

CPSC 314
SHADERS, OPENGL, & JS
RENDERING PIPELINE

UGRAD.CS.UBC.CA/~CS314

slide credits:

Mikhail Bessmeltsev

A 3D rendered scene of a bar. In the foreground, there are several glasses: a tall shot glass with a large ice cube, a wine glass with red wine, a wine glass with white wine, and a beer glass with golden beer. In the background, there is a green bottle lying on its side and two blue dice. The scene is lit with warm, golden light, creating a realistic and detailed environment.

WHAT IS RENDERING?

Generating image from a 3D scene

A 3D rendered scene of a bar. In the foreground, there are several glasses: a tall shot glass with a large ice cube, a wine glass with red wine, a wine glass with white wine, and a tall glass with amber liquid. In the background, there is a green bottle lying on its side and two blue dice on the table. The scene is lit with warm, golden light, creating a realistic and detailed environment.

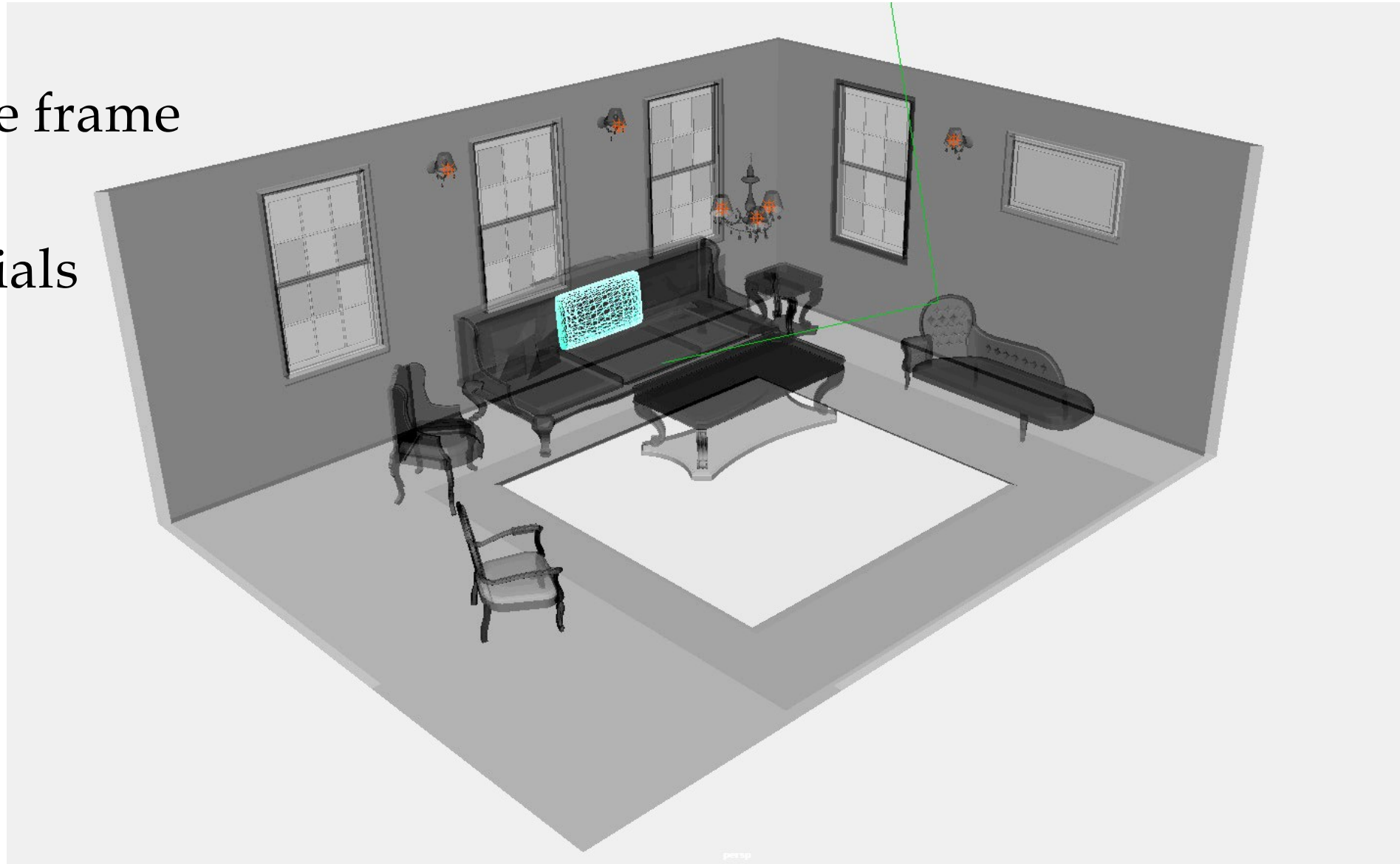
WHAT IS RENDERING?

Generating image from a 3D scene

Let's think HOW.

SCENE

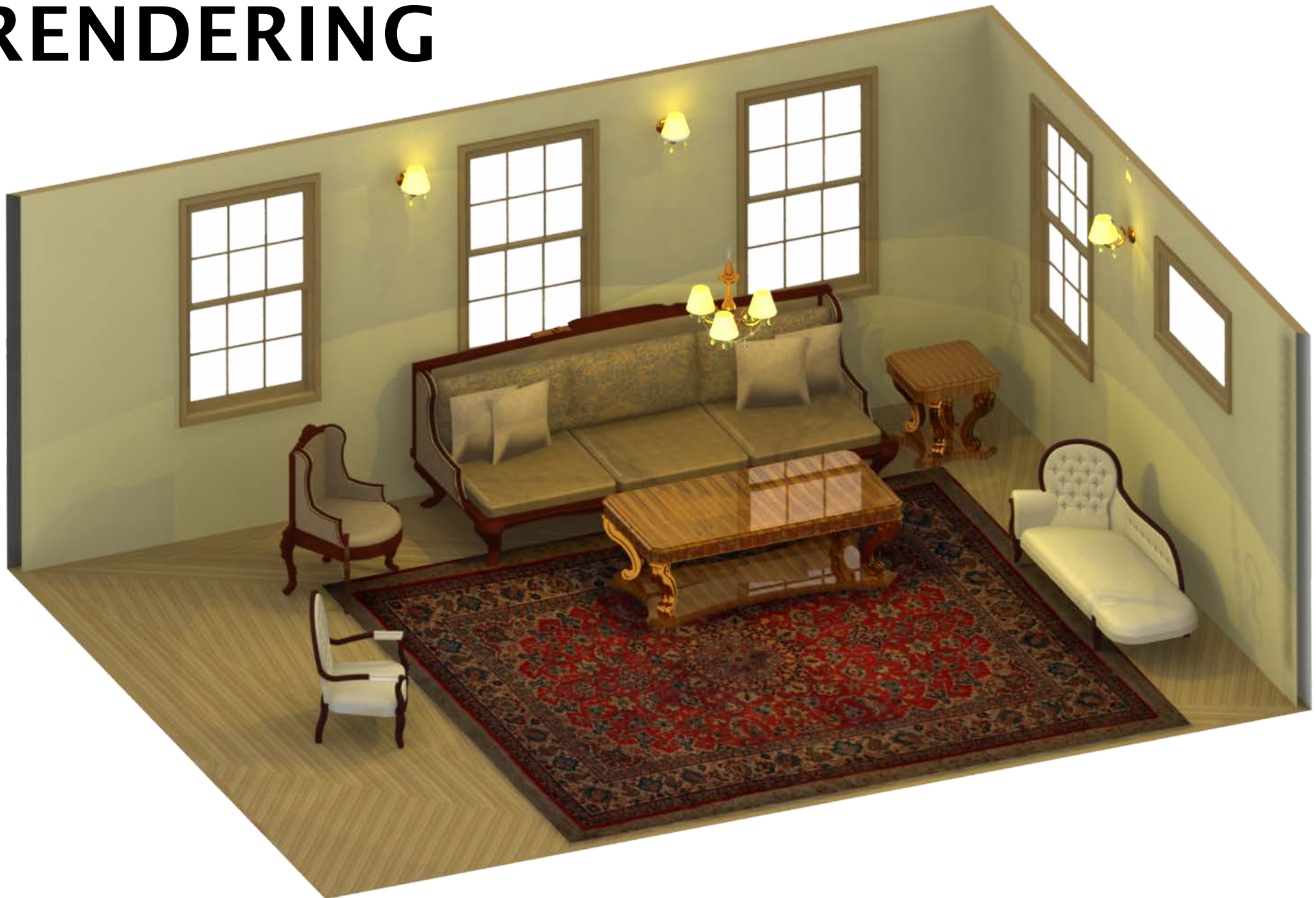
- A coordinate frame
- 3D objects
- Their materials
- Lights
- Cameras



RENDERING



RENDERING



FRAME BUFFER

- Portion of RAM on videocard (GPU)
- What we see on the screen
- Rendering destination

SCREEN

- Displays what's in frame buffer
- Terminology:

Pixel: basic element on device

Resolution: number of rows & columns in device

Measured in

- Absolute values (1K x 1K)
- Density values (300 dots per inch)

Scene

Coordinate Frame
3D objects
Materials
Lights
Cameras

?



Framebuffer

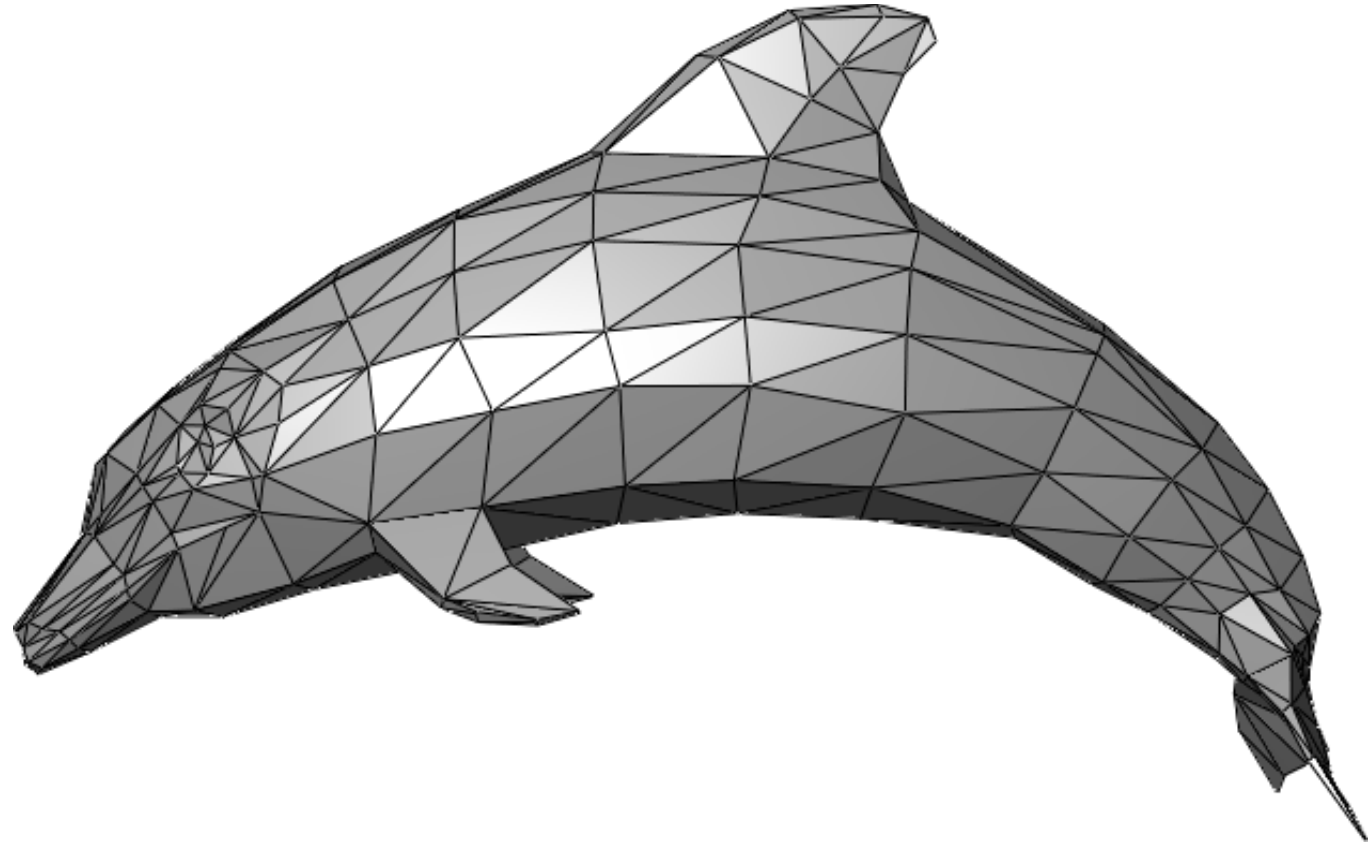
final image

SINGLE OBJECT

- How to describe a single piece of geometry?
- So far geometry has been constructed for you.

SHAPES: TRIANGLE MESHES

- Triangle = 3 vertices
- Mesh = {vertices, triangles}
- Example



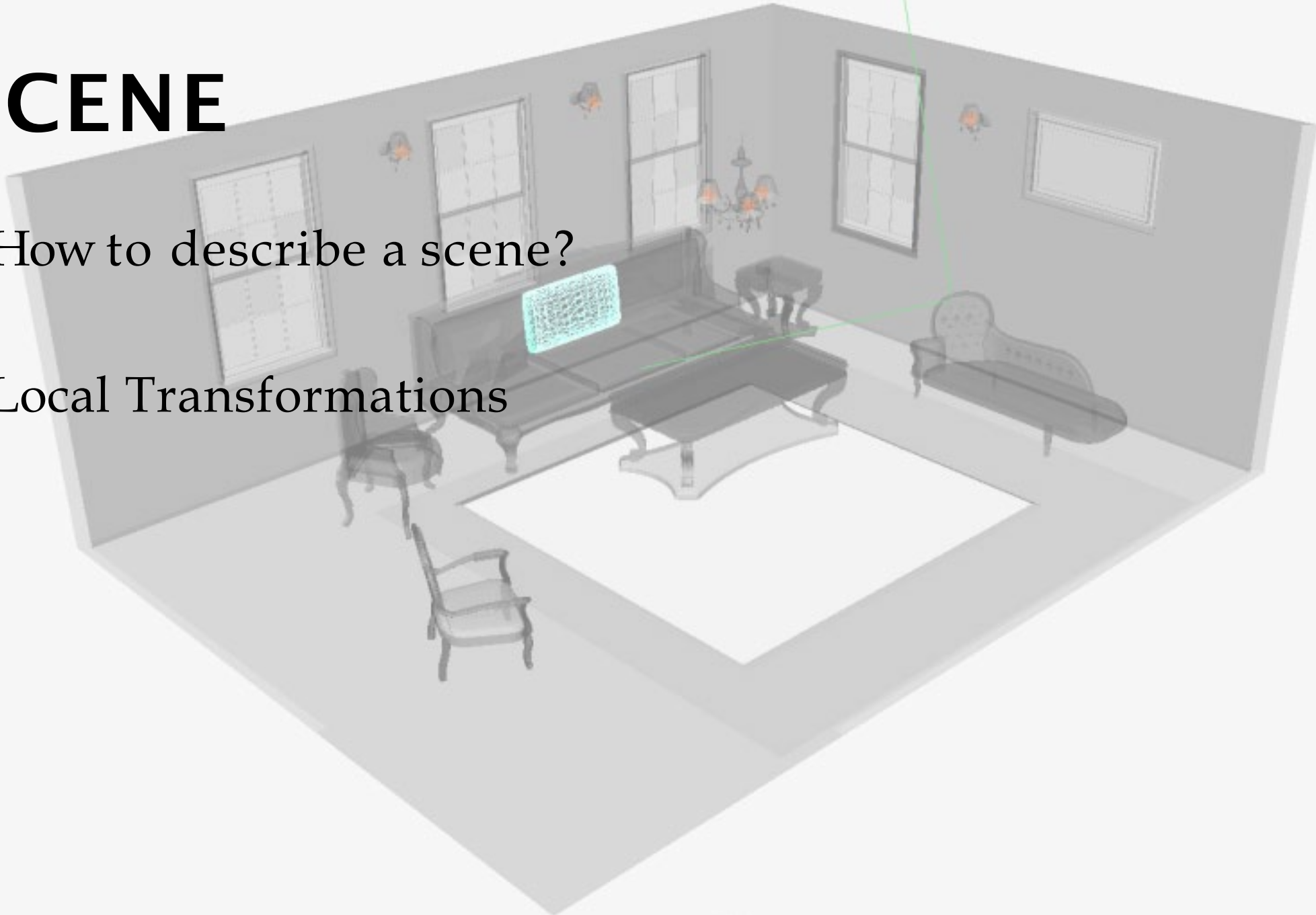
SCENE

- How to describe a scene?



SCENE

- How to describe a scene?
- Local Transformations



Scene

Coordinate Frame
3D objects
Materials
Lights
Cameras

?



