of the object
- Pass 2: Now, render the image to the default framebuffer, i.e., on the screen/canvas so that we can see it
  - Here, we use the rendered image from Pass 1 as a texture map
  - Blend the color from Pass 2 with Pass 1 to get the inverted object
  - Create the illusion of ‘reflection’

# Mid-Semester Recap

# Raster vs Vector Graphics (Images)


Vector Graphics


Raster Graphics

# Raster vs Vector Graphics (Images)

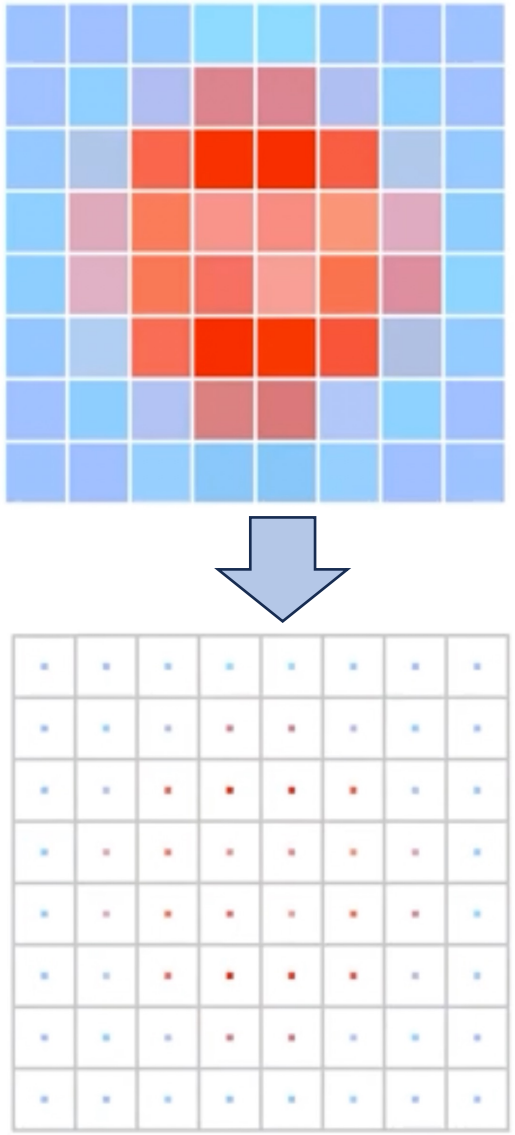


Raster (PNG)

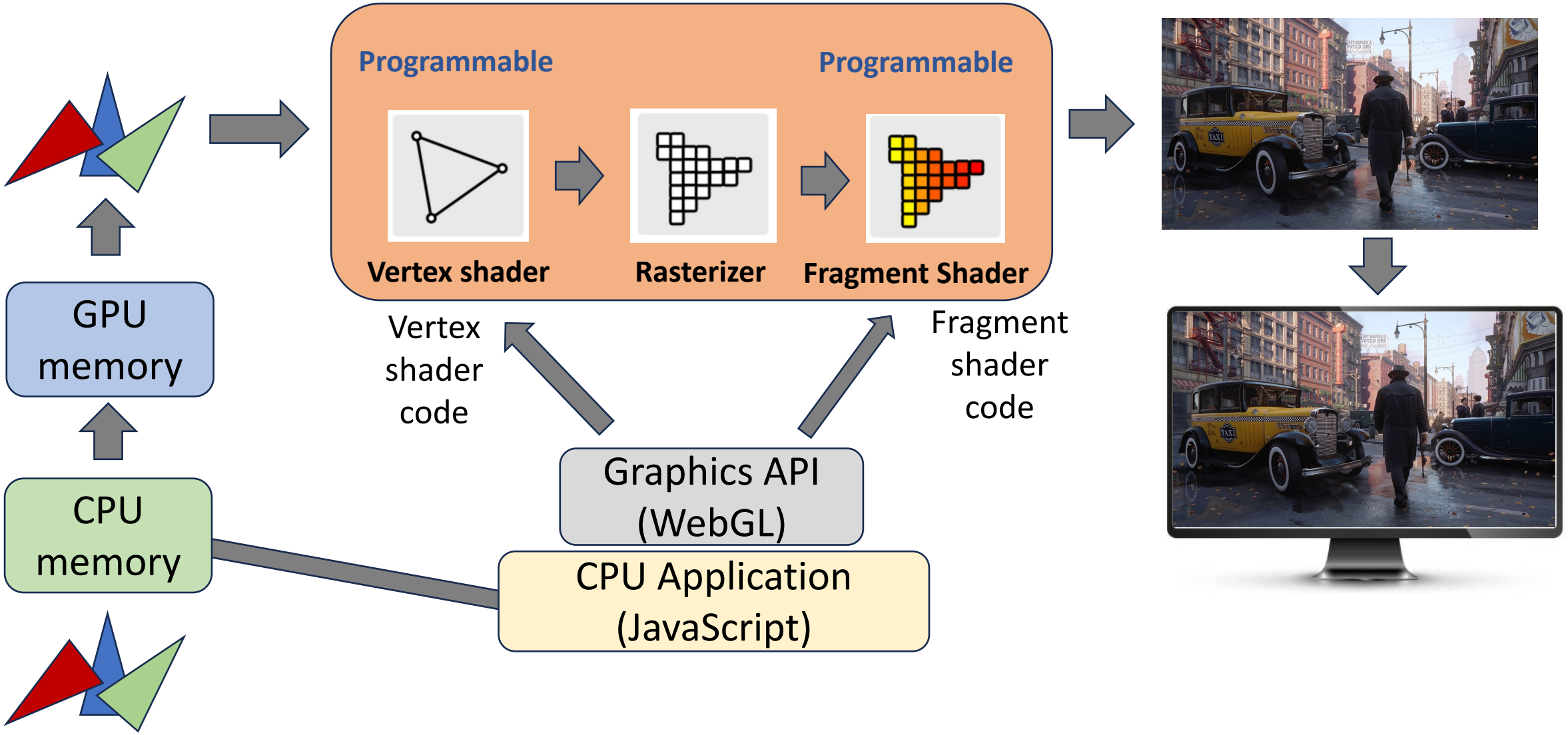
Vector (SVG)

# A Pixel Is Not A Little Square

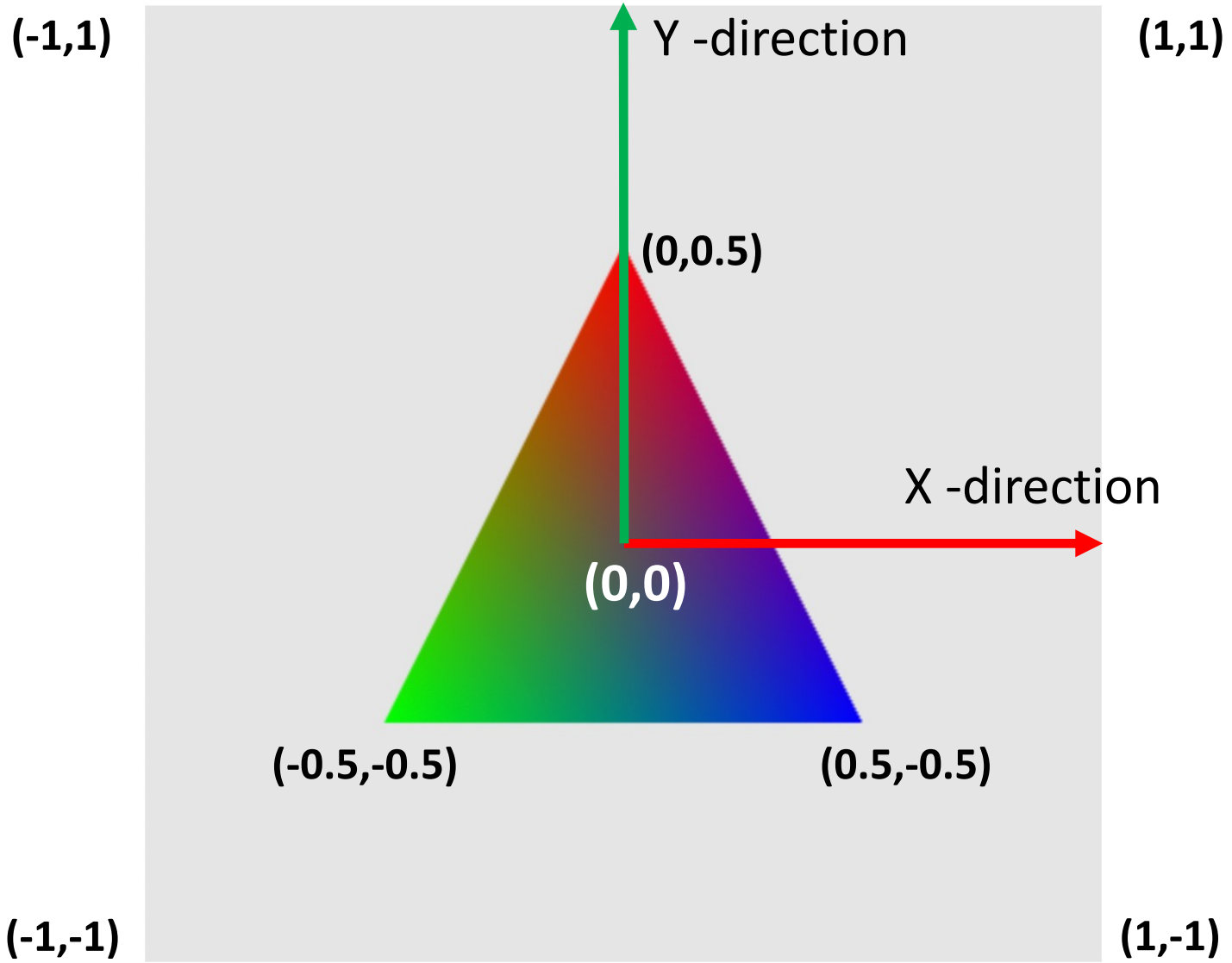
- http://alvyray.com/Memos/CG/Microsoft/6_pixel.pdf
- A pixel is a point sample
  - It exists only at a point
- An image is a rectilinear array of point samples (pixels)
- We can reconstruct a continuous entity from such a discrete entity using an appropriate *reconstruction filter*
  - A truncated Gaussian filter



