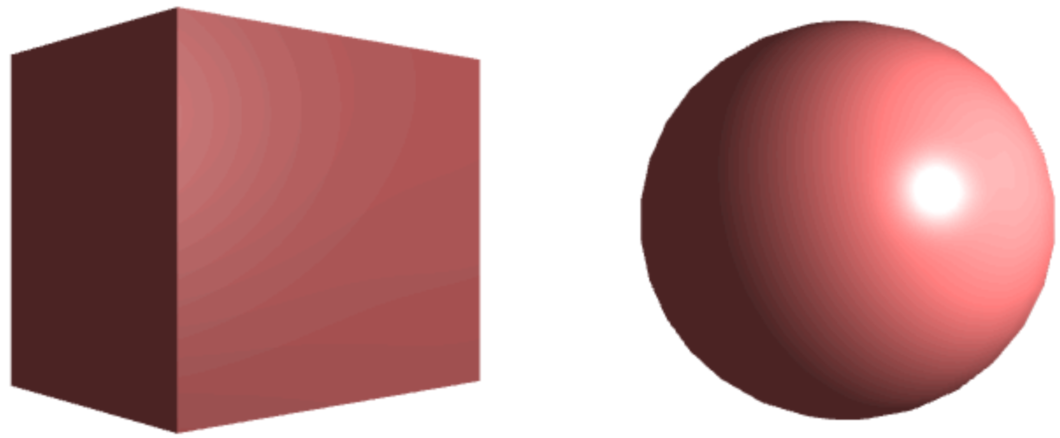


Computer Graphics

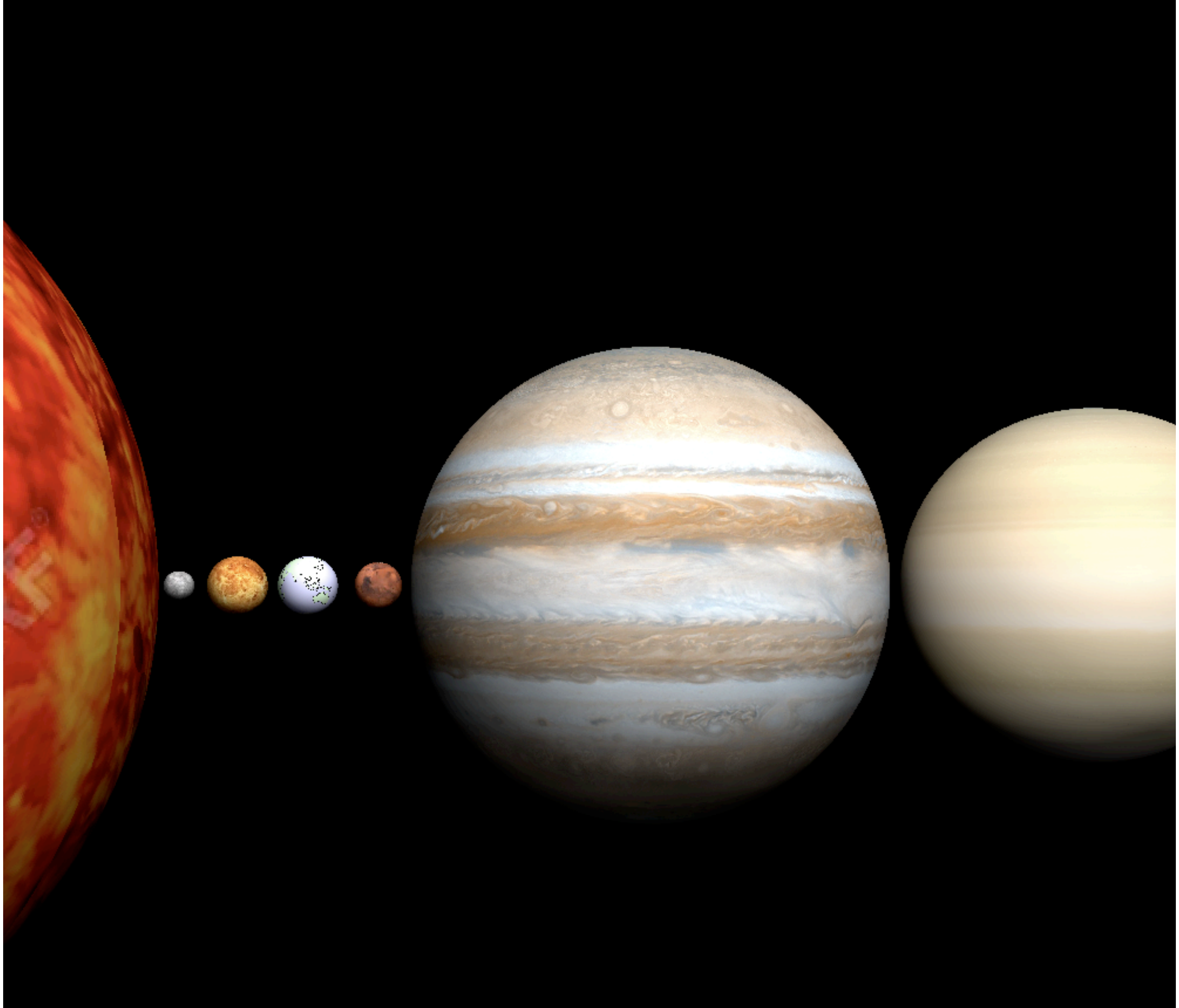
Lecture/Lab 20

Introducing WebGL

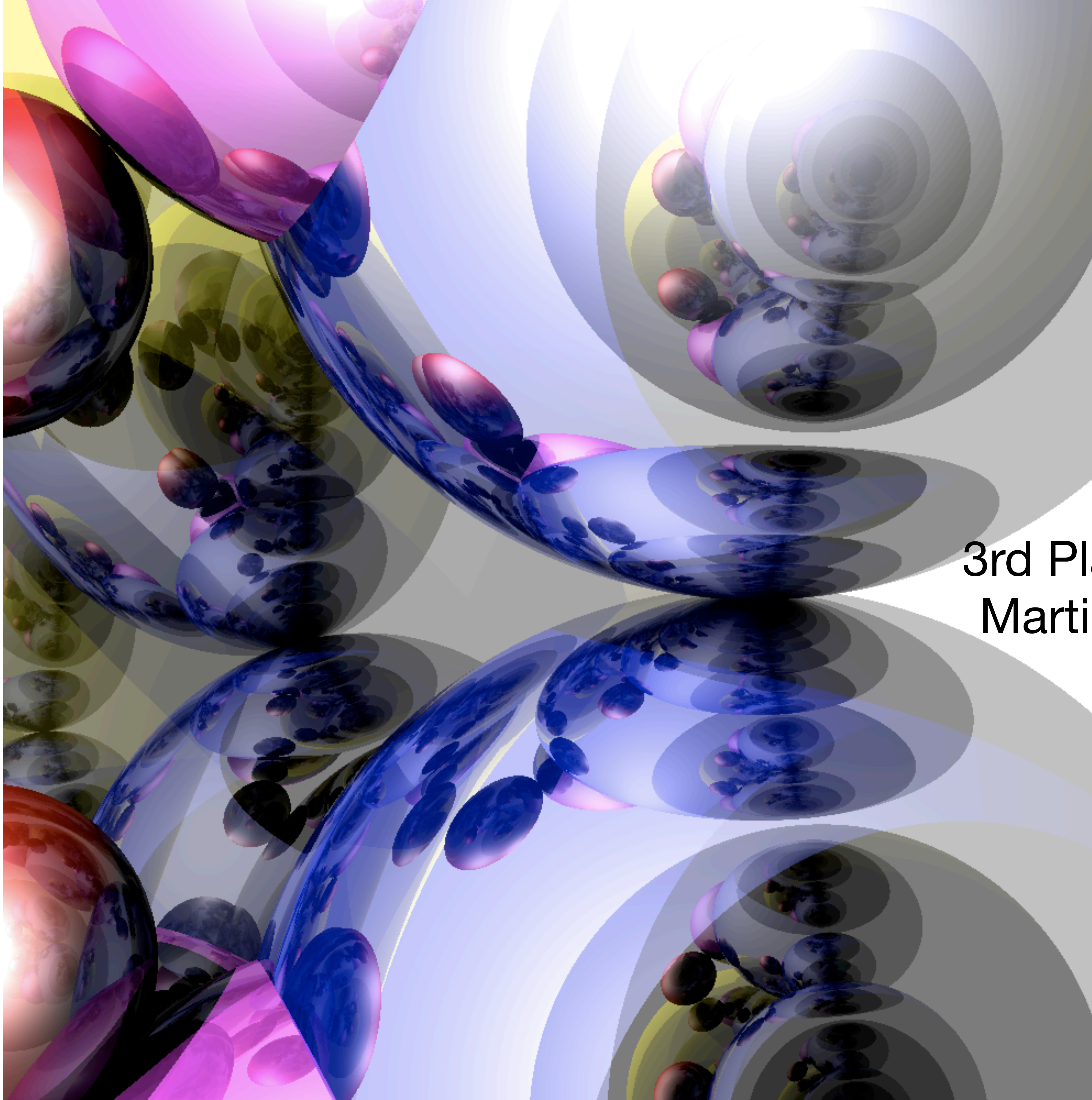
Announcements

- Some links added to the Final Project Proposal writeup on Canvas - more topic ideas, inspiration, resources for learning about what's out there.
- HW1 grading probably done today
- Reminders:
    - Project proposals due Friday
    - Takehome midterm out Friday