Framebuffer

final image

SKETCH OF A RENDERING PIPELINE

- Scene
 - Coordinate frame
 - 3D models
 - Coordinates
 - Local transforms
 - properties (color, material)
 - Lights
 - Camera

SKETCH OF A RENDERING PIPELINE

- **Scene**

- Coordinate frame
- 3D models
 - Coordinates
 - properties (color, material)
- Lights
- Camera

- **Camera View**

- 2D positions of shapes
- Depth of shapes
- Normals

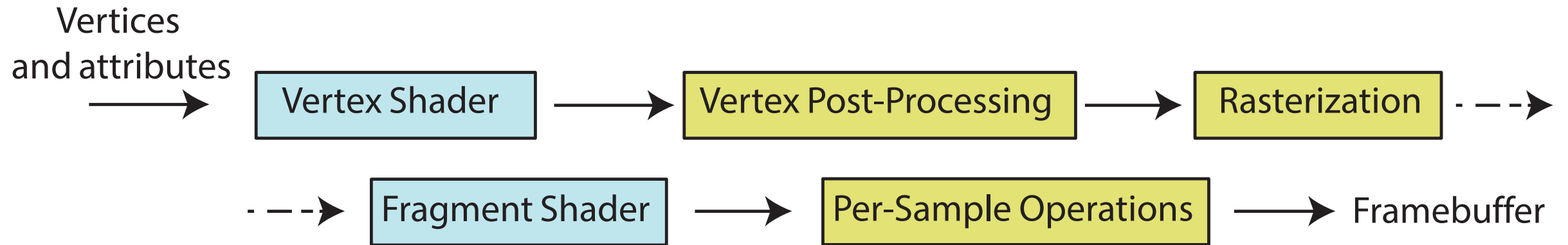
- **Image**

- Shape pixels
- Their color
- Which pixel is visible

OPENGL/WEBGL

- Open Graphics Library
- One of the most popular libraries for 2D/3D rendering
- A software interface to communicate with graphics hardware
- Cross-language API

OPENGL RENDERING PIPELINE



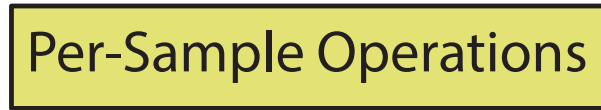
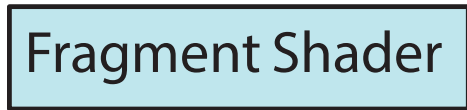
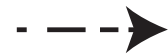
OPENGL RENDERING PIPELINE

Scene

Camera Coords

Device Coords

Vertices
and attributes



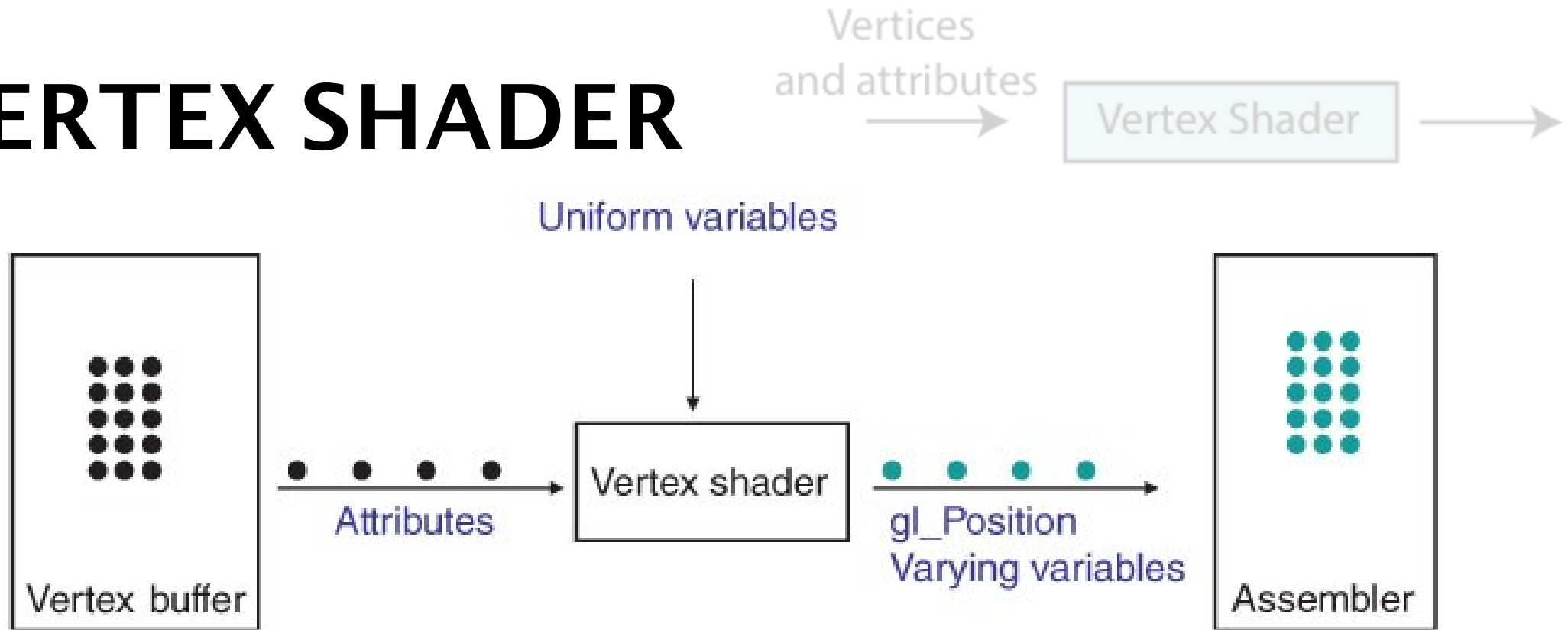
Framebuffer

Image

VERTEX SHADER

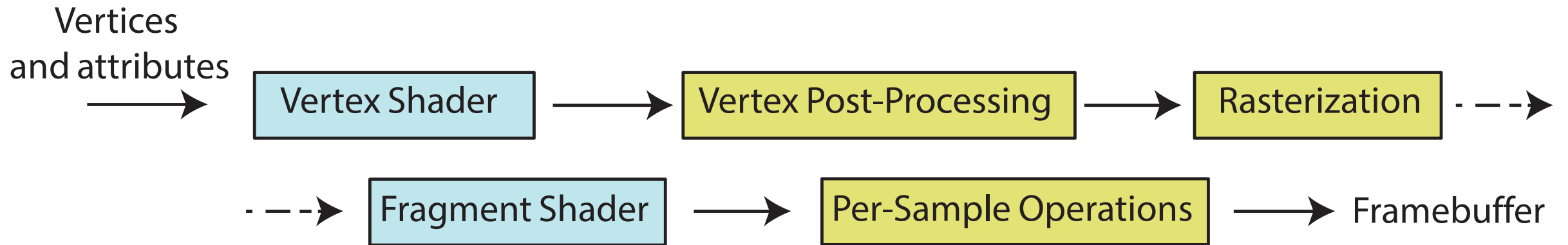


VERTEX SHADER



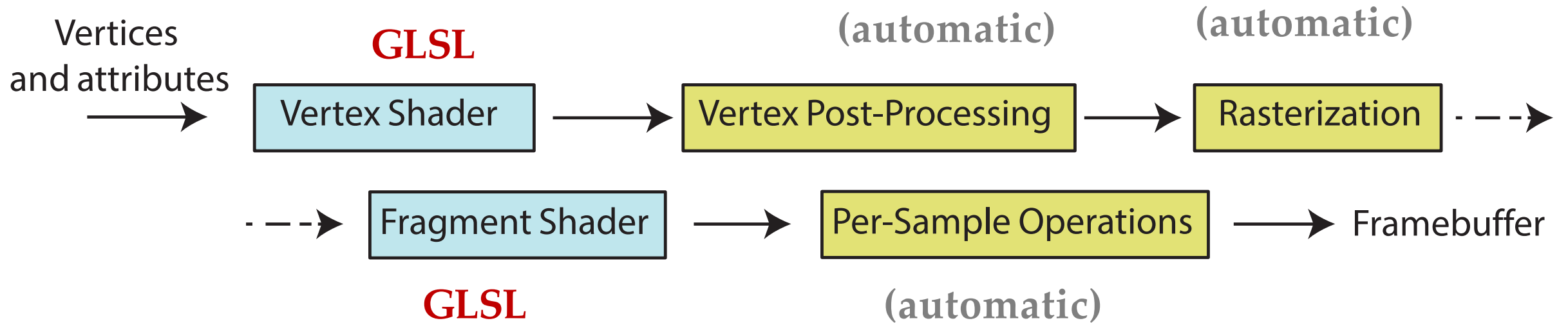
- Vertices are stored in vertex buffer
- Each one is processed by vertex shader
- Outputs 2D position
- May compute per-vertex variables (normal, etc.)

OPENGL RENDERING PIPELINE



OPENGL RENDERING PIPELINE

JavaScript
+ Three.js



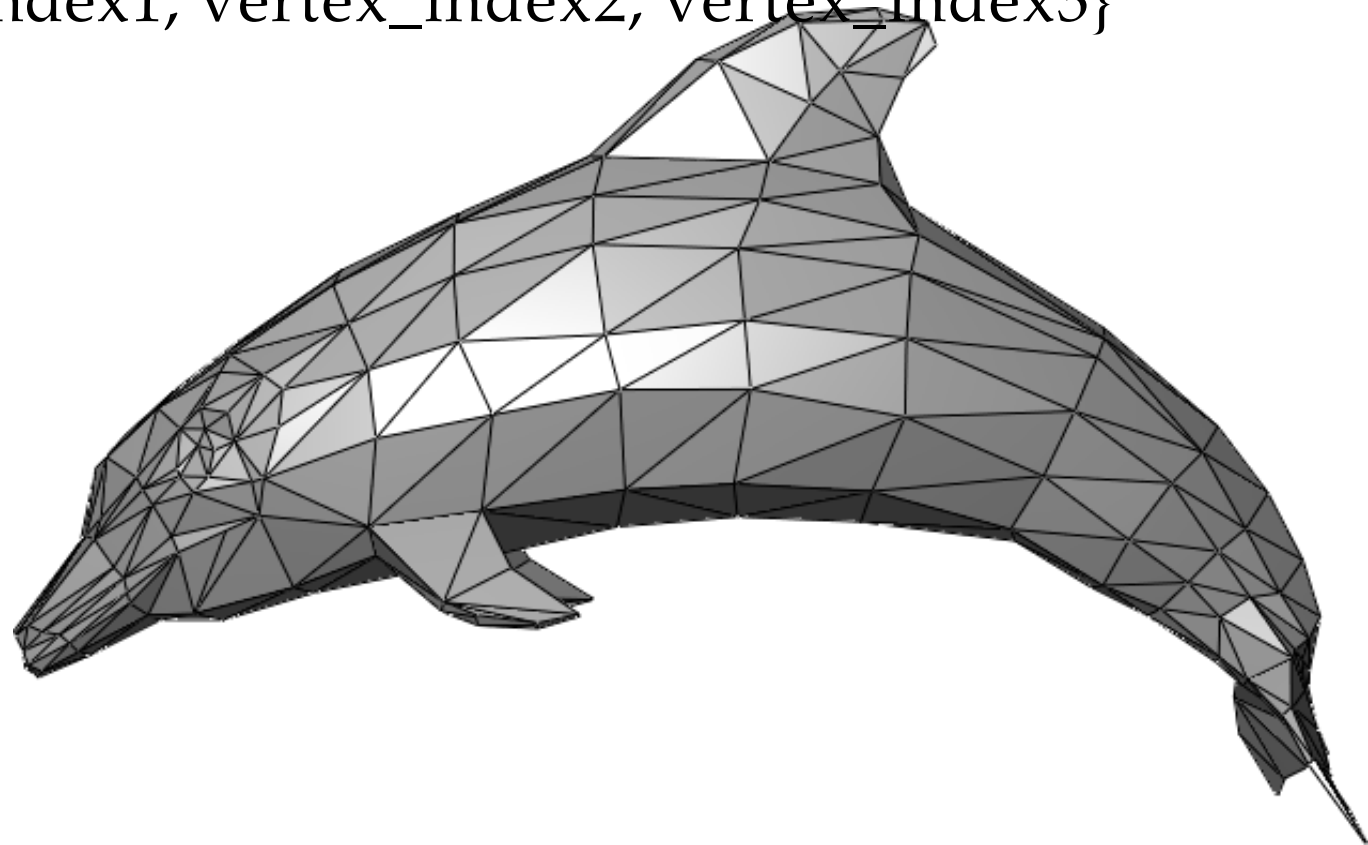
THREE.JS

- High-level library for Javascript
- Uses WebGL for rendering

- Has **Scene**, **Mesh**, **Camera** objects
- **Scene** is hierarchical
- **Mesh** has geometry and material properties
- **Camera** is used for rendering

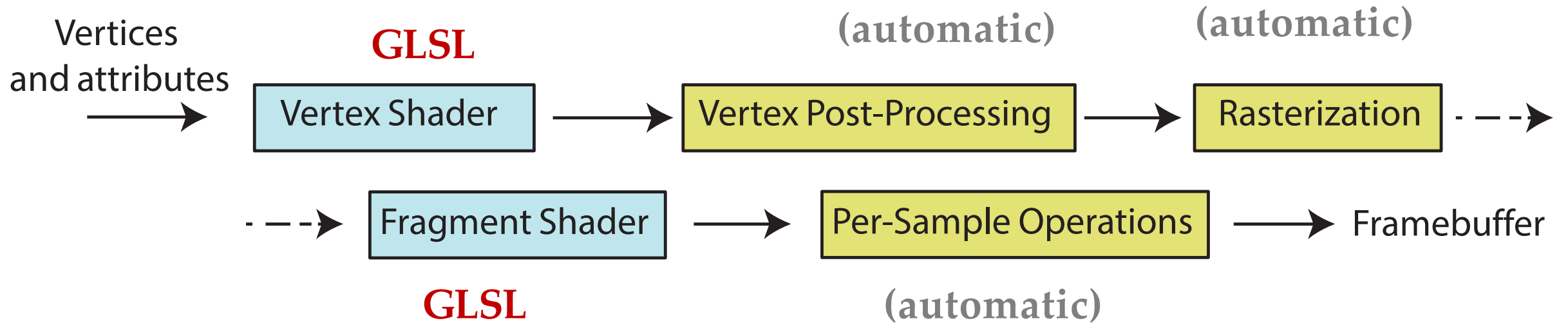
GEOMETRY

- Triangle meshes
 - Set of vertices
 - Triangle defines as {vertex_index1, vertex_index2, vertex_index3}



OPENGL RENDERING PIPELINE

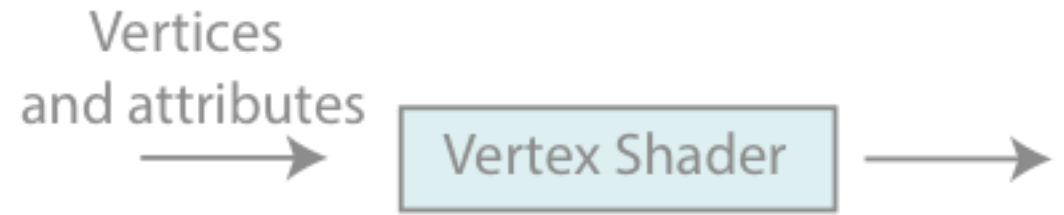
JavaScript
+ Three.js



GLSL

- OpenGL shading language
- Used for Fragment and Vertex shaders
- Lots of useful stuff:
 - `vec3`, `vec4`, `dvec4`, `mat4`, `sampler2D`
 - `mat*vec`, `mat*mat`
 - `Reflect`, `refract`
 - `vec3 v(a.xy, 1)`

VERTEX SHADER



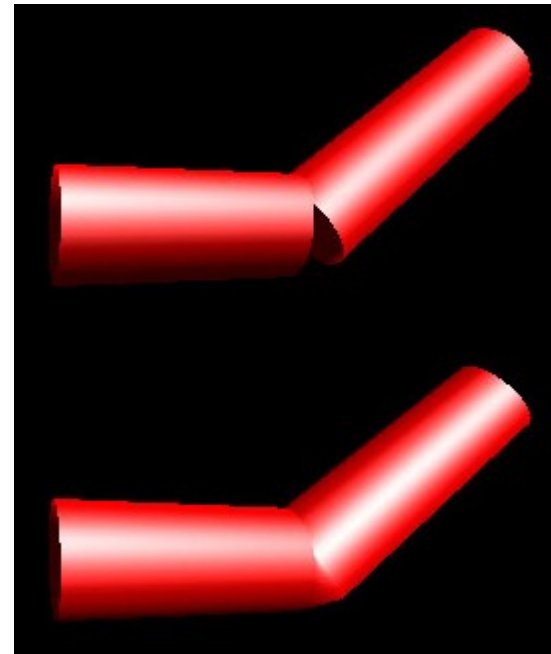
- VS is run for each vertex SEPARATELY
- By default doesn't know connectivity
- Input: vertex coordinates in Object Coordinate System
- Its main goal is to set **gl_Position**

Object coordinates -> WORLD coordinates -> **VIEW coordinates**

VERTEX SHADER



- Except simple conversion to world coordinates
- You can do anything with vertices (or anything that's passed)
 - e.g. deform vertices
 - e.g. skinning!

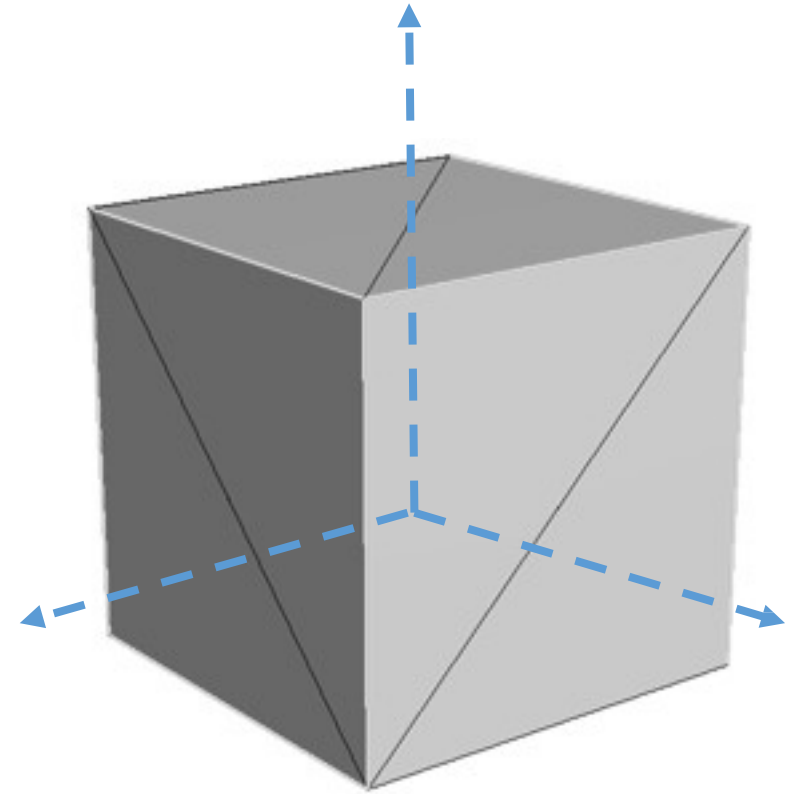


[courtesy NVIDIA]

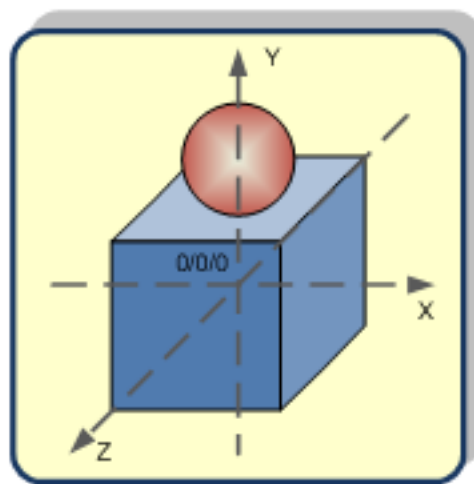
```
var verticesOfCube = [  
  -1,-1,-1,    1,-1,-1,    1,  1,-1,    -1,  1,-1,  
  -1,-1,  1,    1,-1,  1,    1,  1,  1,    -1,  1,  1,  
];
```

```
var indicesOfFaces = [  
  2, 1, 0,    0, 3, 2,  
  0, 4, 7,    7, 3, 0,  
  0, 1, 5,    5, 4, 0,  
  1, 2, 6,    6, 5, 1,  
  2, 3, 7,    7, 6, 2,  
  4, 5, 6,    6, 7, 4  
];
```

```
var geometry = new THREE.PolyhedronGeometry(  
verticesOfCube, indicesOfFaces, 6, 2 );
```



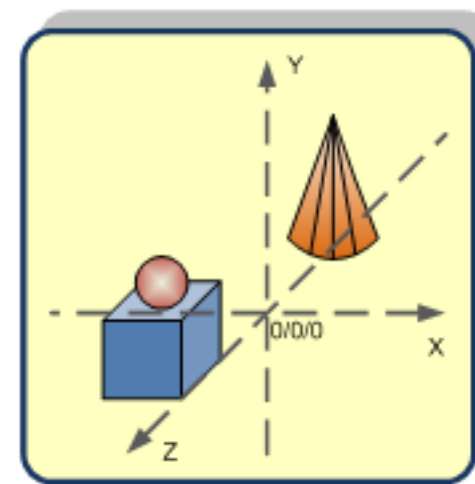
GEOMETRY
(JAVASCRIPT/THRE.JS)