# GPU Pipeline



# WebGL: Draw A Triangle: Output

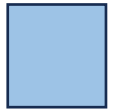


# Transformations (2D and 3D)

# Affine Transformations

- Translation

- Rotation



- Scale

- Skew

  - A combination of rotation and scale

- We will learn about Translation, Rotation, and Scale operations in 2D and 3D
- Affine Transformation
  - Linear combination
  - All lines will remain as lines after affine transformation
  - All parallel lines will remain parallel after affine transformation

# Summary of 2D Transformations



- Translation matrix

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- Scale matrix

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

- Rotation matrix

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

# Transformation Matrix Interpretation



$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

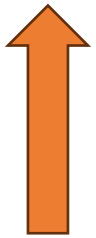
Rotation and Scaling components

Translation components

Affine transformation matrix

# Matrix Operations with glMatrix



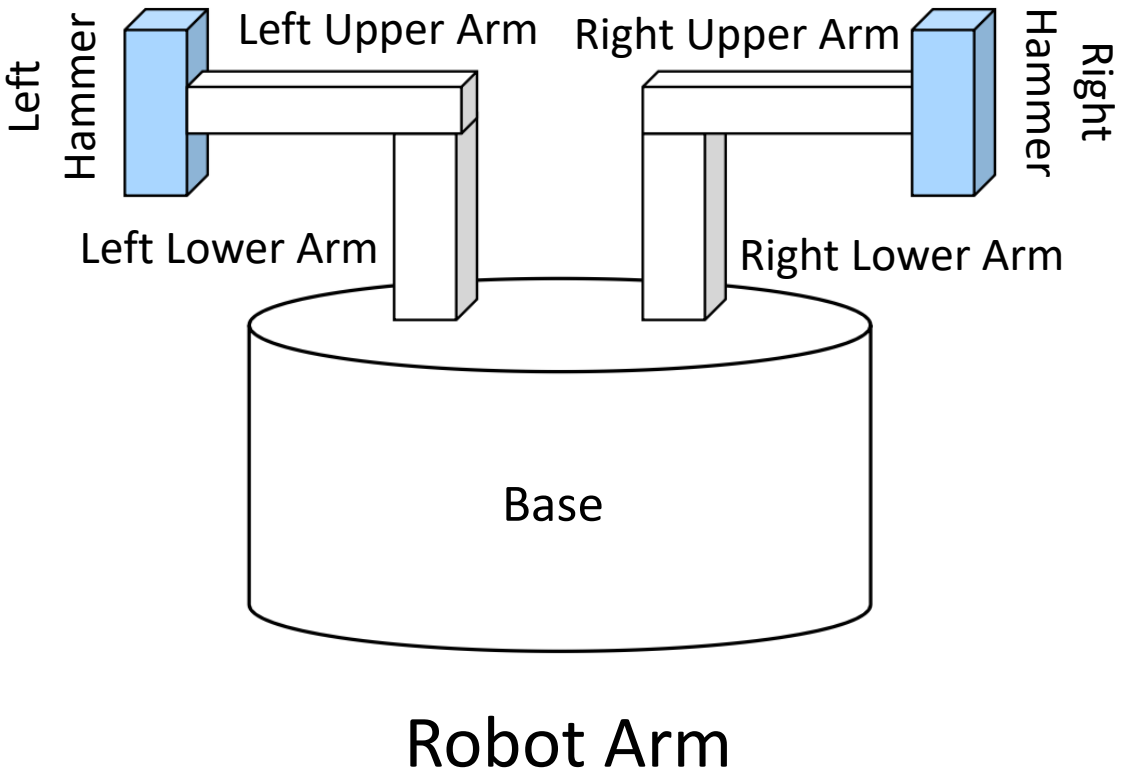



- `mat4.identity(mMatrix);`
- `mMatrix = mat4.translate(mMatrix, [0.1, 0, 0]);`
- `mMatrix = mat4.rotate(mMatrix, degToRad(45), [0, 0, 1]);`
- `mMatrix = mat4.scale(mMatrix, [0.07, 0.25, 1.0]);`

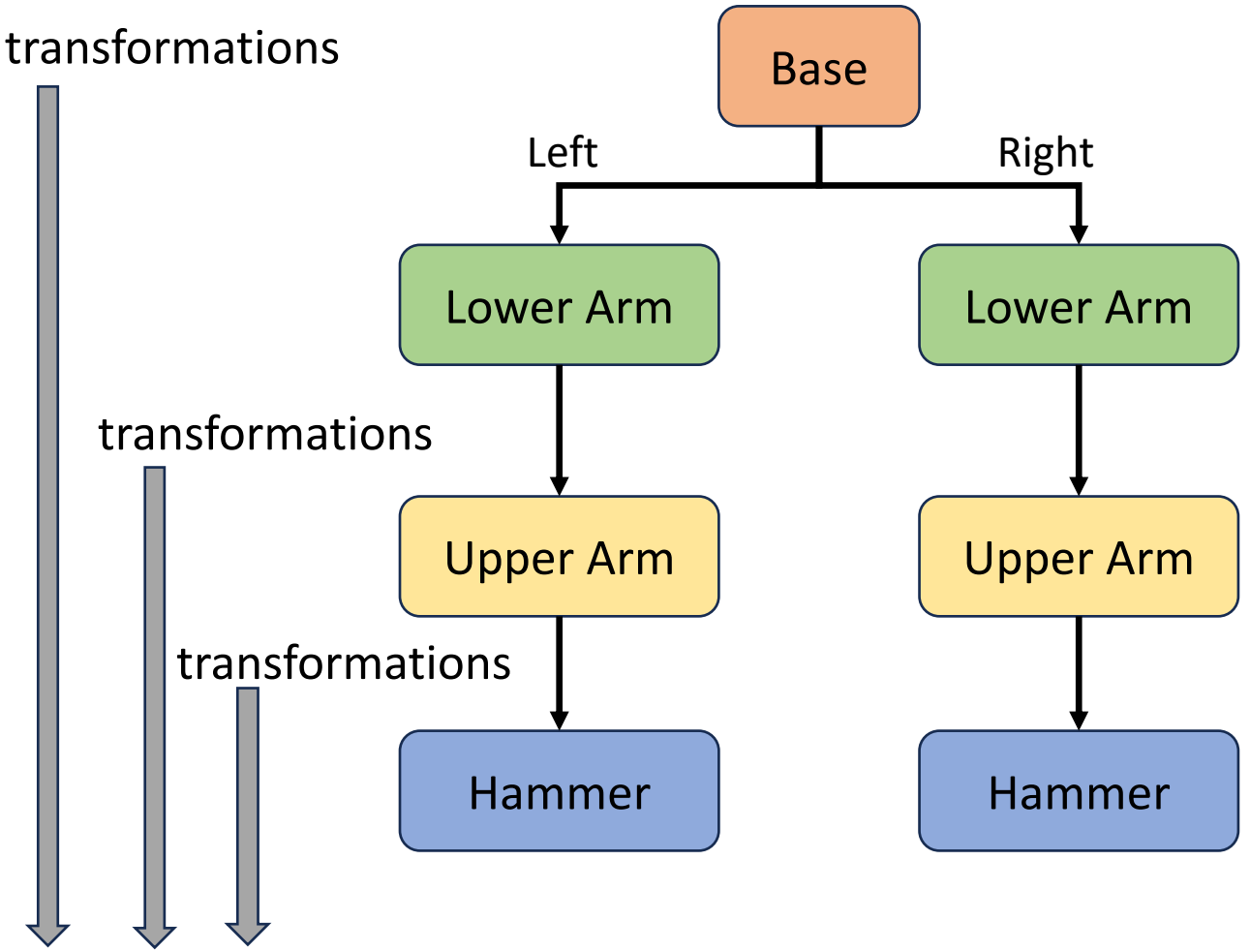
- `mMatrix = translate[0.1, 0, 0]*rotate[45]*scale[0.07, 0.25, 1.0]`
  - Scaling will be applied first, then rotation, and then translation
  - This is the order you should follow in your code when applying transformations to an object in its local/object space
- `newPosition = mMatrix*oldPosition`

# Hierarchical Transformation






Robot Arm



# Summary of 3D Transformations



- Translation matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Scale matrix

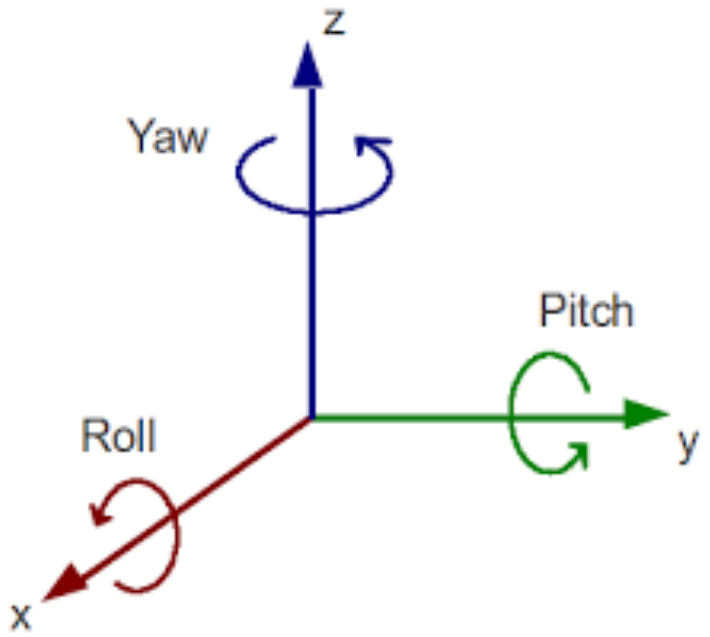
$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation matrix

$R_x$: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & sin\theta & 0 \\ 0 & -sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $R_y$: $\begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $R_z$: $\begin{bmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