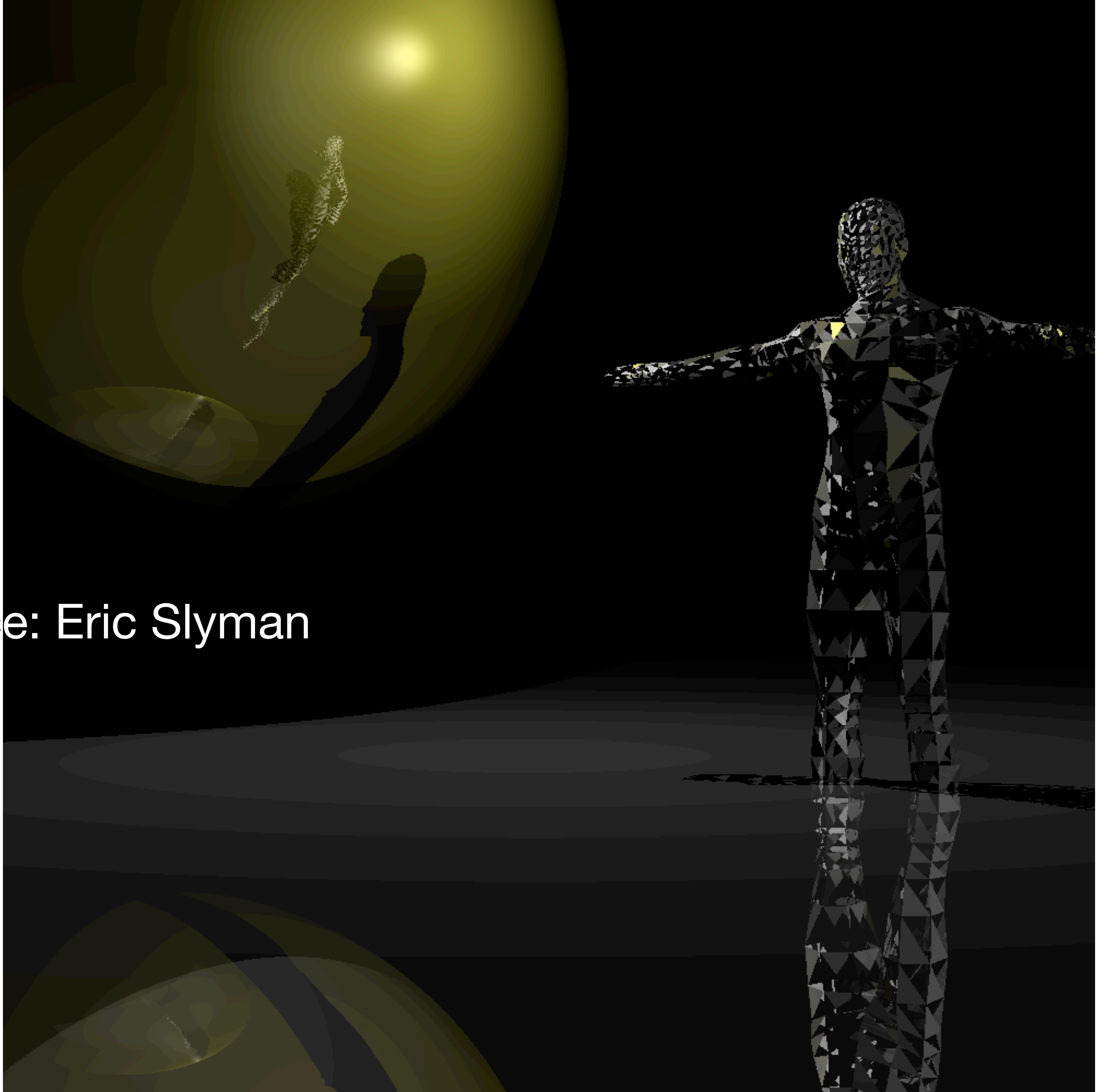
A2 Artifact Results



3rd Place (tie): Sam Burgess

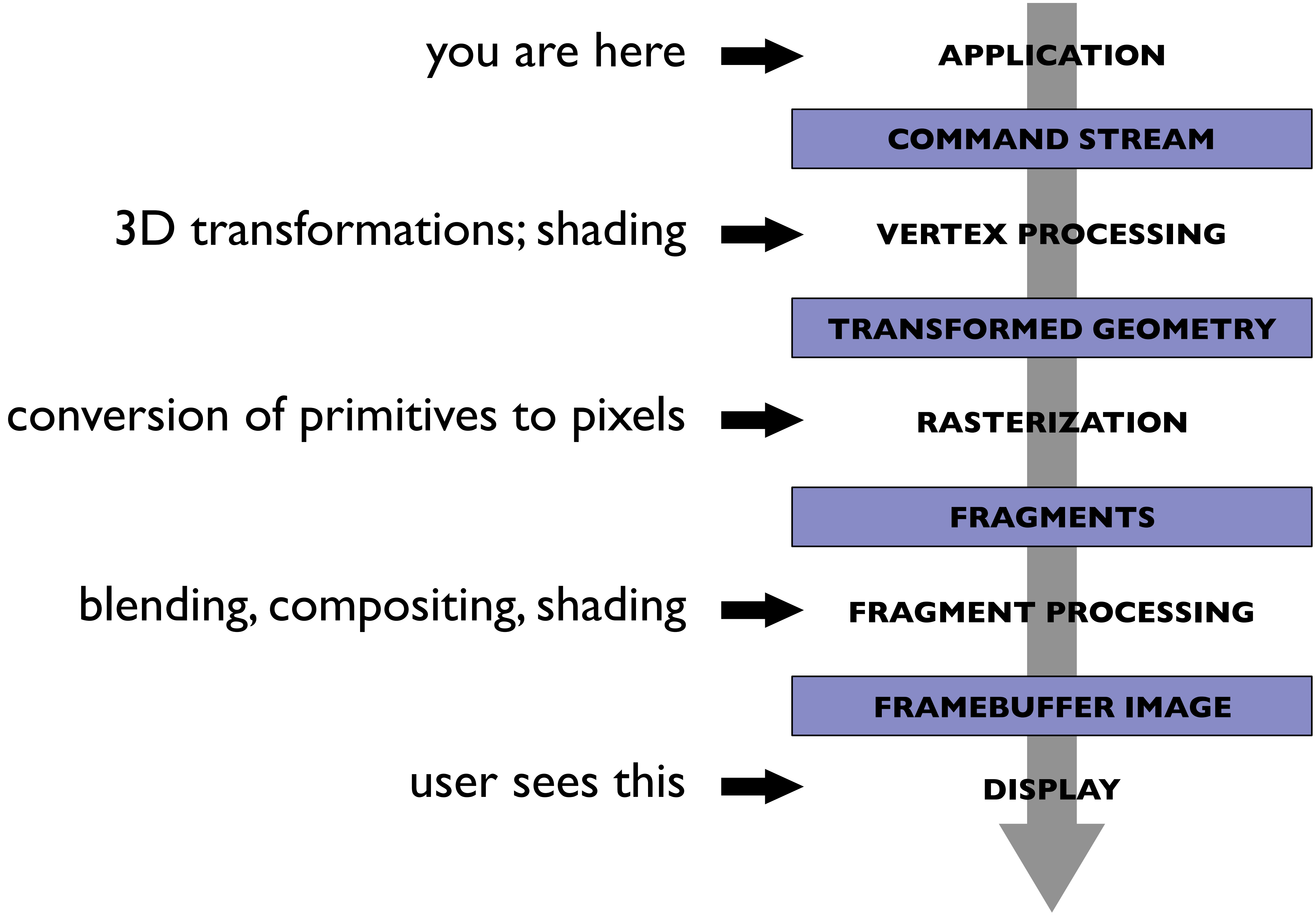


3rd Place (tie):
Martin Smith

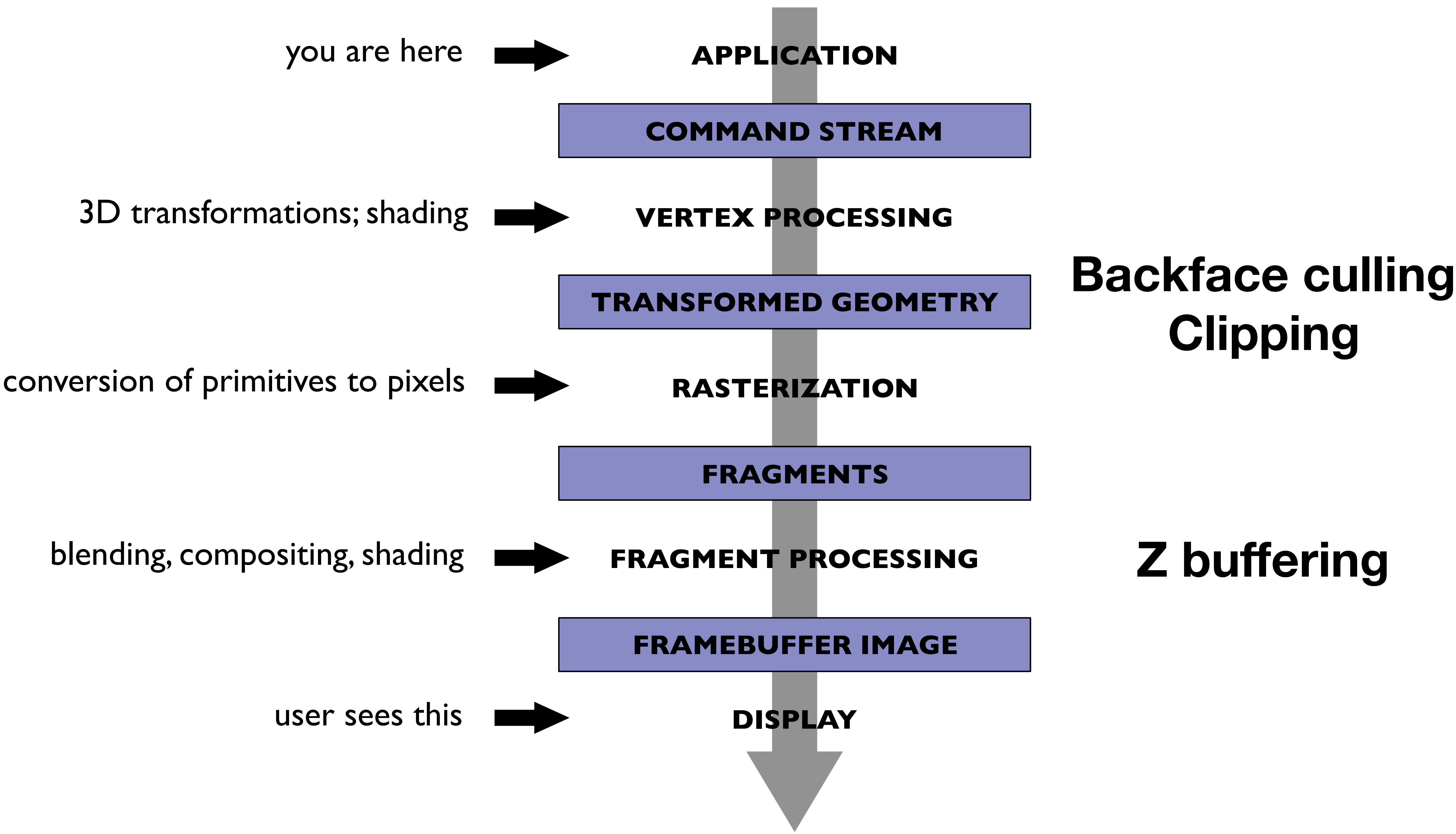


1st place: Eric Slyman

Graphics Pipeline: Overview

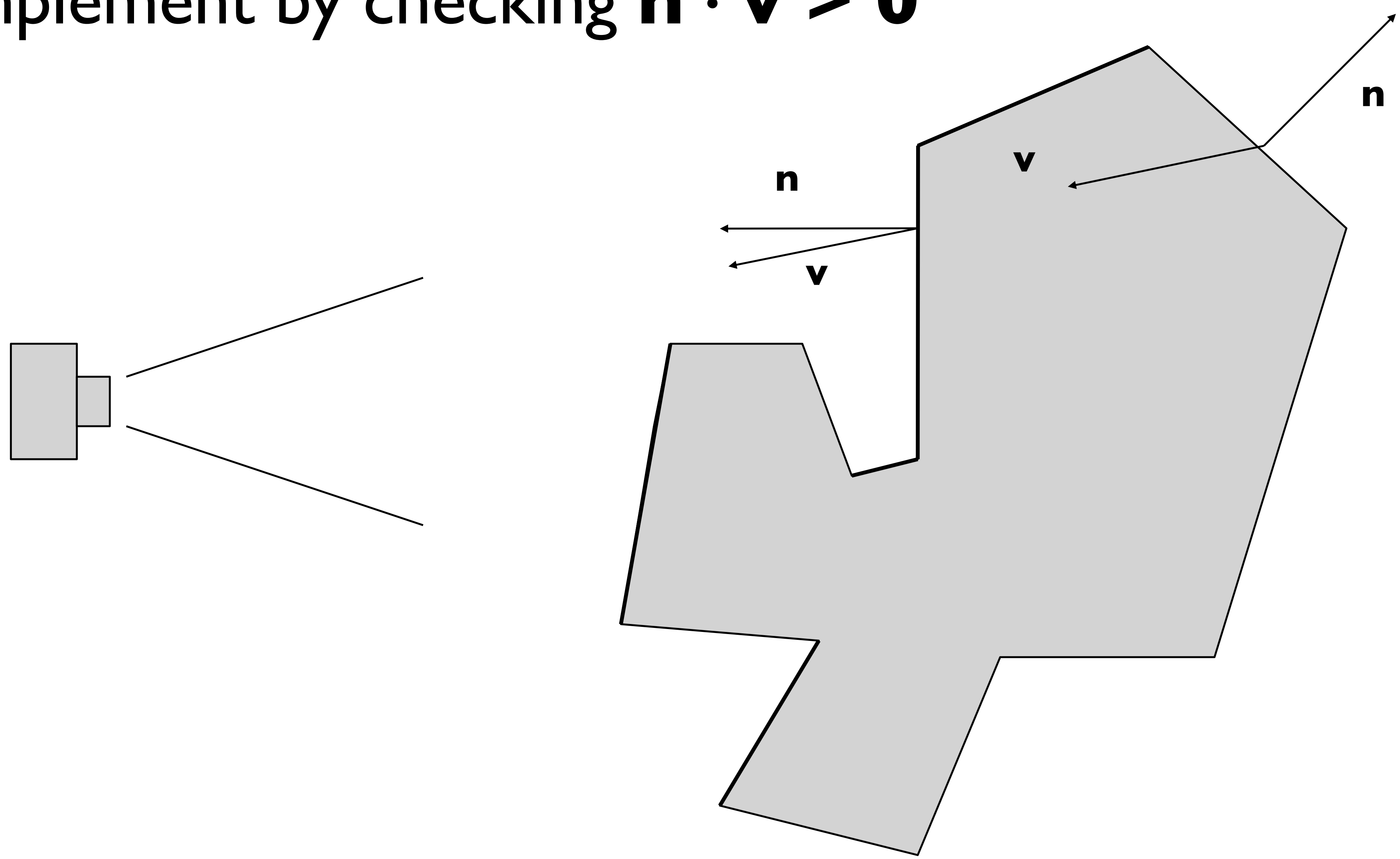


Last time

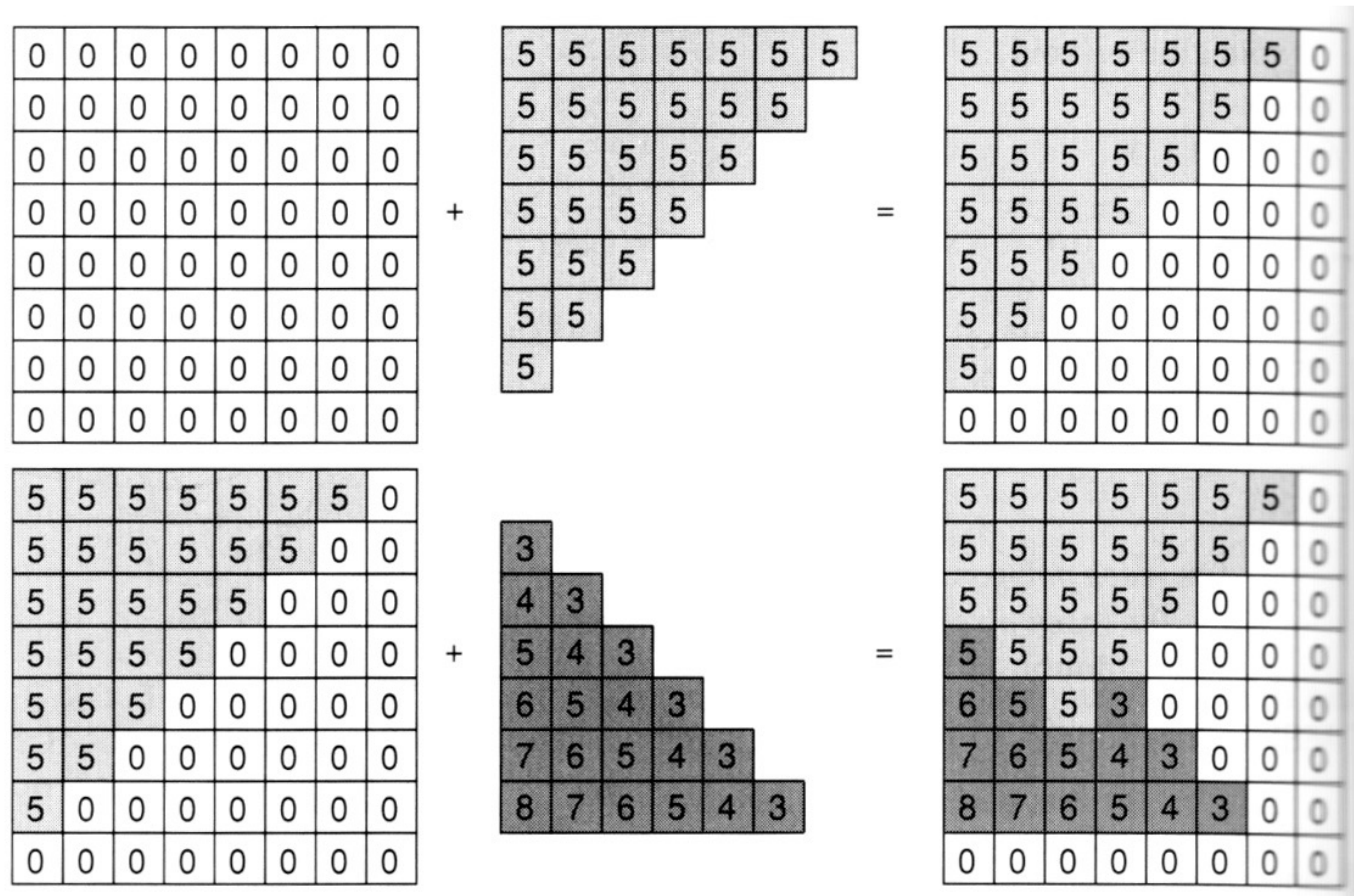

you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
user sees this
DISPLAY
Backface culling
Clipping
Z buffering

Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - implement by checking **n · v > 0**



The z buffer



[Foley et al.]

- another example of a memory-intensive brute force approach that works and has become the standard