Object Space



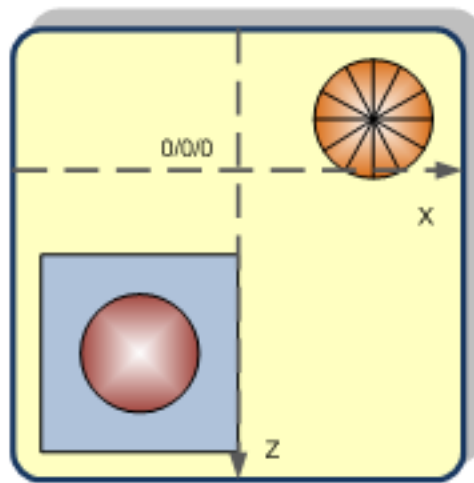
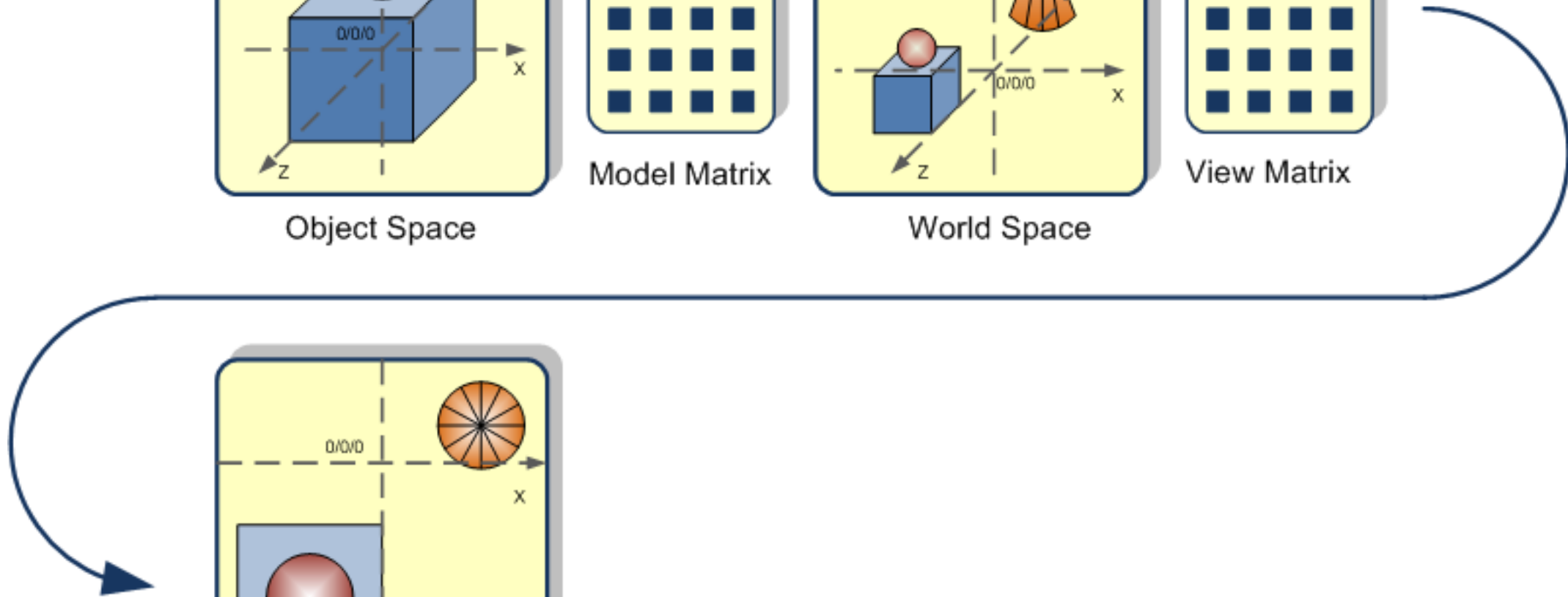
Model Matrix



World Space



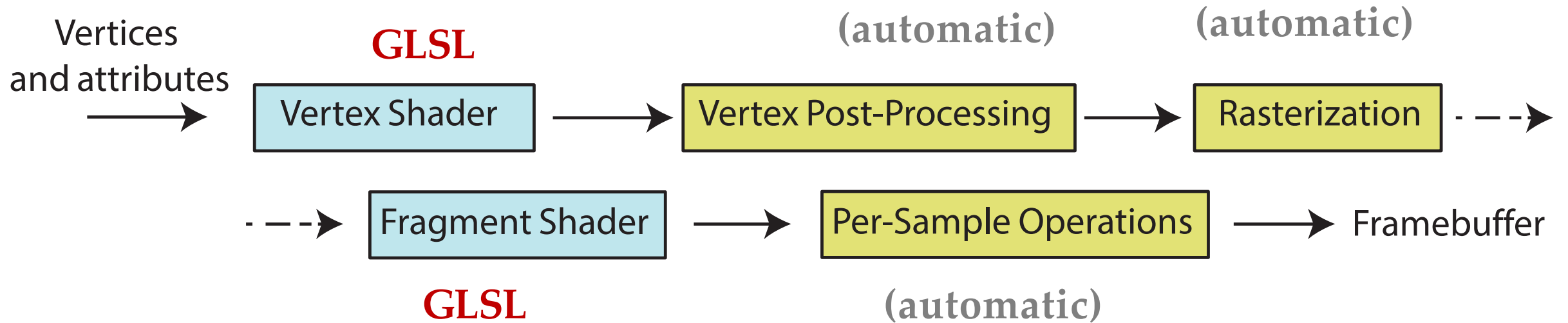
View Matrix



Camera Space

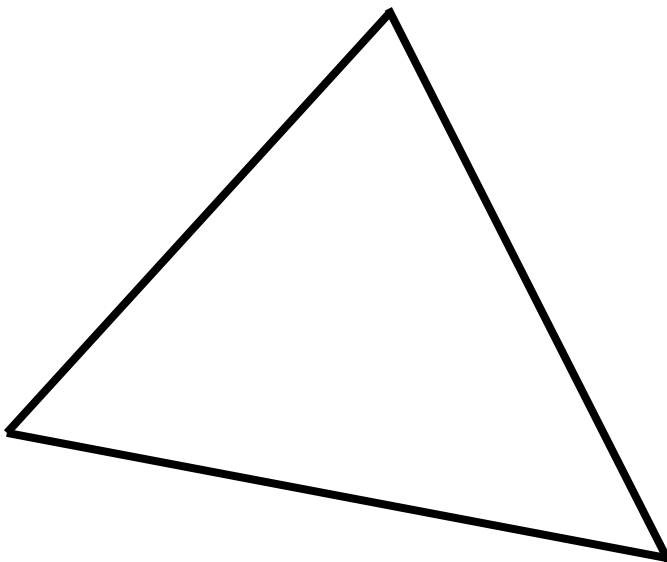
OPENGL RENDERING PIPELINE

JavaScript
+ Three.js

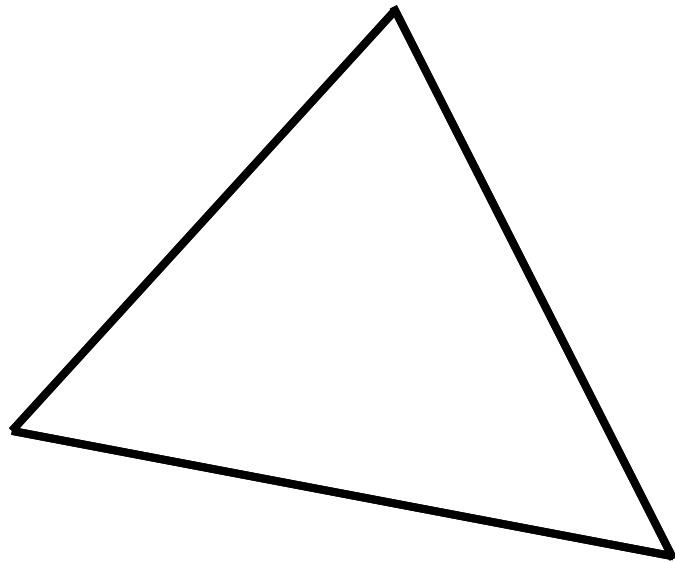




CAMERA VIEW

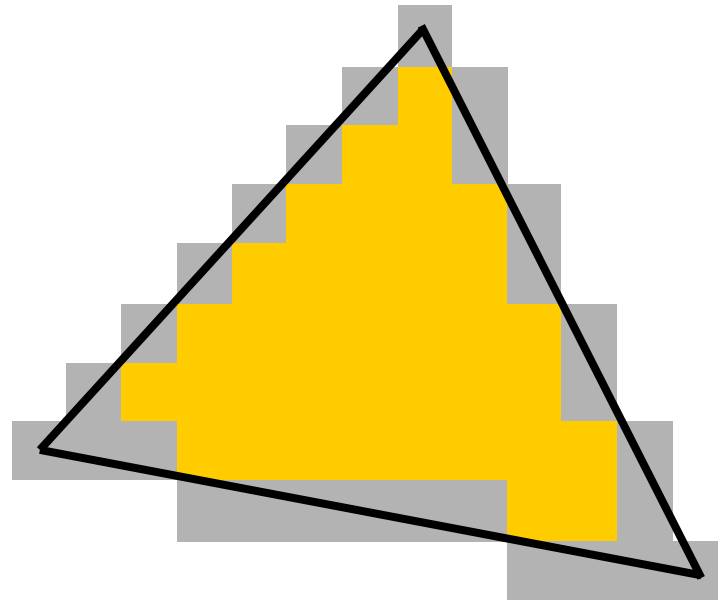


RASTERIZATION



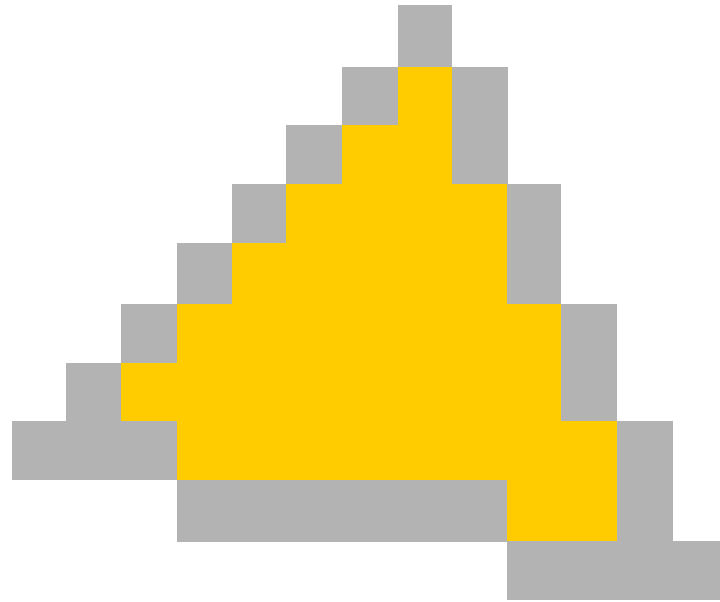
RASTERIZATION

-

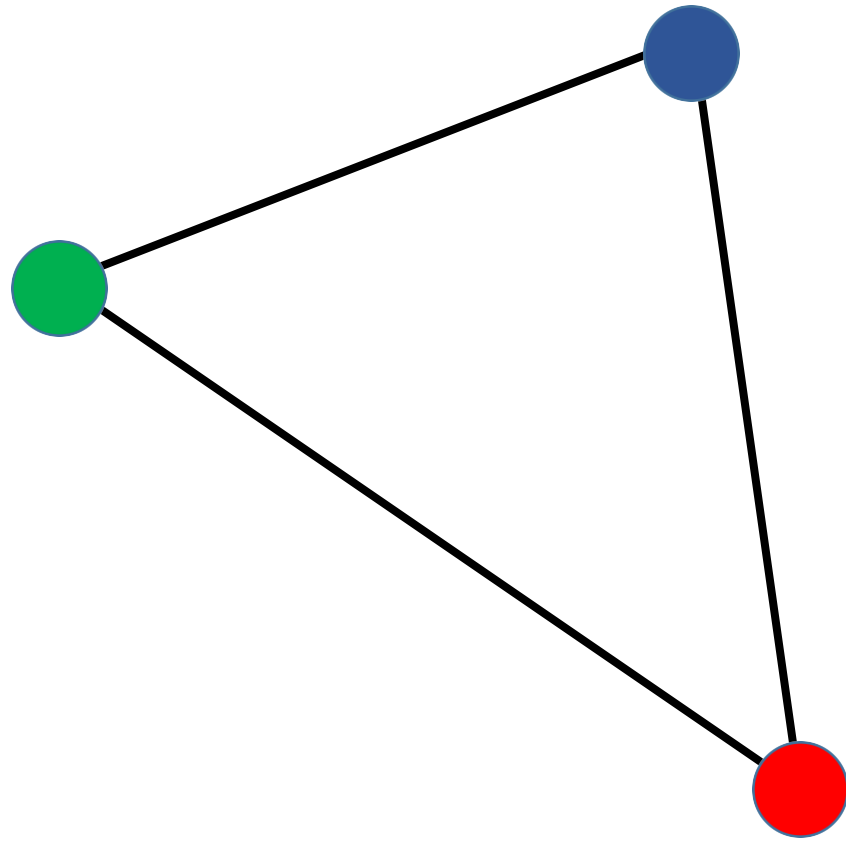


RASTERIZATION

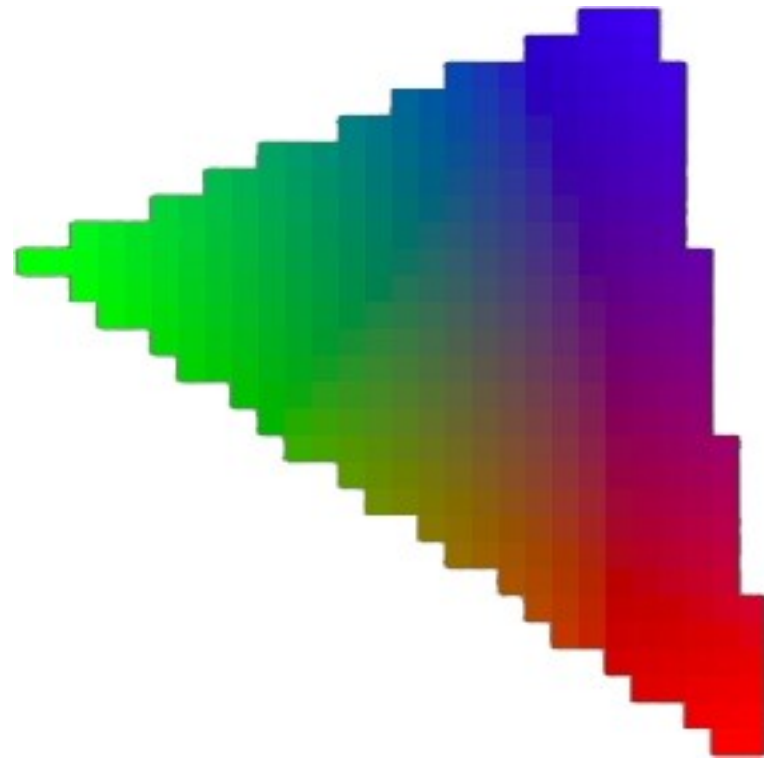
-



RASTERIZATION - INTERPOLATION

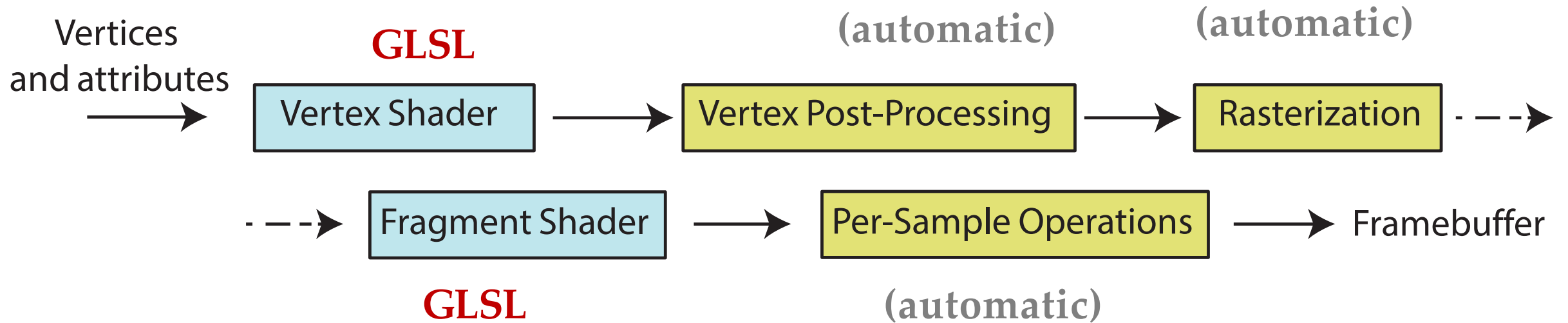


RASTERIZATION - INTERPOLATION



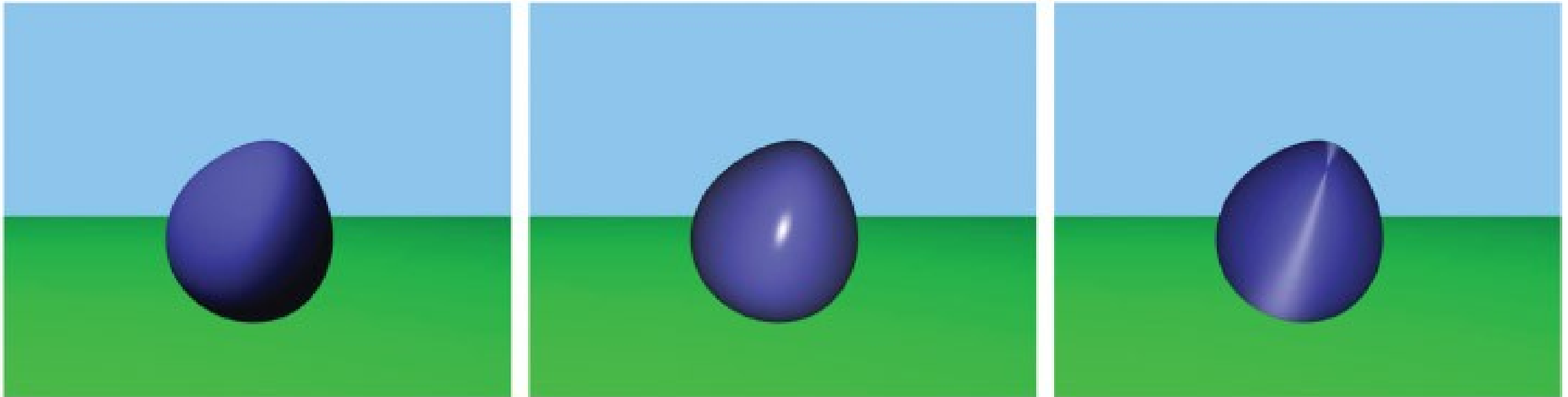
OPENGL RENDERING PIPELINE

JavaScript
+ Three.js



FRAGMENT SHADER

- Fragment = data for drawing a pixel
- Has `gl_FragCoord` - 2D window coords
- May set color!