# Rotation About an Arbitrary Axis

- Rotation about any arbitrary axis can be decomposed into rotation around X-axis, Y-axis, and Z-axis
- The order of applying the rotation is important to get the correct effect
- $\boldsymbol{R} = \boldsymbol{R_z(\alpha) R_y(\beta) R_x(\gamma)}$




**Yaw (Z-axis)**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha & sin\alpha & 0 \\ 0 & -sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Pitch (Y-axis)**

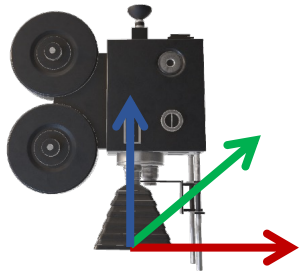
$$\begin{bmatrix} cos\beta & 0 & -sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ sin\beta & 0 & cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Roll (X-axis)**

$$\begin{bmatrix} cos\gamma & sin\gamma & 0 & 0 \\ -sin\gamma & cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

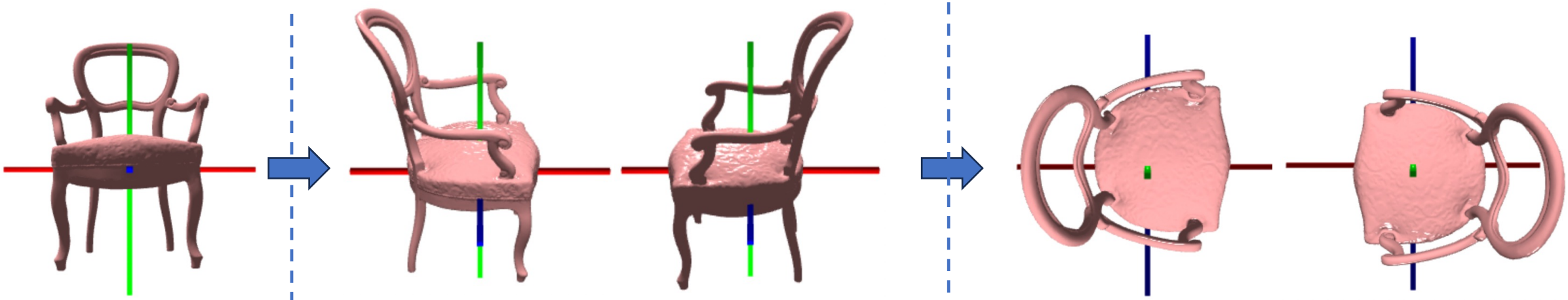
# Viewing and Projection

# Viewing



Model Transformation Matrix

View Transformation Matrix

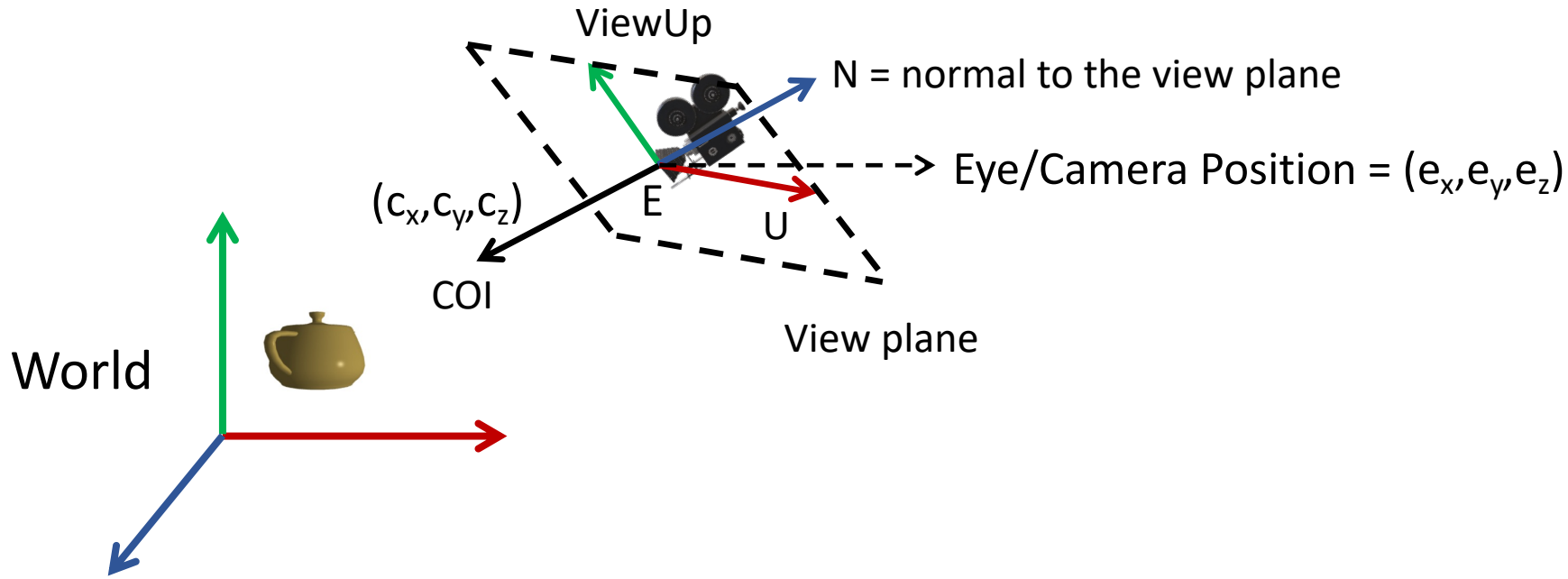


Model/Object Space

World Space

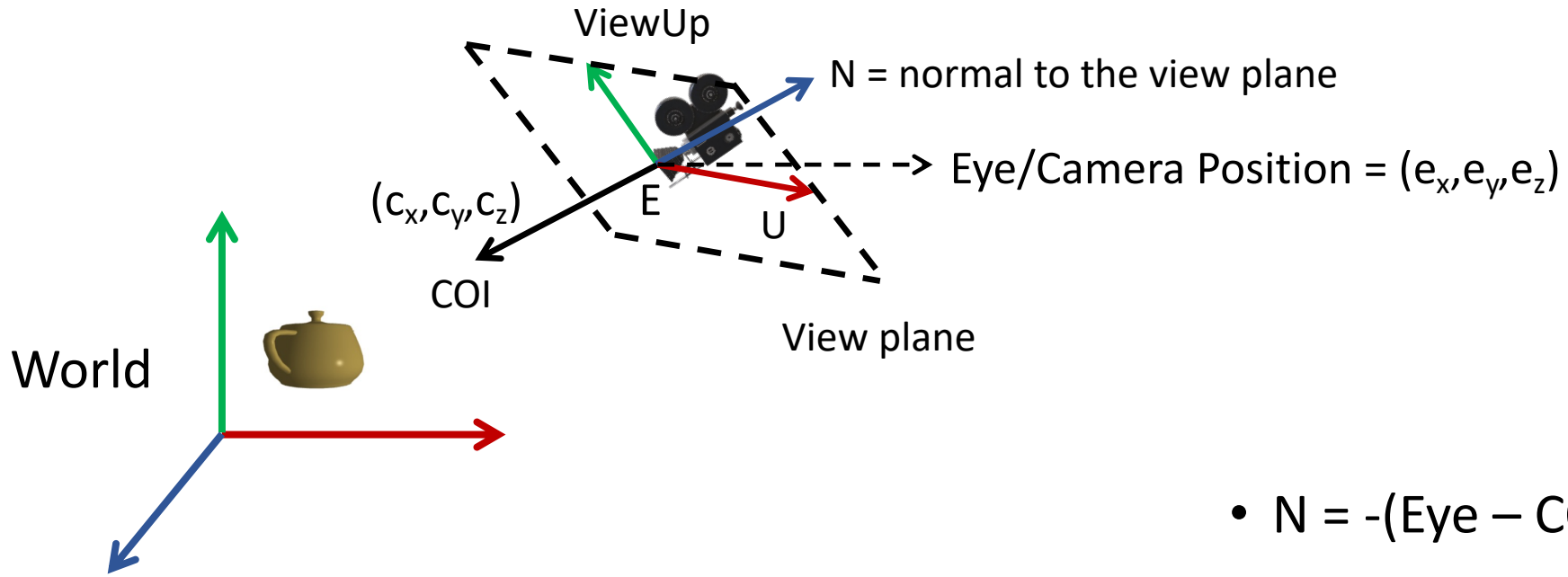
View/Camera Space

# How Do We Form Camera Coordinate System




- Given:
  - COI $(c_x,c_y,c_z)$ is the center of Interest
  - Eye $(e_x,e_y,e_z)$
- Vector N is normal vector to the viewing plane
  - Pointing away from world

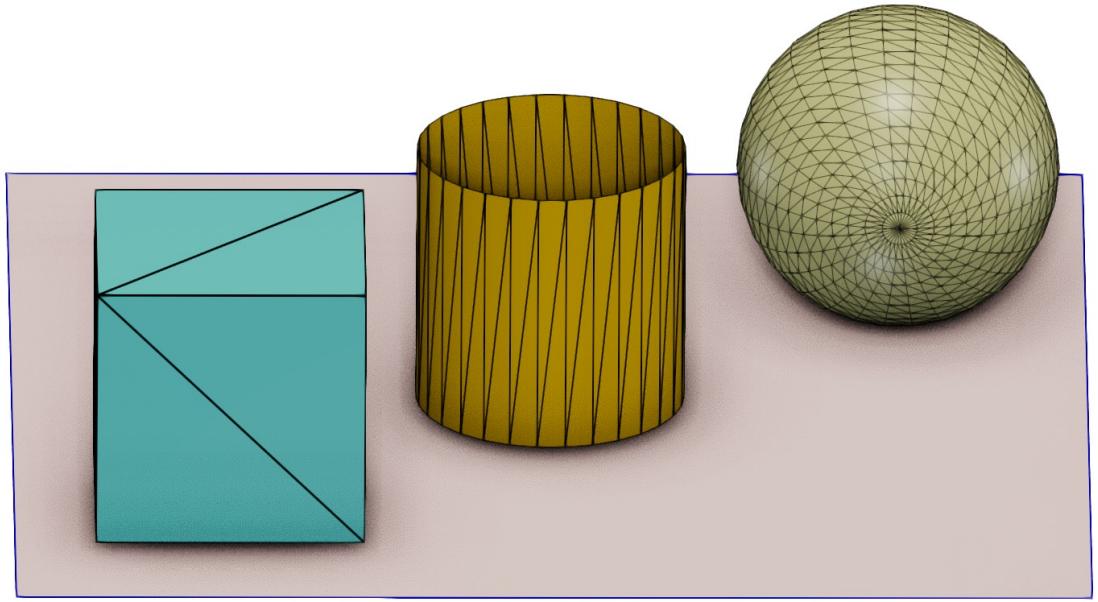
# How Do We Form Camera Coordinate System



