Lecture 2:

The Graphics Pipeline

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

Announcements

- Assignment 1 due T 10:30 am
  - Submit via Canvas
- Assignment 2: Due T 7 Feb
- Before next week, find a computer/compiler/graphics card that you'll be using
- Compile and run "wesbench"
  - https://codeforge.lbl.gov/frs/?group_id=67 [get the instructional version NOT the full version]
  - Right now it uses GLUT. I would like to fix this.
  - Link will also be in Assignment 2, which I will post by next T
- Problems? Let's work them out sooner rather than later.

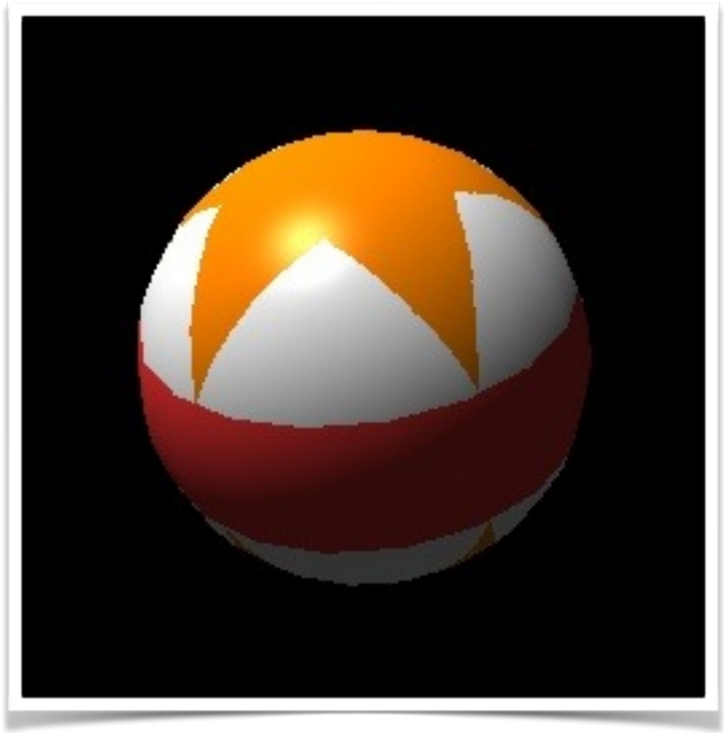
Purpose of graphics

```
Surface beachball(
      uniform float Ka = 1, Kd = 1;
      uniform float Ks = .5, roughness = .1;
      uniform color starcolor = color (1,.5,0);
      uniform color bandcolor = color (1,.2,.2);
      uniform float rmin = .15, rmax = .4;
      uniform float npoints = 5;) {
    color Ct; float angle, r, a, in_out; vector d1;

    uniform float starangle = 2*PI/npoints;
    uniform point p0 = rmax*point(cos(0),sin(0),0);
    uniform point p1 = rmin*point(cos(starangle/2),
                                  sin(starangle/2),0);
    uniform vector d0 = p1 - p0;
    angle = 2*PI * s;    r = .5-abs(t-.5);
    a = mod(angle, starangle)/starangle;

    if (a >= 0.5) a = 1 - a;
    d1 = r*(cos(a), sin(a),0) - p0;
    in_out = step(0, zcomp(d0^d1));
    Ct = mix(mix(Cs, starcolor, in_out), bandcolor, step(rmax,r));

    normal Nf = normalize(faceforward(N,I));
    Oi = Os;
    Ci = Os * (Ct * (Ka * ambient() + Kd * diffuse(Nf)) +
               Ks * specular(Nf,-normalize(I),roughness));
}
```



[Courtesy of Marc Olano]

Outline

- API Decisions
- Motivating OpenGL
- The Graphics Pipeline

Decision: How to specify scenes

Declarative (What, not How)

Describe the scene

For example: virtual camera

- RenderMan scene description
- Inventor and Performer scene graphs
- PHIGS

Imperative (How, not What)

Emit a sequence of drawing commands

For example: load model-view matrix

- OpenGL
- PostScript and Xlib

Decision: Rendering basis

Light transport

(raytracing)

[rendered with POVRay by "Resch"]

Decision: Rendering basis

Images

(image based rendering)

[rendered by David McAllister / University of North Carolina]

Decision: Rendering basis

Geometry

Why well-suited for real-time?

[cover of the OpenGL Programming Guide]

Decision: Primitives

0D Points

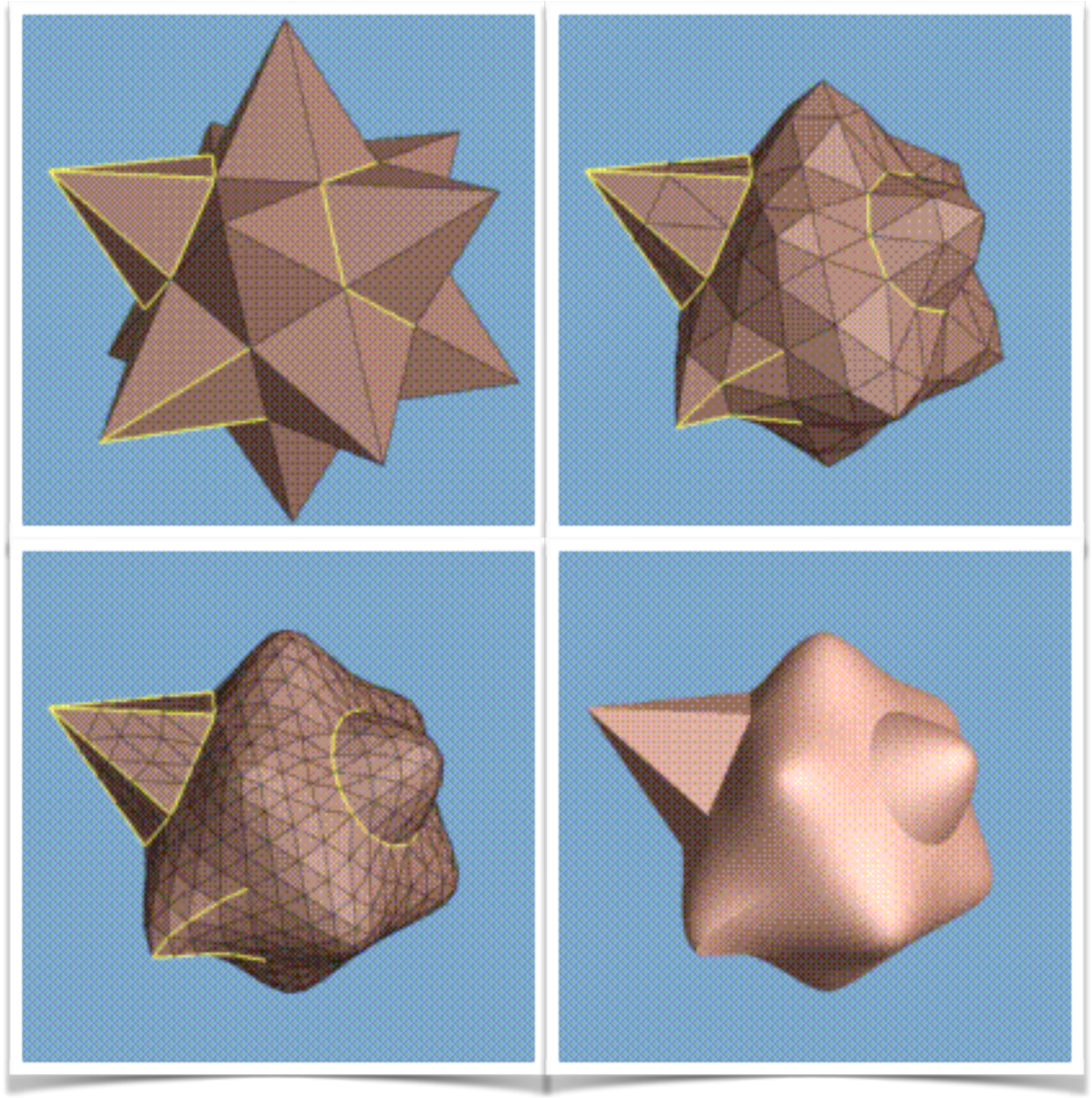
[Surfel-based renderer, courtesy of Hanspeter Pfister/ MERL]

[QSplat, courtesy of Szymon Rusinkiewicz]

Decision: Primitives

2D Geometry

- Triangles
- Quads
- Higher-order surfaces
- Images



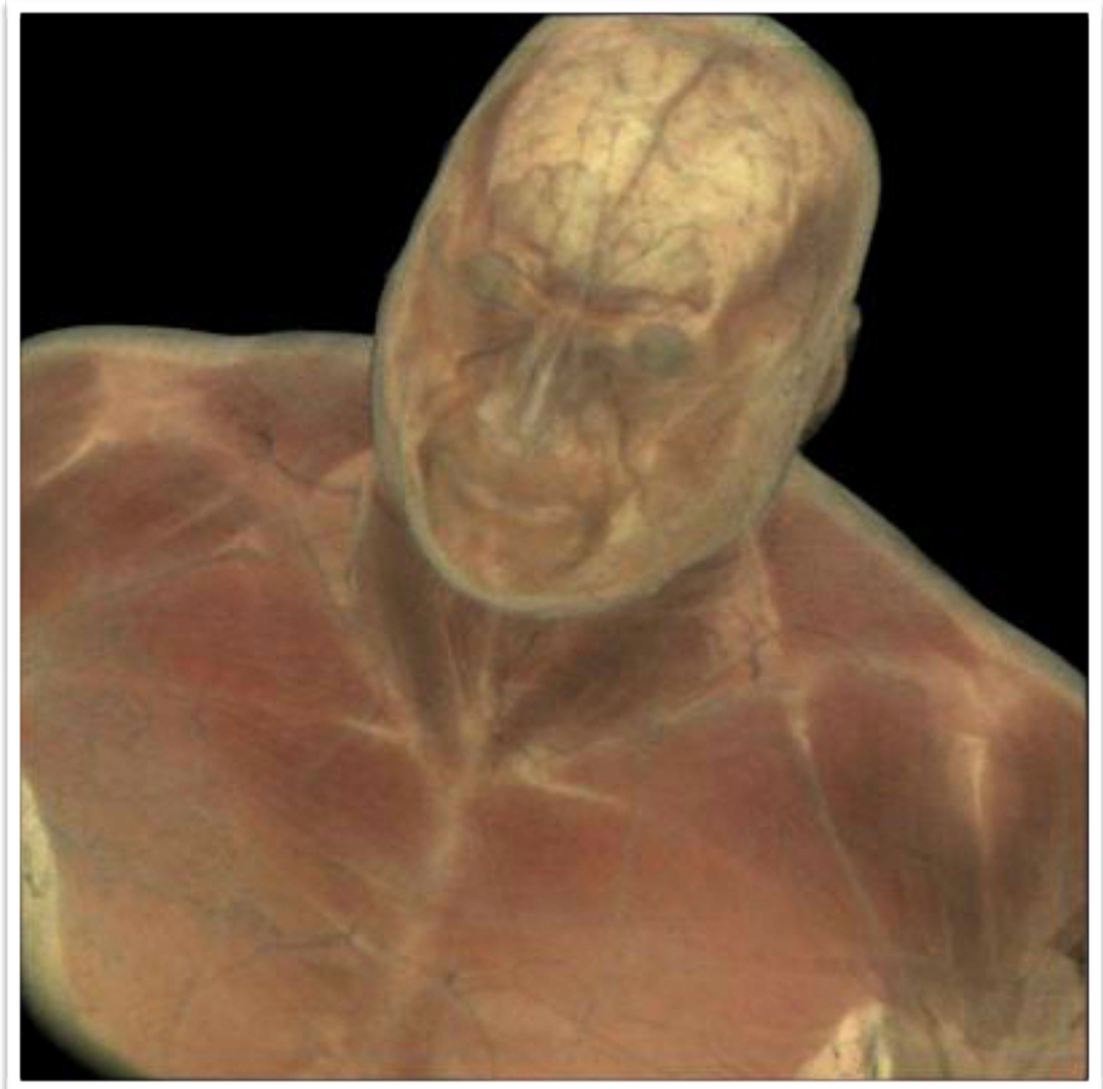
[Subdivision surfaces courtesy of UW GRAIL]

Decision: Primitives

3D Volume rendering

"… in 10 years, all rendering will be volume rendering."

—Jim Kajiya at SIGGRAPH '91



[Visible Human rendered at the San Diego Supercomputer Center]

Rendering Spectrum

Appearance Based — *Physically Based*

Images — *Geometric models*

Light field

Texture mapping

Image Morphing

View-dependent textures

Geometric level-of-detail

Sprites

Radiosity

Layered depth images

Monte Carlo ray tracing

[Jed Lengyel, "The Convergence of Graphics and Vision", IEEE Computer, July 1998]

Decision: Immediate vs. retained mode

Immediate mode—when program evaluates graphics command, command is sent to graphics system

Retained mode—graphics commands saved on server, bundled up

- Interpreted vs. compiled?

OpenGL, historically—immediate, but has display lists

OpenGL, today—essentially retained mode (vertex buffers/ arrays)

Direct3D—both modes

Decision: Ordering

Should graphics commands execute in the order they are specified?

[Courtesy of AMD "Mecha" demo]

Decision: Framebuffer

Framebuffers have storage available for each pixel.

Typically store color and depth information per pixel.

Single/double buffering.

But not required—beam-chasing or sorting algorithms do not require them!

Decision: Framebuffer

"A Characterization of Ten Hidden-Surface Algorithms", Ivan Sutherland, Robert Sproul, and Robert Schumacker (ACM Computing Surveys, March 1974)

hardware Thus the only variation of interest here is Newell et al, an order of magnitude less "costly" and the brute-force approach which is already ridiculously expensive.

Decision: Graphics State

- Minimizes data transmission
- Resources (shared or global, persistent)
  - Fonts
  - Texture
  - Display lists
- Attributes
  - Appearance: Lights, Materials, Colors, . . .
  - Transformation: camera, model, texture, . . .
  - Options: fb formats, constant per-frame

Decision: Graphics State

Ideally small and bounded

- e.g. maximum number of lights
- OpenGL: ~12kb

Distributed throughout the pipeline

- Difficult to manage (forces major design decisions)
- Must often be broadcast; must be consistent
- State changes not free!
- Multiple states active at once
- Difficult to query state

Hard to switch contexts

- OpenGL has a single context; X has multiple contexts
- One drawing process; windows are difficult
- Often pushed to window system

Decision: Invariance

- OpenGL does NOT precisely define drawing commands
  - For example:
    - Does not specify what pixels are inside a triangle
    - Does not specify the precision of intermediate calculations
  - Image drawn by two systems may differ
- Does require invariance across modes
  - Image drawn in two modes must be the “same”

Outline

API Decisions

Motivating OpenGL

The Graphics Pipeline

Motivating OpenGL

"The Design of the OpenGL Graphics Interface", Mark Segal and Kurt Akeley '94

http://www.opengl.org/developers/documentation/overviews.html

Segal/Akeley say graphics standards must:

- Be implementable on platforms w/ varying capabilities w/o compromising performance
- Natural interface for tersely describing rendering operations
- Gracefully accommodate extensions

Motivating OpenGL

"The Design of the OpenGL Graphics Interface", Mark Segal and Kurt Akeley '94

http://www.opengl.org/developers/documentation/overviews.html

Design considerations:

- Performance
- Orthogonality
- Completeness
- Interoperability
- Extensibility
- Acceptance

Motivating OpenGL

"The Design of the OpenGL Graphics Interface", Mark Segal and Kurt Akeley '94

http://www.opengl.org/developers/documentation/overviews.html

"Provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex object themselves."

Motivating OpenGL

"The Design of the OpenGL Graphics Interface", Mark Segal and Kurt Akeley '94

http://www.opengl.org/developers/documentation/overviews.html

OpenGL makes several design decisions:

- Immediate mode
- Ordered
- Has graphics state
- Triangle as primitive
- Framebuffer

Geometry

- Application generates list of geometry (triangles made of vertices).
- Command processor sequences them.
- Start with triangle/vertex geometry in *object* space.
- Operations on vertices (transformations, shading)
  - This is the *vertex program*
- Operations on triangles (clipping, culling)
- Project on to screen. Result is triangles in screen space.

Rasterization

Now geometry is in screen (image) space.

Object -> Image space transformation vital.

Now, map triangle to covered screen pixels.

"Rasterization"

Generates fragments.

Shade fragments with fixed-function calls or *fragment program*

But what about occlusion?

Composition

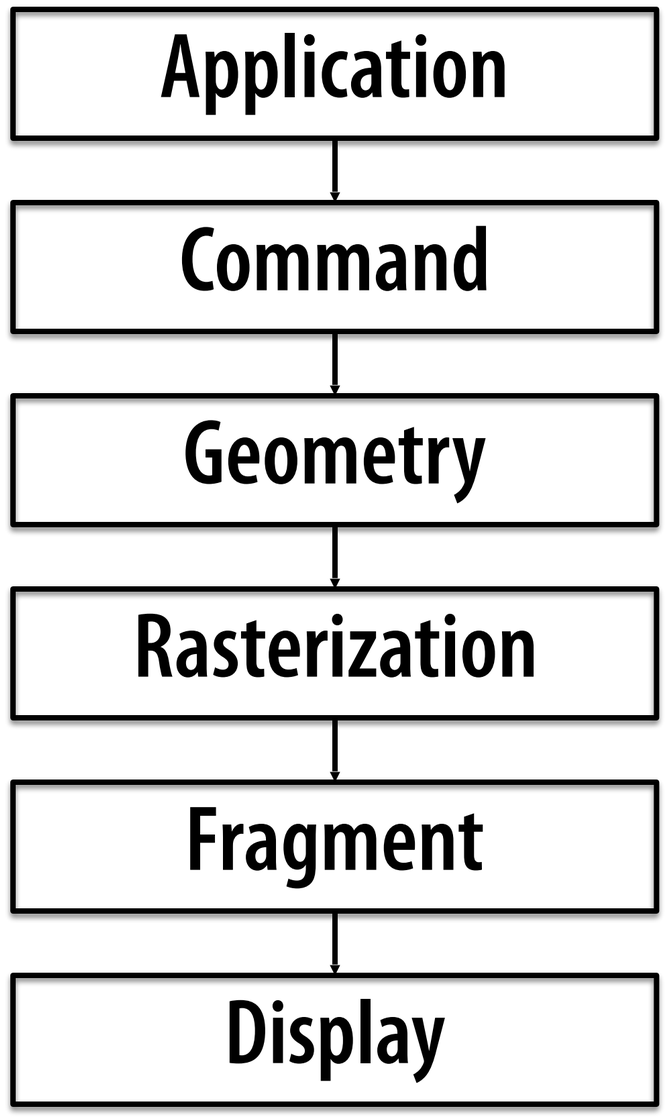
Begin with list of fragments.

Process each fragment.