FRAGMENT SHADER

- Common Tasks:
 - texture mapping
 - per-pixel lighting and shading
- Synonymous with Pixel Shader

MINIMAL VERTEX SHADER

```
void main()  
{  
    // Transforming The Vertex  
    gl_Position = modelViewMatrix * position;  
}
```

MINIMAL FRAGMENT SHADER

```
void main()  
{  
    // Setting Each Pixel To Red  gl_FragColor =  
    vec4(1.0, 0.0, 0.0, 1.0);  
}
```

MINIMAL VERTEX SHADER

```
void main()  
{  
    // Transforming The Vertex  
    gl_Position = modelViewMatrix * position;  
}
```

defined by Three.JS

MINIMAL FRAGMENT SHADER

```
void main()  
{  
    // Setting Each Pixel To Red  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

MINIMAL VERTEX SHADER

```
void main()  
{  
    // Transforming The Vertex  
    gl_Position = modelViewMatrix * position;  
}
```

defined by Three.JS

$$\begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

MINIMAL FRAGMENT SHADER

```
void main()  
{  
    // Setting Each Pixel To Red  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

MINIMAL VERTEX SHADER

```
void main()  
{  
    // Transforming The Vertex  
    gl_Position = modelViewMatrix * position;  
} view coordinate system defined by Three.JS
```

$$\begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

MINIMAL FRAGMENT SHADER

```
void main()  
{  
    // Setting Each Pixel To Red  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

MINIMAL VERTEX SHADER

```
void main()  
{  
    // Transforming The Vertex  
    gl_Position = modelViewMatrix * position;  
}
```

view coordinate system defined by Three.JS

$$\begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

MINIMAL FRAGMENT SHADER

```
void main()  
{  
    // Setting Each Pixel To Red  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

Red Green Blue Alpha

VERTEX SHADER – EXAMPLE 2

```
uniform float uVertexScale; attribute  
vec3 vColor; varying vec3 fColor;
```

```
void main() {  
    gl_Position = vec4(position.x * uVertexScale, position.y, 0.0, 1.0);  
    fColor = vColor;  
}
```

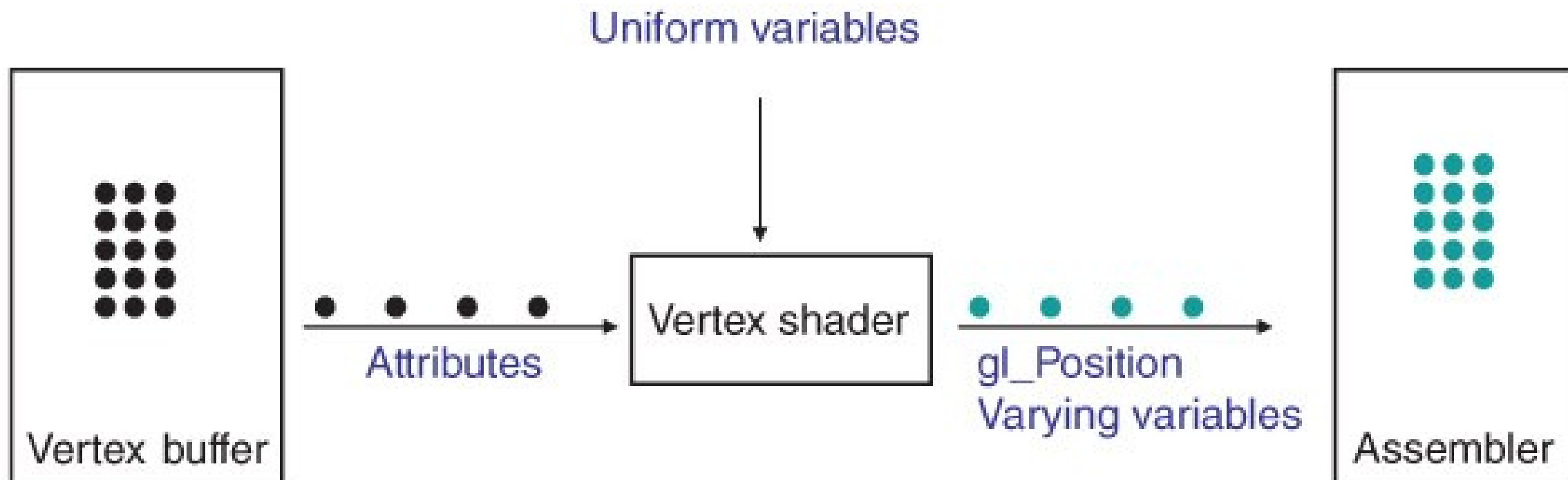

CONCEPTS

- uniform
 - same for all vertices
- varying
 - computed per vertex, automatically interpolated for fragments
- attribute
 - some values per vertex
 - available only in Vertex Shader

CONCEPTS

- **uniform** JS + Three.js → Vertex Shader → Fragment Shader
 - same for all vertices
- **varying** Vertex Shader → Fragment Shader
 - computed per vertex, automatically interpolated for fragments
- **attribute** JS + Three.js → Vertex Shader
 - some values per vertex
 - available only in Vertex Shader

VERTEX SHADER

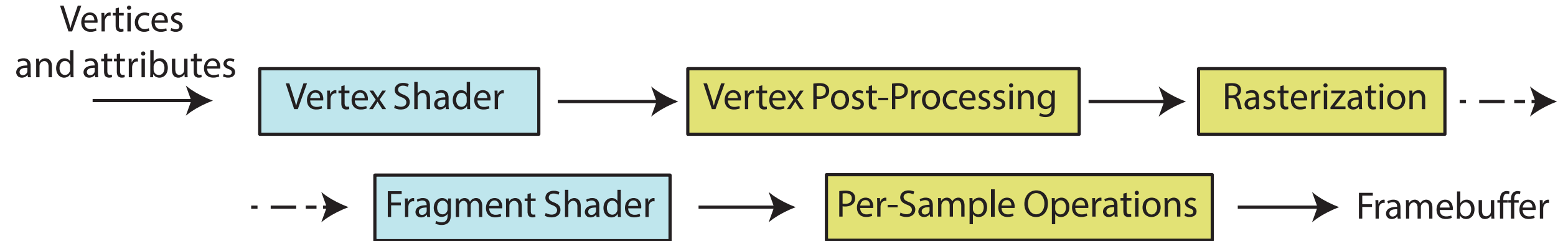


ATTACHING SHADERS

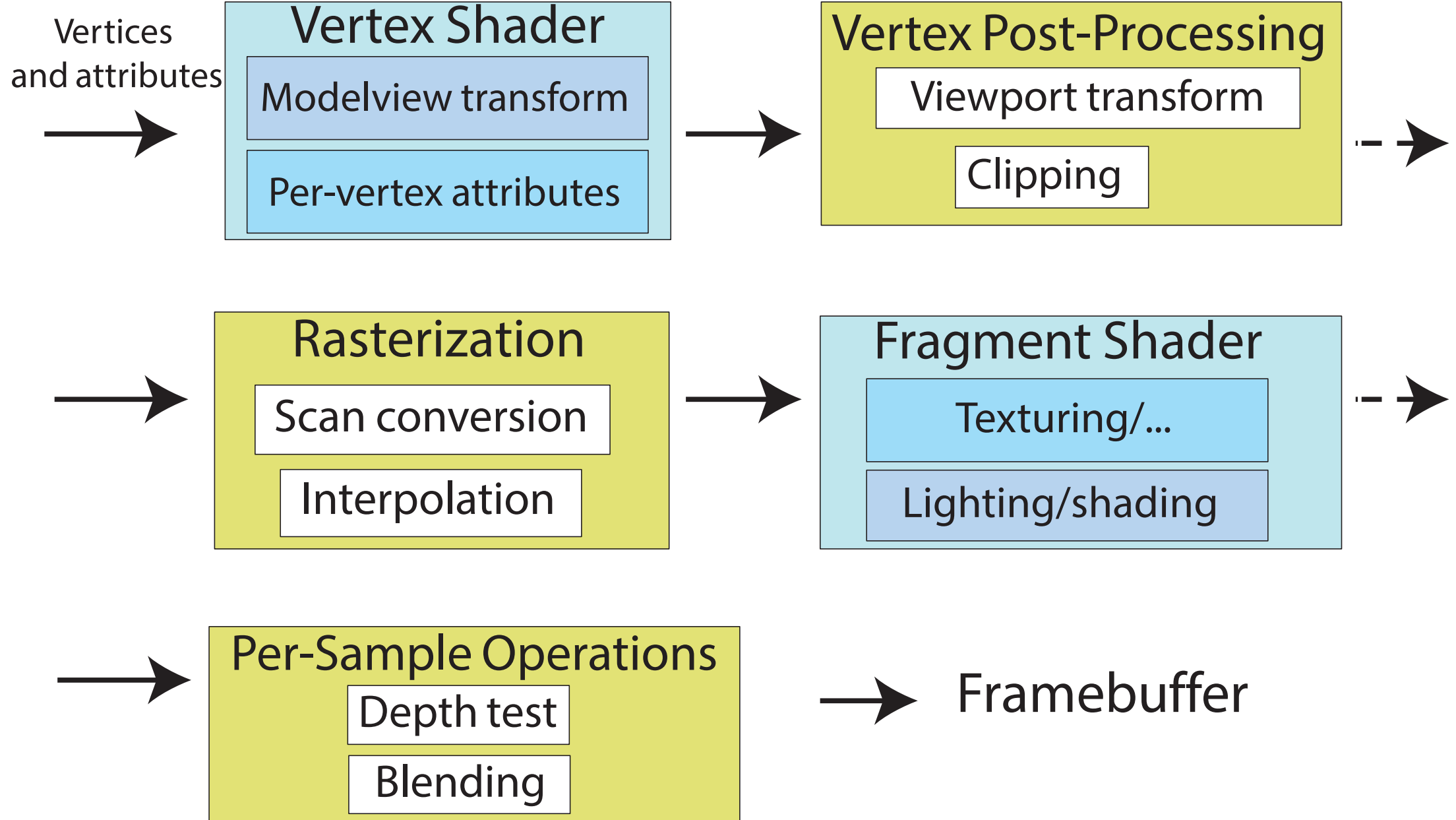
```
var remoteMaterial = new THREE.ShaderMaterial({
  uniforms: {
    remotePosition: remotePosition,
  },});
//here goes loading shader files into shaders[] ...
remoteMaterial.vertexShader = shaders['glsl/remote.vs.glsl'];
remoteMaterial.fragmentShader = shaders['glsl/remote.fs'];
var remoteGeometry = new THREE.SphereGeometry(1, 32, 32);
var remote = new THREE.Mesh(remoteGeometry, remoteMaterial);

scene.add(remote);
```