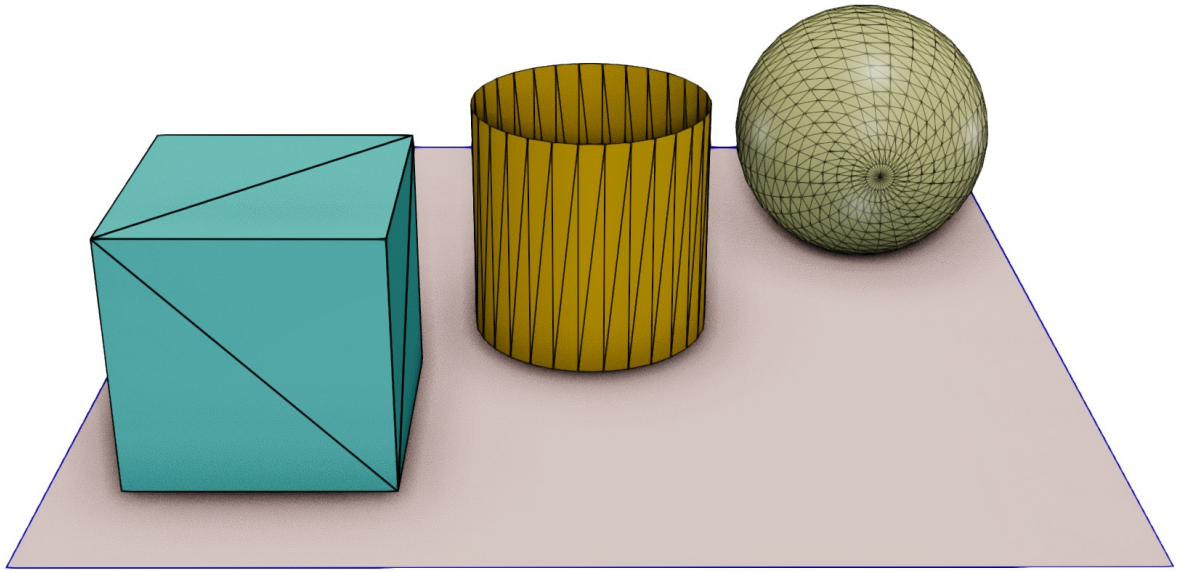

- Given:
  - COI $(c_x,c_y,c_z)$ is the center of Interest
  - Eye $(e_x,e_y,e_z)$
- Vector N is normal vector to the viewing plane
  - Pointing away from world

- N = -(Eye – COI), n = N/|N|
- ViewUp: Direction of head up
- U = ViewUp X n, U = U/|U|

# Projection



Orthographic Projection



Perspective Projection

# Orthographic Projection



In Canonical View Volume Space

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{f+n}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

In Camera Space

# Perspective Projection

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

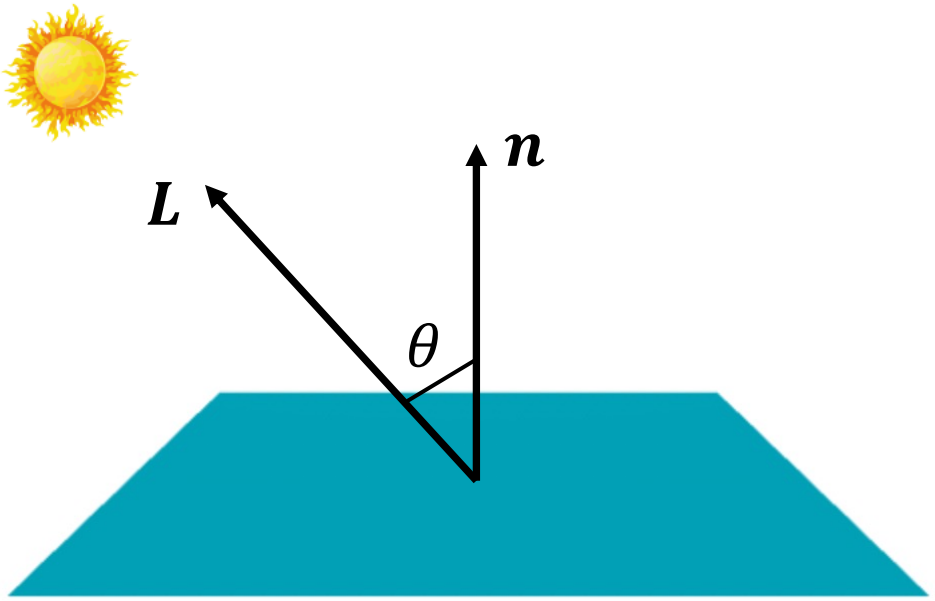
Perspective Projection

Orthographic Projection

Perspective Transformation

# Shading

# Lambertian Diffuse Material and Lighting




$K_d cos\theta = K_d(\boldsymbol{n} . \boldsymbol{L})$

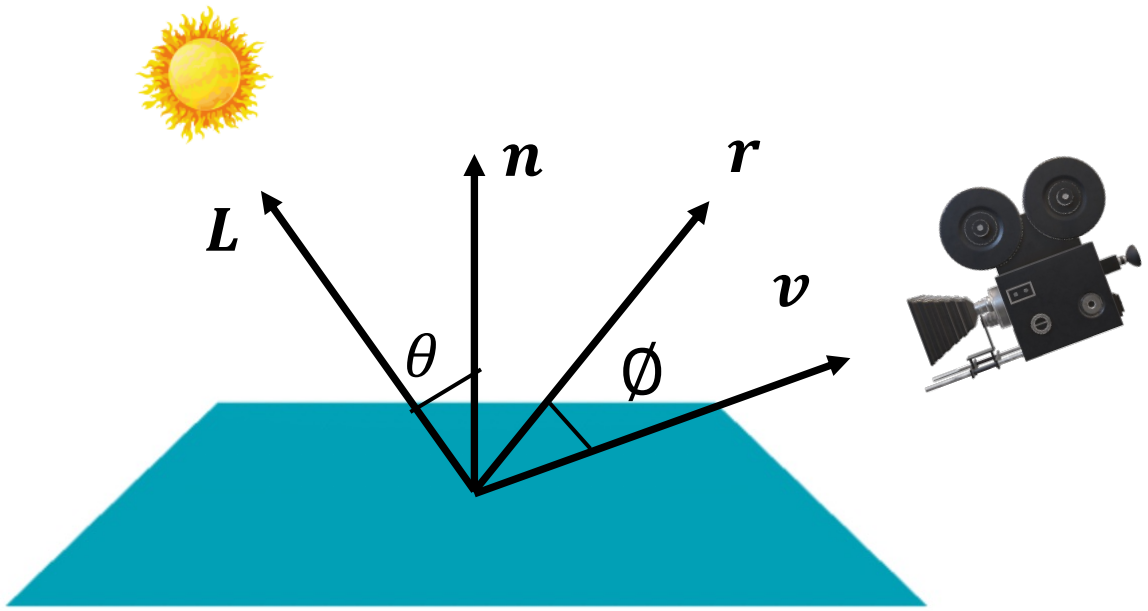
$C_d = IK_d cos\theta$

$I$ = Light Intensity

$\boldsymbol{n}$ and $\boldsymbol{L}$ are unit vectors

$K_d$ =

# Phong Specular Reflection Lighting




$$(cos\emptyset)^{\alpha} = (\boldsymbol{v} \,.\, \boldsymbol{r})^{\alpha}$$

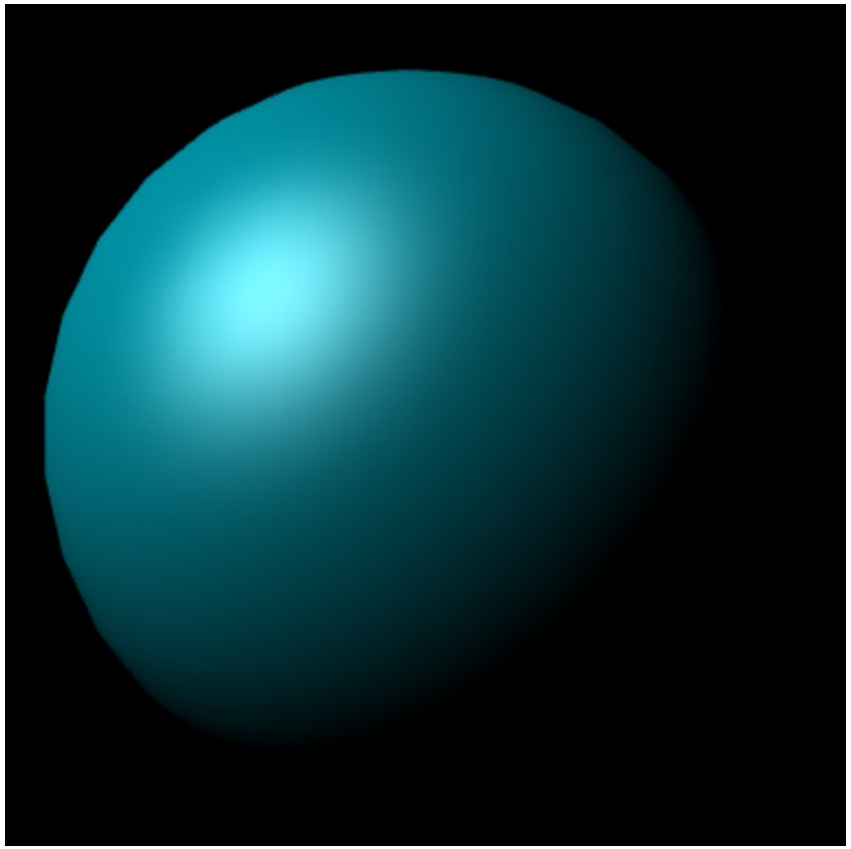
$$K_s(cos\emptyset)^{\alpha} = K_s(\boldsymbol{v} \,.\, \boldsymbol{r})^{\alpha}$$