- store z as an integer for speed and memory efficiency (at the expense of precision!)

OpenGL: Nowadays

OpenGL: Nowadays

- Send buffers full of data to GPU

OpenGL: Nowadays

- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)

OpenGL: Nowadays

- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is location in **clip space**)

OpenGL: Nowadays

- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is location in **clip space**)

= normalized device coordinates

OpenGL: Nowadays

- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**

OpenGL: Nowadays

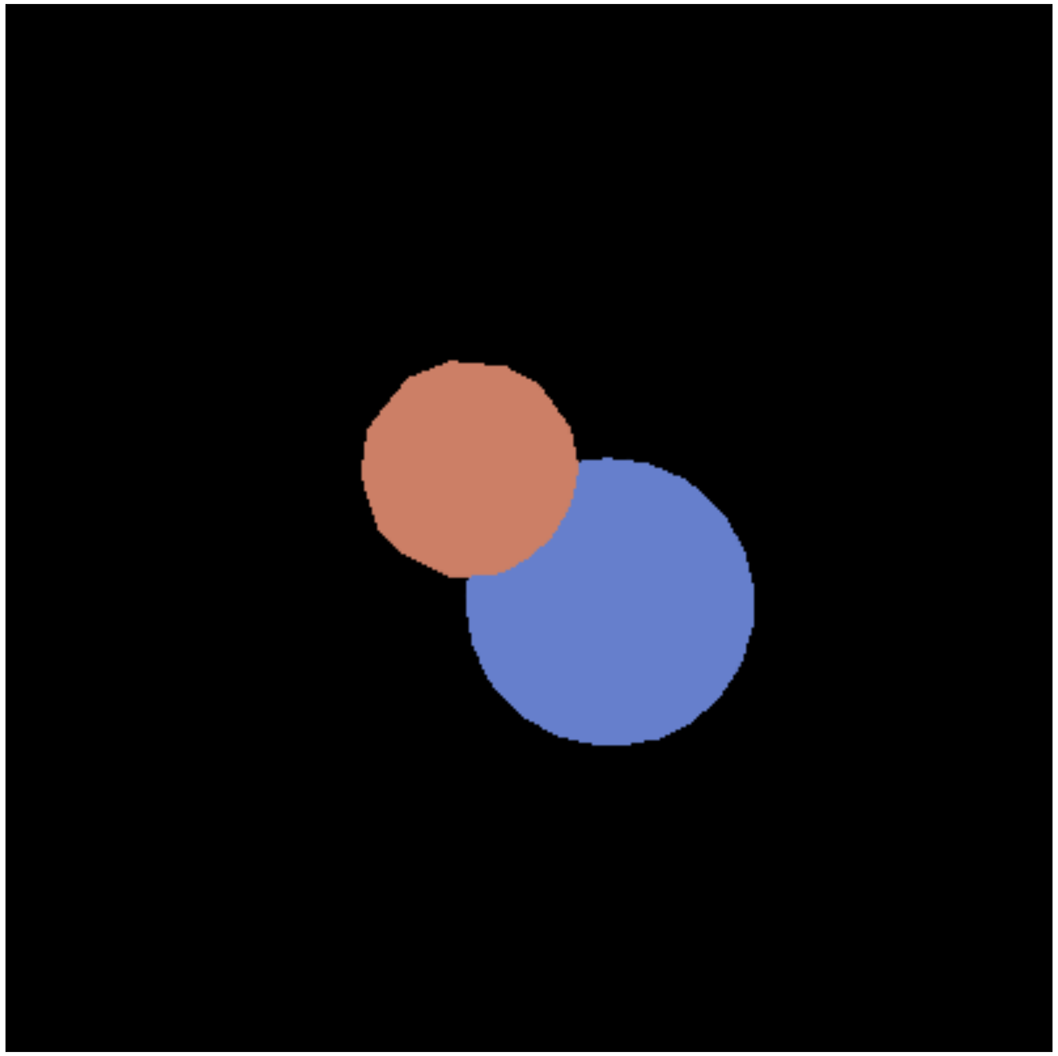
- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.

OpenGL: Nowadays

- Send buffers full of data to GPU
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