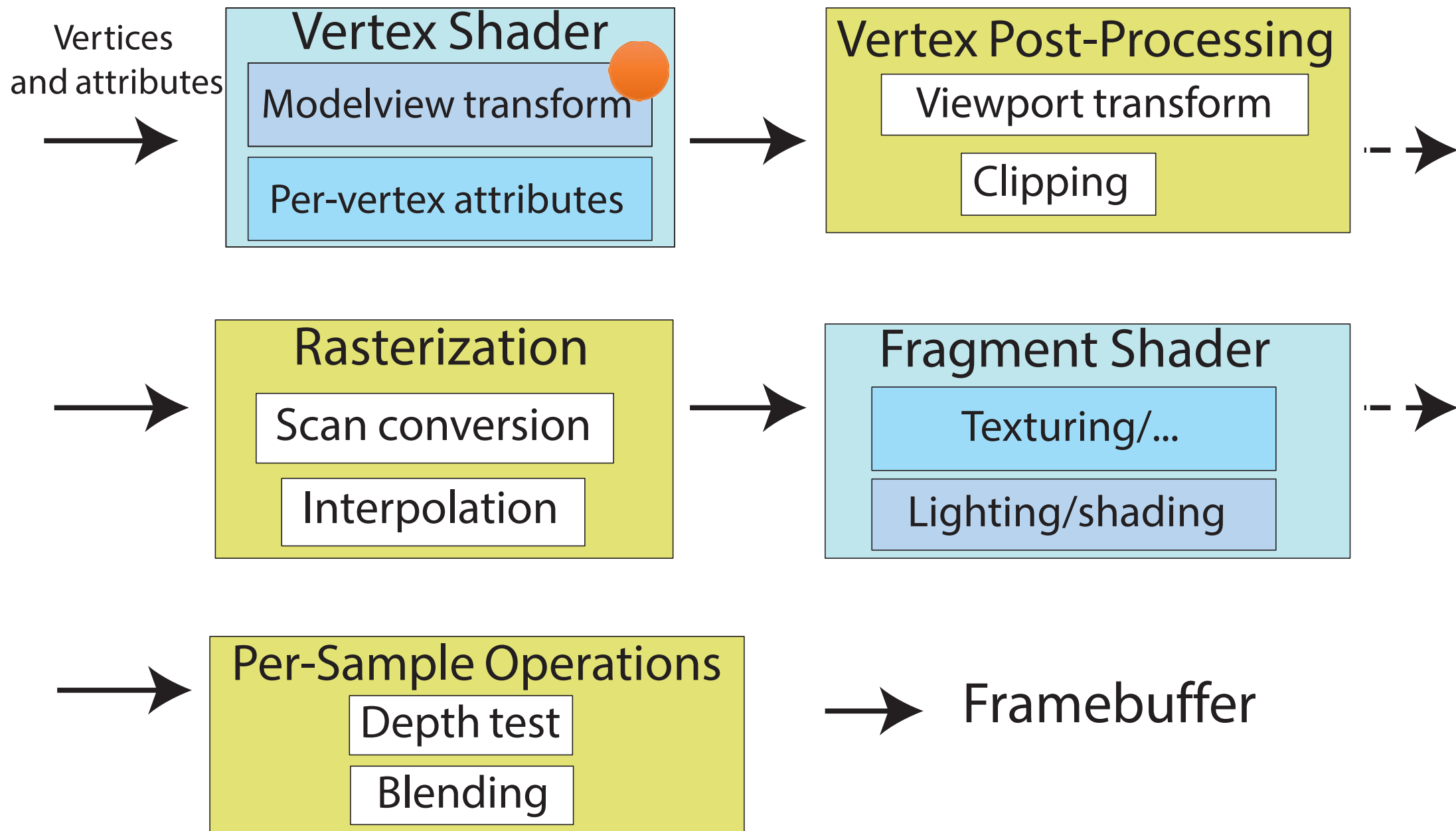
PIPELINE: MORE DETAILS



OPENGL RENDERING PIPELINE

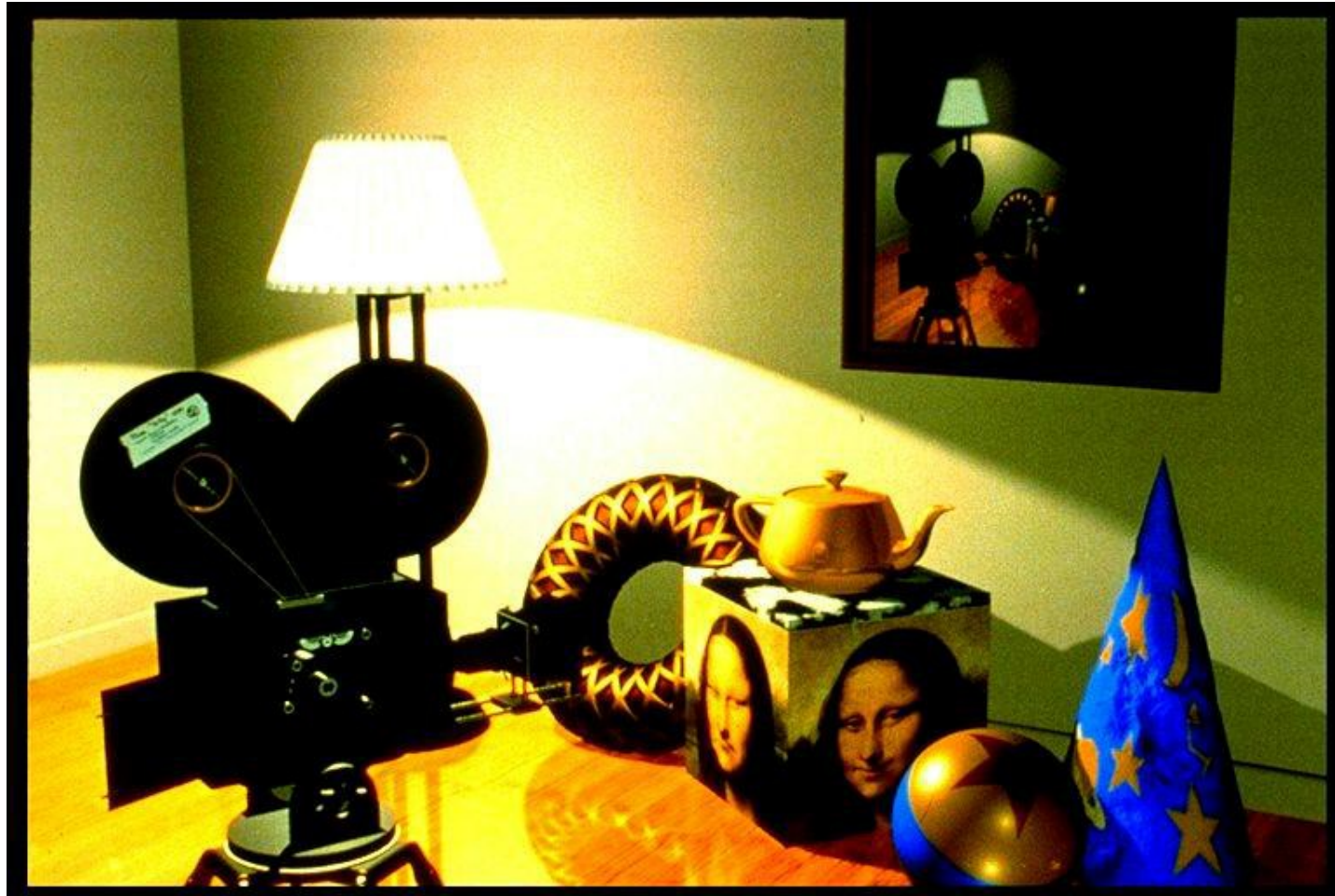


PIPELINE: MORE DETAILS

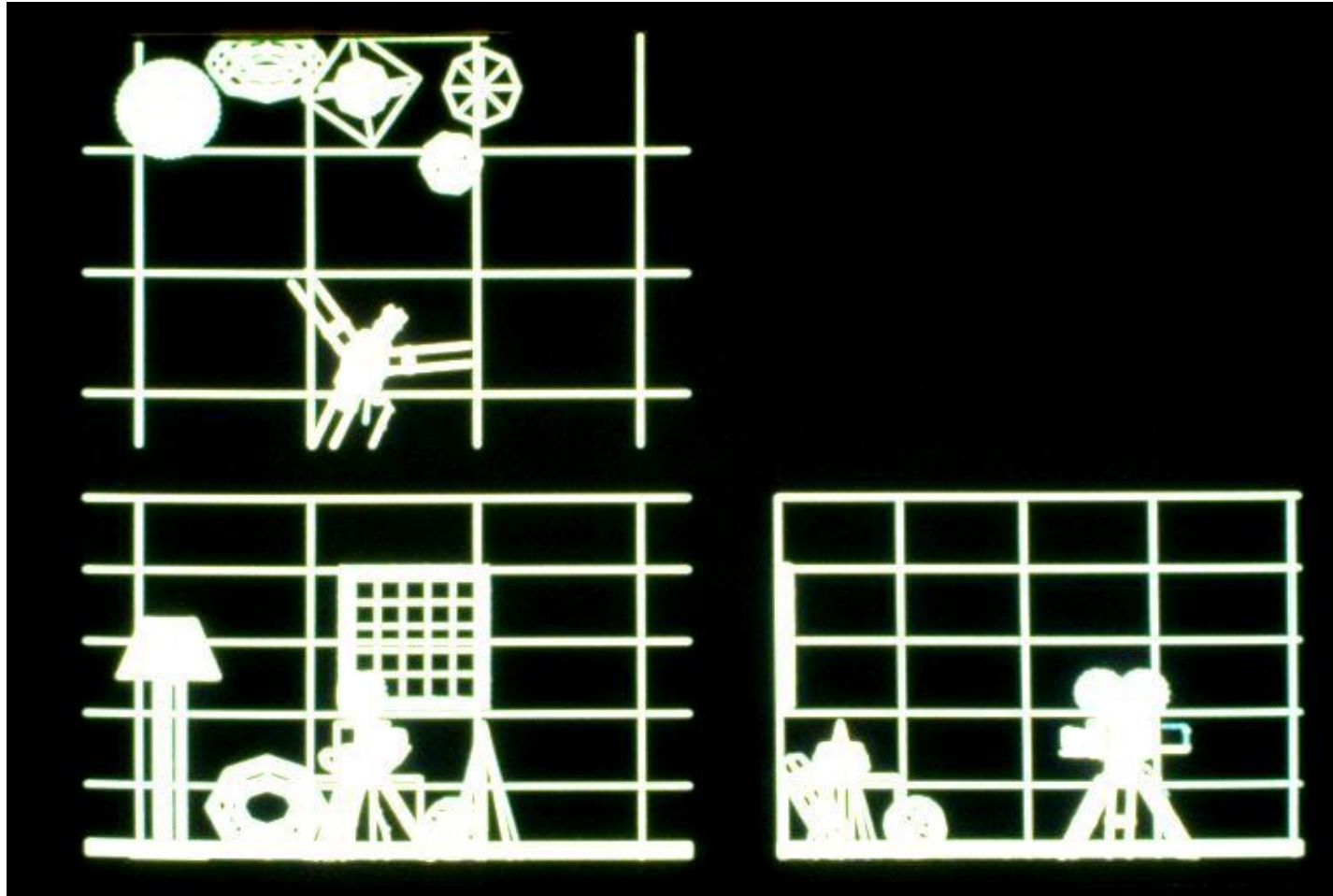


MODELING AND VIEWING TRANSFORMATIONS

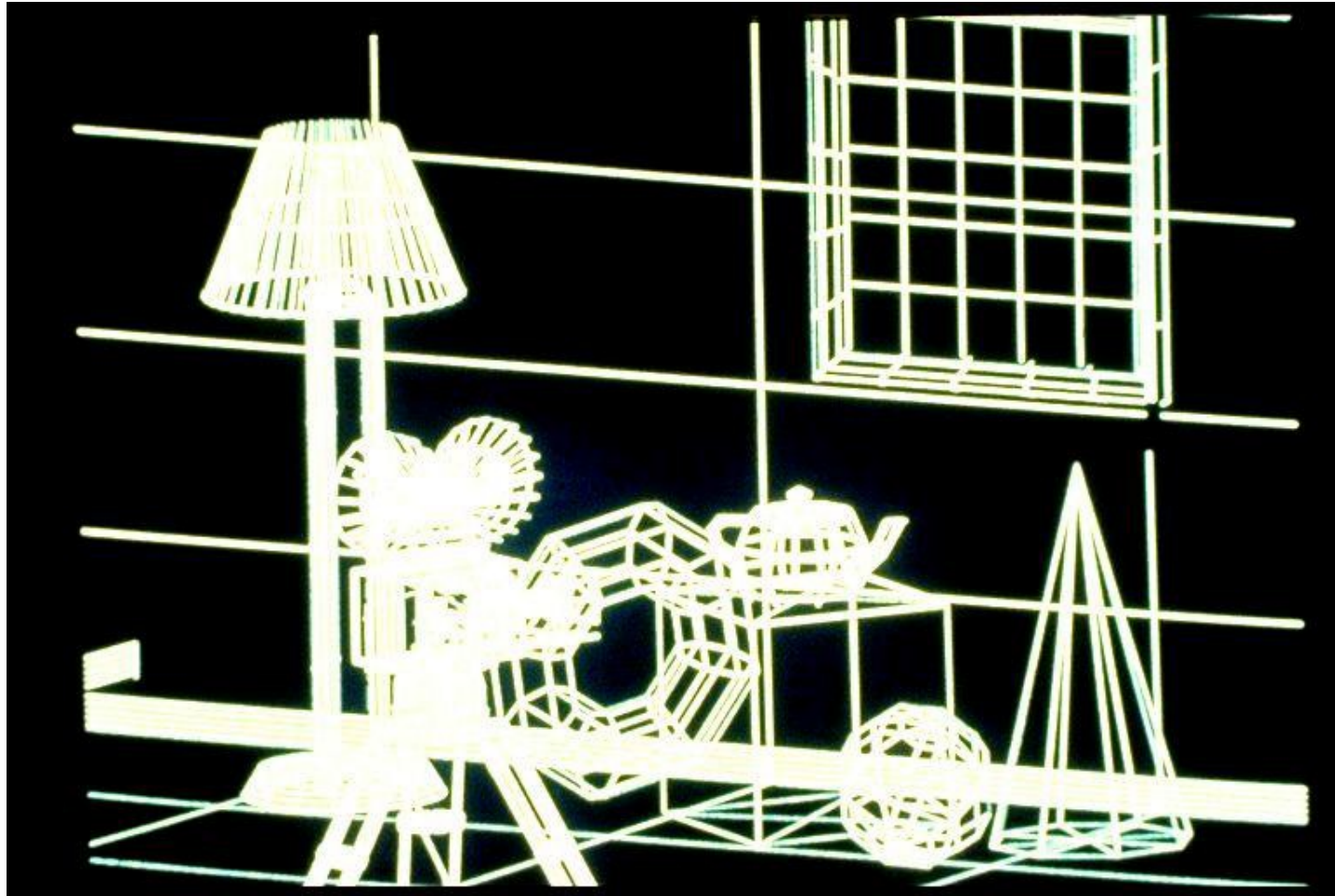
- Placing objects - Modeling transformations
 - Map points from object coordinate system to world coordinate system
- Looking from the camera - Viewing transformation
 - Map points from world coordinate system to camera (or eye) coordinate system



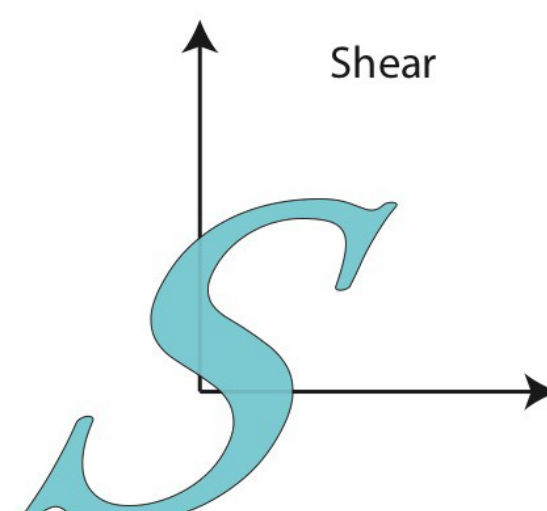
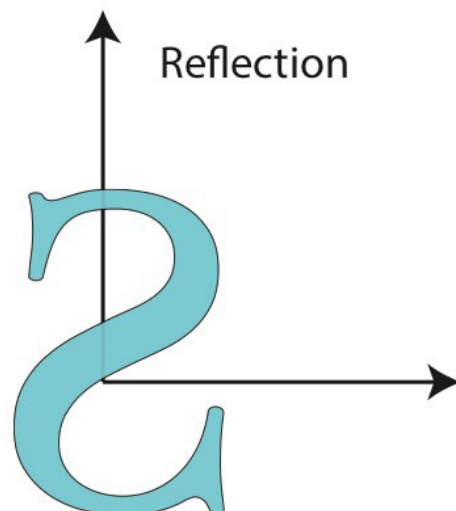
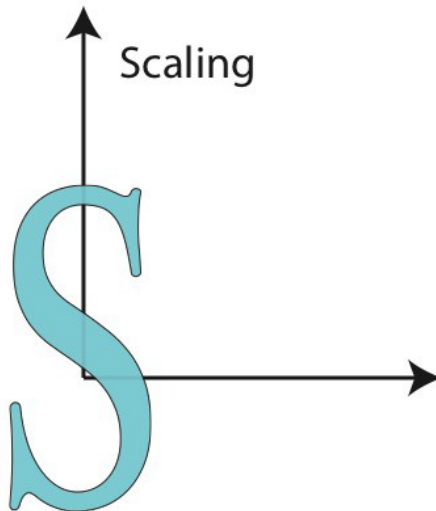
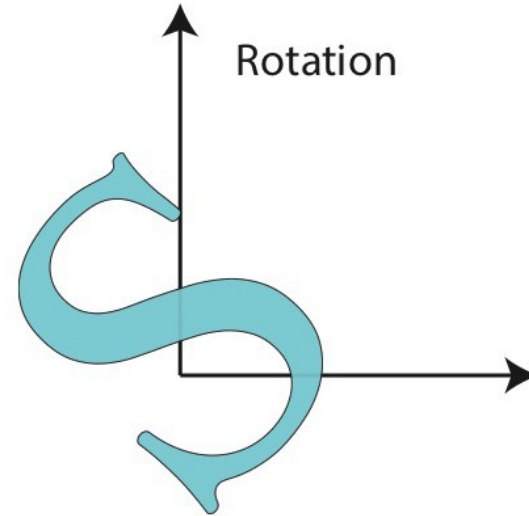
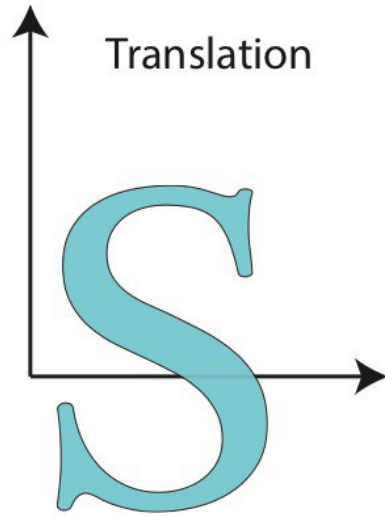
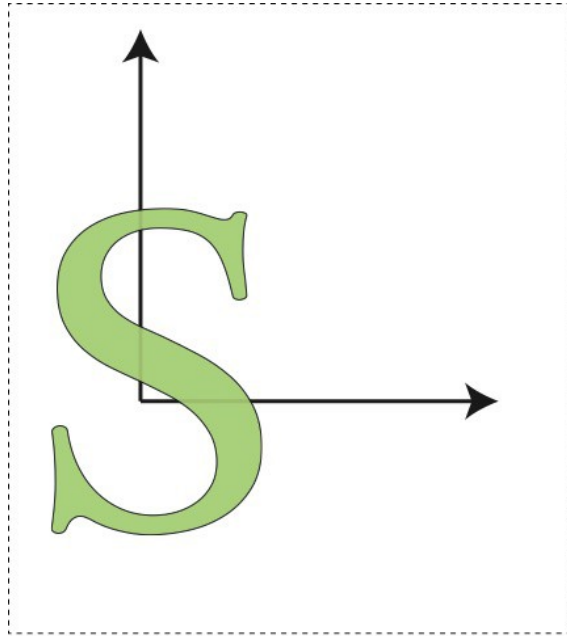
MODELING TRANSFORMATIONS: OBJECT PLACEMENT



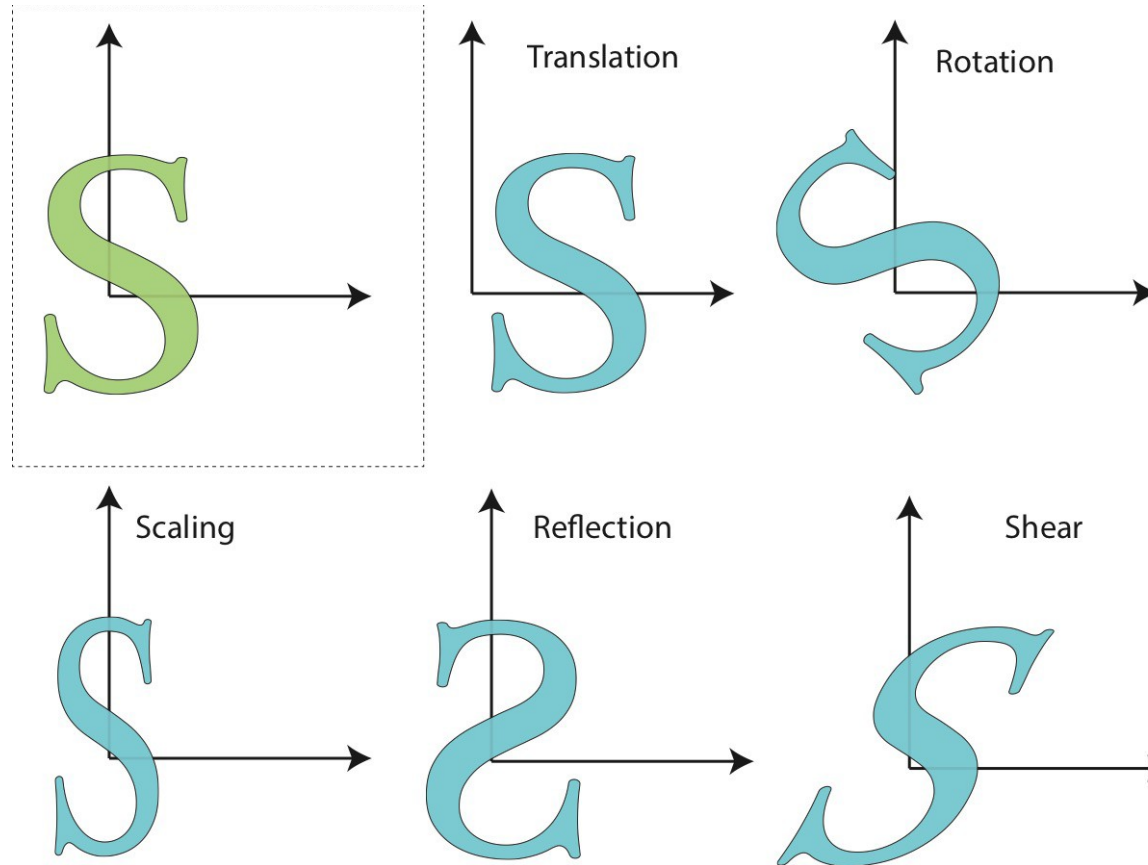
VIEWING TRANSFORMATION: LOOKING FROM A CAMERA



MODELING & VIEWING TRANSFORMATIONS



MODELING & VIEWING TRANSFORMATIONS



- Other transformations (not handled by rendering pipeline):
 - Freeform deformation



MODELING & VIEWING TRANSFORMATION

- Linear transformations
 - Rotations, scaling, shearing
 - Can be expressed as 3x3 matrix
 - E.g. scaling (non uniform):

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

MODELING & VIEWING TRANSFORMATION

- Affine transformations
 - Linear transformations + translations
 - Can be expressed as 3x3 matrix + 3 vector
 - E.g. scale+ translation:

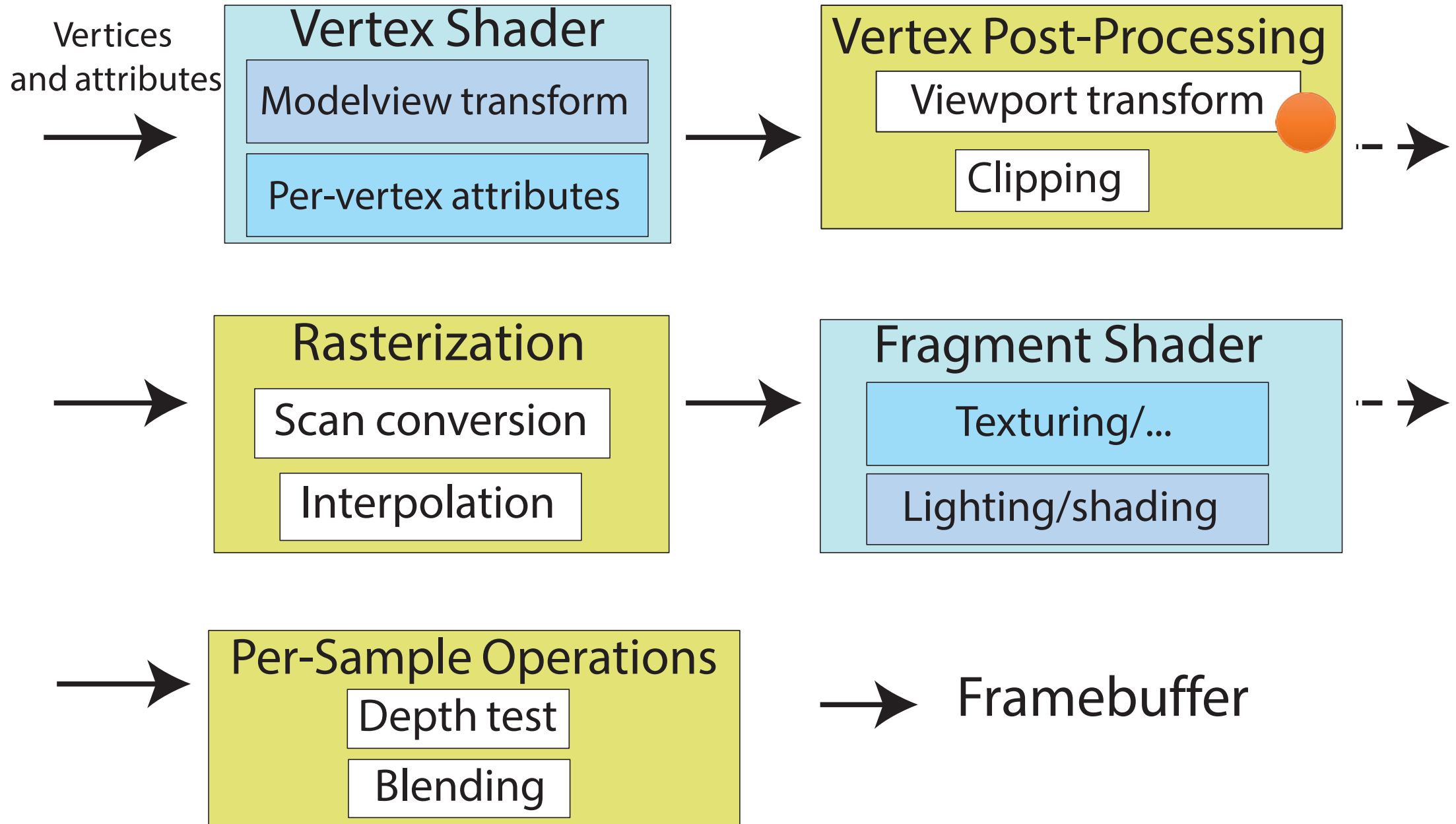
$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

- Another representation: 4x4 homogeneous matrix

MATRICES

- Object coordinates \rightarrow World coordinates
 - **Model Matrix**
 - One per object
- World coordinates \rightarrow Camera coordinates
 - **View Matrix**
 - One per camera

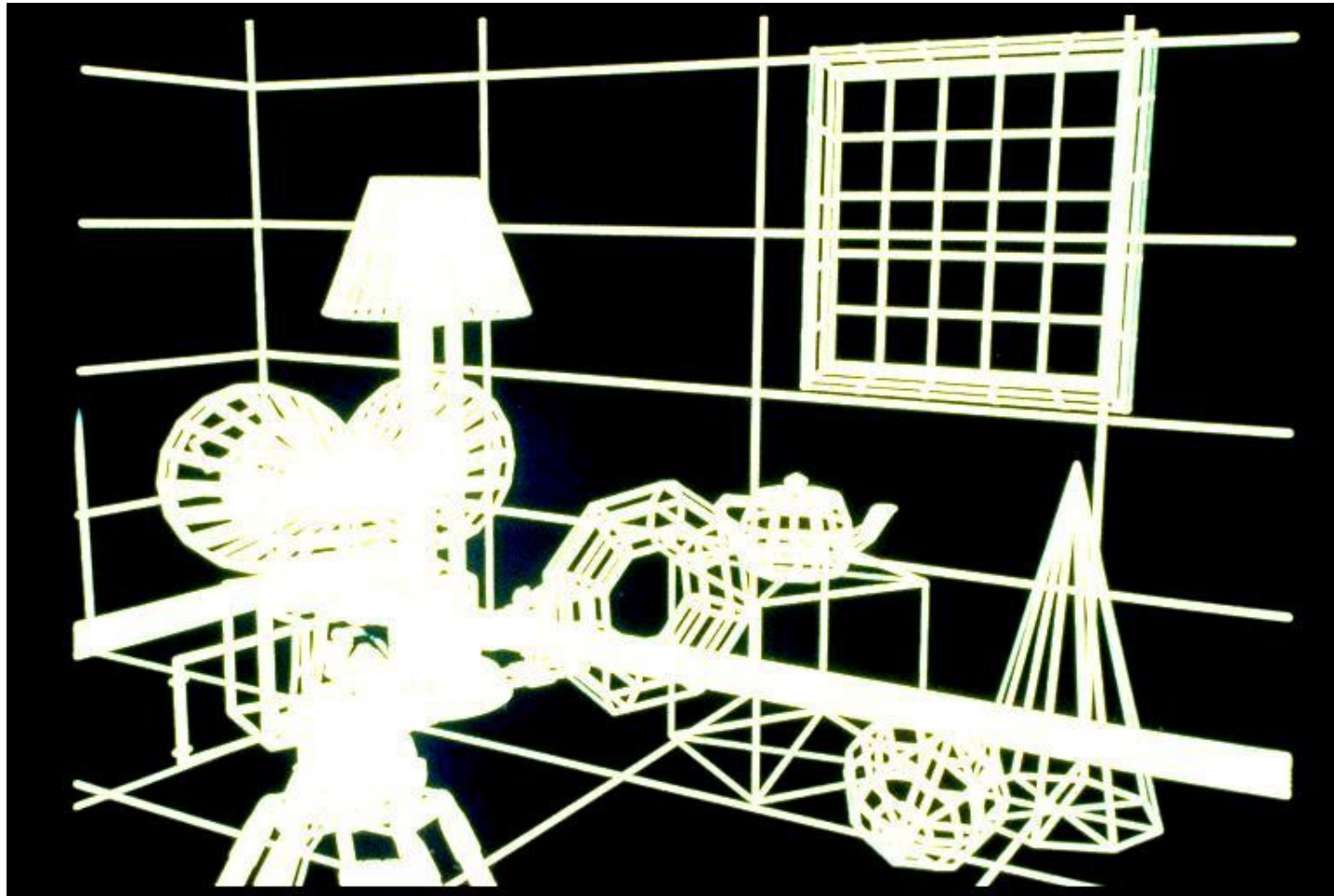
PIPELINE: MORE DETAILS



PERSPECTIVE TRANSFORMATION

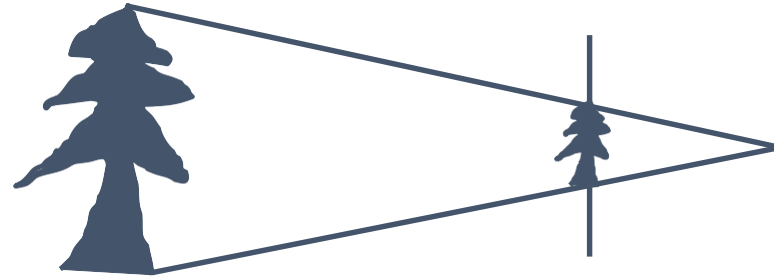
- Purpose:
 - Project 3D geometry to 2D image plane
 - Simulates a camera
- Camera model:
 - Pinhole camera (single view point)
 - More complex camera models exist, but are less common in CG