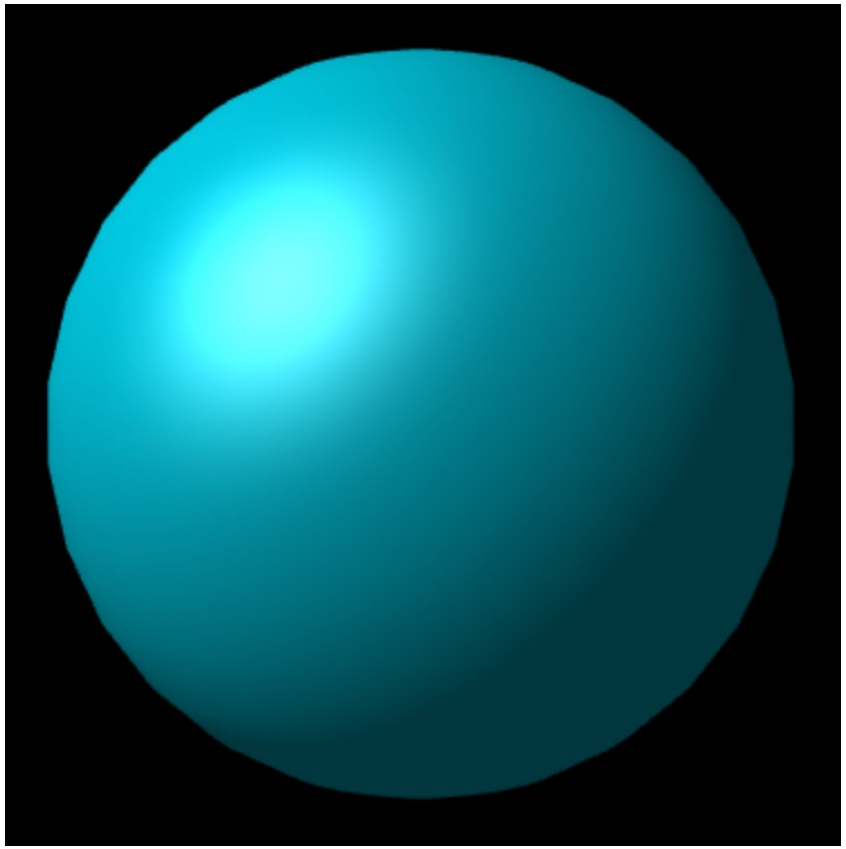
$$C_s = IKs(\boldsymbol{v} \,.\, \boldsymbol{r})^{\alpha}$$

$I$ = Light Intensity

Specular Reflections are reflections of light source

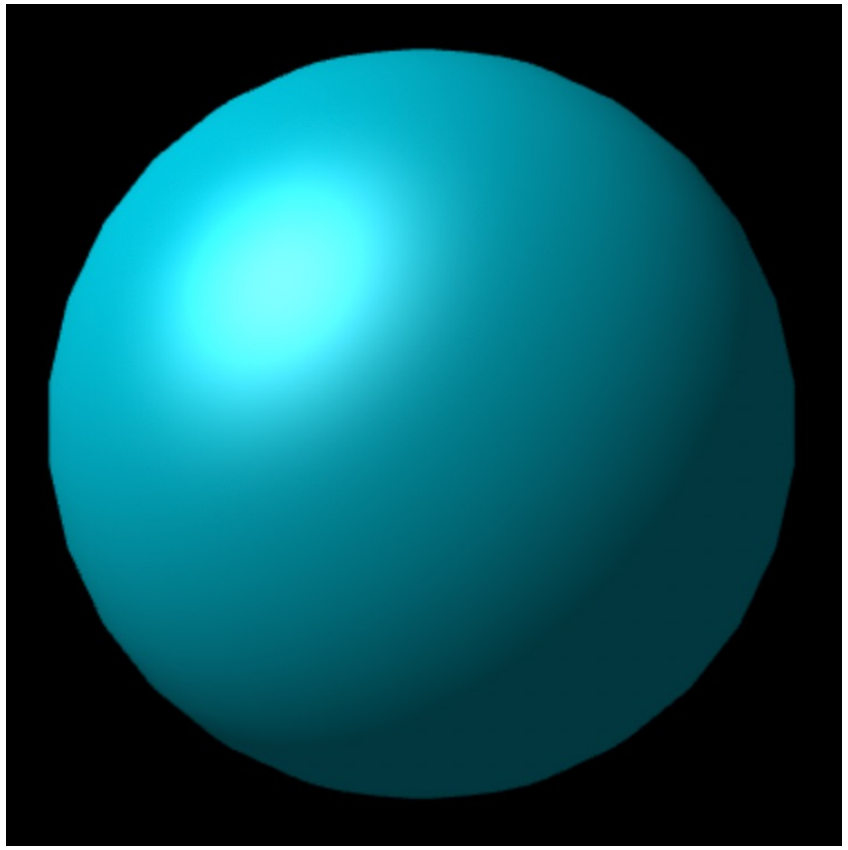
# Ambient Light





- Add a small constant light contribution to our object, called ambient light
- An easy approximation of global lighting

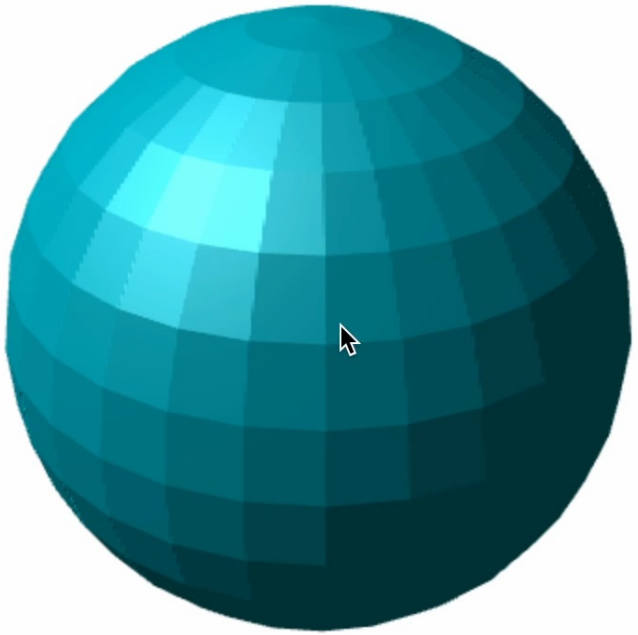
# The Complete Phong Model



$$C = IK_d max(0, cos\theta) + IK_s(cos\emptyset)^{\alpha} + \mathrm{IK_a}$$

Most of the time, we assume ambient color is same as diffuse color

# Different Types of Shading



**Change Background Color:**

Grey | Black | White

**Change Shading/Lighting Mode:**

Flat (Per-Face) | Per-Vertex | Per-Fragment | Toon

# Shading Transformation Matrices



View Space ⟸ Model Space

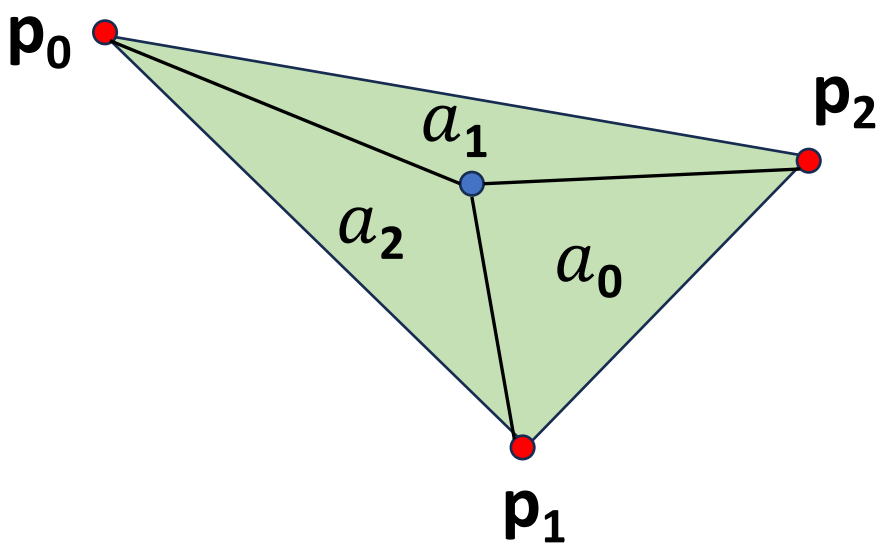
$$\boldsymbol{p}' = \boldsymbol{M}\boldsymbol{p} \quad \Longleftarrow \text{ Position}$$

$$\boldsymbol{n}' = (\boldsymbol{M}_{3x3}^{-1})^{T}\,\boldsymbol{n} \quad \Longleftarrow \text{ Normal}$$

$\boldsymbol{M}$ = ModelView Matrix

# Barycentric Coordinates of Triangles

- $(\alpha, \beta, \gamma)$ are called Barycentric Coordinates
- How do we compute these coordinates?
- Area of the triangle $=a$




$$p = \alpha \boldsymbol{p_0} + \beta \boldsymbol{p_1} + \gamma \boldsymbol{p_2}$$

$$\alpha + \beta + \gamma = 1$$

$$\alpha = \frac{a_0}{a} \qquad \beta = \frac{a_1}{a} \qquad \gamma = \frac{a_2}{a}$$

$$a = a_0 + a_1 + a_2$$

# Triangular Meshes: Attributes



- Vertex position
- Vertex Normal
- Texture Coordinate
- Color

| Vertex pos X | Vertex pos Y | Vertex pos Z |
|---|---|---|
| $v_{x1}$ | $v_{y2}$ | $v_{z3}$ |
| $v_{x2}$ | $v_{y2}$ | $v_{z2}$ |
| $v_{x3}$ | $v_{y3}$ | $v_{z3}$ |

| Vertex normal X | Vertex normal Y | Vertex normal Z |
|---|---|---|
| $N_{x1}$ | $N_{y2}$ | $N_{z3}$ |
| $N_{x2}$ | $N_{y2}$ | $N_{z2}$ |
| $N_{x3}$ | $N_{y3}$ | $N_{z3}$ |