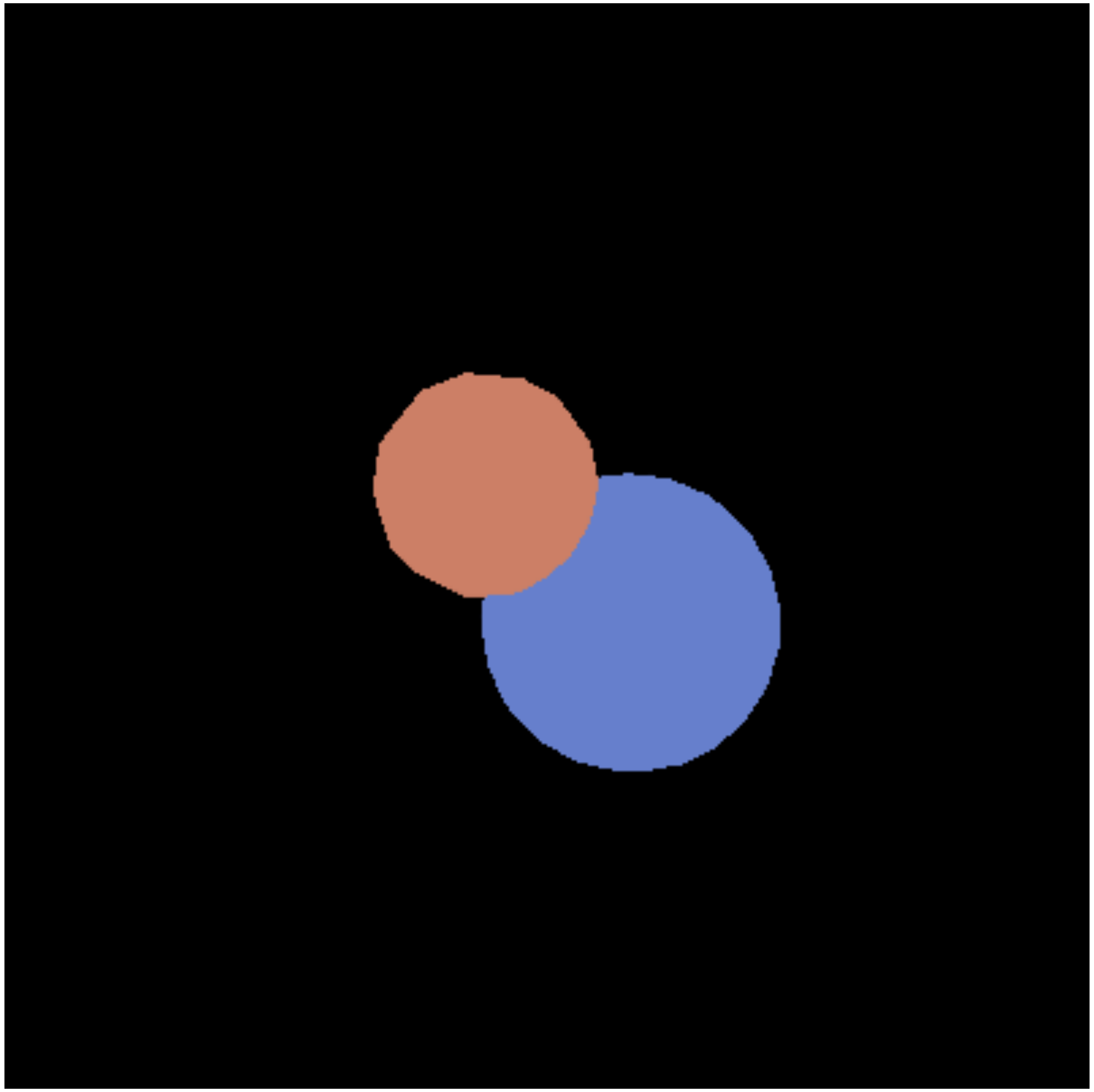
Pipeline for minimal operation

- Vertex stage (input: position / vtx; color / tri)
  – transform position (object to screen space)
  – pass through color
- Rasterizer
  – pass through color
- Fragment stage (output: color)
  – write to color planes



Result of minimal pipeline

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci480_20w/pipeline_demo/



OpenGL: Your job, conceptually

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is **location in clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- Execute custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine is **location in clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- Execute custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)

(write vertex shader)

- GL executes custom-written **vertex shader** program on each vertex (to determine is **location in clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- Execute custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)

(write vertex shader)

- GL executes custom-written **vertex shader** program on each vertex (to determine is **location in clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**

(write fragment shader)

- Execute custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

Terminology, so far

- Clipping
- Rasterization
- Interpolation
- Fragment
- Shader

WebGL: Your Jobs

- Send geometry
- Write a vertex shader
- Write a fragment shader

WebGL: Your Jobs

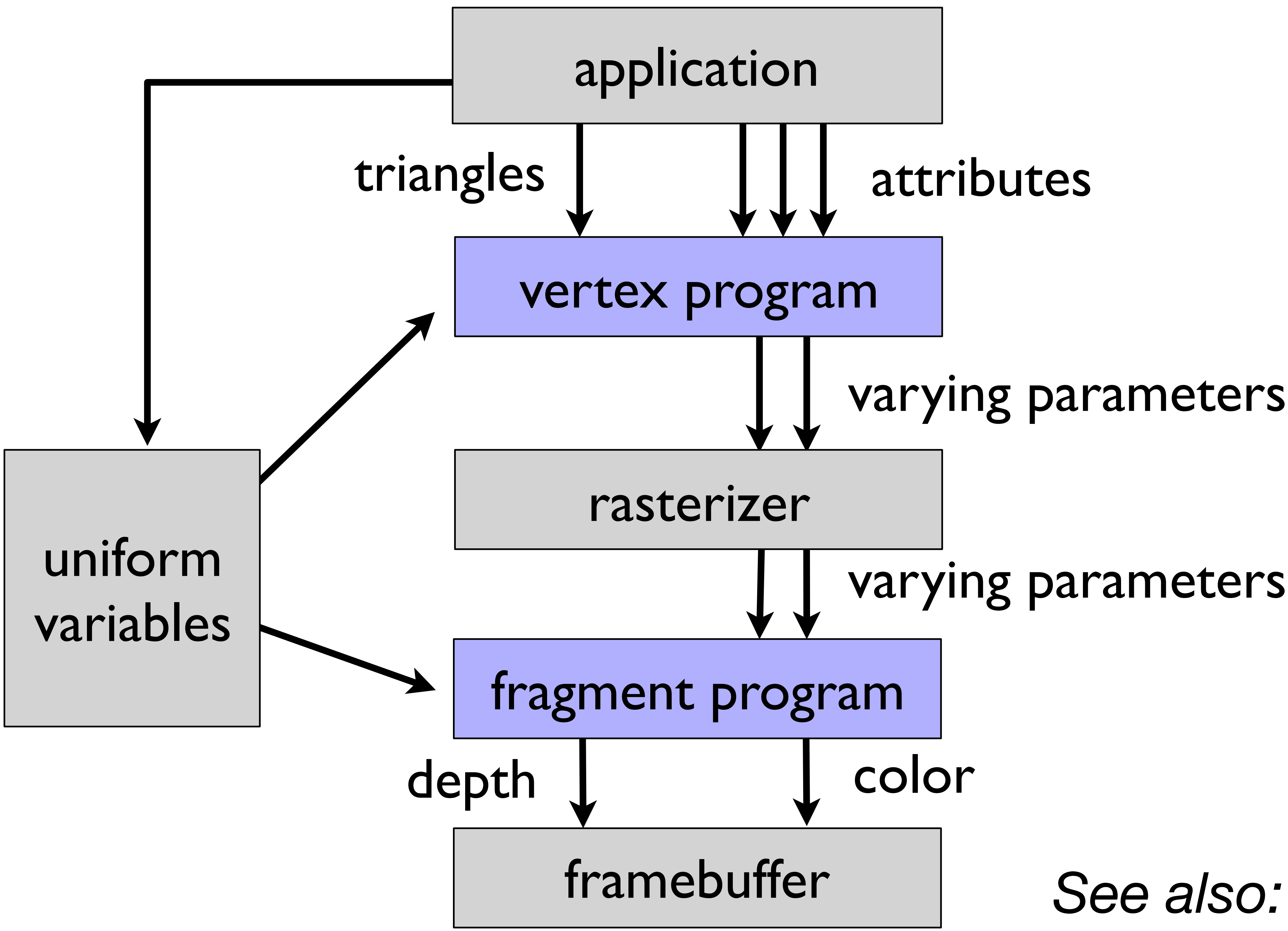
- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

WebGL: Your Jobs

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

WebGL Data Plumbing: Overview



See also: today's lecture notes

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

A first pass at the lab code...

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

okay so we saw some unfamiliar words in there:

buffer
attribute

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

okay so we saw some unfamiliar words in there:

buffer
attribute

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

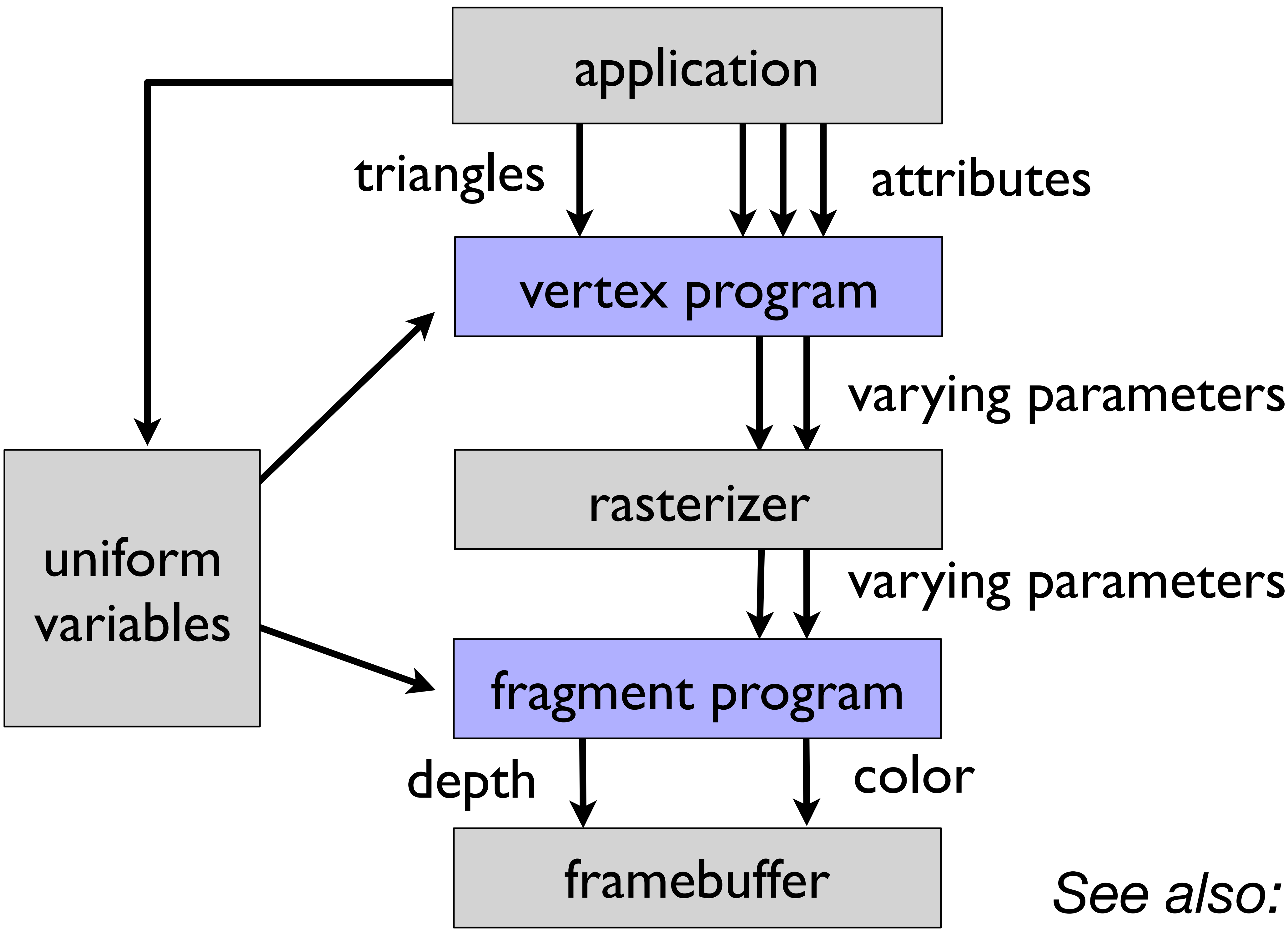
in **GLSL**, the GL shader language

A first pass at the lab code...

okay so we saw some unfamiliar words in there:

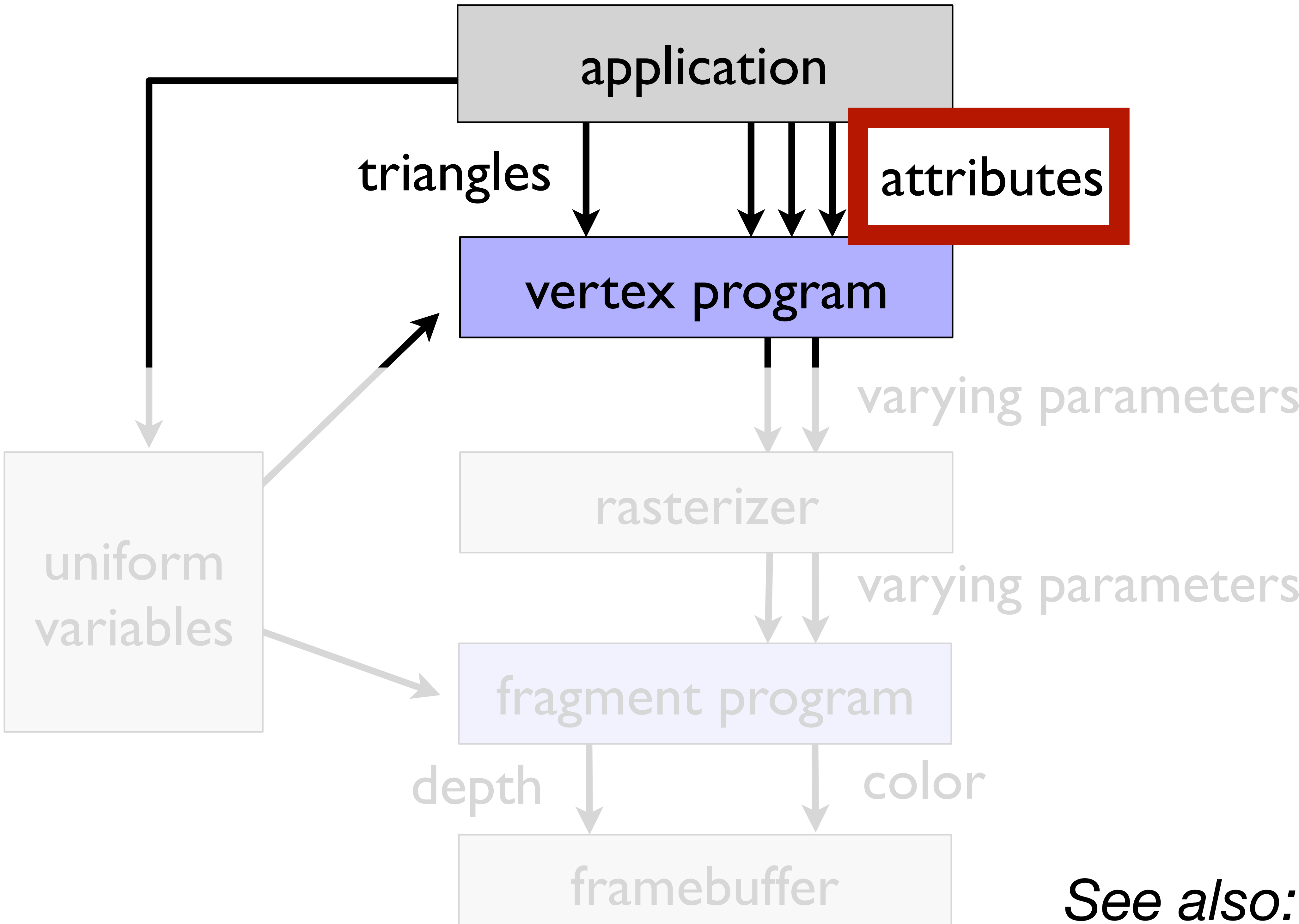
buffer
attribute

WebGL Data Plumbing: Overview



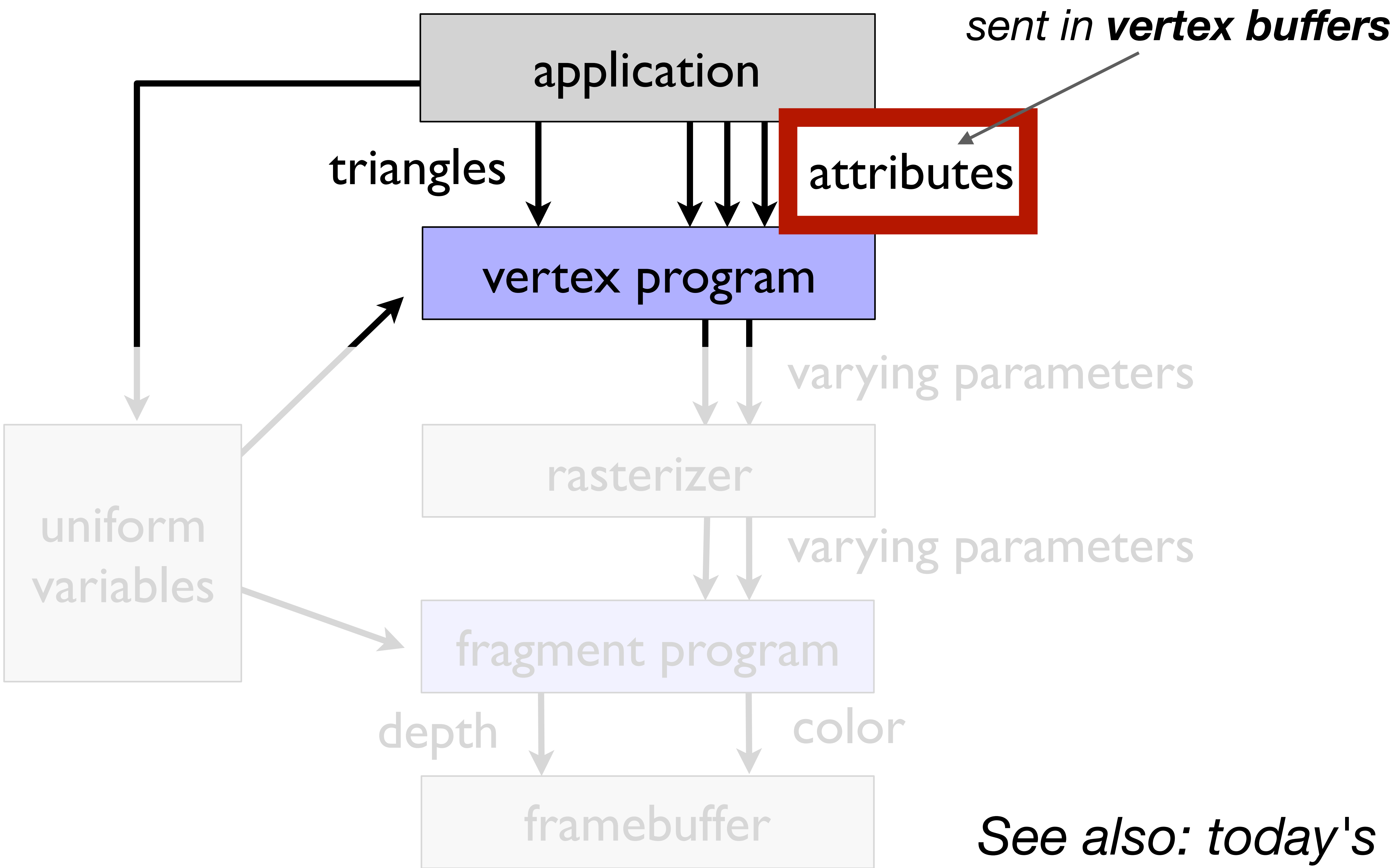
See also: today's lecture notes

WebGL Data Plumbing



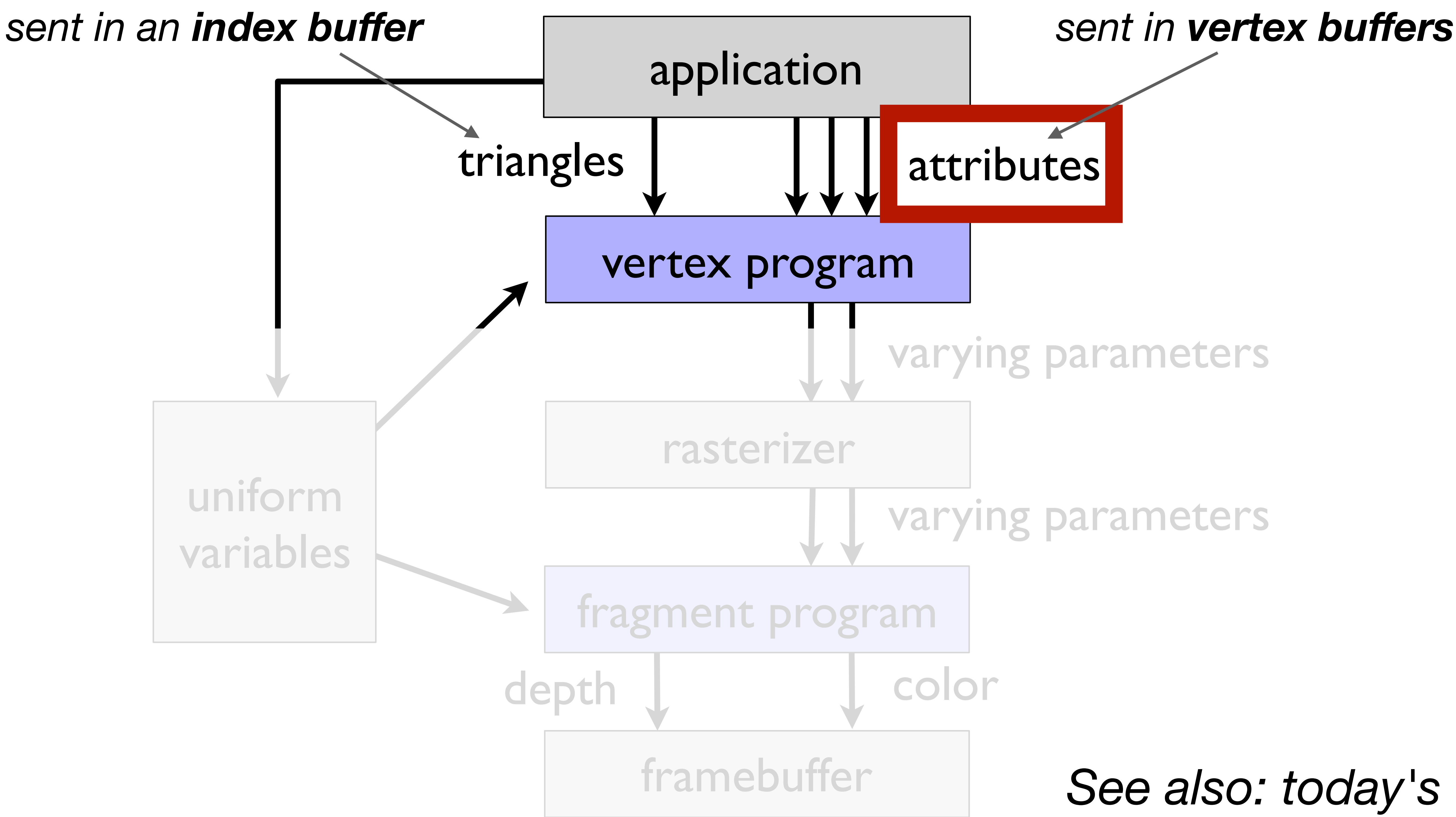
See also: today's lecture notes

WebGL Data Plumbing



See also: today's lecture notes

WebGL Data Plumbing



See also: today's lecture notes

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

WebGL: Hello, Triangle!

- Send geometry by calling `gl` functions
- Write a vertex shader
- Write a fragment shader