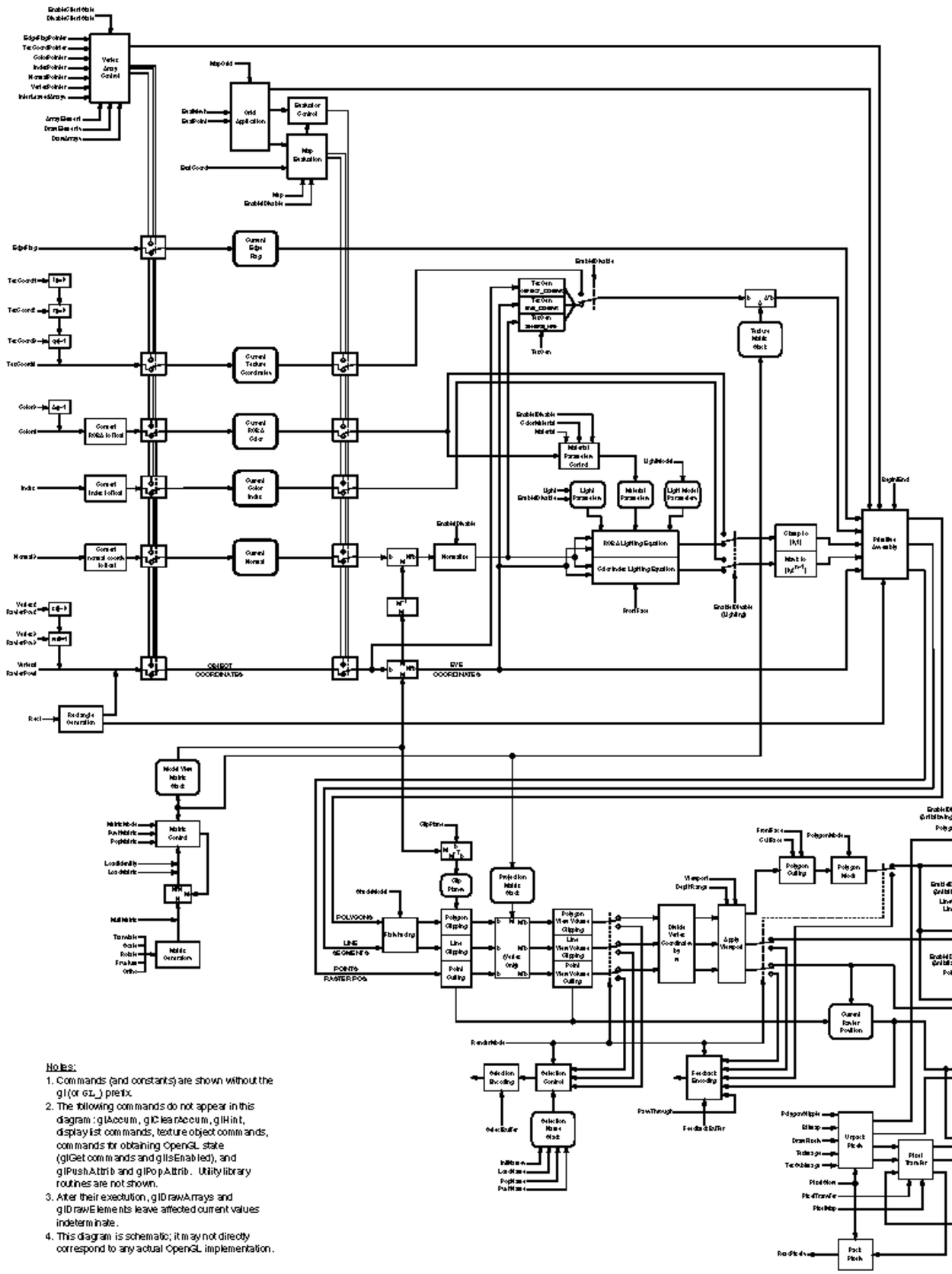
Assemble fragments into final image.

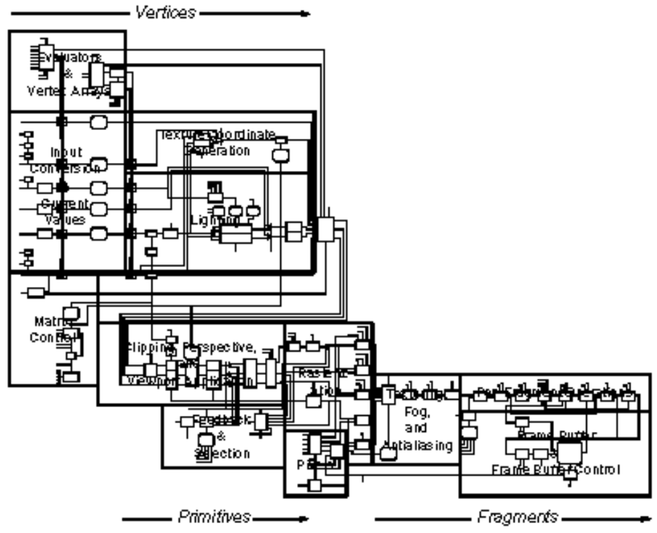
In OpenGL, store depth-per-pixel. Pick closest pixel to camera.

Display result.



The OpenGL® Machine





Key to OpenGL Operations

Notes:

1. Commands (and constants) are shown without the gl (or GL_) prefix.
2. The following commands do not appear in this diagram: glAccum, glClearAccum, glHint, display list commands, texture object commands, commands for obtaining OpenGL state (glGet commands and glIsEnabled), and glPushAttrib and glPopAttrib. Utility library routines are not shown.
3. After their execution, glDrawArrays and glDrawElements leave affected current values indeterminate.
4. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.

Outline

- API Decisions
- Motivating OpenGL
- The Graphics Pipeline (twice)!
- Computational Costs

Application

Simulation

Input event handlers

Modify data structures

Database traversal

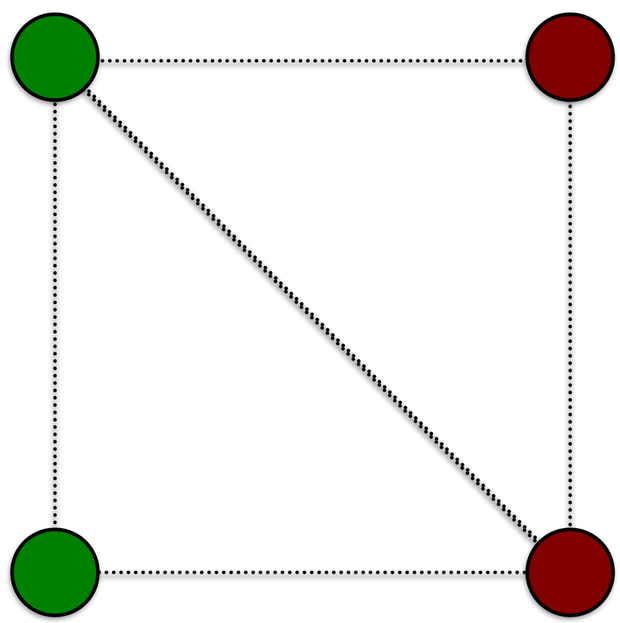
Primitive generation

Utility functions

Command

- Command buffering
- Command interpretation
- Unpack and perform format conversion
- Maintain graphics state

```
        glLoadIdentity( );
        glMultMatrix( T );
glBegin( GL_TRIANGLE_STRIP );
 glColor3f ( 0.0, 0.5, 0.0 );
 glVertex3f( 0.0, 0.0, 0.0 );
 glColor3f ( 0.5, 0.0, 0.0 );
 glVertex3f( 1.0, 0.0, 0.0 );
 glColor3f ( 0.0, 0.5, 0.0 );
 glVertex3f( 0.0, 1.0, 0.0 );
 glColor3f ( 0.5, 0.0, 0.0 );
 glVertex3f( 1.0, 1.0, 0.0 );
                …
          glEnd( );
```

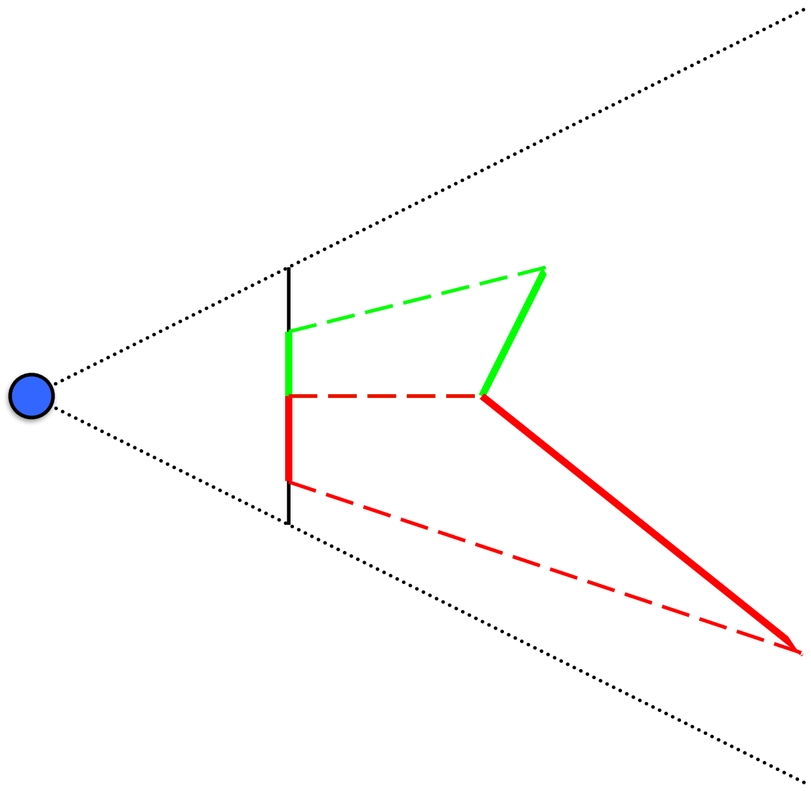


Geometry

Evaluation of polynomials for curved surfaces

Transform and projection

Clipping, culling and primitive assembly



Geometry

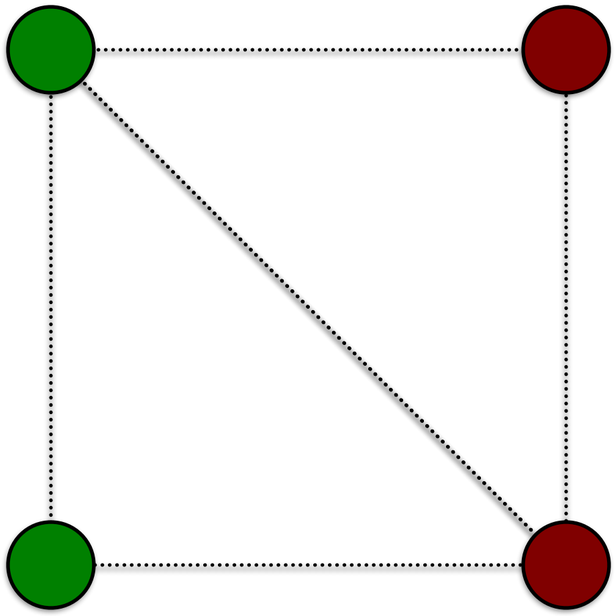
Evaluation of polynomials for curved surfaces

Transform and projection (object -> image space)

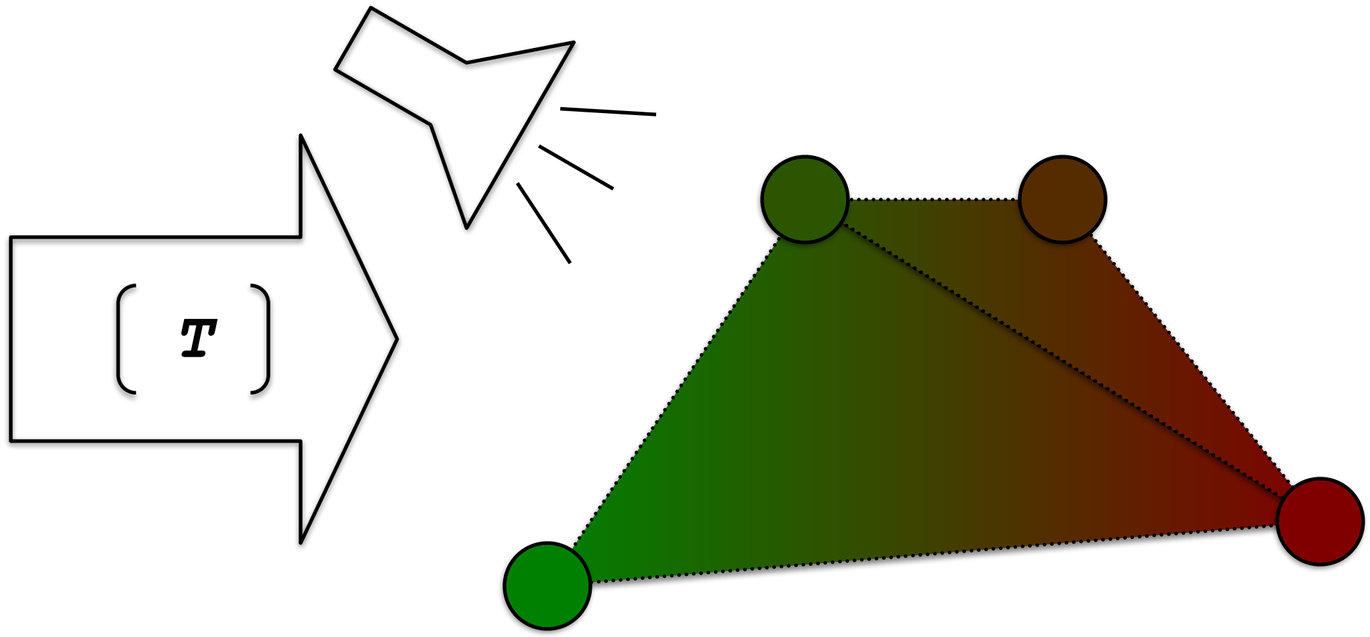
Clipping, culling and primitive assembly

Lighting

Texture coordinate generation



Object-space triangles



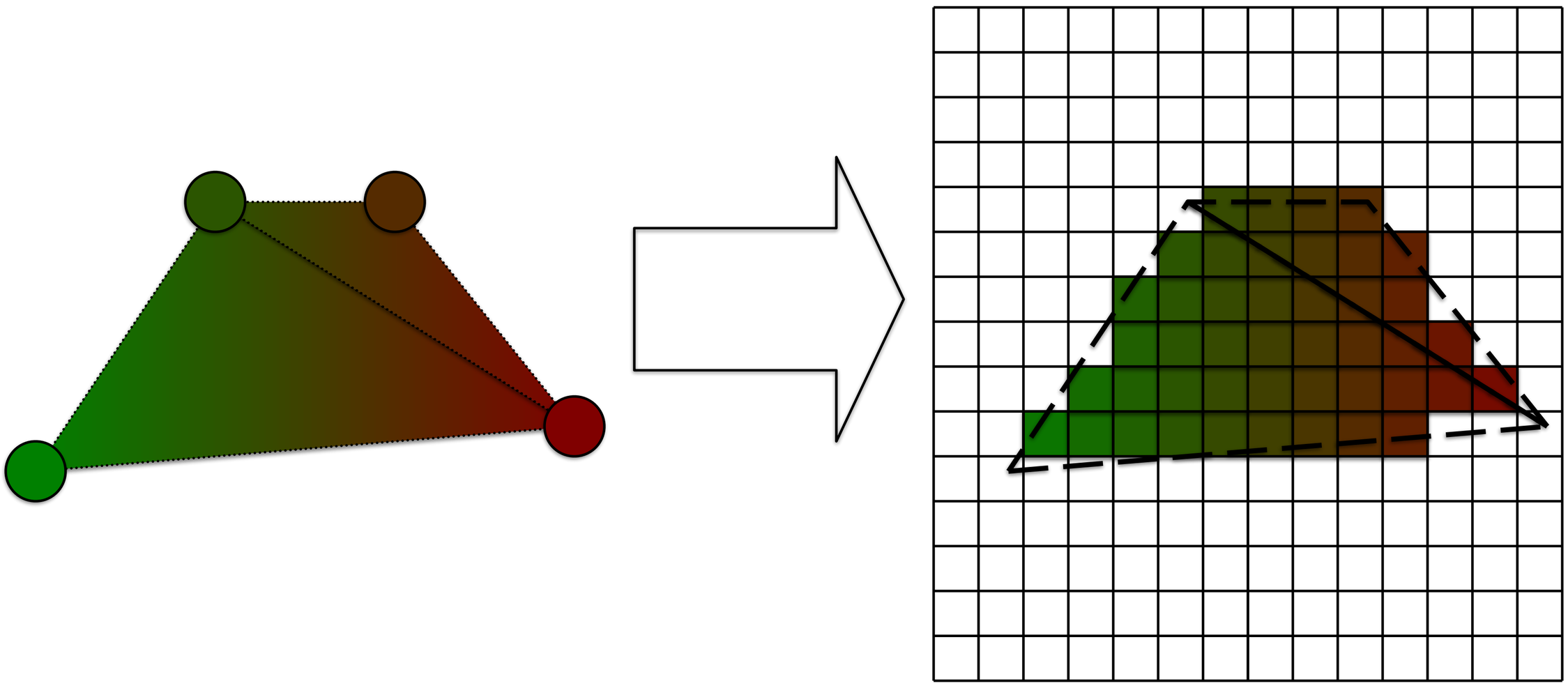
Screen-space lit triangles

Rasterization

Setup (per-triangle)

Sampling (triangle = {fragments})

Interpolation (interpolate colors and coordinates)