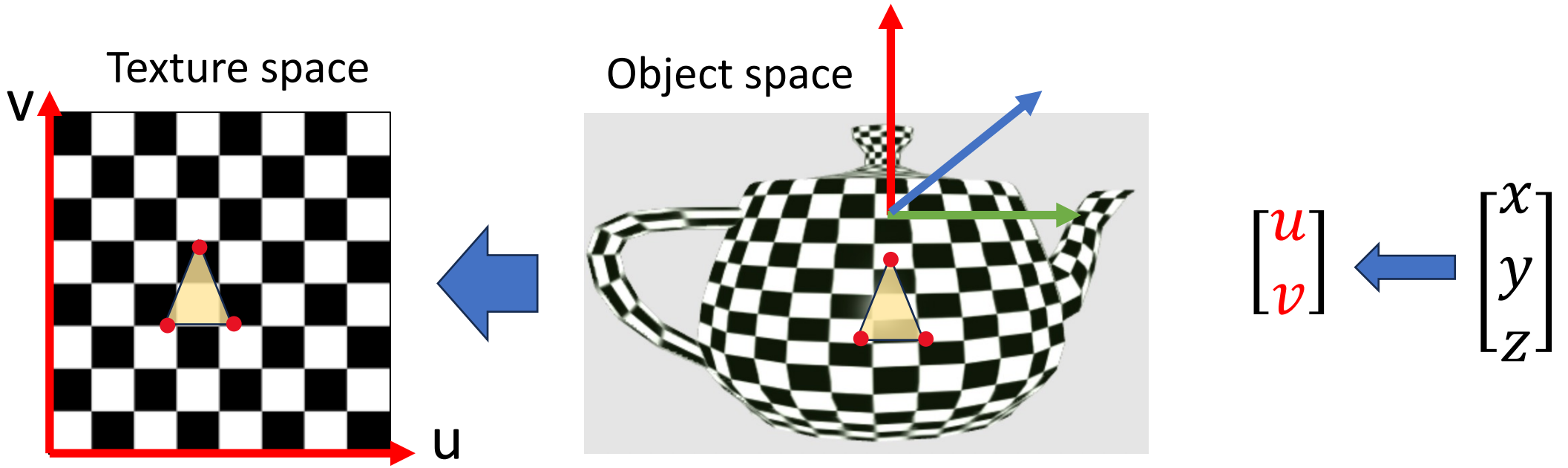
| Vertex Color R | Vertex Color G | Vertex Color B |
|---|---|---|
| $C_r$ | $C_g$ | $C_b$ |
| $C_r$ | $C_g$ | $C_b$ |
| $C_r$ | $C_g$ | $C_b$ |

| Vertex texture U | Vertex normal V |
|---|---|
| $T_{x1}$ | $T_{y2}$ |
| $T_{x2}$ | $T_{y2}$ |
| $T_{x3}$ | $T_{y3}$ |

# Texture Mapping

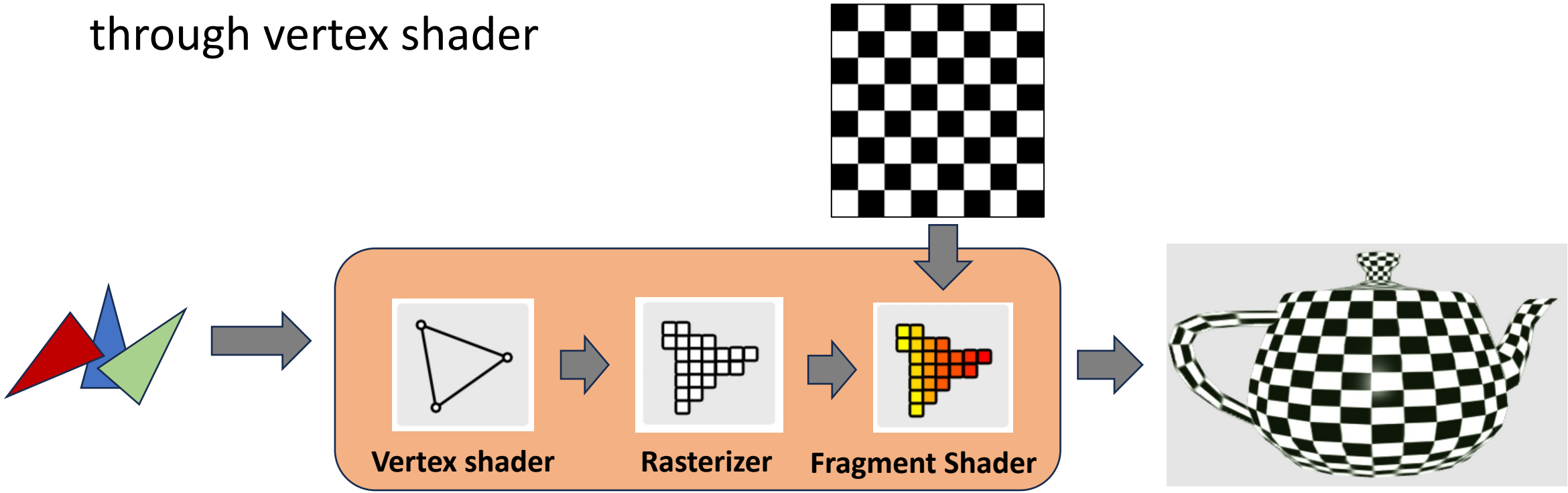
# Texture Mapping

- A mapping from **Texture space** to **Object space**
- Texture mapping is sometimes hard
- There are many variants to do this correctly
- Even then, texture mapping on complex shaped objects is not straightforward

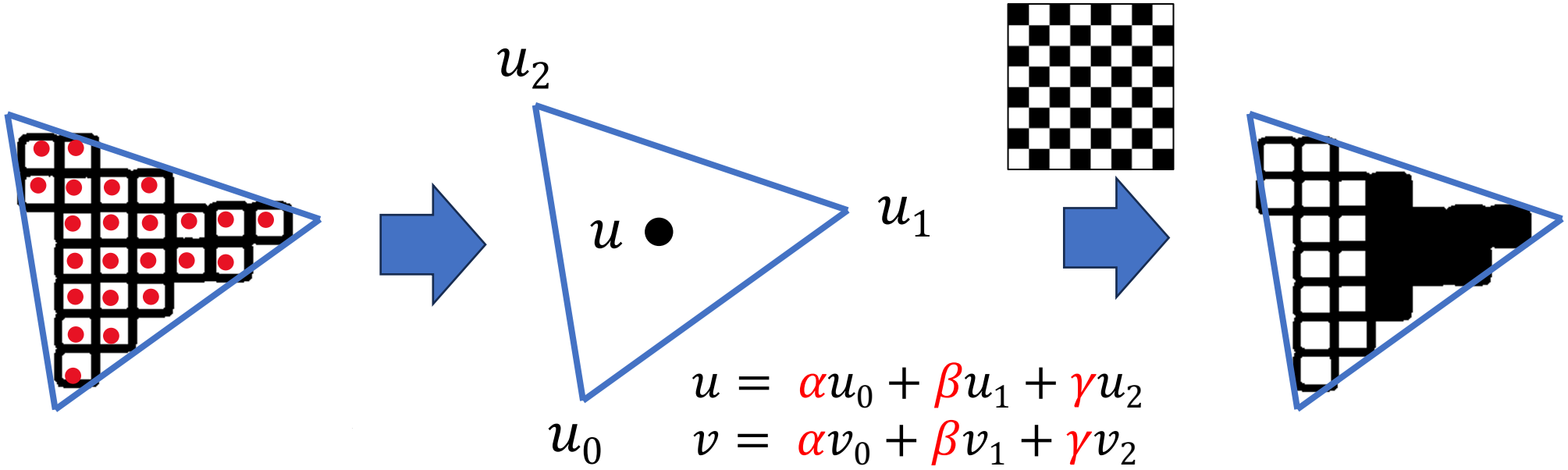


# Texture Mapping: GPU Pipeline

- Texture mapping happens in fragment shader
- Pass vertex texture coordinates through vertex shader



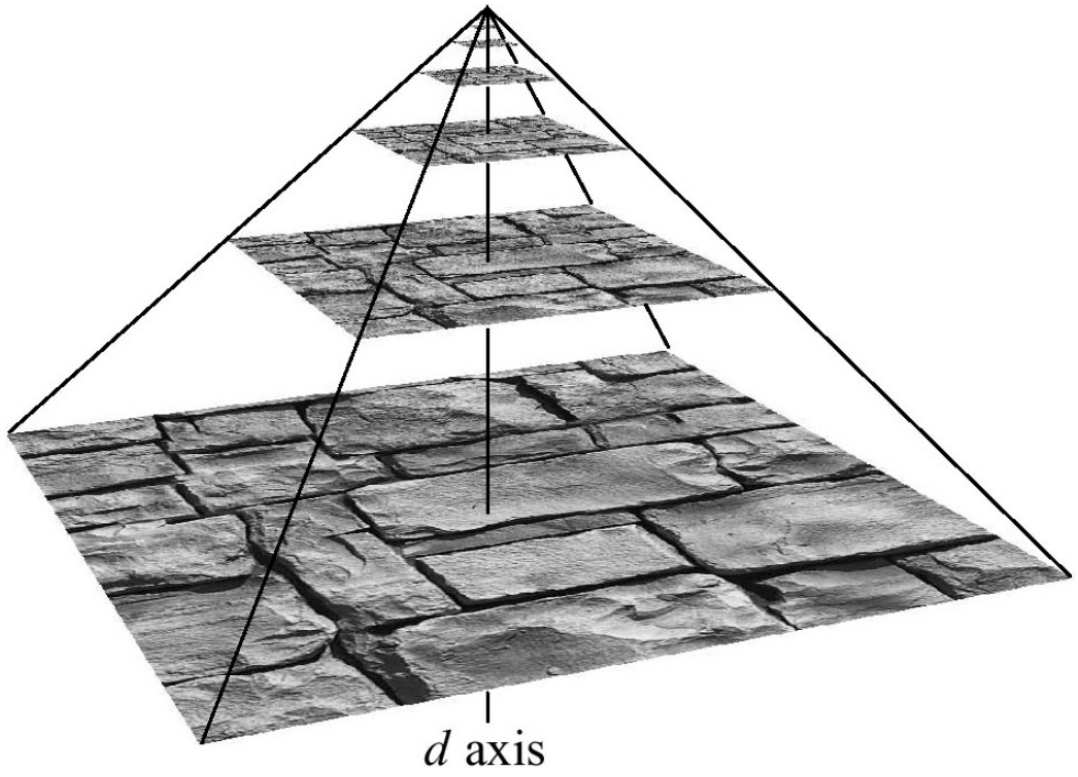
# Texture Mapping: GPU Pipeline




- Use Barycentric coordinates to compute texture coordinates at each fragment inside each triangle
  - This will be done by hardware for us
- Then we look up the color using the $(u, v)$ coordinate from the texture and shade the fragment