in **GLSL**, the GL shader language

A first look at the shader code...

Shader Responsibilities

The **vertex shader's job** is to:

- assign a value to **`gl_Position`**,
  which specifies the vertex's position

- assign values to any **varying** parameters needed later

The **fragment shader's job** is to:

- assign a value to **`gl_FragColor`**,
  which specifies the fragment's color

GLSL - GL Shader Language

- A C-like mini-language
- Basic program looks like:

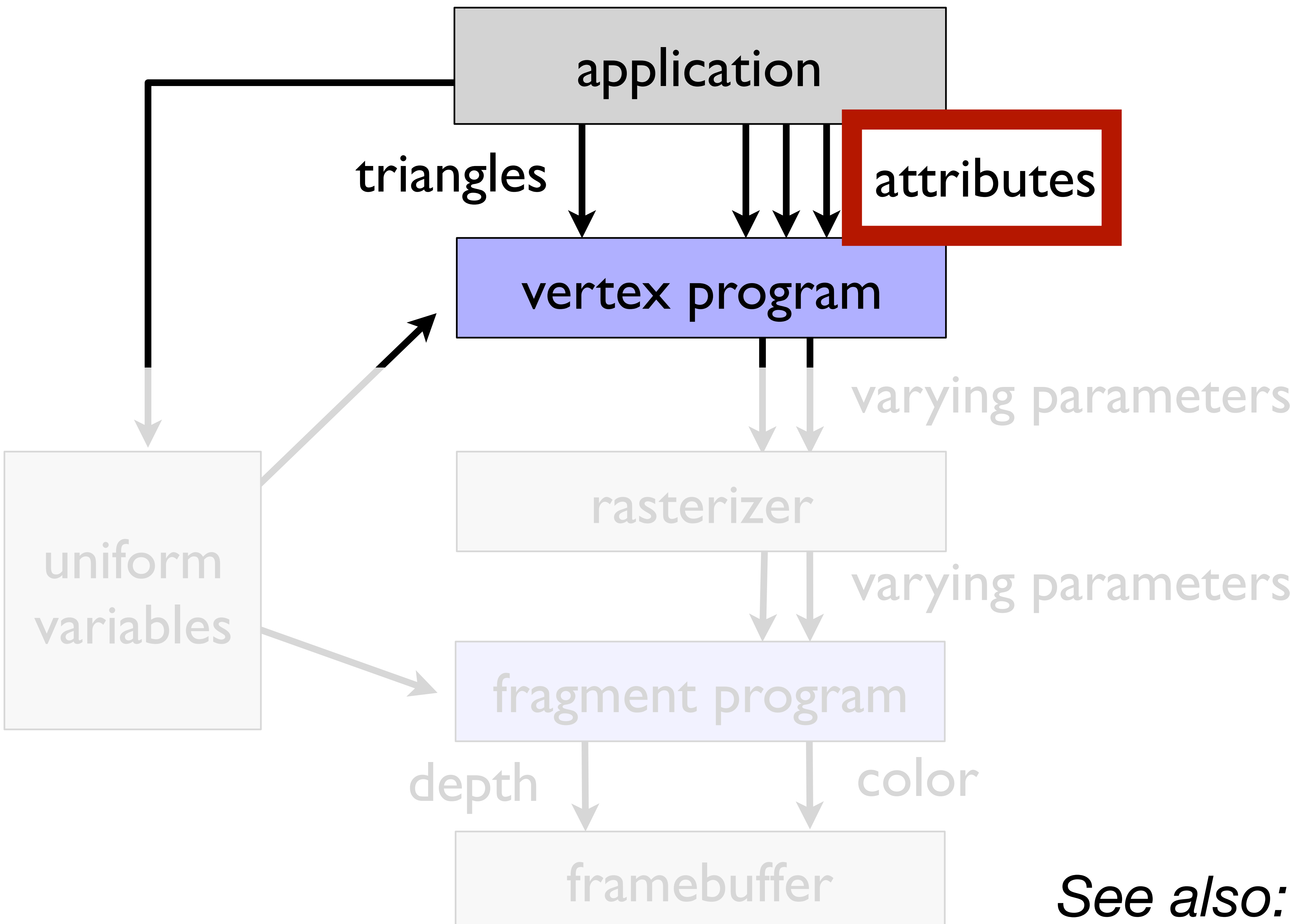
```
// some declarations

void main() {
    // main program
}
```