Screen-space triangles

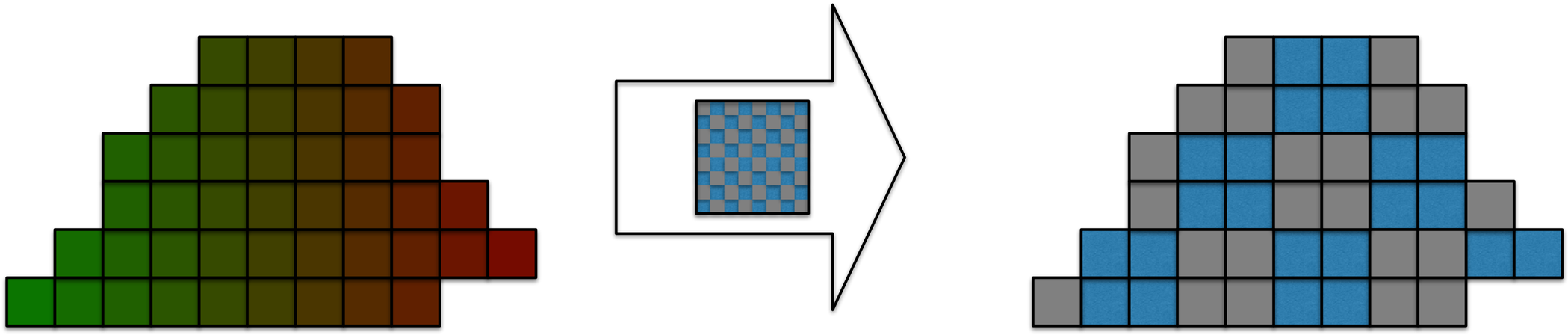
Fragments

Texture

Texture transformation and projection

Texture address calculation

Texture filtering



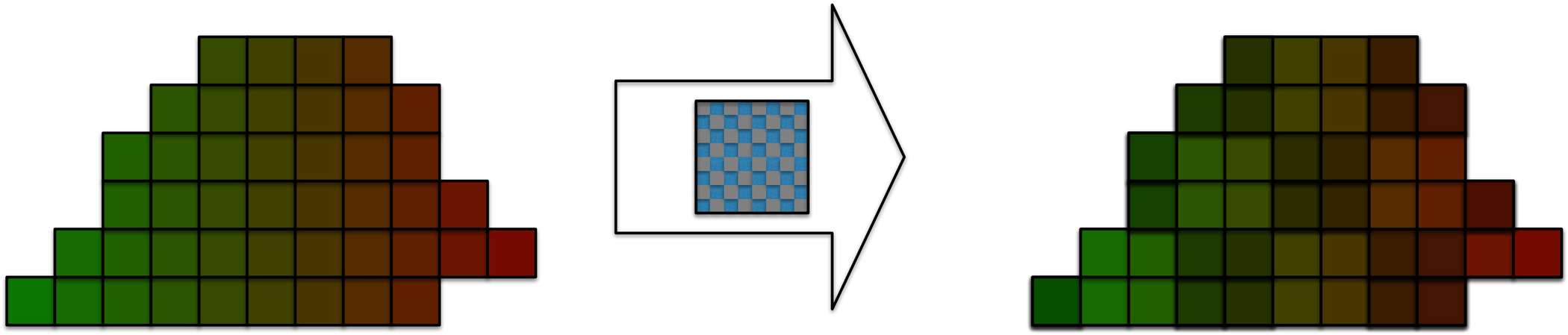
Fragments

Texture (to apply to fragments)

Fragment

Final fragment color = function(interpolated color, texture)

Generalize to arbitrary function at each fragment (fragment program)



Fragments

Textured Fragments

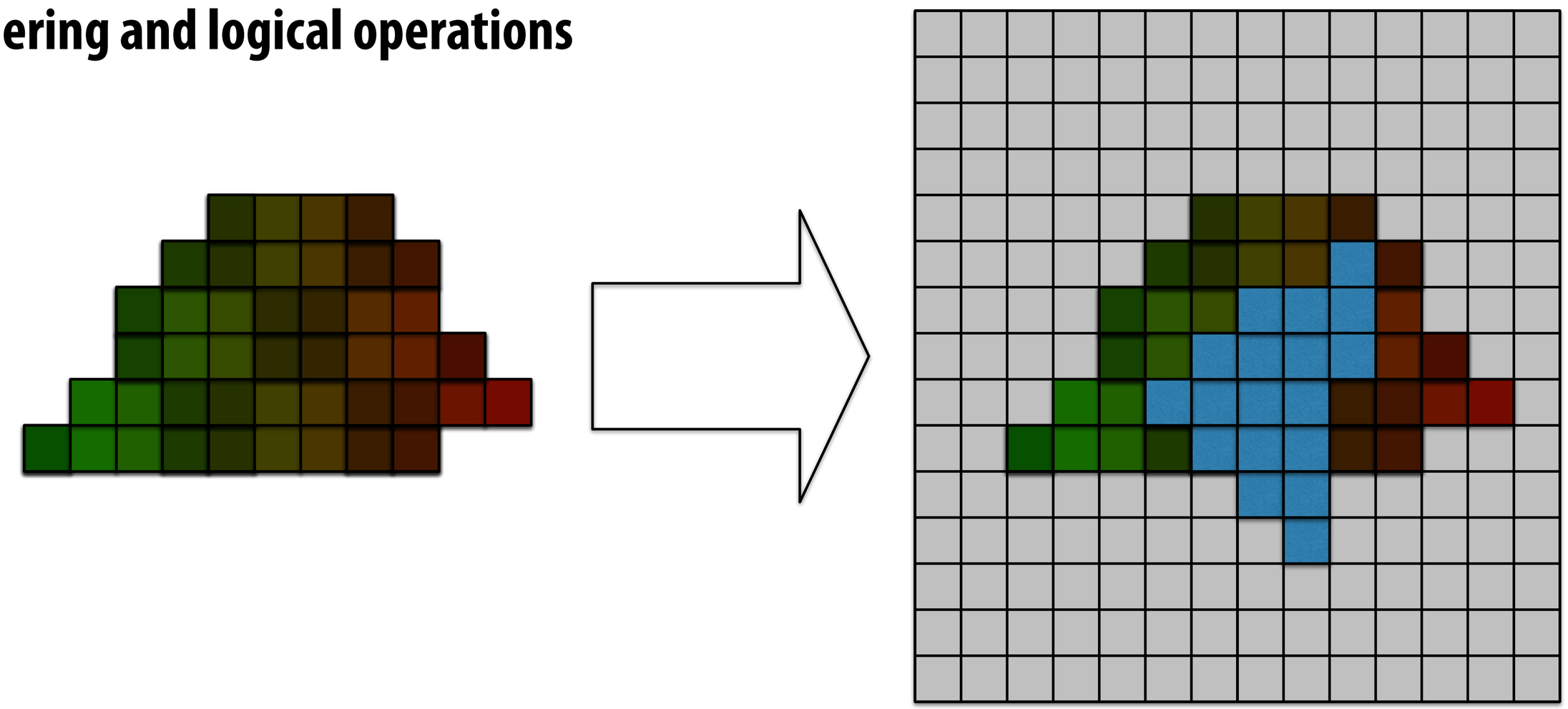
Fragment

Texture application and fog

Owner, scissor, depth, alpha and stencil tests

Blending or compositing

Dithering and logical operations



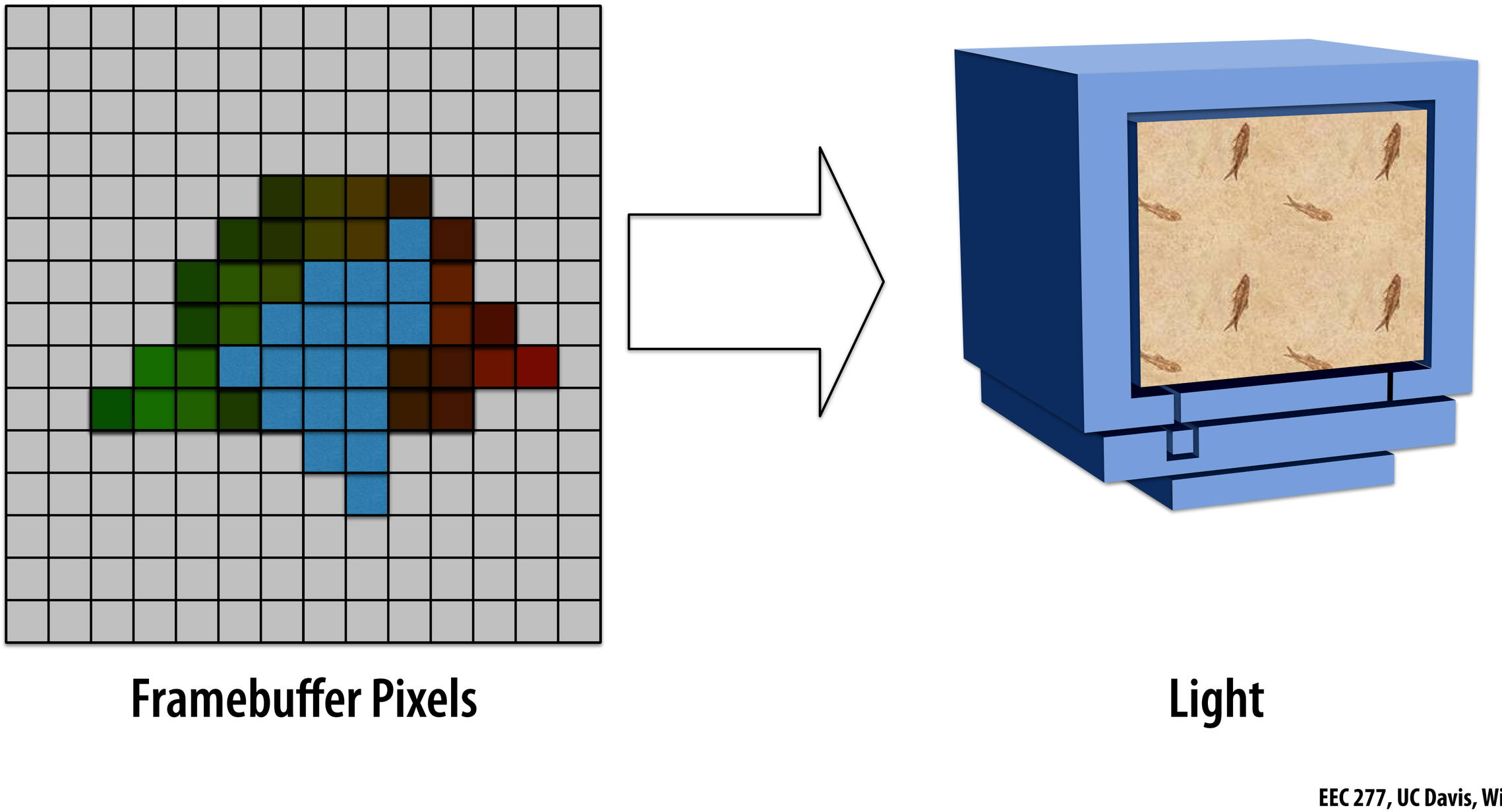
Textured Fragments

Framebuffer Pixels

Display

Gamma correction

Digital to analog conversion



Framebuffer Pixels

Light

That was a pass through the pipeline # 1

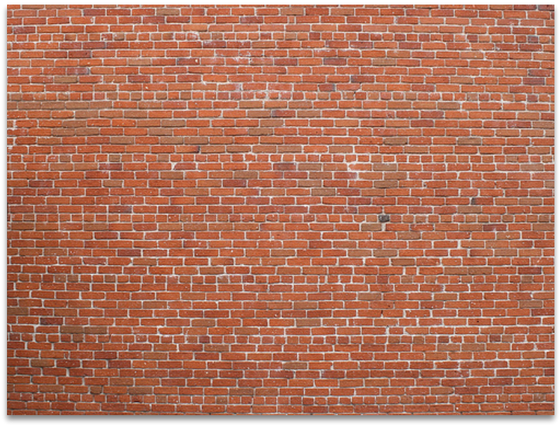
- Questions?

Command: draw these triangles!

Inputs:

```
list_of_positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x   };
```

```
list_of_texcoords = {
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v   };
```



Texture map

Object-to-camera-space transform: $\mathbf{T}$

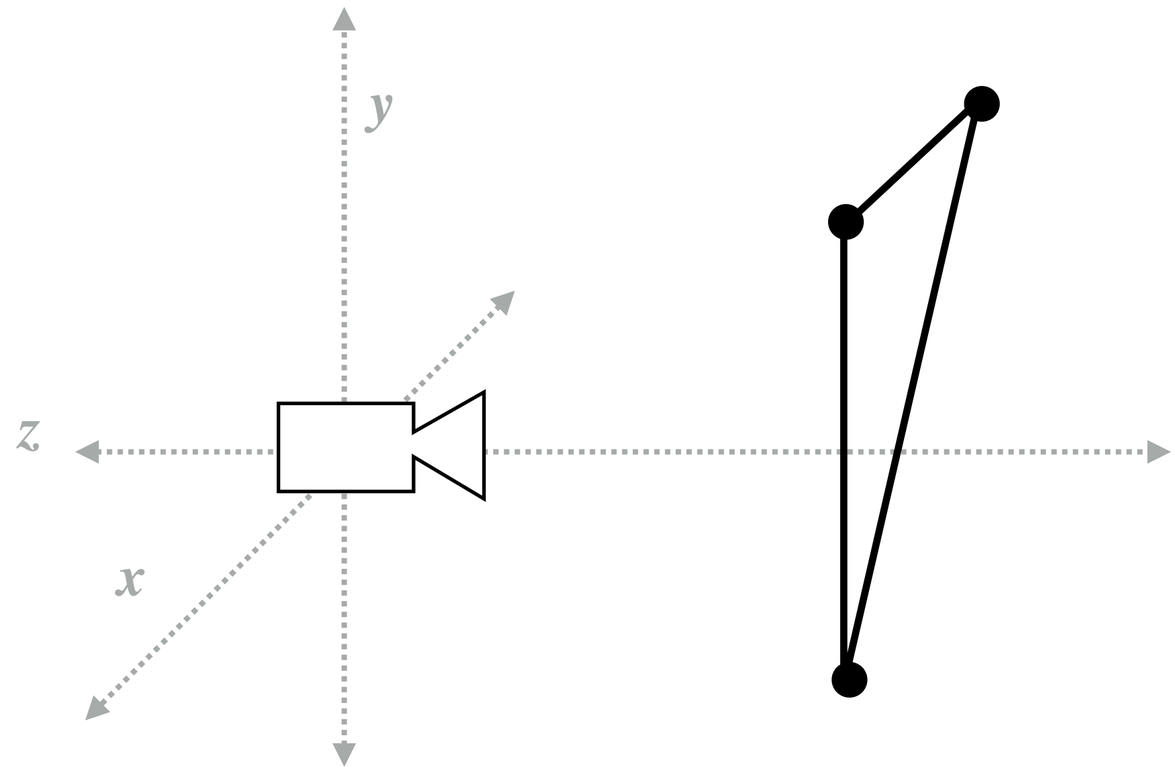
Perspective projection transform $\mathbf{P}$

Size of output image (W, H)

Use depth test /update depth buffer: YES!

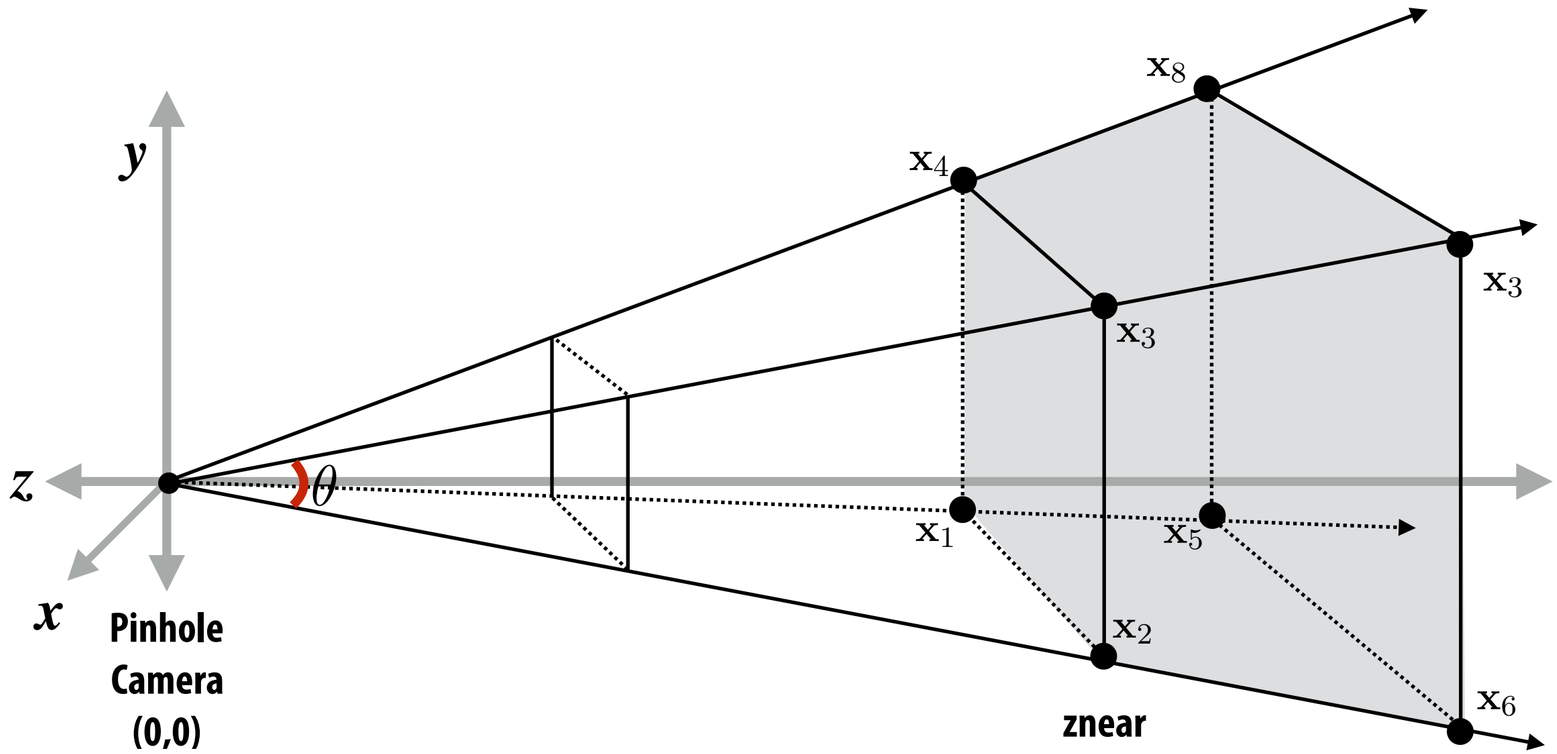
Step 1:

Transform triangle vertices into camera space

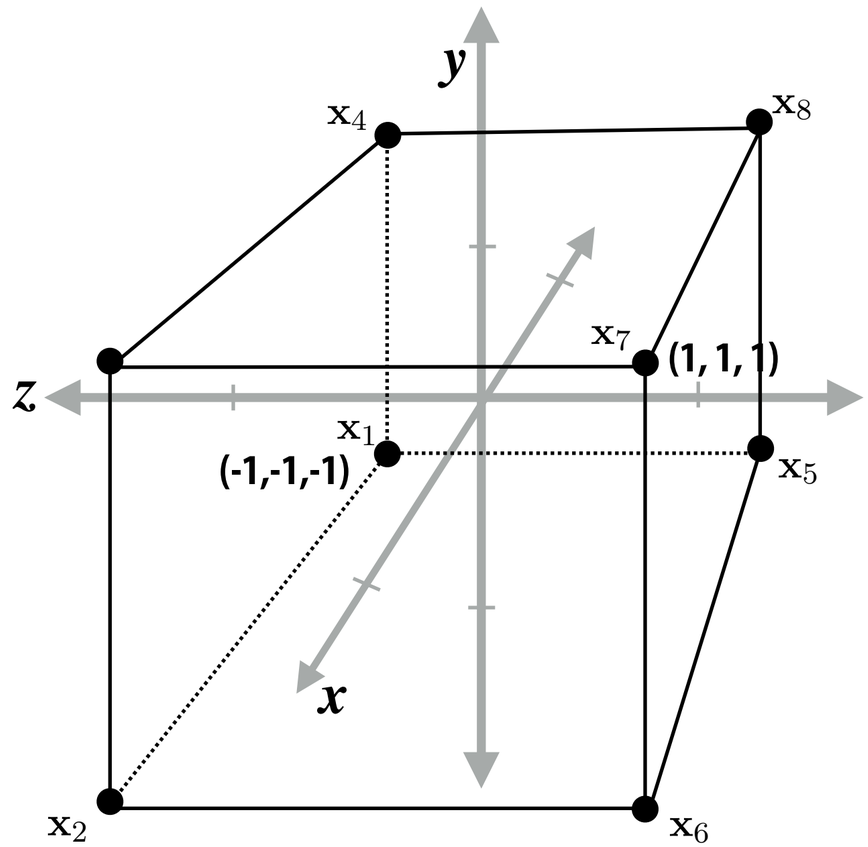


Step 2:

Apply perspective projection transform to transform triangle vertices into normalized coordinate space



Camera-space positions: 3D



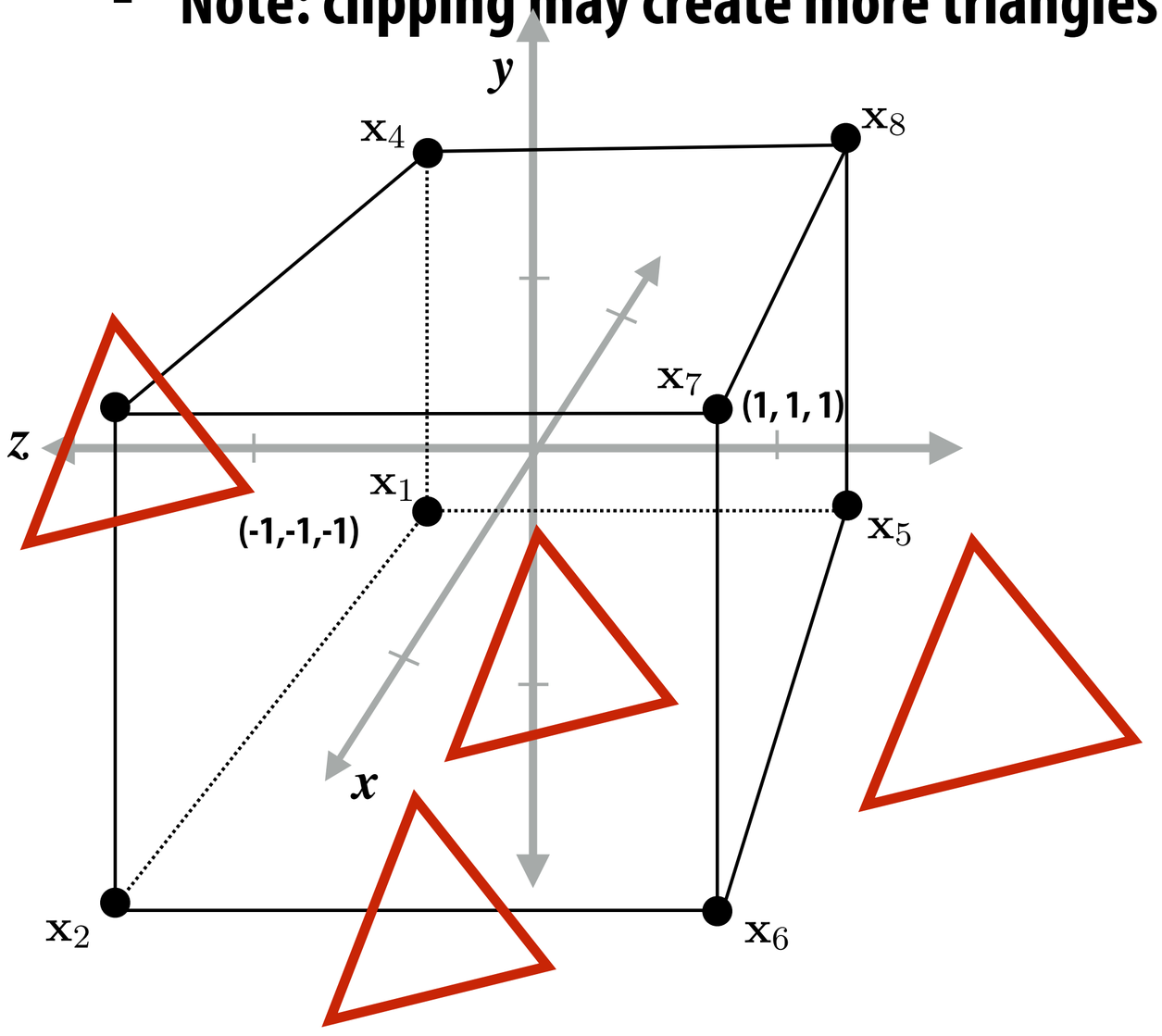
Normalized space positions

Note: I'm illustrating normalized 3D space after the homogeneous divide, it is more accurate to think of this volume in 3D-H space as defined by:

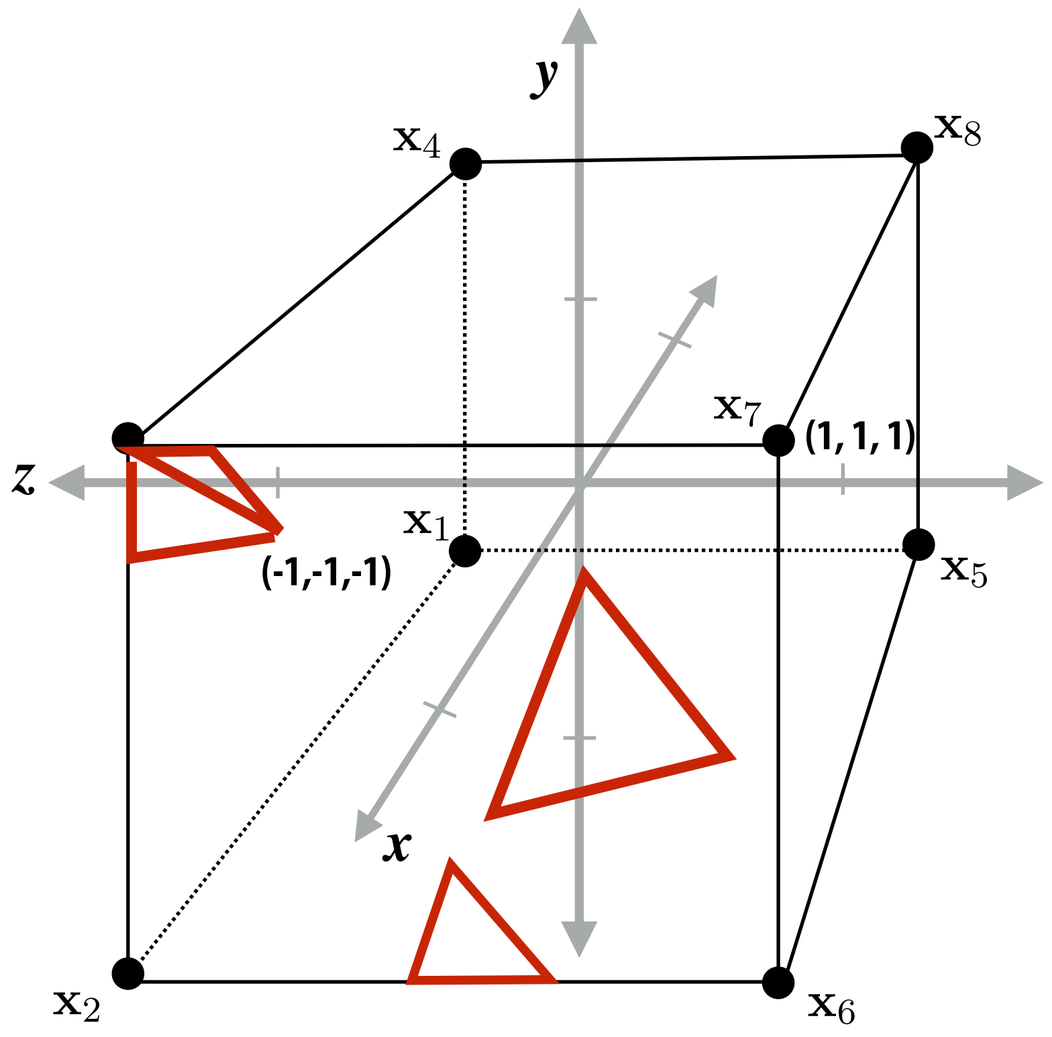
(-w, -w, -w, w) and (w, w, w, w)

Step 3: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - Note: clipping may create more triangles



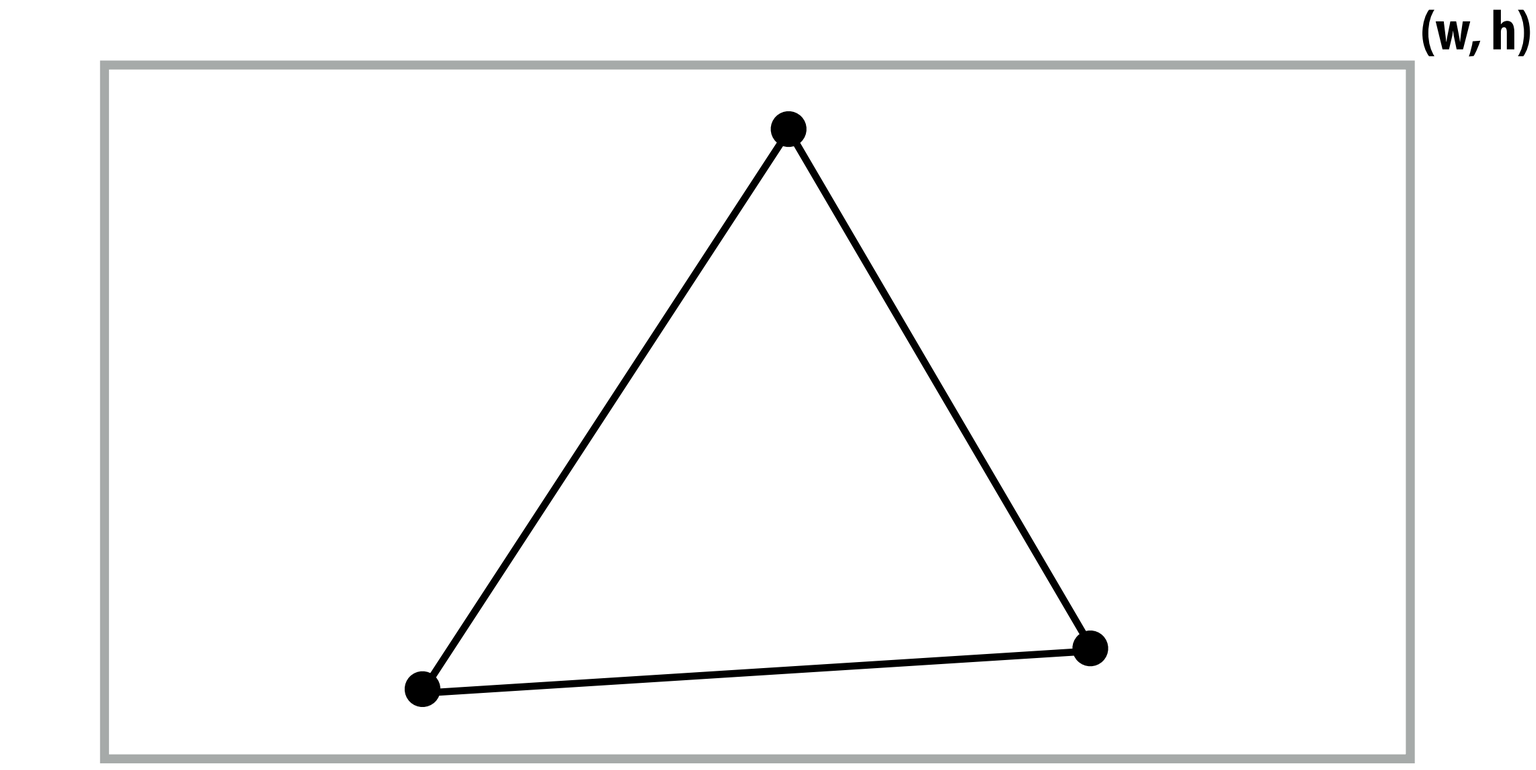
Triangles before clipping



Triangles after clipping

Step 4: transform to screen coordinates

Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



Step 5: setup triangle (triangle preprocessing)