PERSPECTIVE PROJECTION



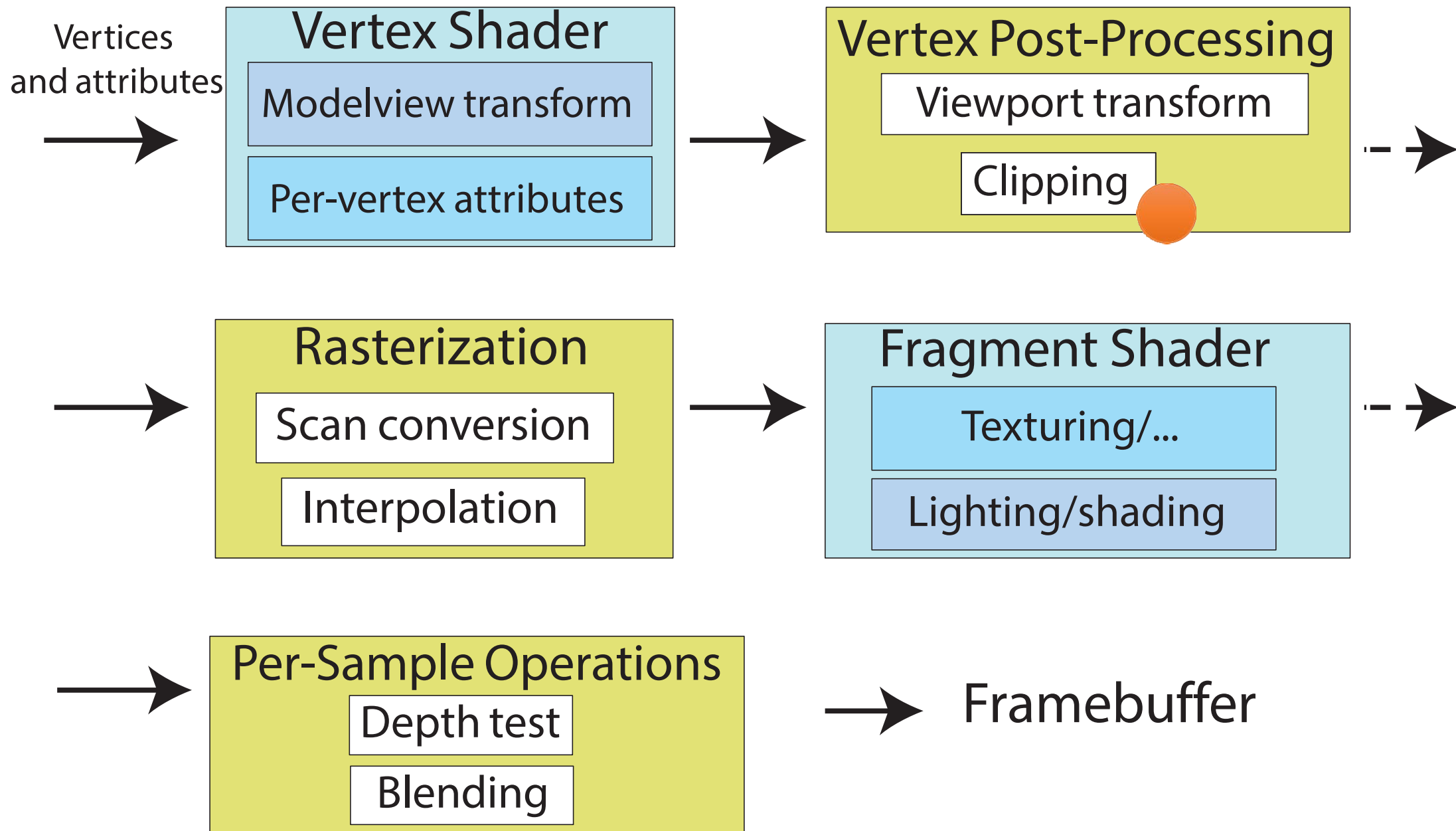
PERSPECTIVE TRANSFORMATION

- In computer graphics:
 - Image plane conceptually in front of center of projection



- Perspective transformation is **one of** projective transformations
- Linear & affine transformations also belong to this class
- All projective transformations can be expressed as 4x4 matrix operations

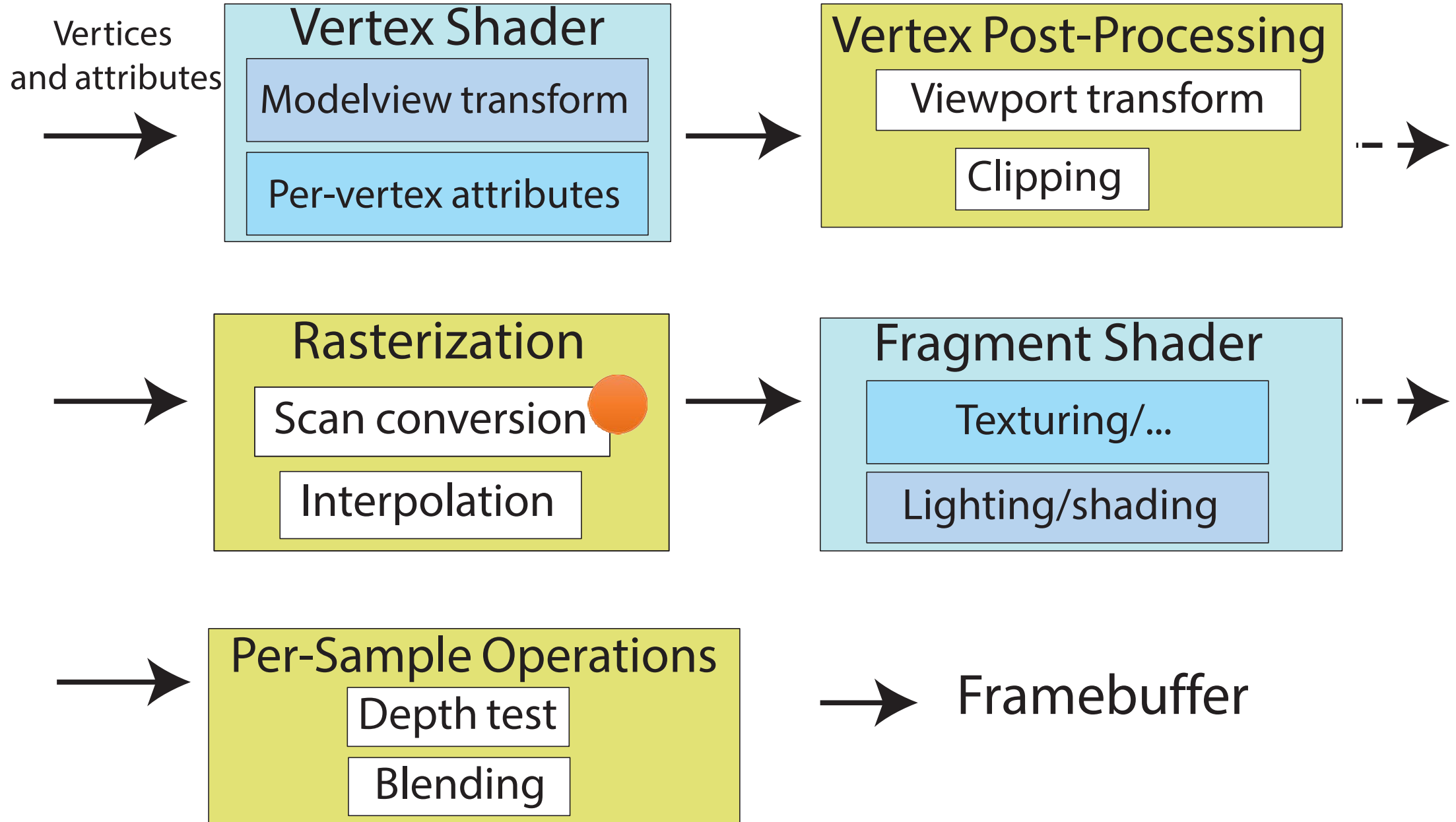
PIPELINE: MORE DETAILS



CLIPPING

- Removing invisible geometry
 - Geometry outside viewing frustum
 - Plus too far or too near one
- Optimization

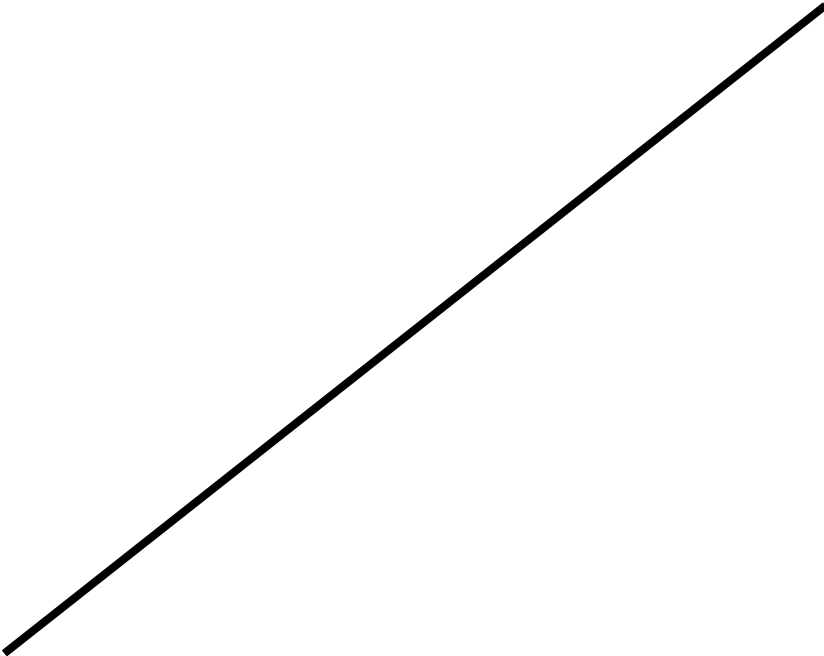
PIPELINE: MORE DETAILS



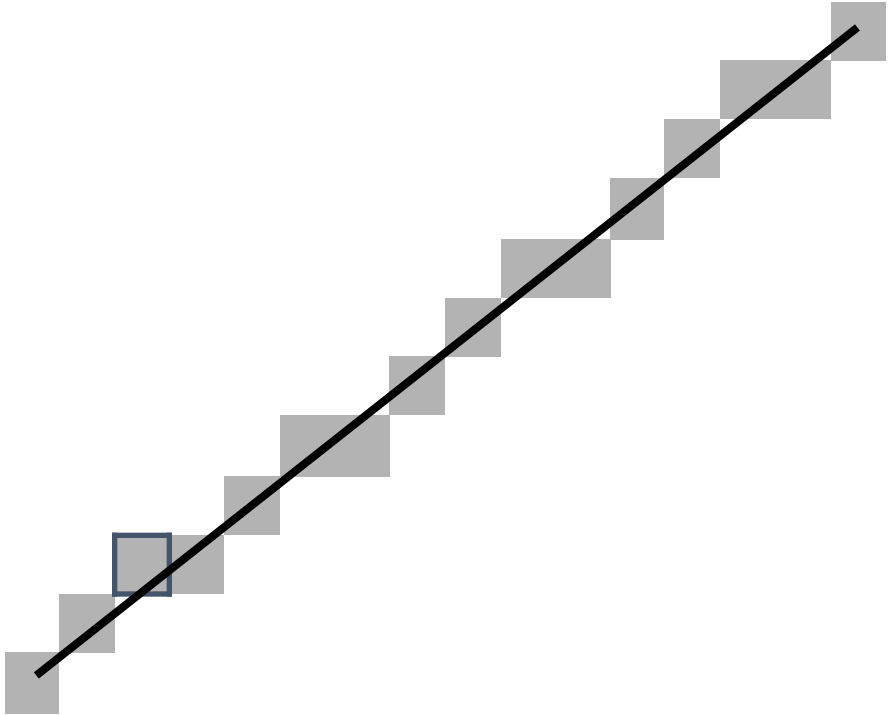
SCAN CONVERSION/RASTERIZATION

- Convert continuous 2D geometry to discrete
- Raster display – discrete grid of elements
- Terminology
 - **Screen Space:** Discrete 2D Cartesian coordinate system of the screen pixels

SCAN CONVERSION



SCAN CONVERSION

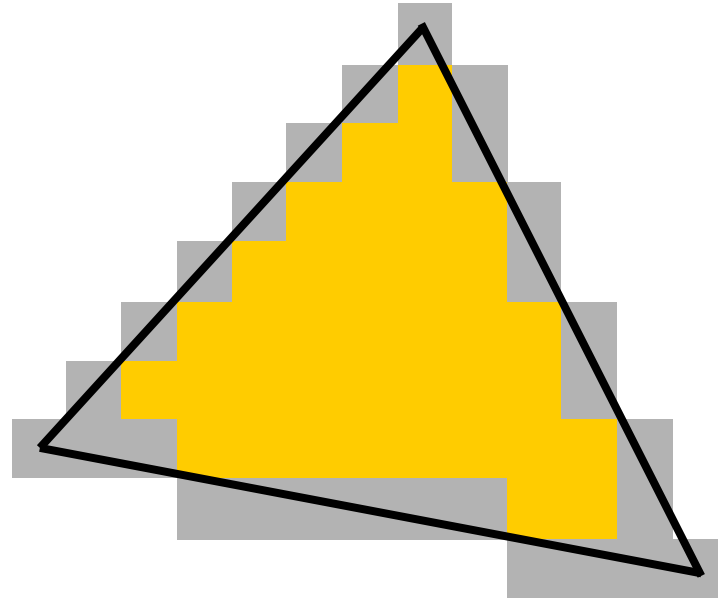


SCAN CONVERSION

- Problem:
 - Line is infinitely thin, but image has finite resolution
 - Results in steps rather than a smooth line
 - Jaggies
 - Aliasing
 - One of the fundamental problems in computer graphics

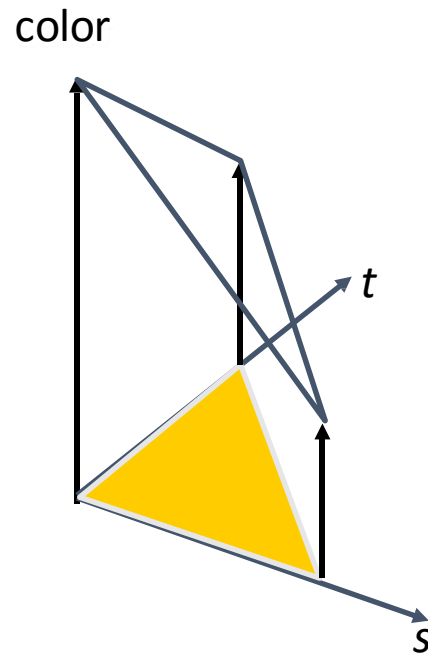
SCAN CONVERSION

-



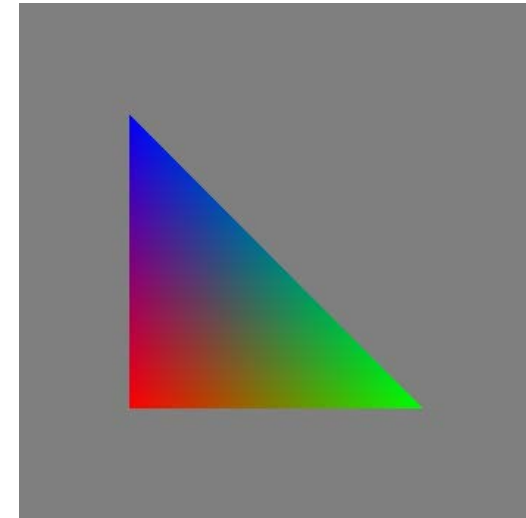
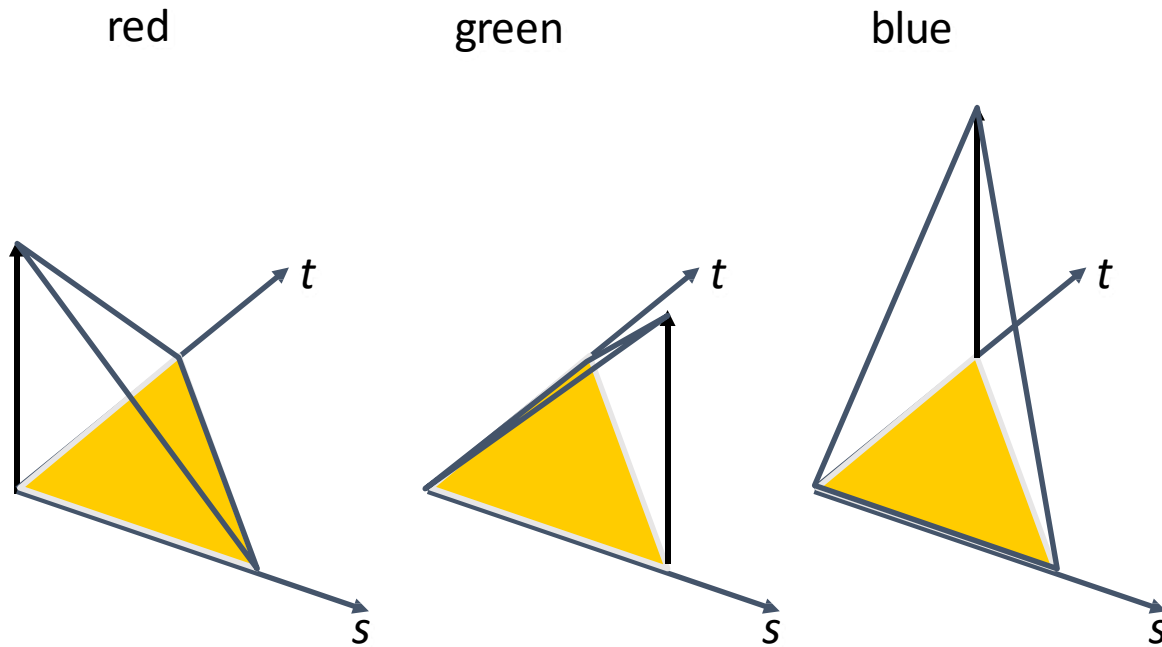
COLOR INTERPOLATION