# Mipmaps

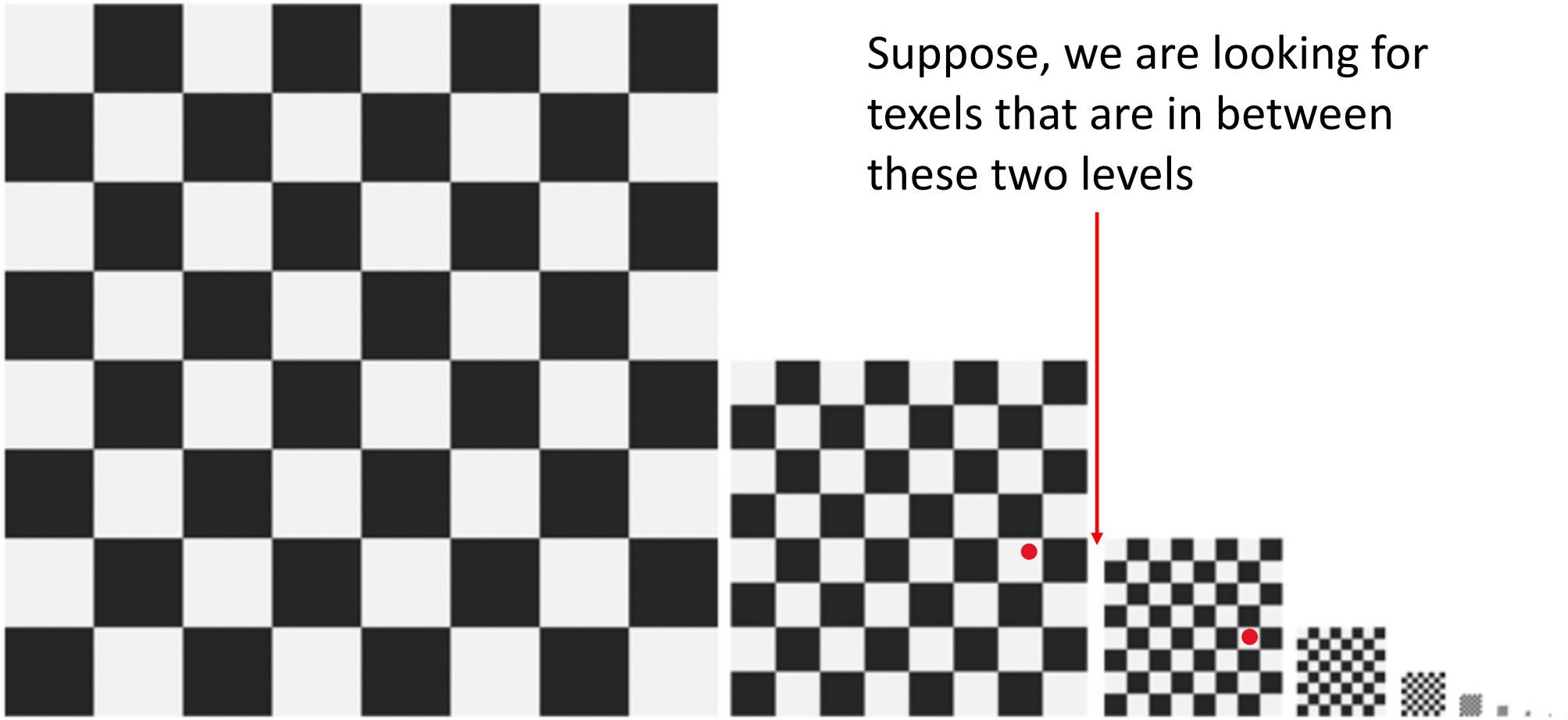





Mipmap levels

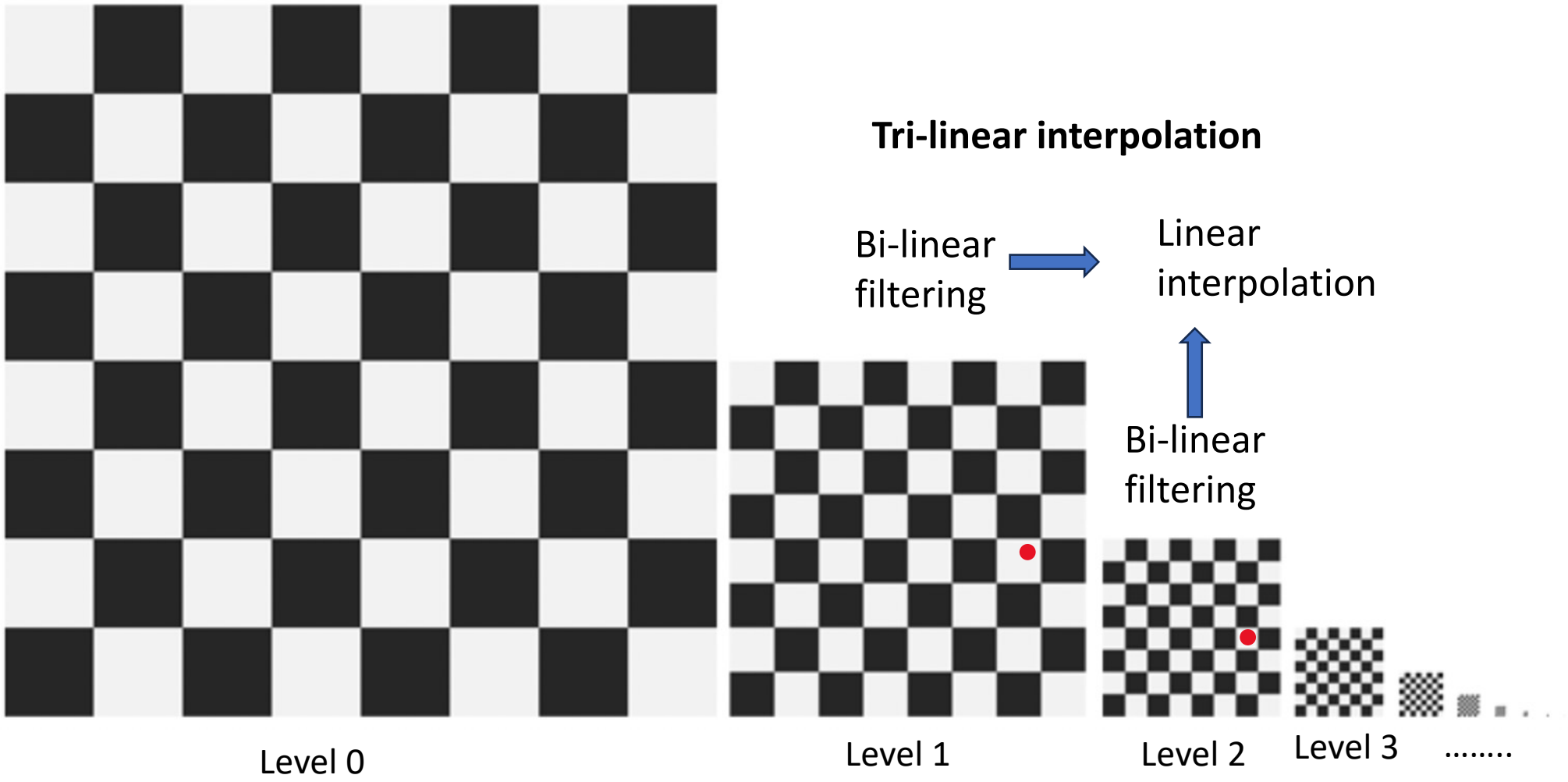
- Pre-generate several texture images at different level of detail