Compute triangle edge equations

Compute triangle attribute equations

$$\mathbf{E}_{01}(x,y)$$

$$\mathbf{E}_{12}(x,y)$$

$$\mathbf{E}_{20}(x,y)$$

$$\frac{1}{\mathbf{w}}(x,y)$$

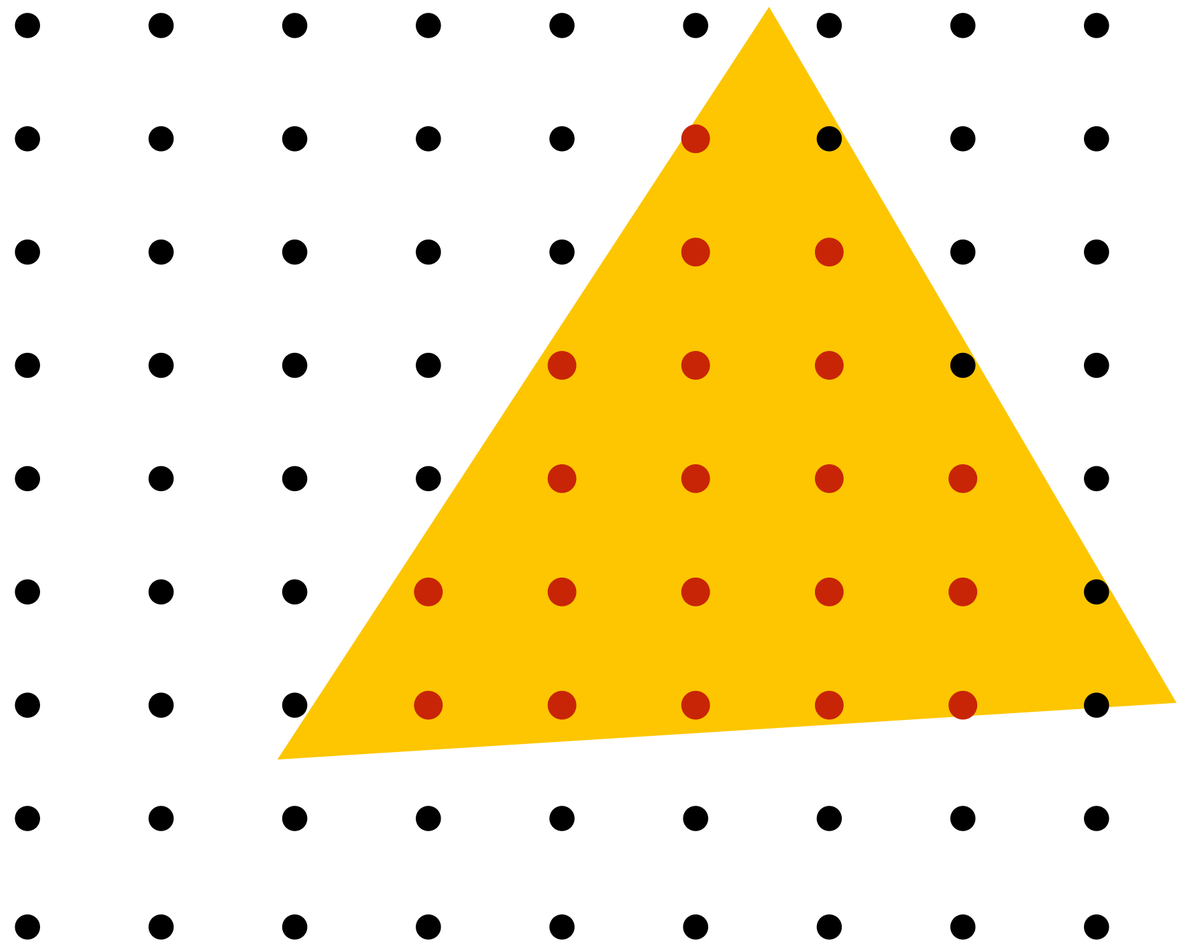
$$\mathbf{Z}(x,y)$$

$$\mathbf{U}(x,y)$$

$$\mathbf{V}(x,y)$$

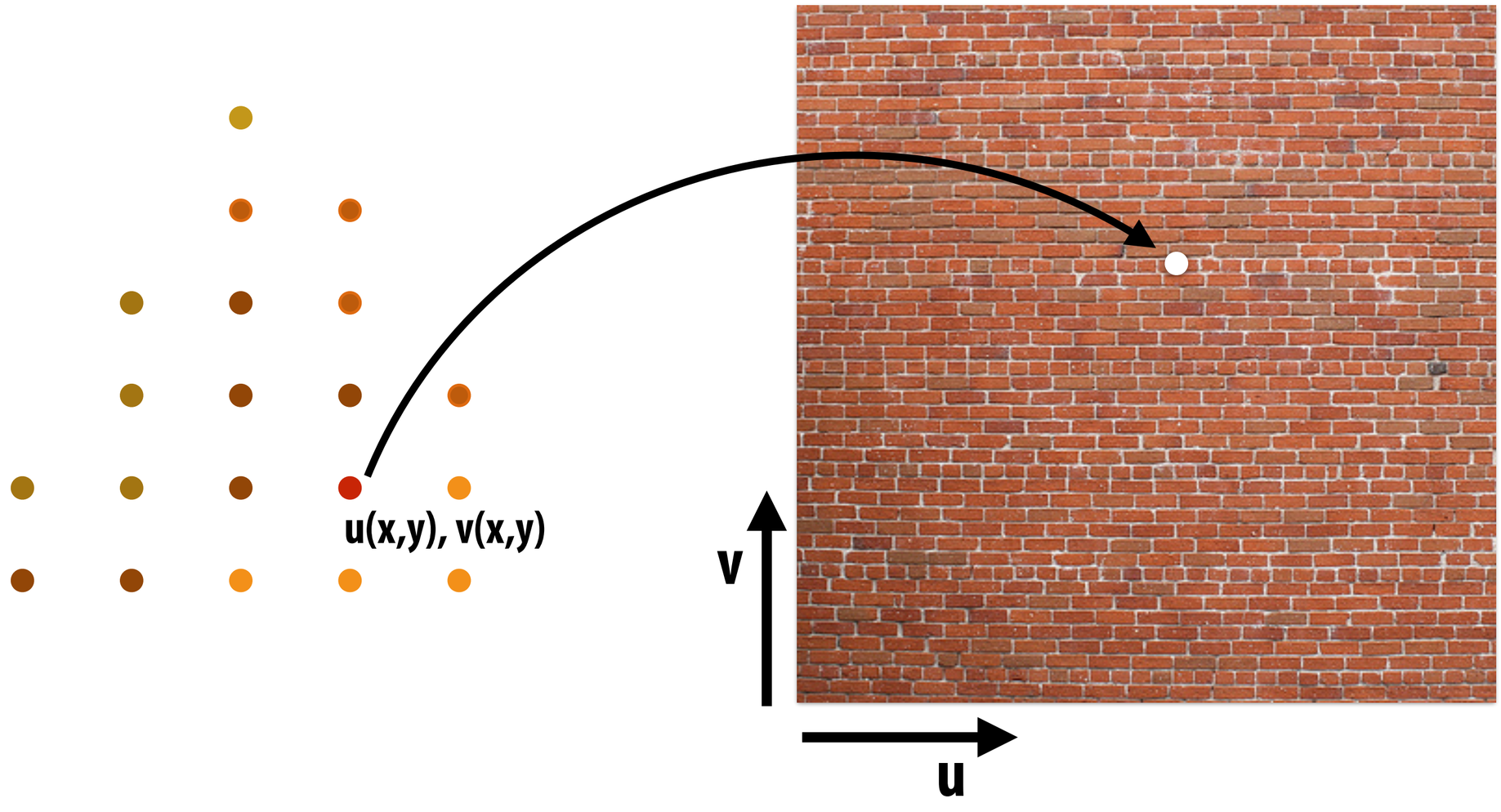
Step 6: sample coverage

Evaluate attributes z, u, v at all covered samples



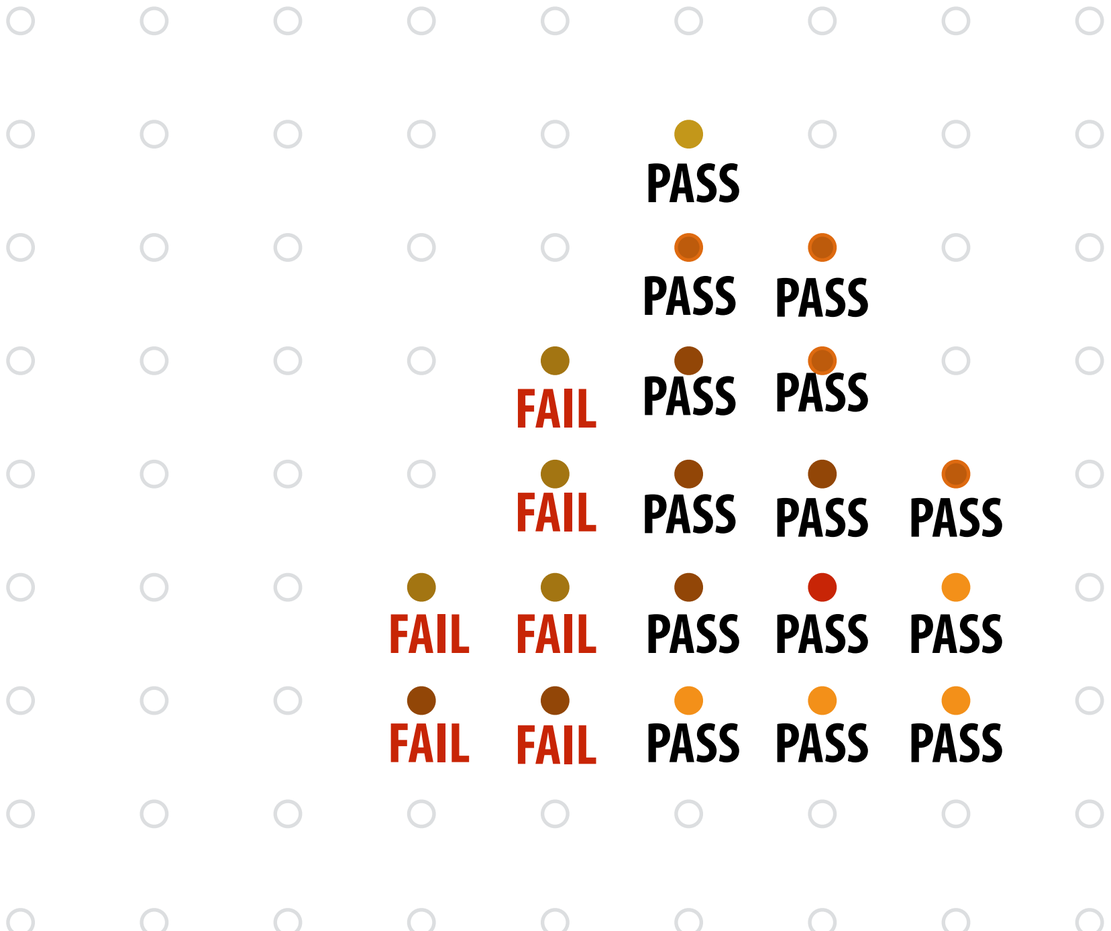
Step 6: compute triangle color at sample point

e.g., sample texture map

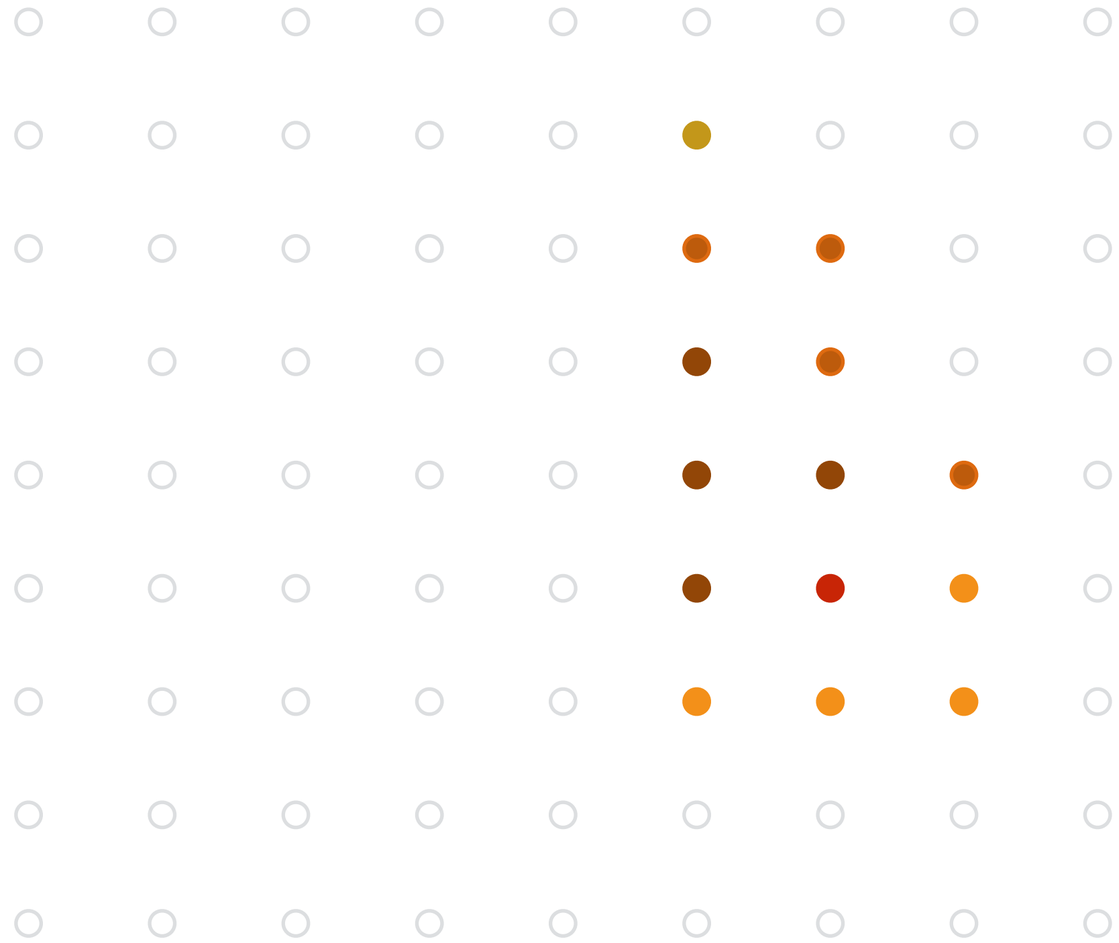


Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)

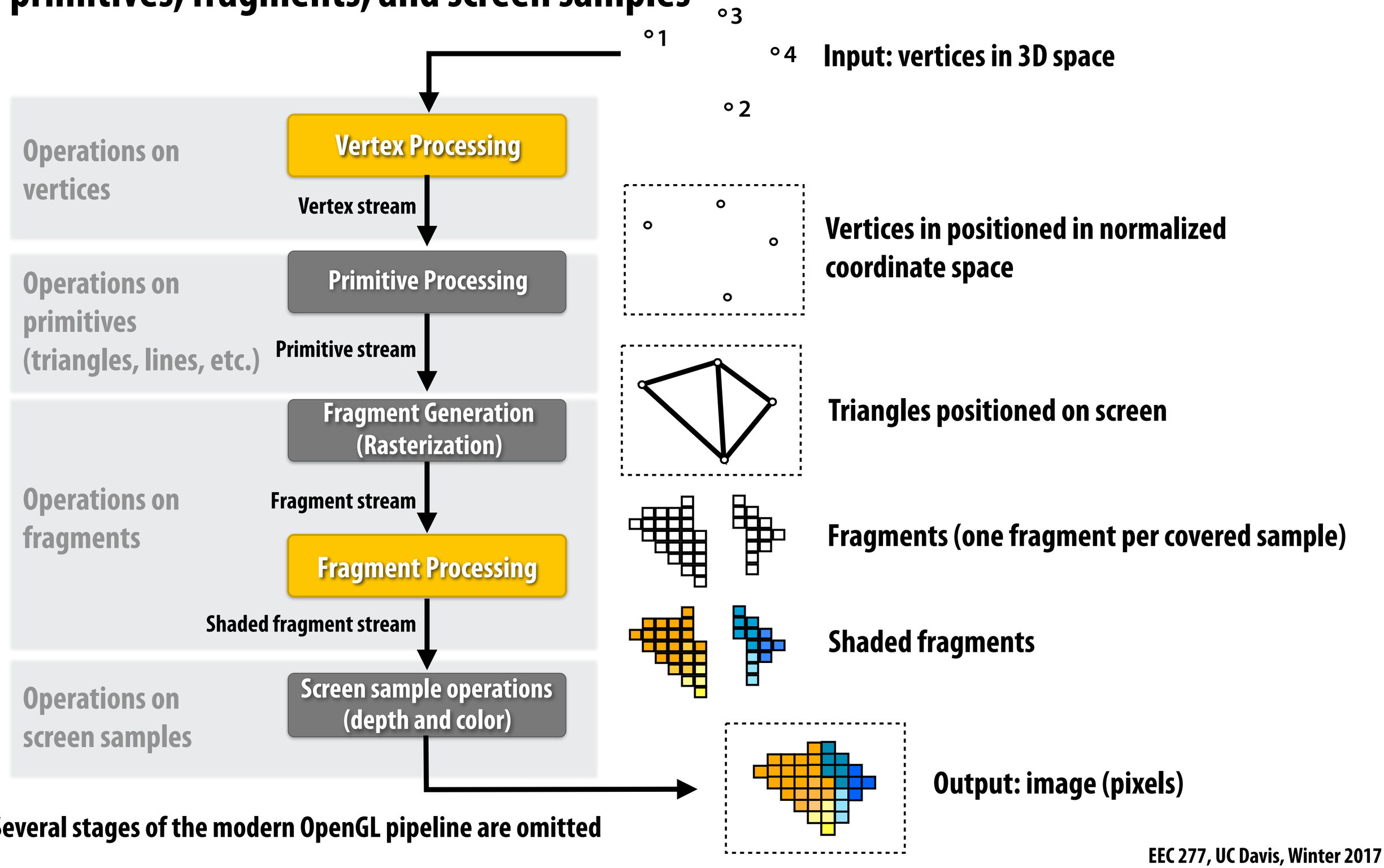


Step 8: update color buffer (if depth test passed)



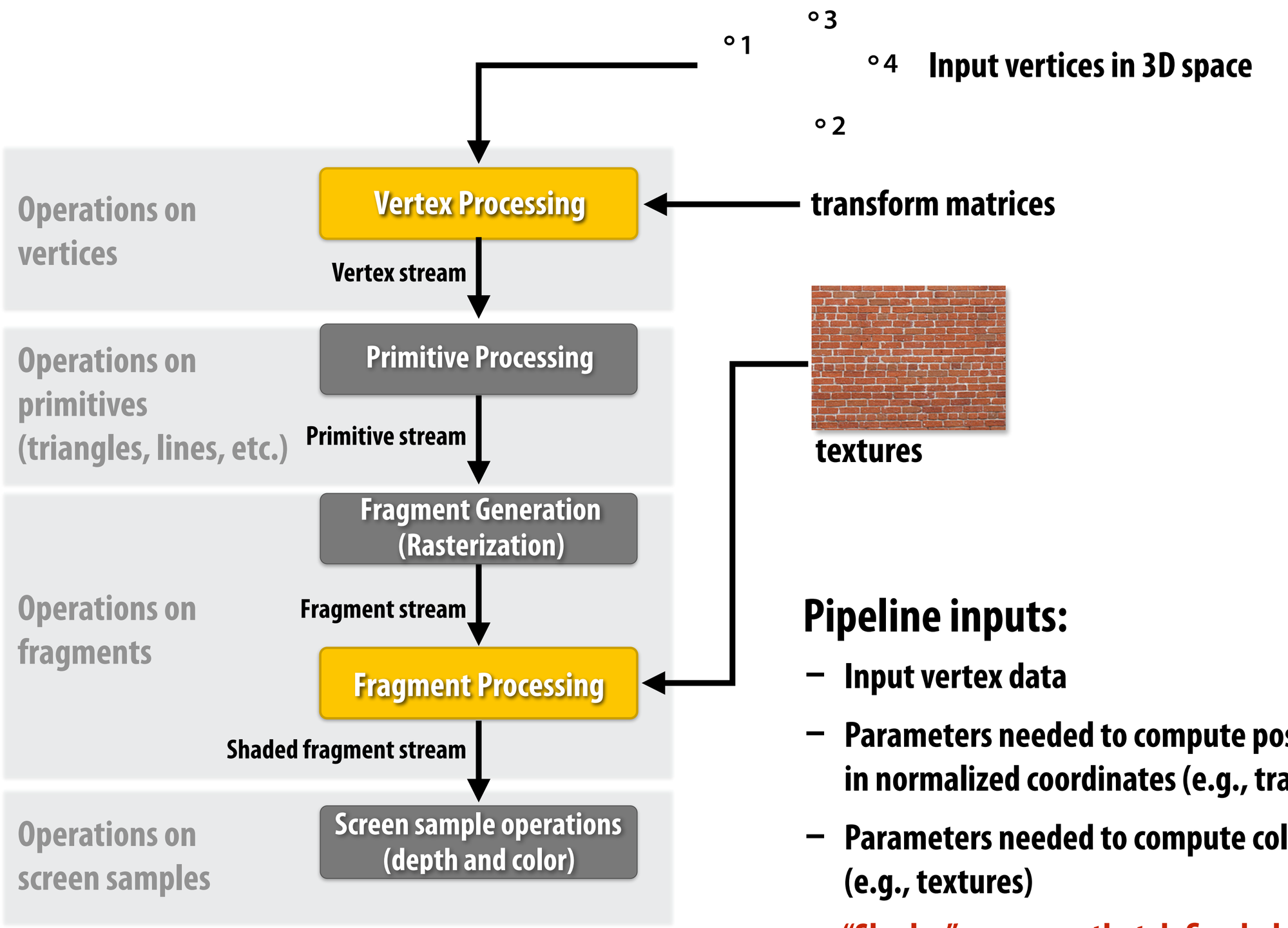
OpenGL/Direct3D graphics pipeline *

Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples



* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *



Pipeline inputs:

- Input vertex data
- Parameters needed to compute position on vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)
- "Shader" programs that define behavior of vertex and fragment stages

* several stages of the modern OpenGL pipeline are omitted

Shader programs

Define behavior of vertex processing and fragment processing stages

Describe operation on a single vertex (or single fragment)

Example GLSL fragment shader program

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

Program parameters

Per-fragment attributes (interpolated by rasterizer)

Sample surface albedo (reflectance color) from texture

Shader outputs surface color

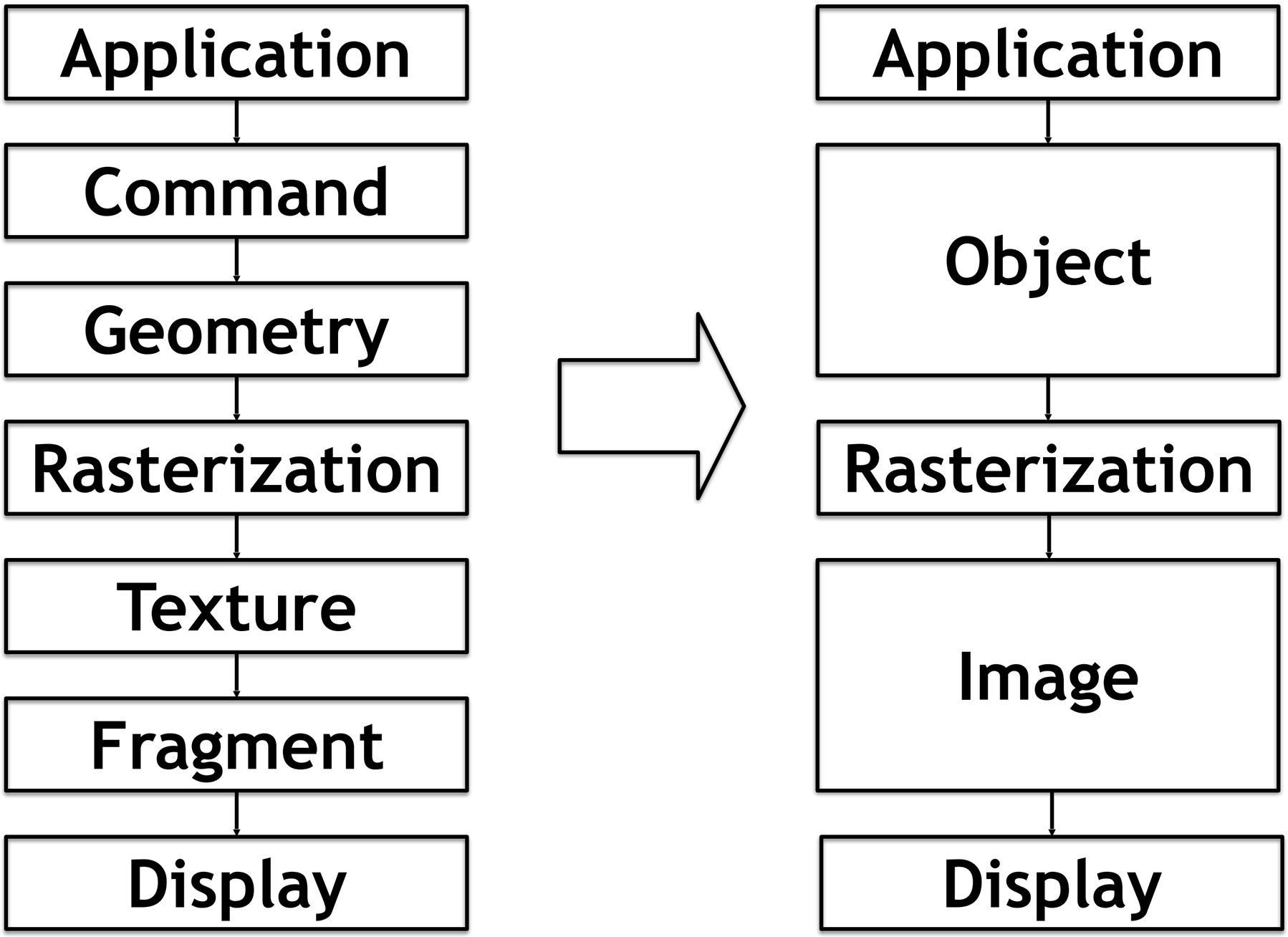
Modulate surface albedo by incident irradiance (incoming light)

Shader function executes once per fragment.