Linearly interpolate per-pixel color from vertex color values
Treat every channel of RGB color separately

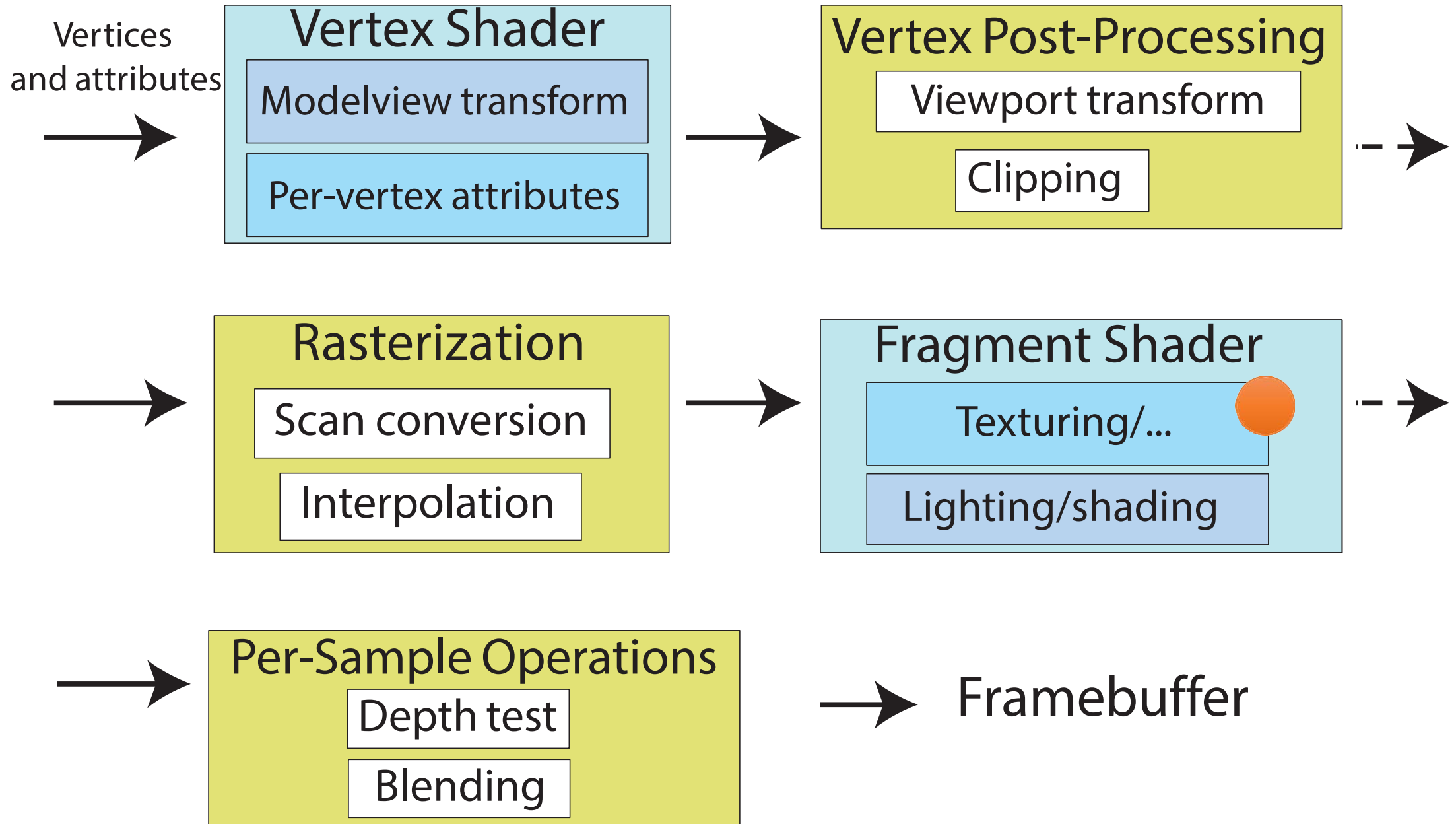


COLOR INTERPOLATION

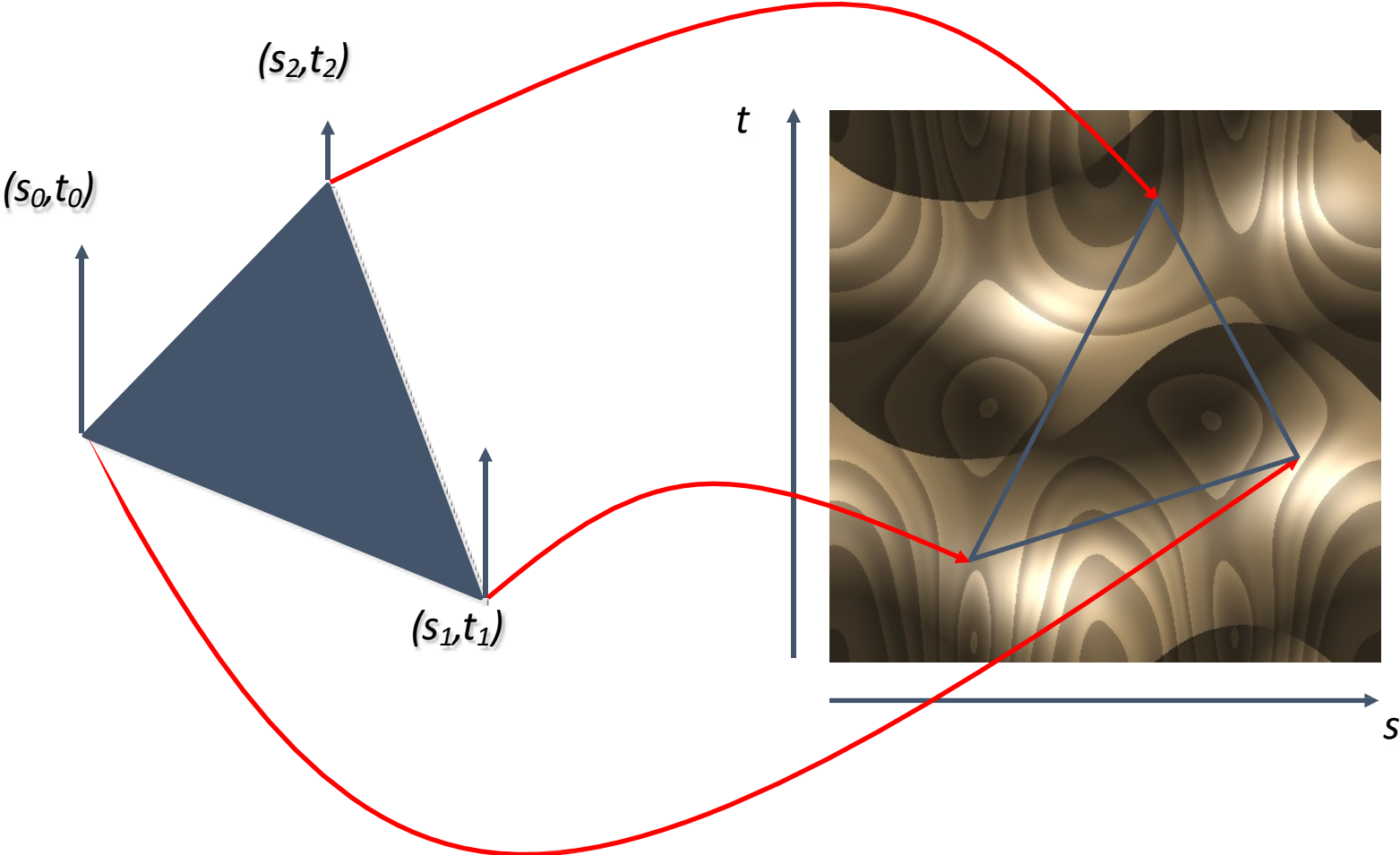
- Example:



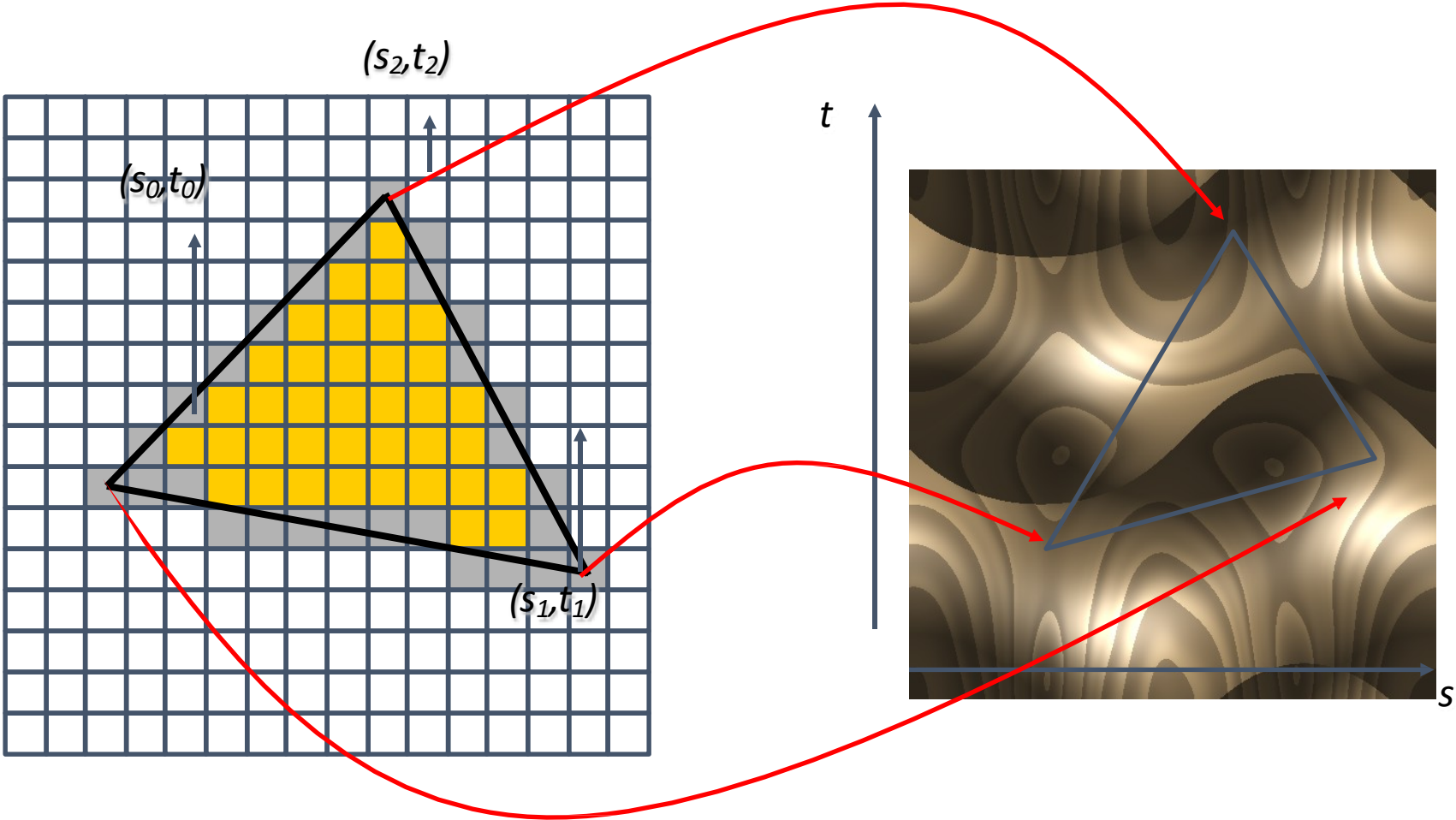
PIPELINE: MORE DETAILS



TEXTURING



TEXTURING



TEXTURE MAPPING



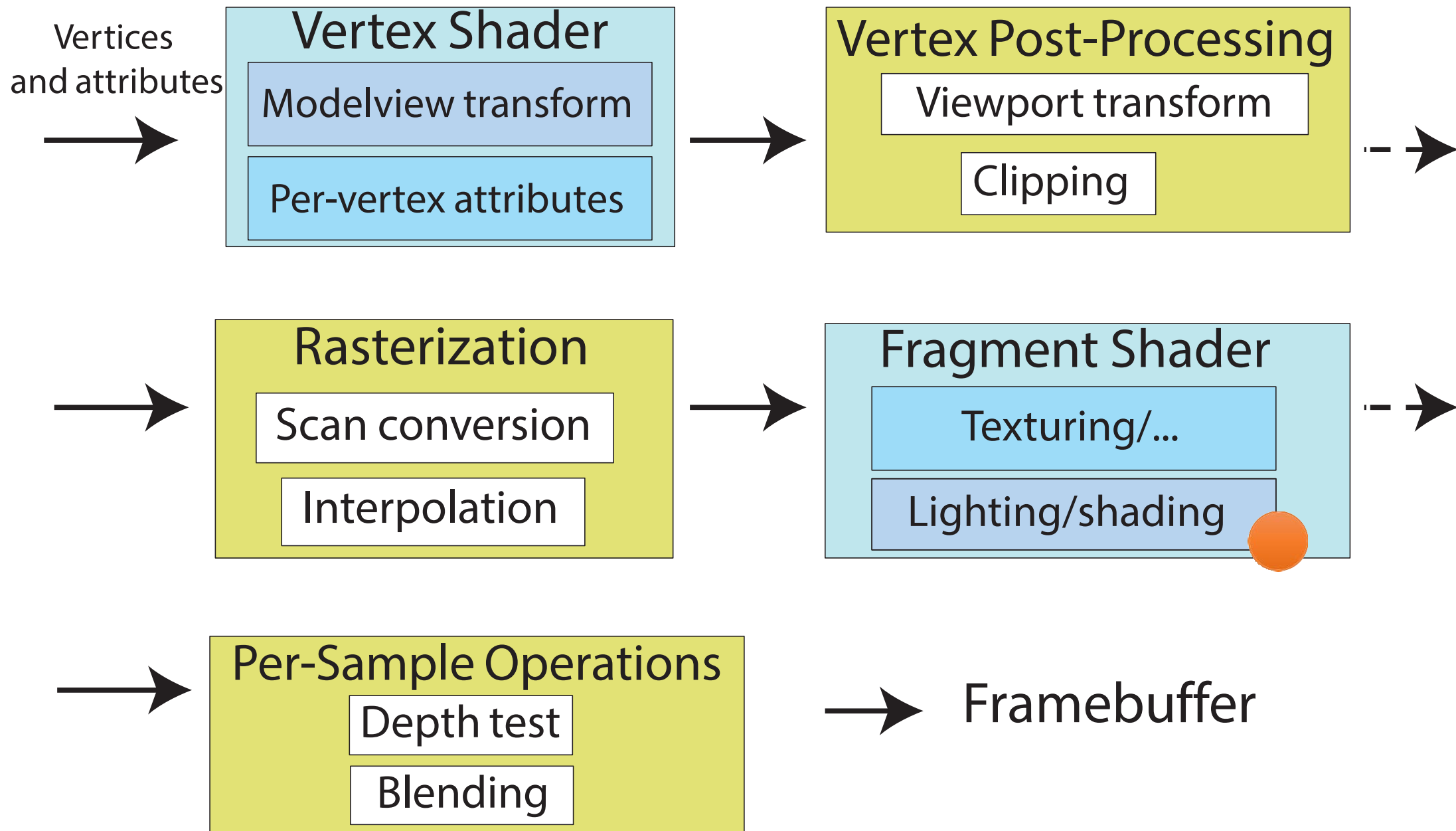
DISPLACEMENT MAPPING



TEXTURING

- Issues:
 - Computing 3D/2D map (low distortion)
 - How to map pixel from texture (texels) to screen pixels
 - Texture can appear widely distorted in rendering
 - Magnification / minification of textures
 - Filtering of textures
 - Preventing aliasing (anti-aliasing)