# Mipmaps: How Do We Look Up?

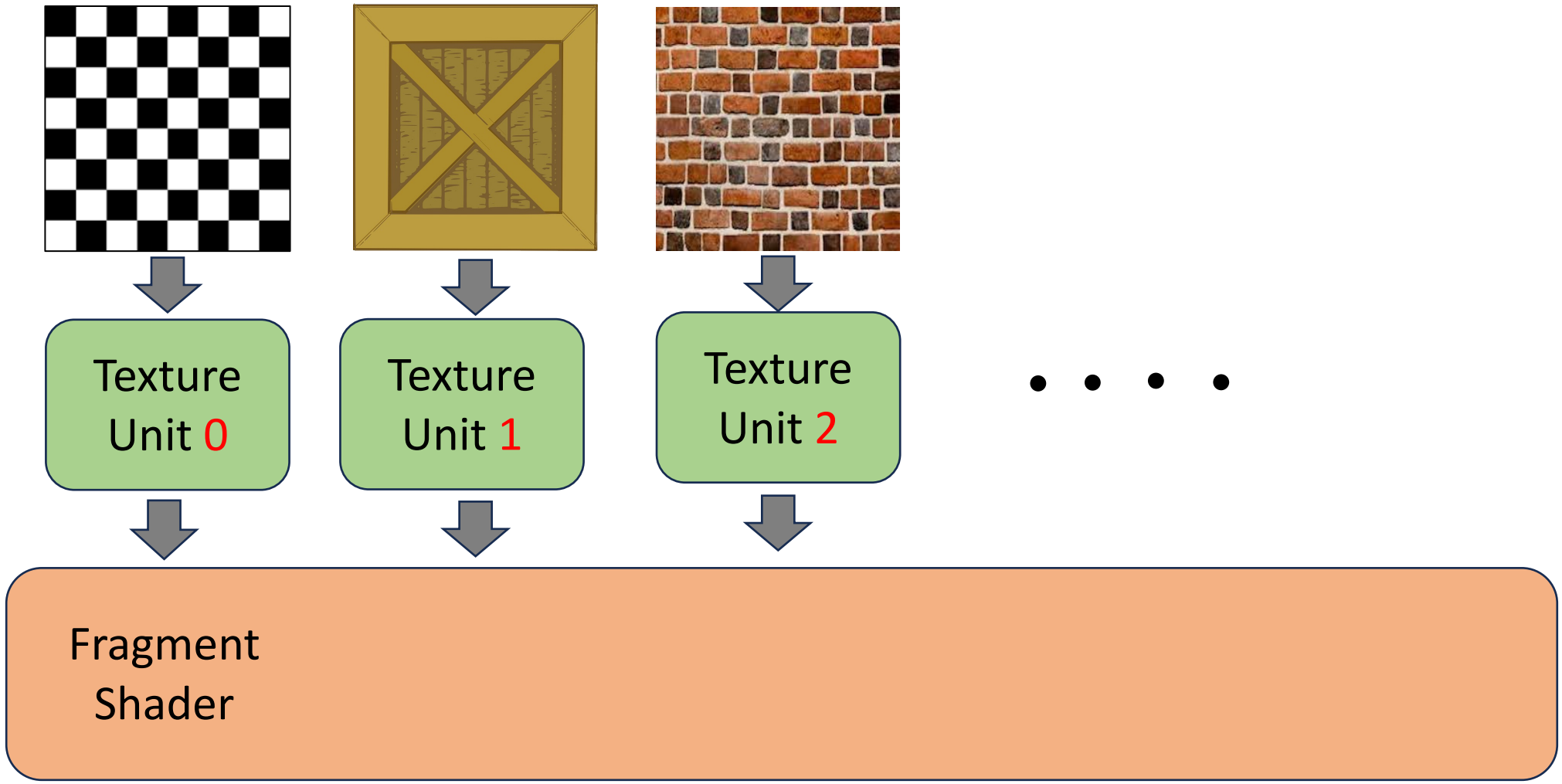


Suppose, we are looking for texels that are in between these two levels

# Mipmaps: How Do We Look Up?

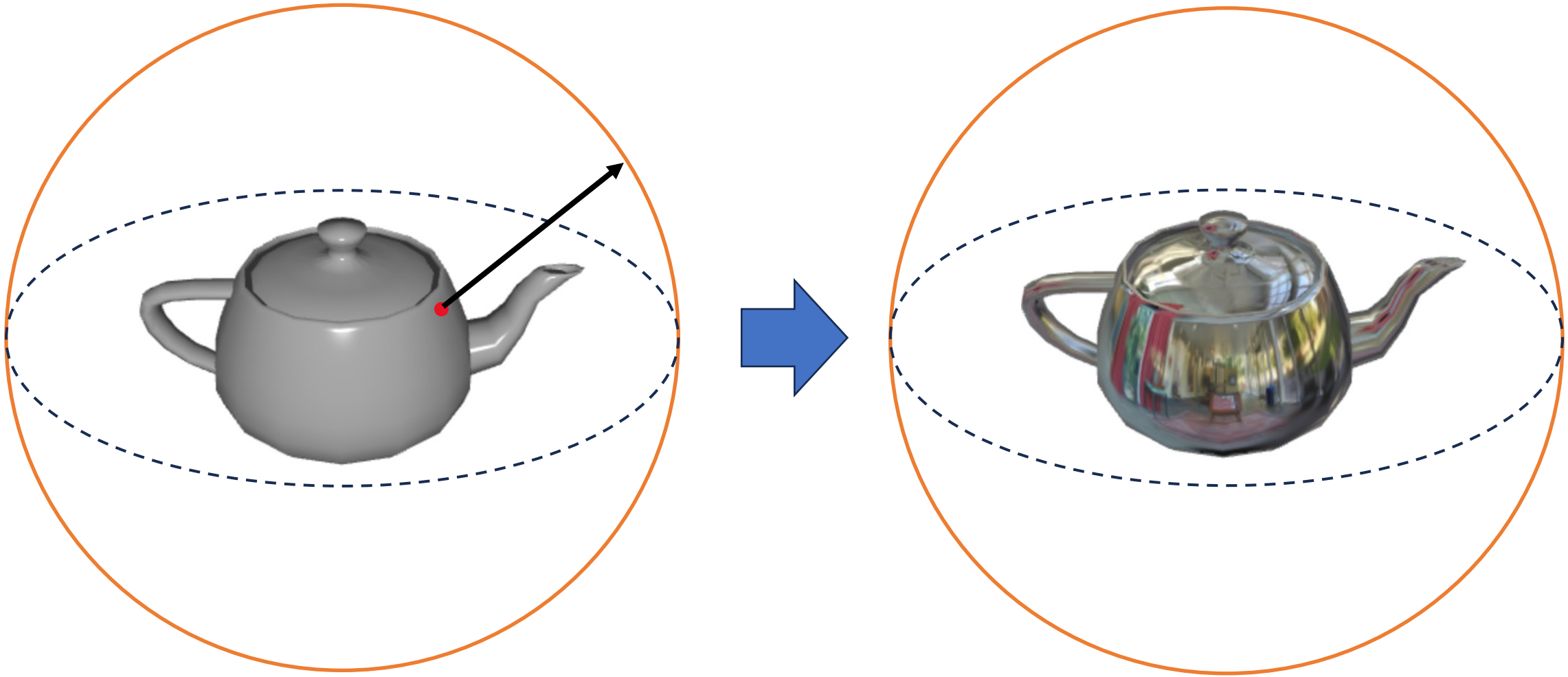


# Texture Units: GPU Pipeline

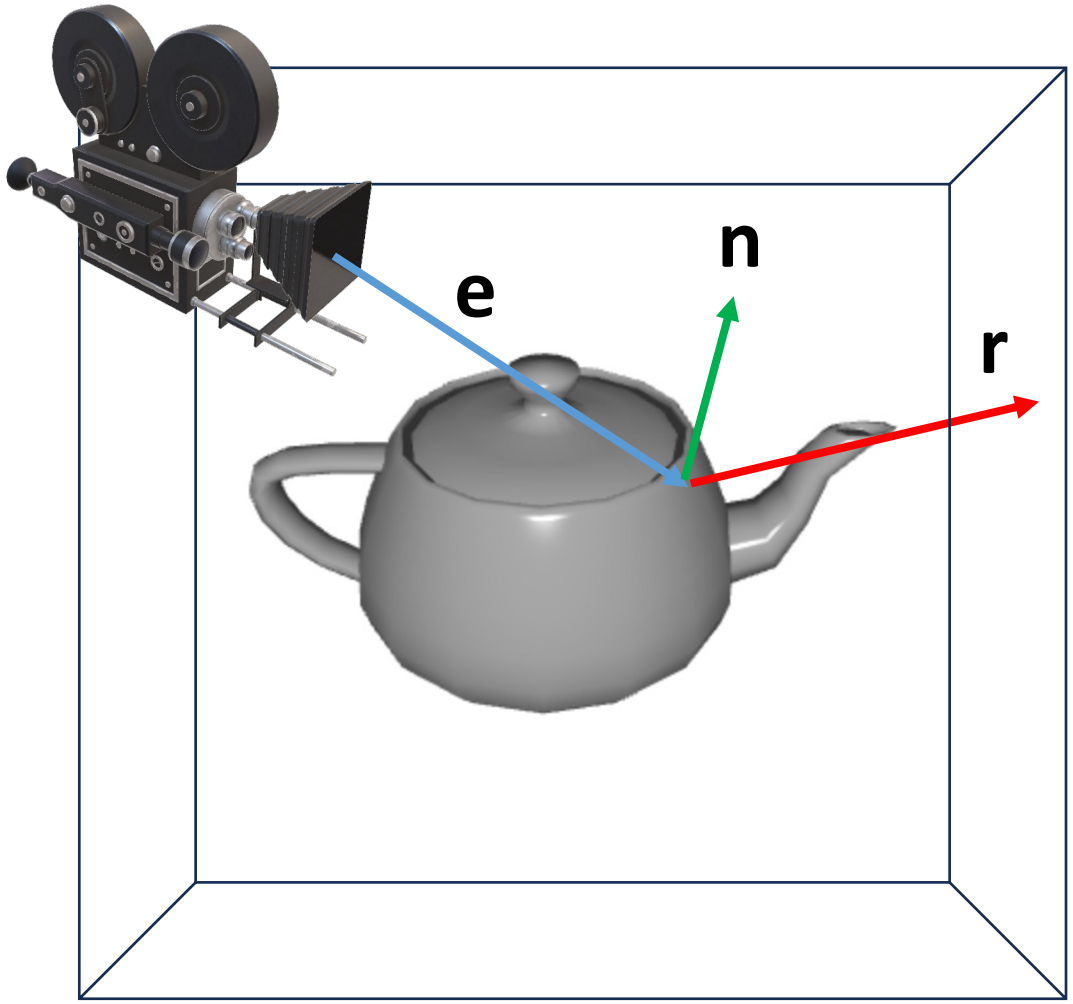


# Environment Mapping and Reflection

# General Idea: Sphere Mapping



# General Idea: Cube Environment Mapping




- From the camera/eye position, we are looking at the object
- We want to look up using the reflection vector (**r**) as shown
- All we need is the reflection vector from each point of the object