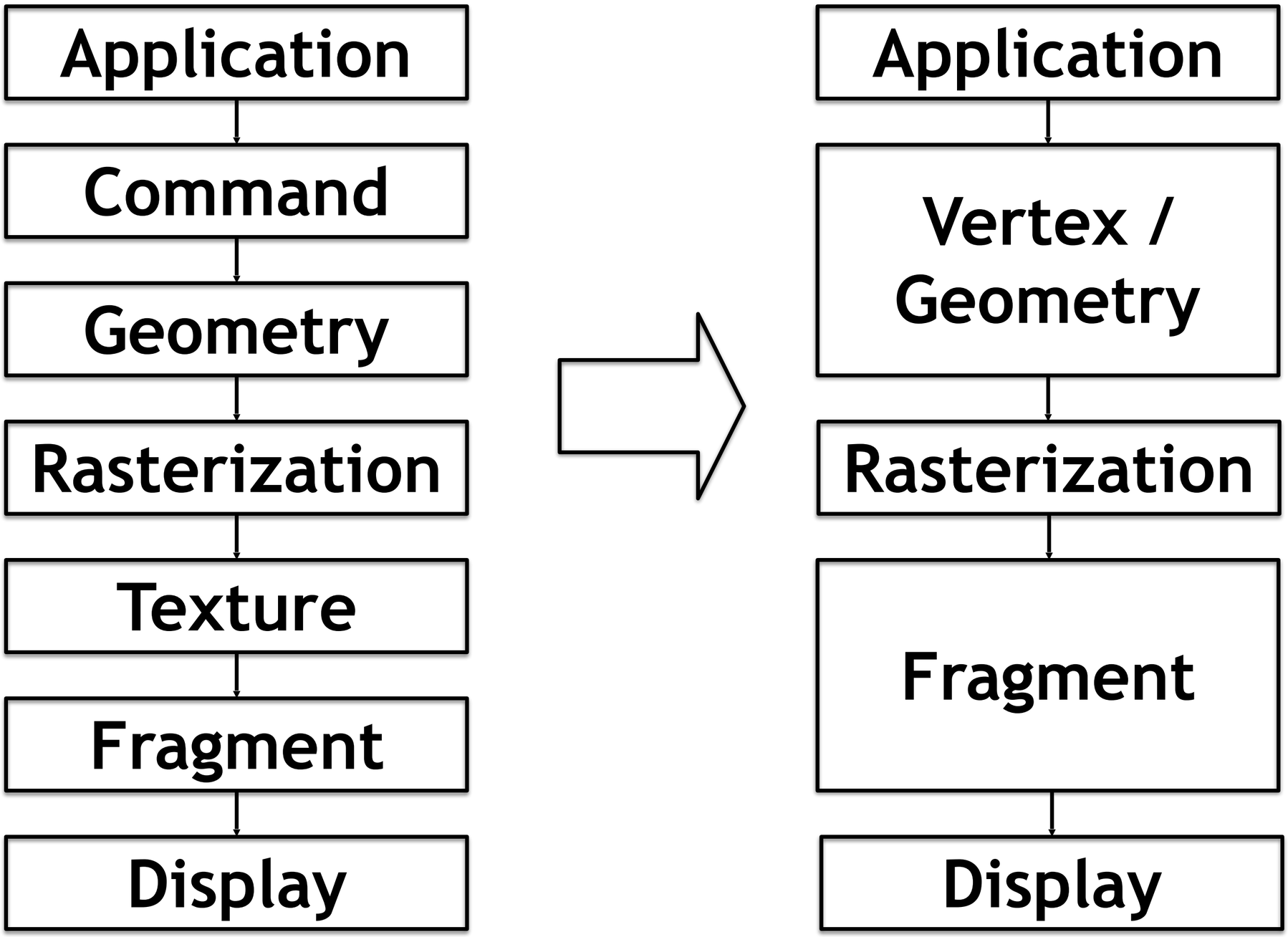
Outputs color of surface at sample point corresponding to fragment.

(this shader performs a texture lookup to obtain the surface's material color at this point, then performs a simple lighting computation)

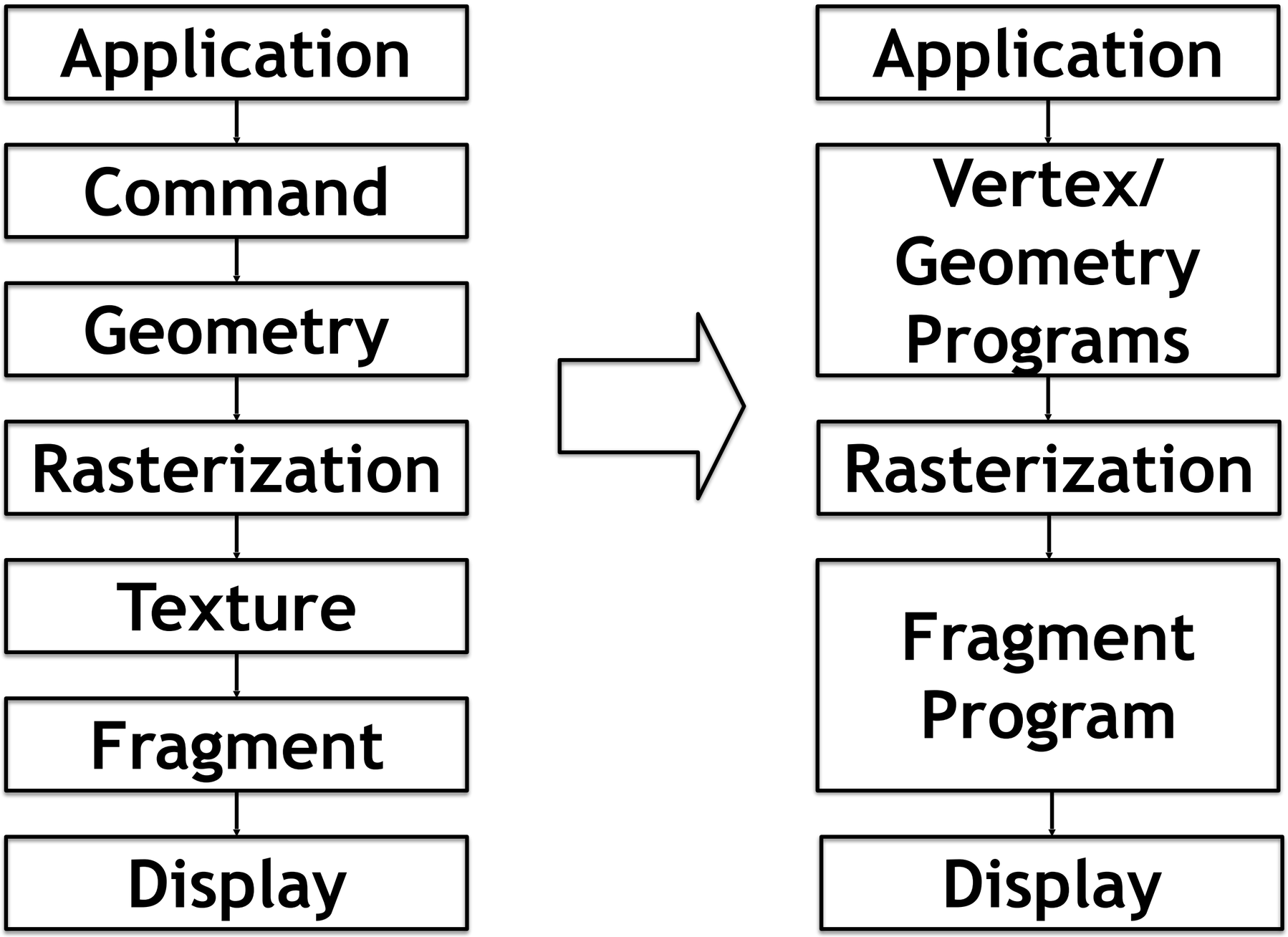
Graphics Pipeline



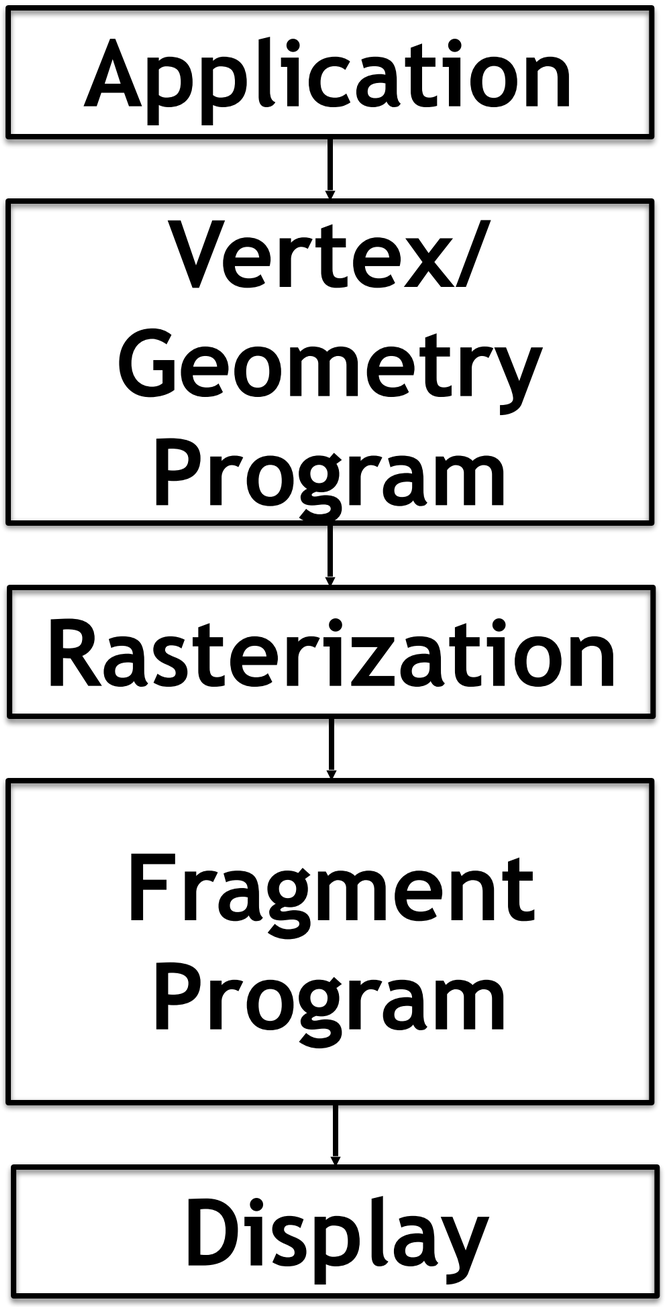
Graphics Pipeline



Programmable Graphics Pipeline

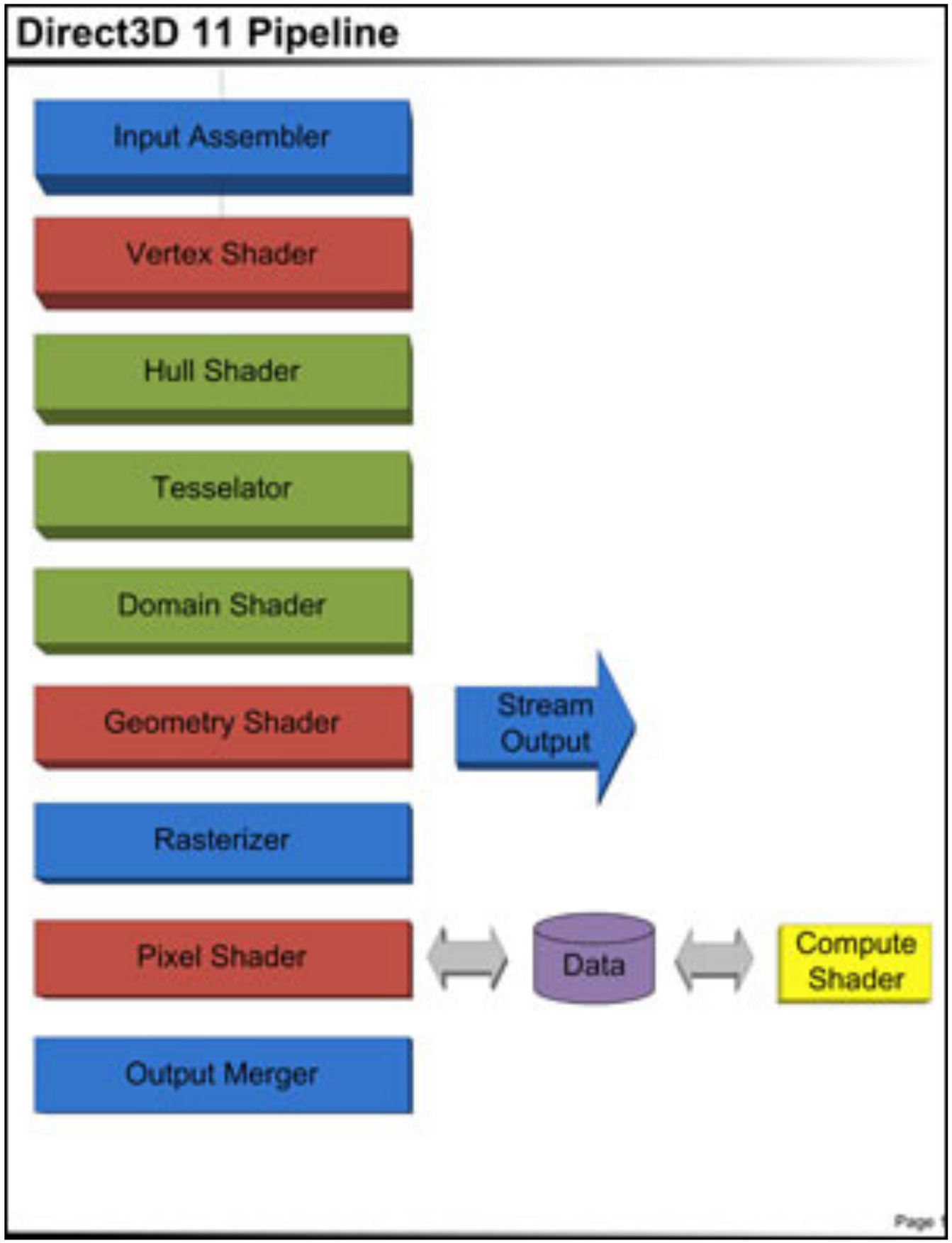


Abstracting the Pipeline

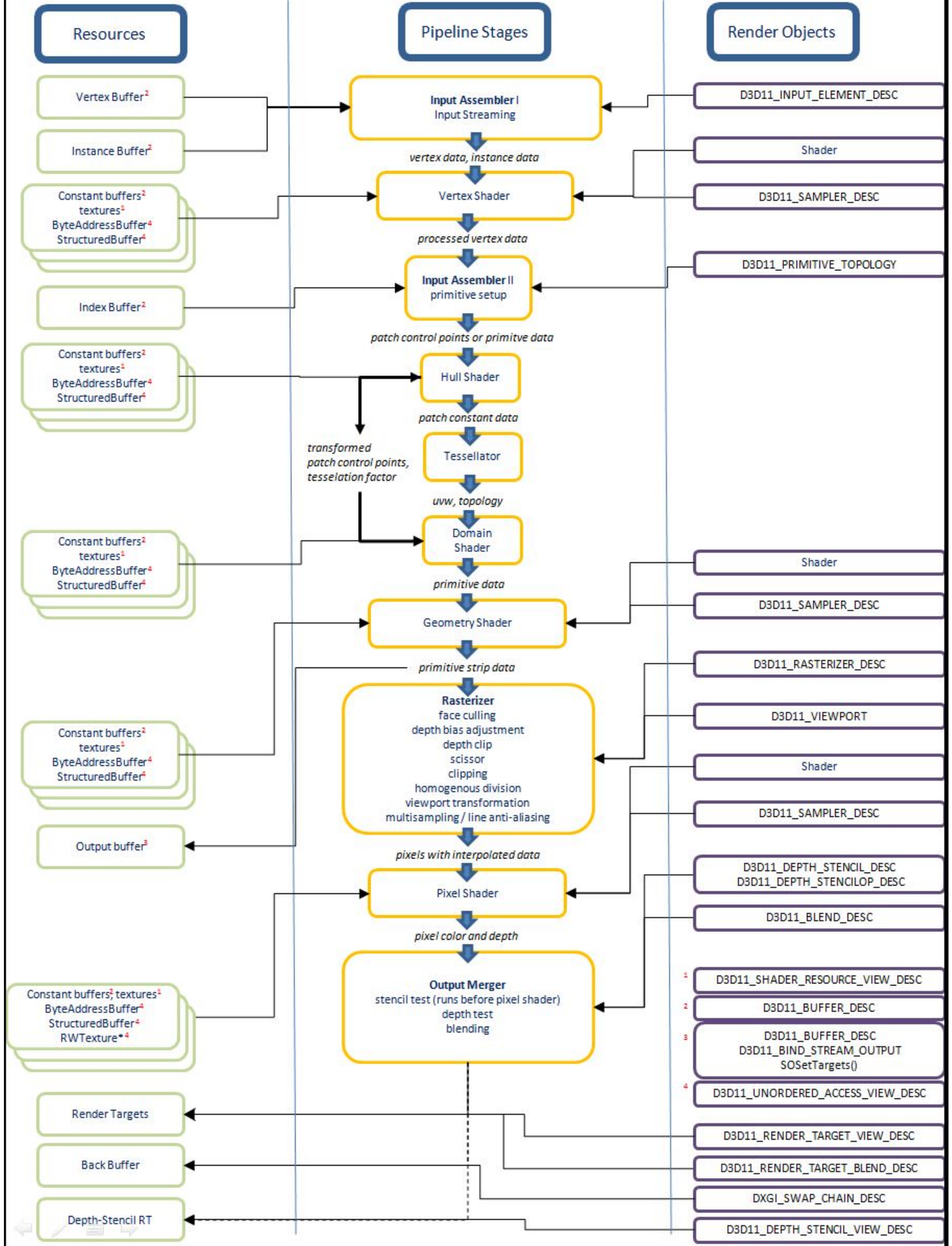


- The vertex, geometry, and fragment programs are:
  - Programmable (but restrictive)
  - Data parallel
  - High performance
- We can thus map general purpose computation to them
- But all we have available is graphics commands
- New programming environments (CUDA, OpenCL, etc.) bridge this gap
- Future lectures!