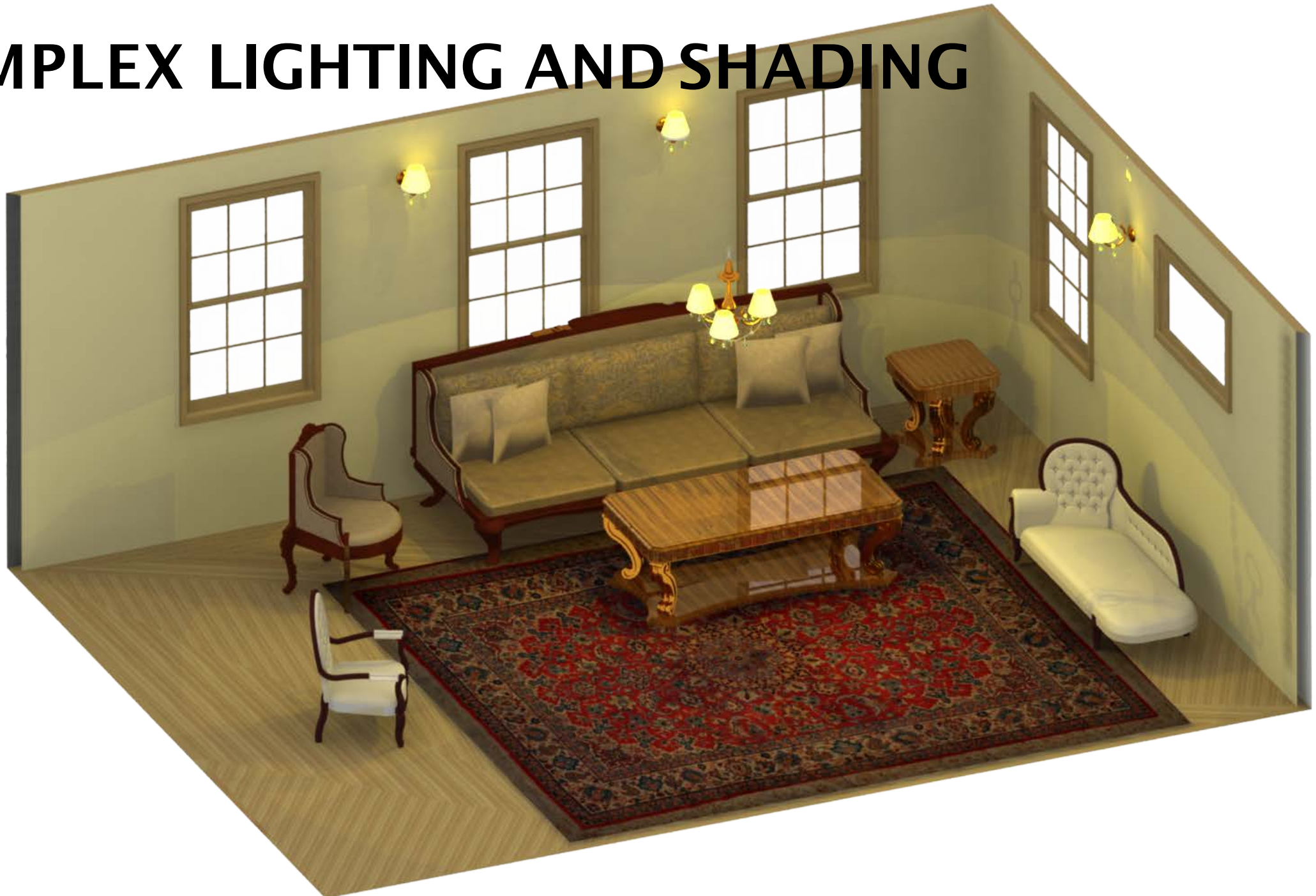
PIPELINE: MORE DETAILS



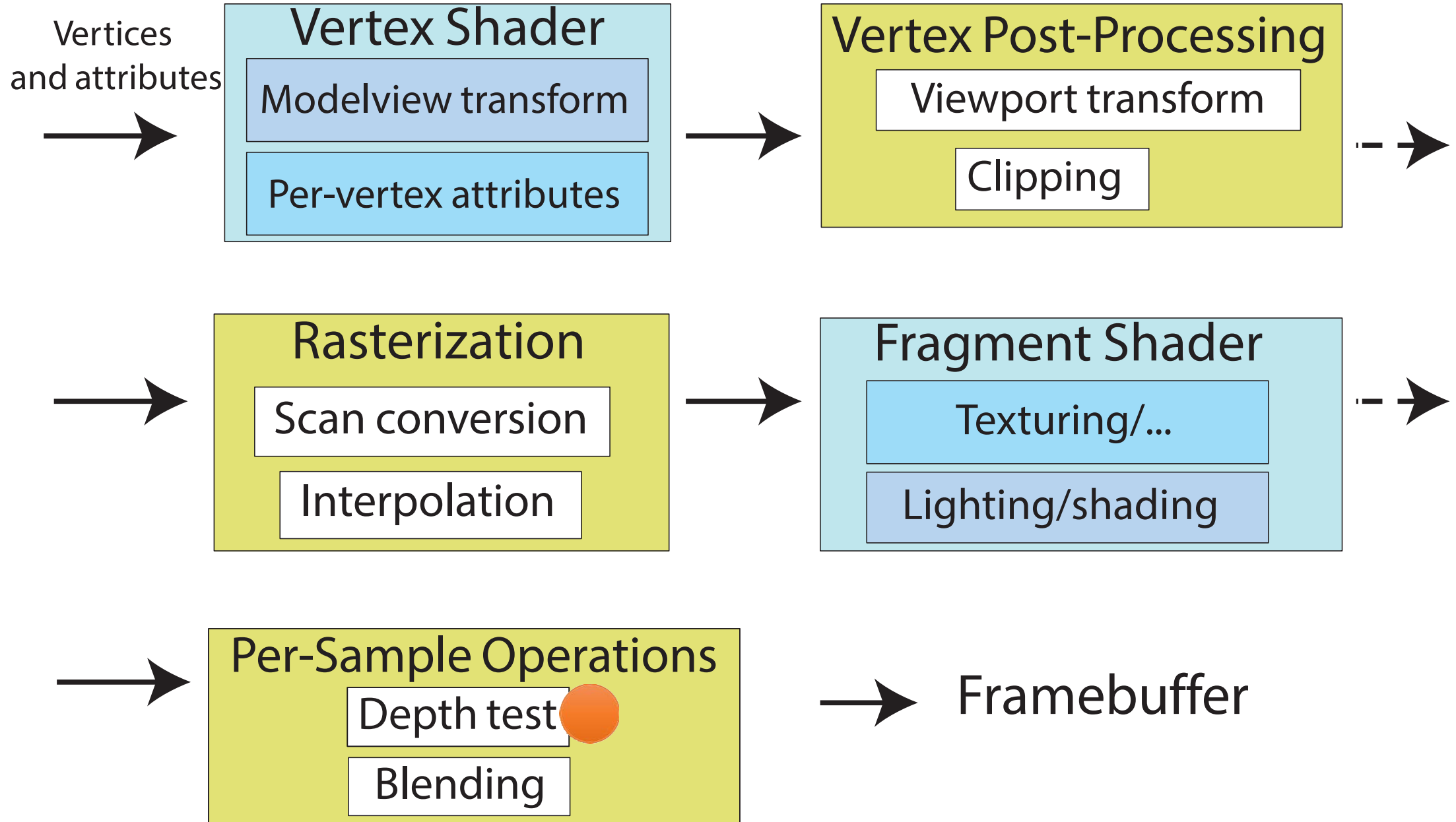
LIGHTING



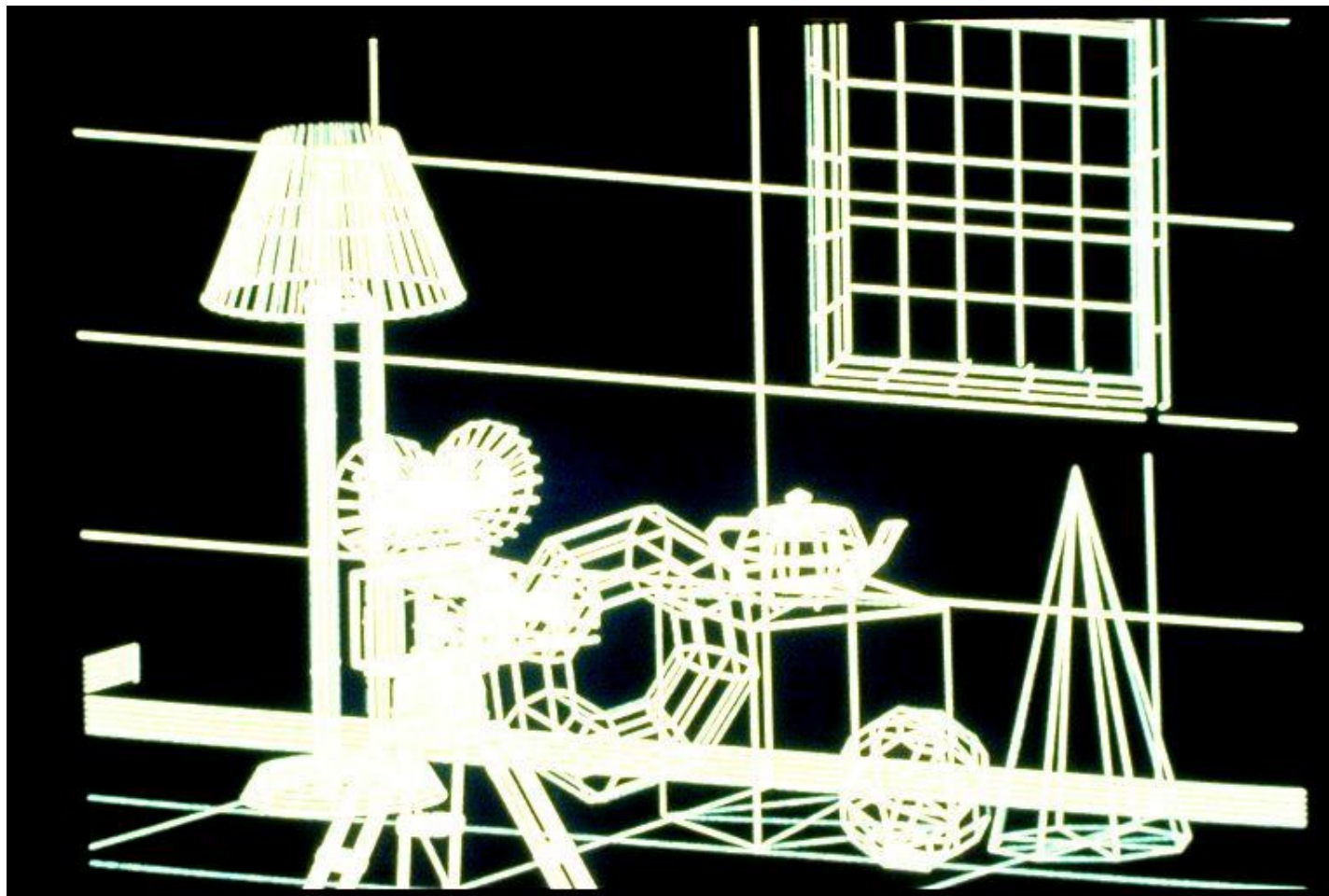
COMPLEX LIGHTING AND SHADING



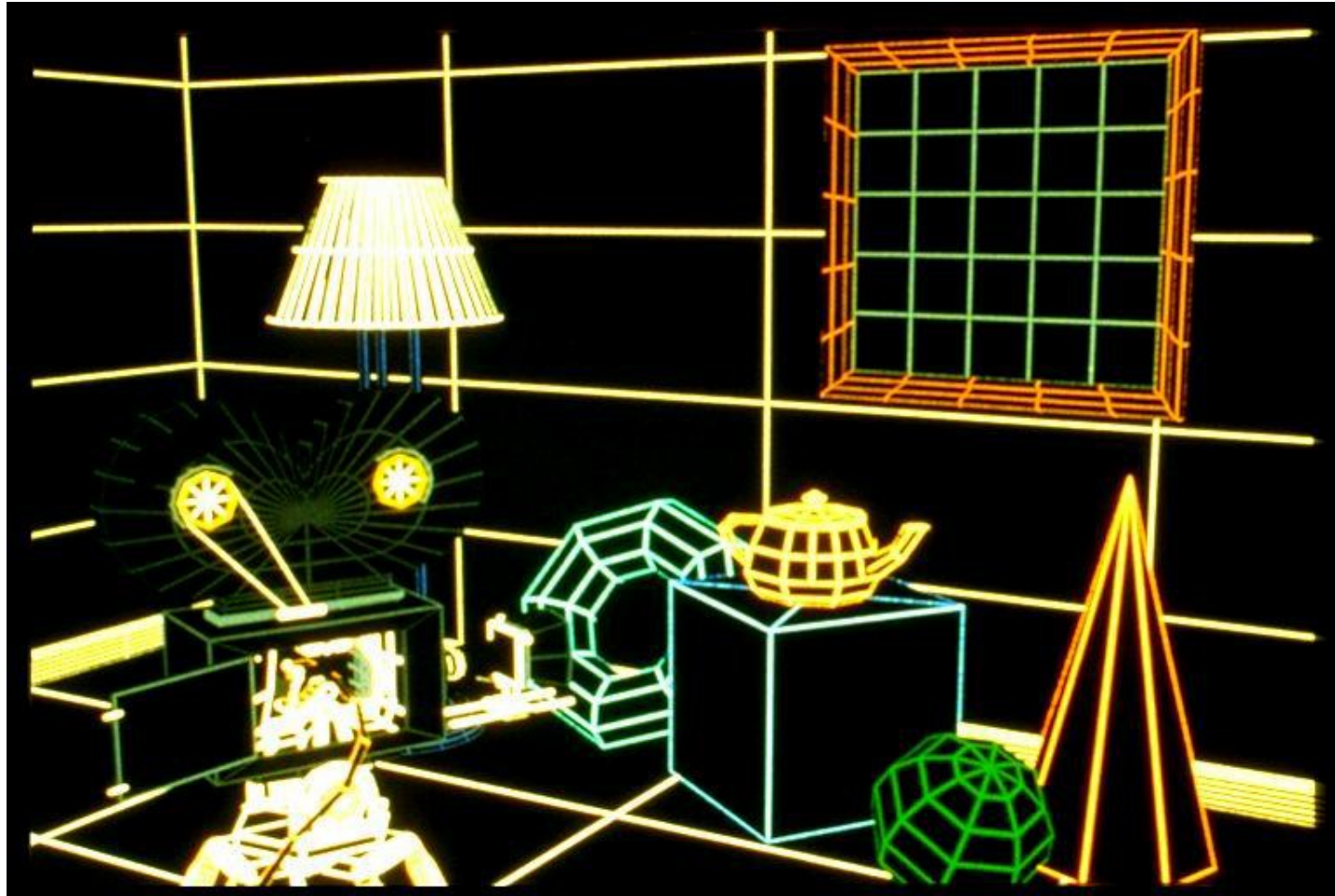
PIPELINE: MORE DETAILS



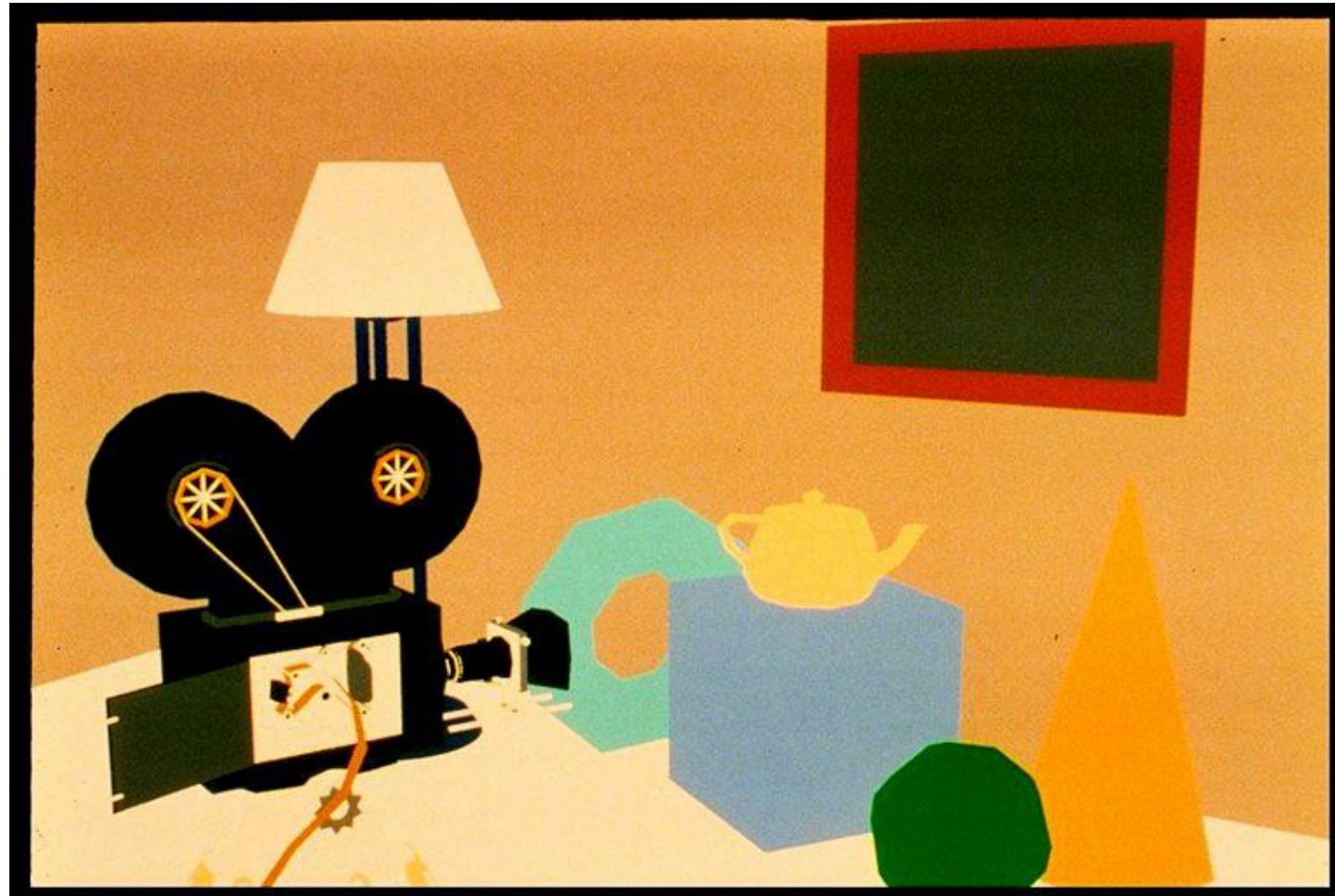
WITHOUT HIDDEN LINE REMOVAL



HIDDEN LINE REMOVAL



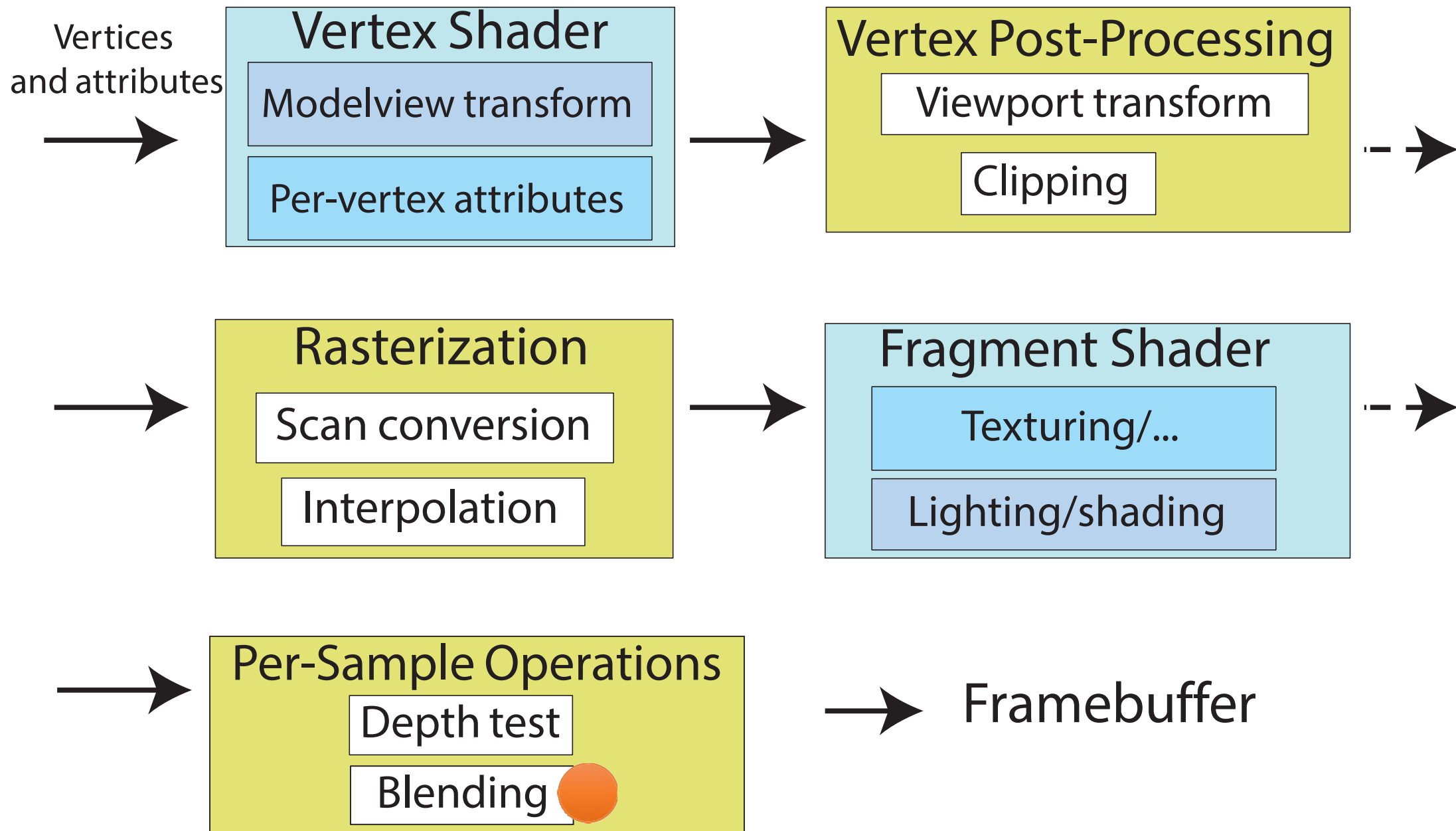
HIDDEN SURFACE REMOVAL



DEPTH TEST /HIDDEN SURFACE REMOVAL

- Remove invisible geometry
 - Parts that are hidden behind other geometry
- Possible Implementations:
 - Pixel level decision
 - Depth buffer
 - Object space decision
 - E.g. intersection order for ray tracing

PIPELINE: MORE DETAILS



BLENDING

- Blending:
 - Fragments -> Pixels
 - Draw from farthest to nearest
 - No blending – replace previous color
 - Blending: combine new & old values with some arithmetic operations
- Frame Buffer : video memory on graphics board that holds resulting image & used to display it

REFLECTION/SHADOWS