DX11


Direct3D 11 Pipeline
Input Assembler
Vertex Shader
Hull Shader
Tesselator
Domain Shader
Geometry Shader
Stream Output
Rasterizer
Pixel Shader
Data
Compute Shader
Output Merger

DX11


Resources
Pipeline Stages
Render Objects
Vertex Buffer[2]
Instance Buffer[2]
Input Assembler I
Input Streaming
D3D11_INPUT_ELEMENT_DESC
vertex data, instance data
Shader
Constant buffers[2]
textures[1]
ByteAddressBuffer[4]
StructuredBuffer[4]
Vertex Shader
D3D11_SAMPLER_DESC
processed vertex data
Index Buffer[2]
Input Assembler II
primitive setup
D3D11_PRIMITIVE_TOPOLOGY
patch control points or primitve data
Constant buffers[2]
textures[1]
ByteAddressBuffer[4]
StructuredBuffer[4]
Hull Shader
patch constant data
transformed
patch control points,
tesselation factor
Tessellator
uvw, topology
Constant buffers[2]
textures[1]
ByteAddressBuffer[4]
StructuredBuffer[4]
Domain
Shader
Shader
primitive data
D3D11_SAMPLER_DESC
Geometry Shader
primitive strip data
D3D11_RASTERIZER_DESC
Rasterizer
face culling
depth bias adjustment
depth clip
scissor
clipping
homogenous division
viewport transformation
multisampling / line anti-aliasing
D3D11_VIEWPORT
Constant buffers[2]
textures[1]
ByteAddressBuffer[4]
StructuredBuffer[4]
Shader
D3D11_SAMPLER_DESC
Output buffer[3]
pixels with interpolated data
D3D11_DEPTH_STENCIL_DESC
D3D11_DEPTH_STENCILOP_DESC
Pixel Shader
D3D11_BLEND_DESC
pixel color and depth
Output Merger
stencil test (runs before pixel shader)
depth test
blending
1 D3D11_SHADER_RESOURCE_VIEW_DESC
2 D3D11_BUFFER_DESC
Constant buffers[2]; textures[1]
ByteAddressBuffer[4]
StructuredBuffer[4]
RWTexture*[4]
3 D3D11_BUFFER_DESC
D3D11_BIND_STREAM_OUTPUT
SOSetTargets()
4 D3D11_UNORDERED_ACCESS_VIEW_DESC
Render Targets
D3D11_RENDER_TARGET_VIEW_DESC
Back Buffer
D3D11_RENDER_TARGET_BLEND_DESC
DXGI_SWAP_CHAIN_DESC
Depth-Stencil RT
D3D11_DEPTH_STENCIL_VIEW_DESC

Design alternatives

Hidden-surface elimination

- painters algorithm (host) = hide-first
- z-buffer = hide-last

Texturing

- Fragment textures = texture-last
- Vertex textures = texture-first

Shading

- Vertex shading = shade-first or shade-first-vertex
- Fragment shading = shade-last-fragment
- Deferred shading = shade-last-pixel

Outline

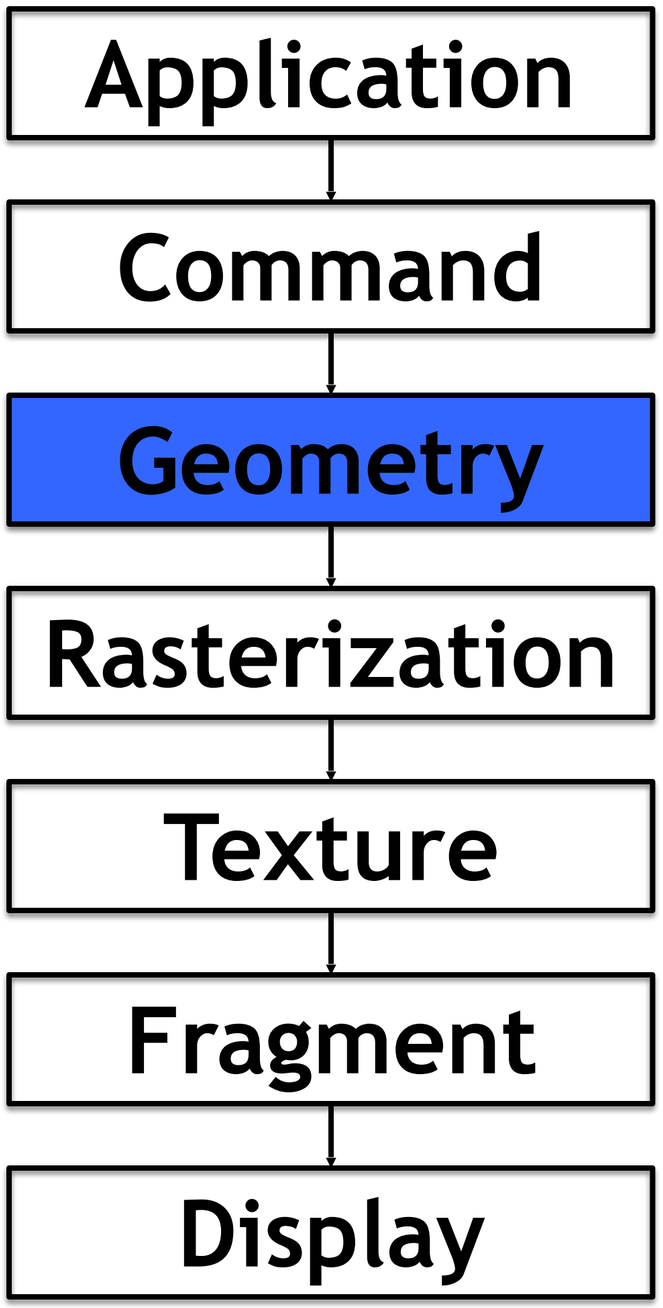
API Decisions

Motivating OpenGL

The Graphics Pipeline

Computational Costs (courtesy Akeley/Hanrahan)

Computational Requirements



Geometry (per-vertex)

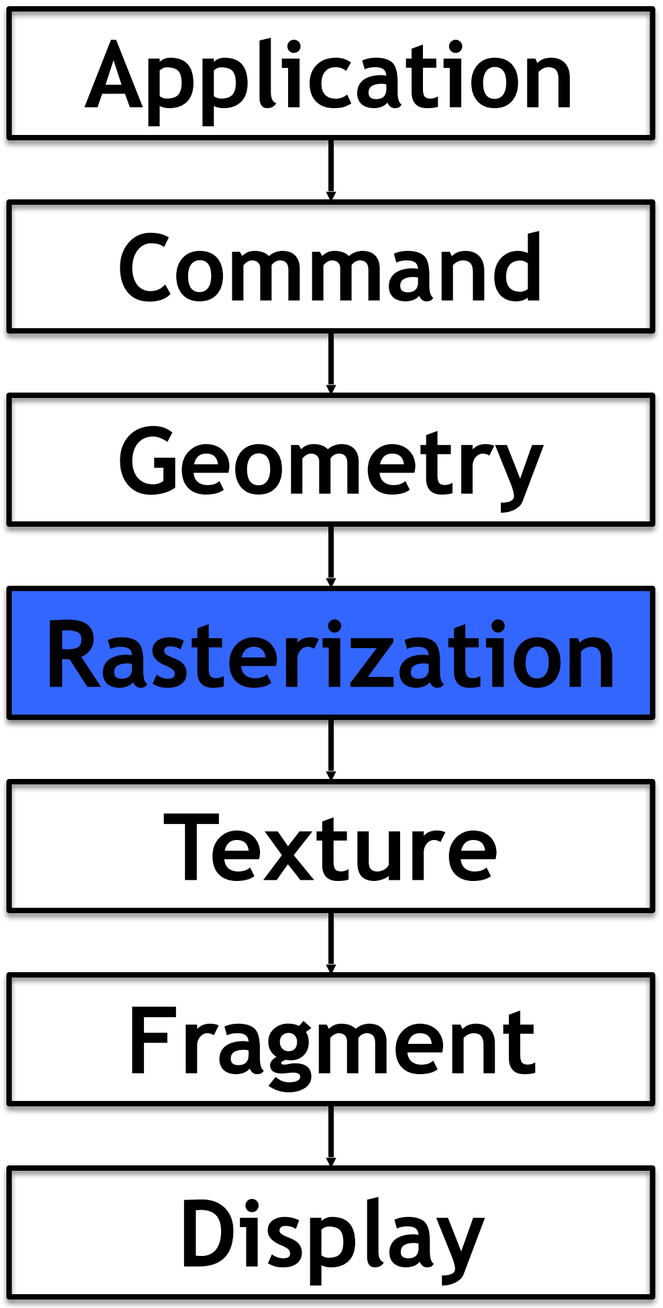
Assumptions:

- 1 infinite light
- Texture coordinates

ADD	CMP	MUL	DIV	SPE
40	8	53	1	1

Rough estimate: 100 ops

Computational Requirements

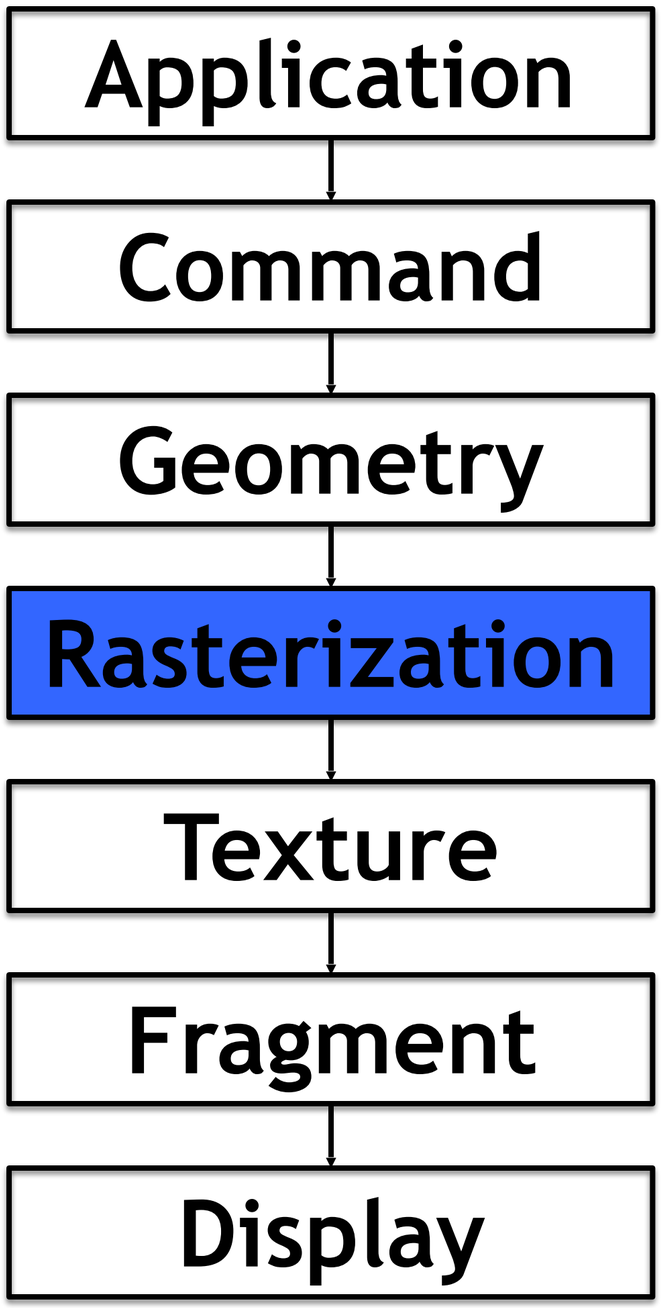


Rasterization: per-vertex
Assumptions:
- 7 interpolants (z,r,g,b,s,t,q)

ADD	CMP	MUL	DIV	SPE
62	22	55	4	0

Rough estimate

Computational Requirements



Rasterization: per-fragment

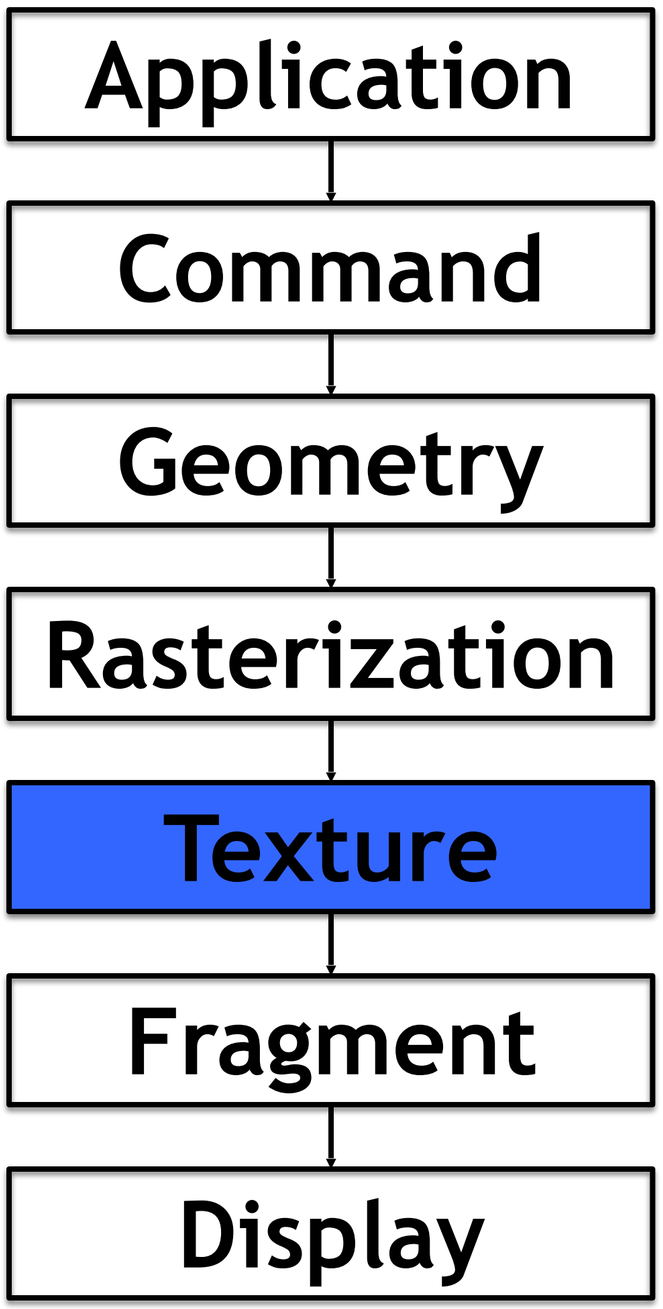
Assumptions:

- 7 interpolants (z,r,g,b,s,t,q)

ADD	CMP	MUL	DIV	SPE
16	3	6	0	0

Rough estimate

Computational Requirements



Texture: per-fragment

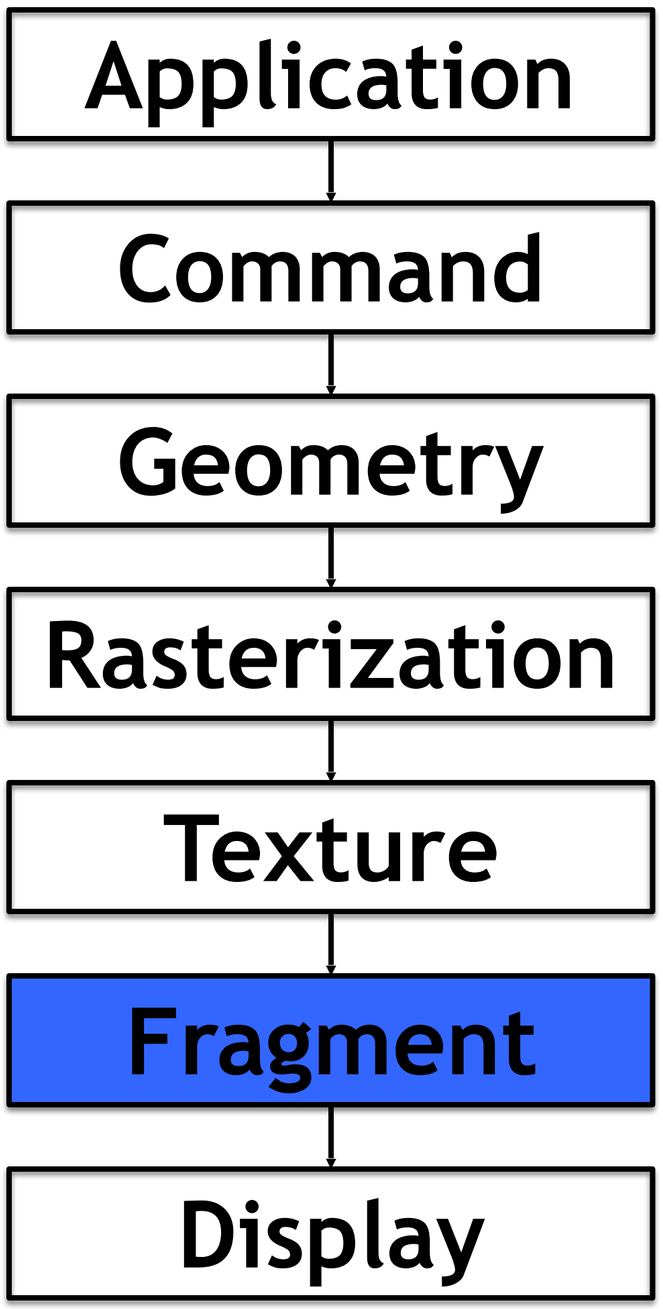
Assumptions:

- Projective texture mapping
- Level of detail calculation
- Trilinear interpolation

ADD	CMP	MUL	DIV	SPE
42	5	48	1	3

Rough estimate

Computational Requirements



Fragment: per-fragment

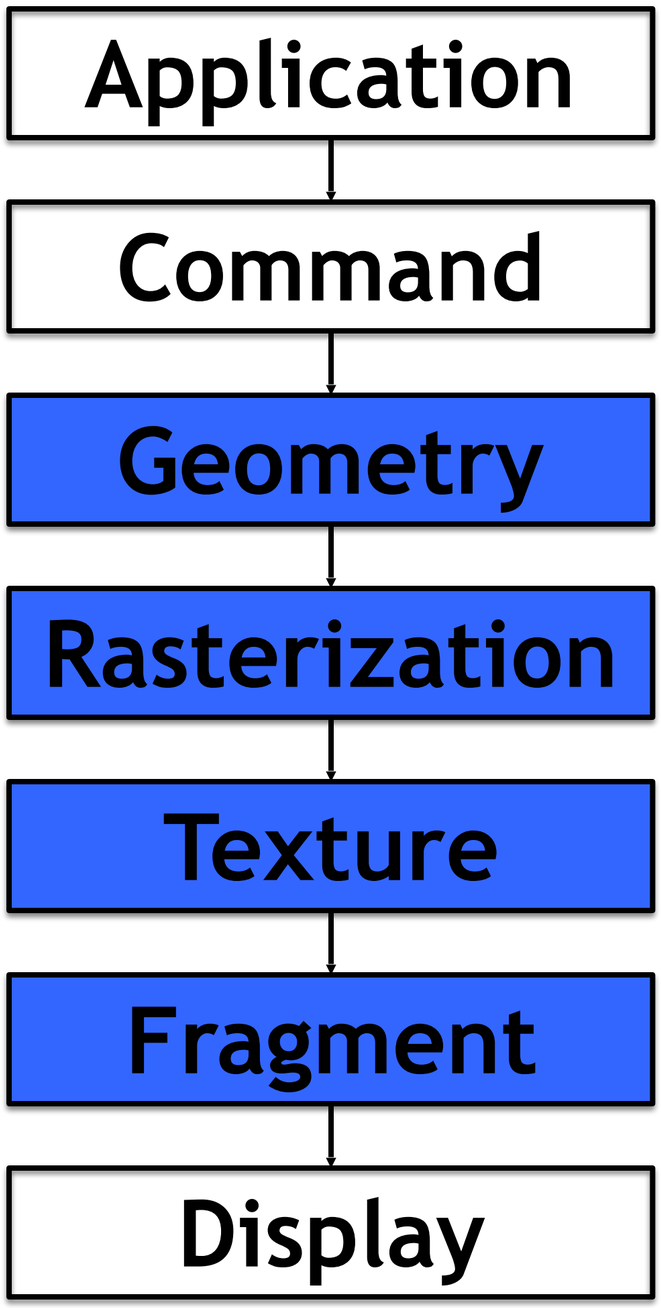
Assumptions:

- Texture blending
- Color blending
- Depth buffering

ADD	CMP	MUL	DIV	SPE
8	1	16	0	0

Rough estimate

Computational Requirements



Per-Vertex

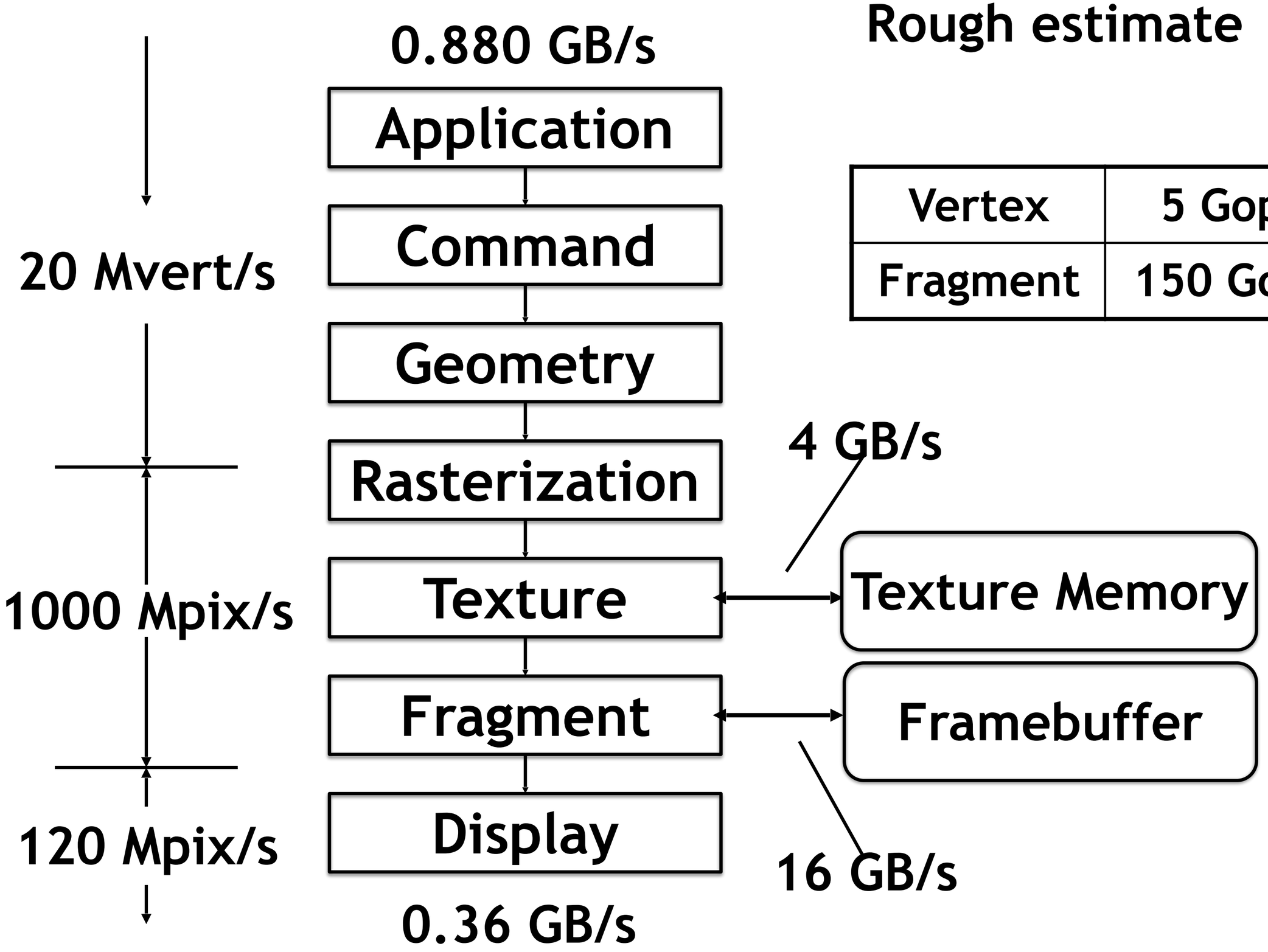
ADD	CMP	MUL	DIV	SPE
102	30	108	5	1

Per-Fragment

ADD	CMP	MUL	DIV	SPE
66	9	70	1	3

Rough estimate

Communication Requirements

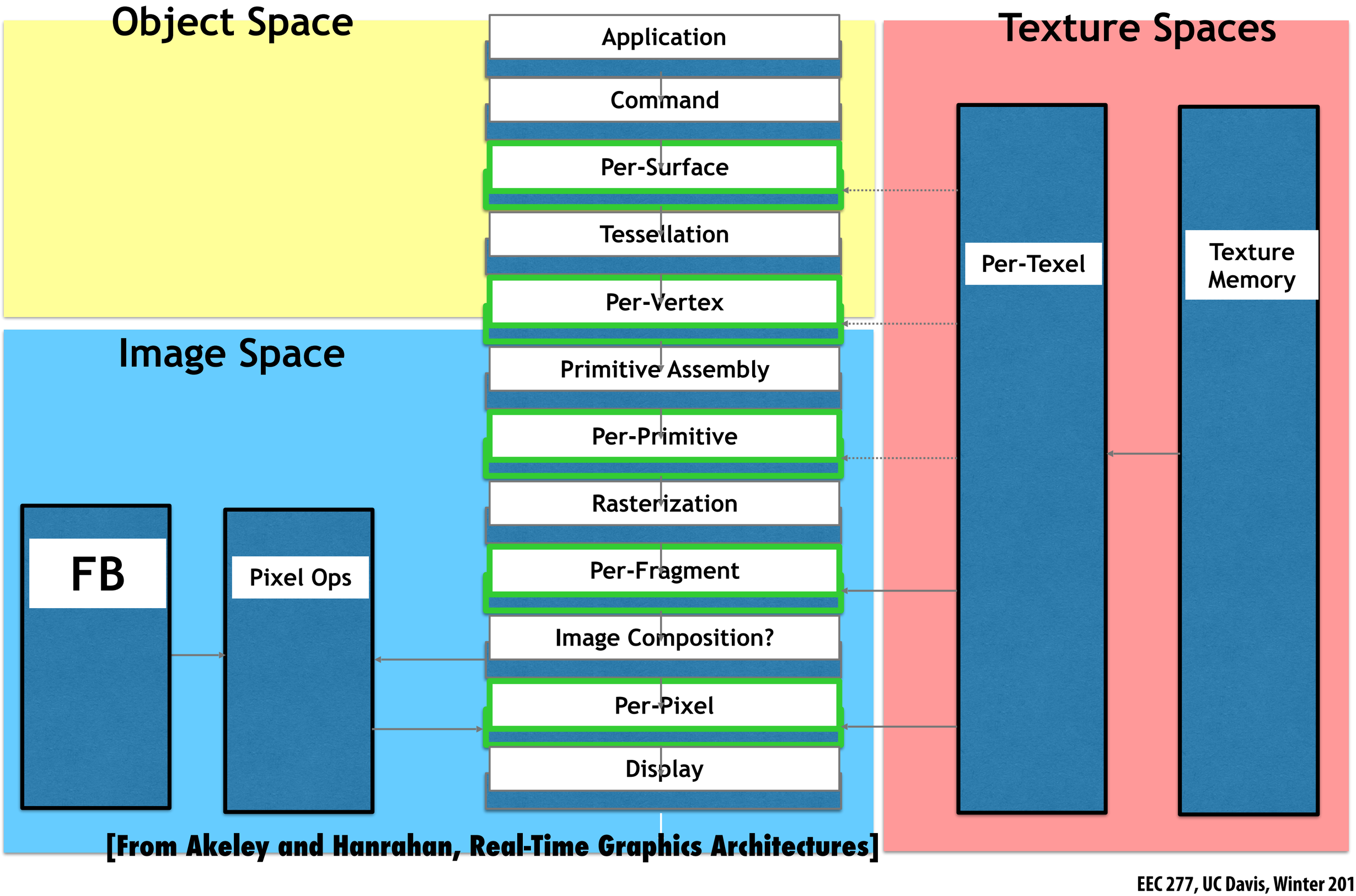


Rough estimate

Vertex	5 Gops
Fragment	150 Gops

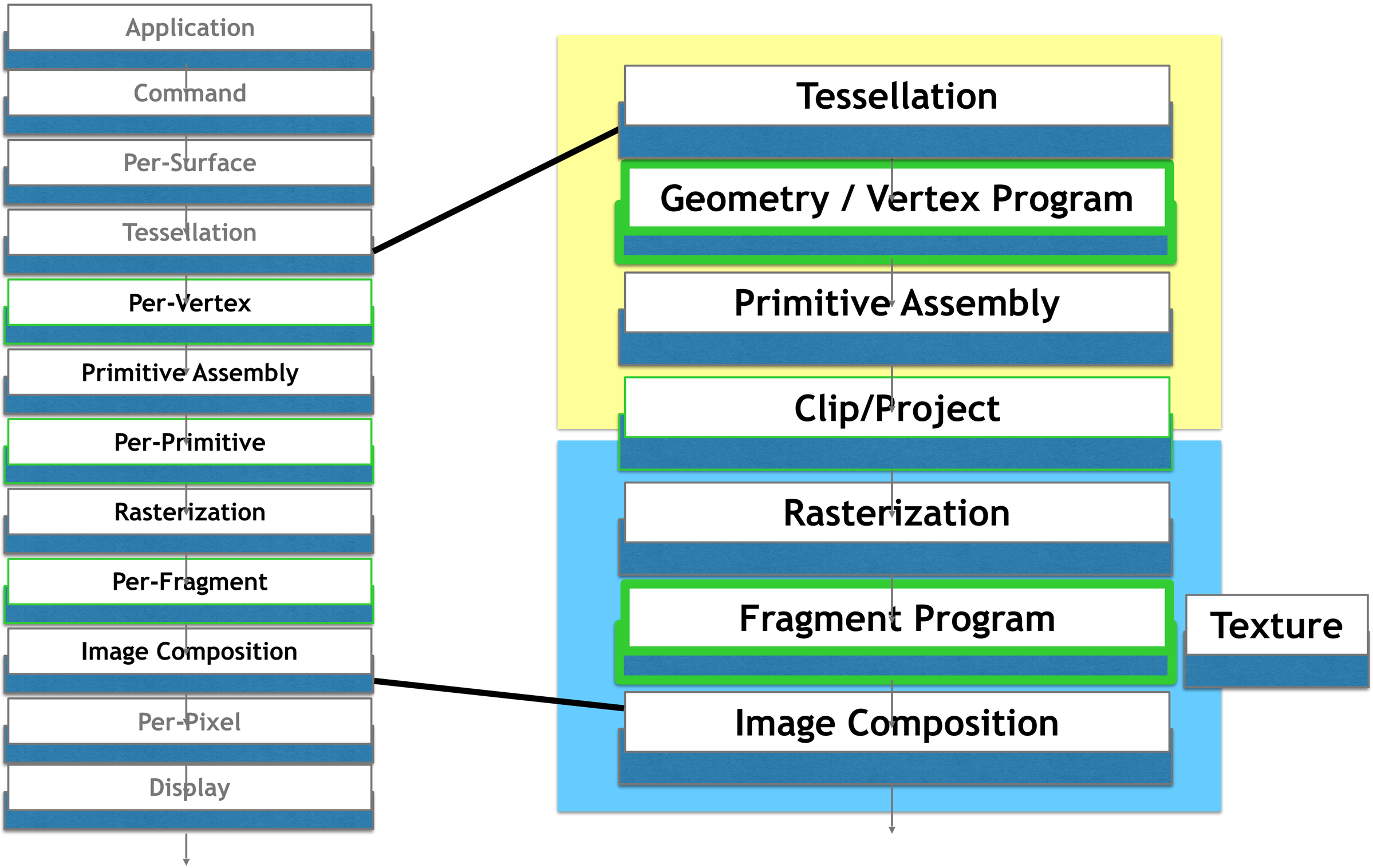
Next Lecture

Generalized Graphics Pipeline?

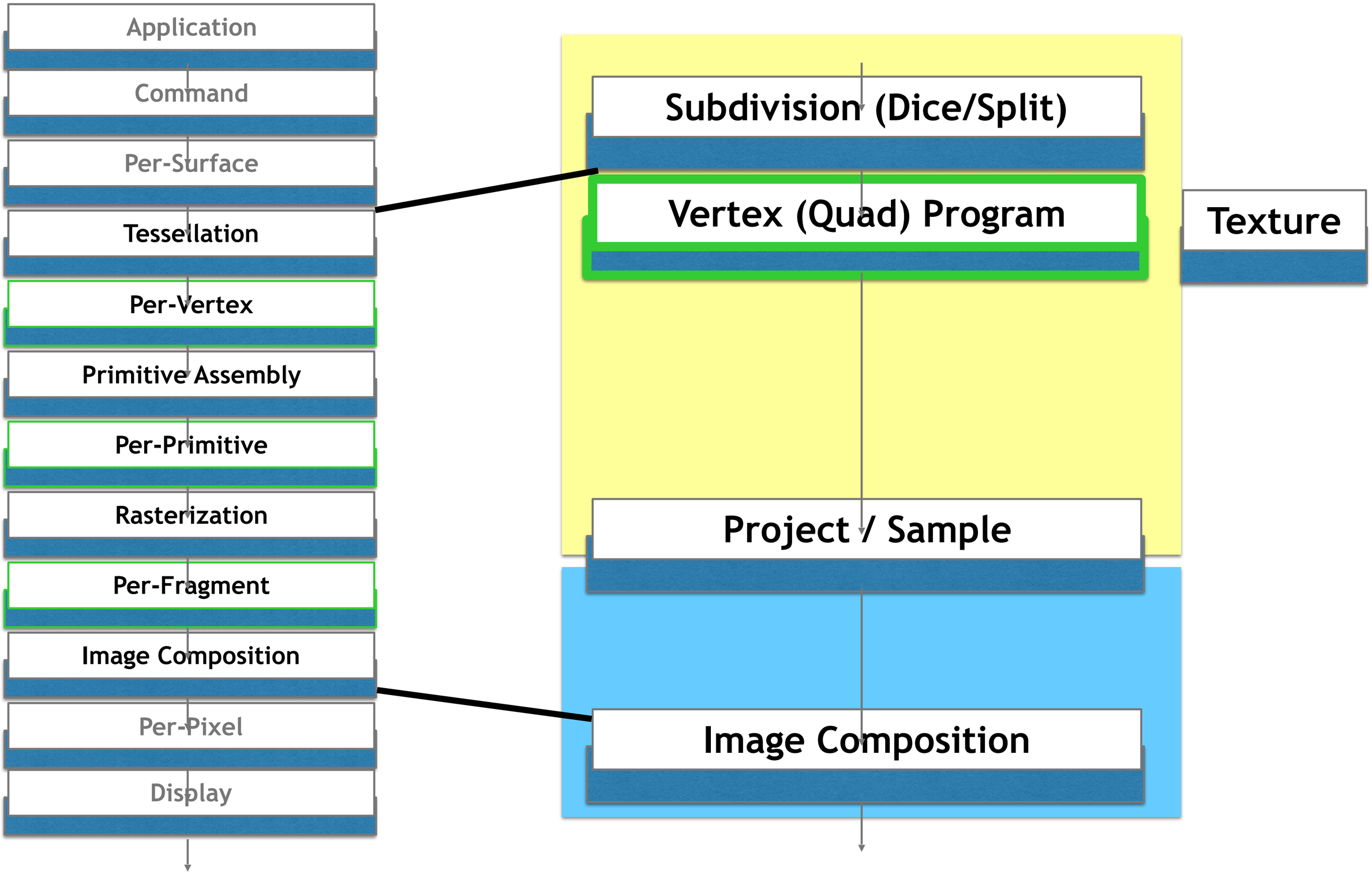


[From Akeley and Hanrahan, Real-Time Graphics Architectures]

OpenGL in Generalized Pipeline



? In Generalized Pipeline



? in Generalized Pipeline