- Built-in types for small vectors/matrices (e.g., `vec3`, `mat4`)

Task 1: Turn the triangle black

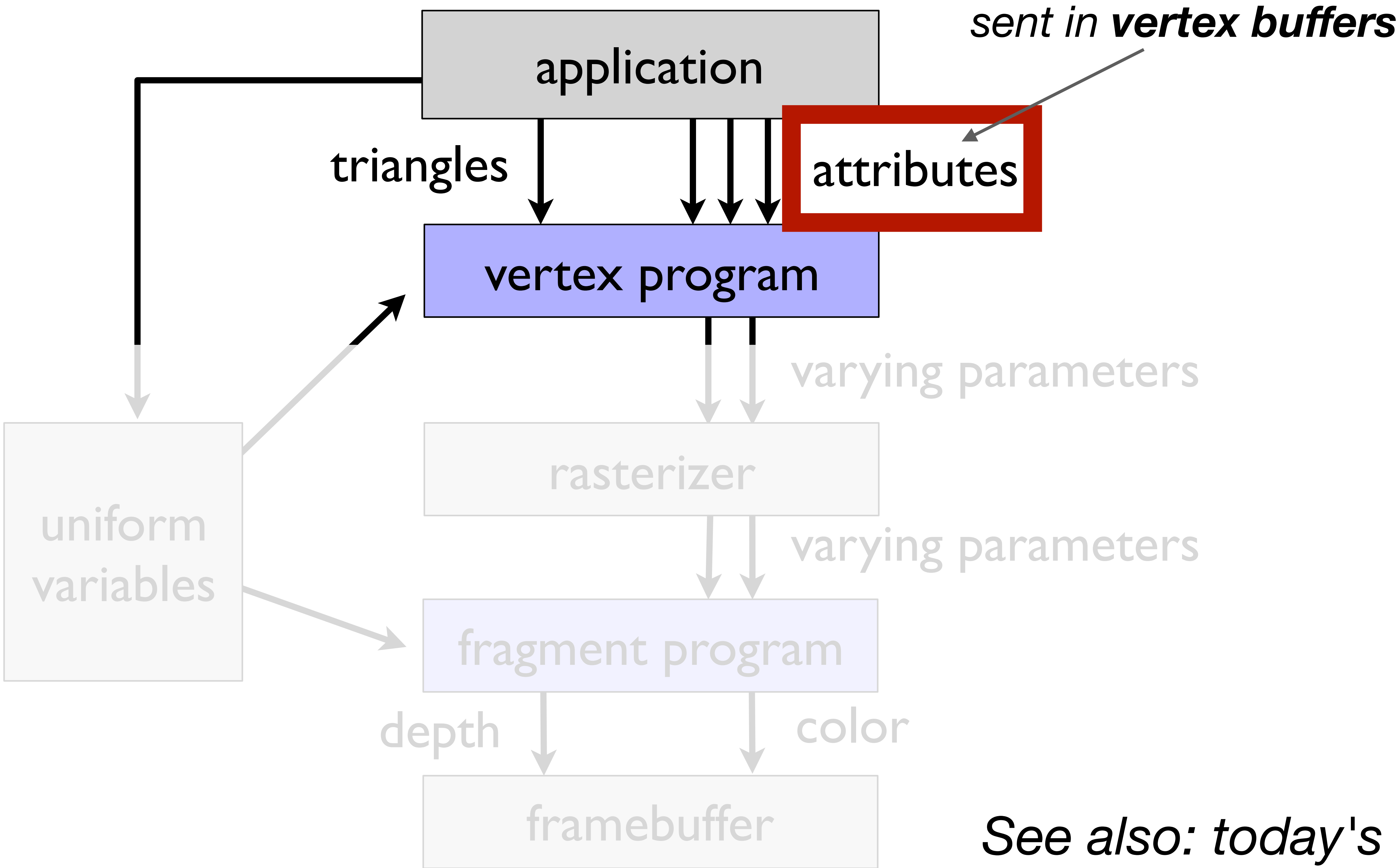
- Change the fragment shader's source code to set the triangle color to black instead of white.
- *Note*: colors are vec4s; the 4th channel is transparency ("alpha"):
  - 0.0 is fully transparent, 1.0 is fully opaque

WebGL Data Plumbing



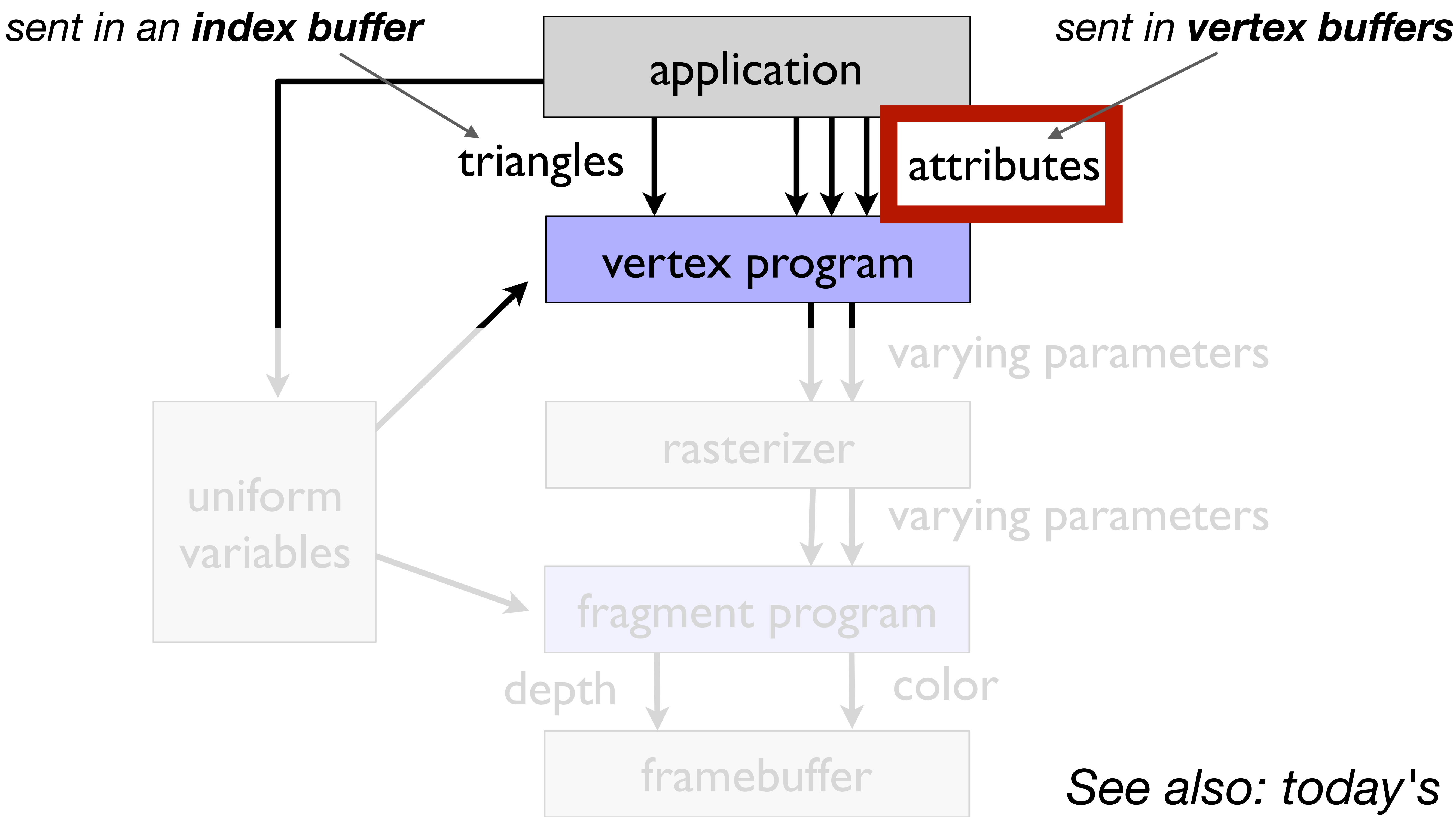
See also: today's lecture notes

WebGL Data Plumbing



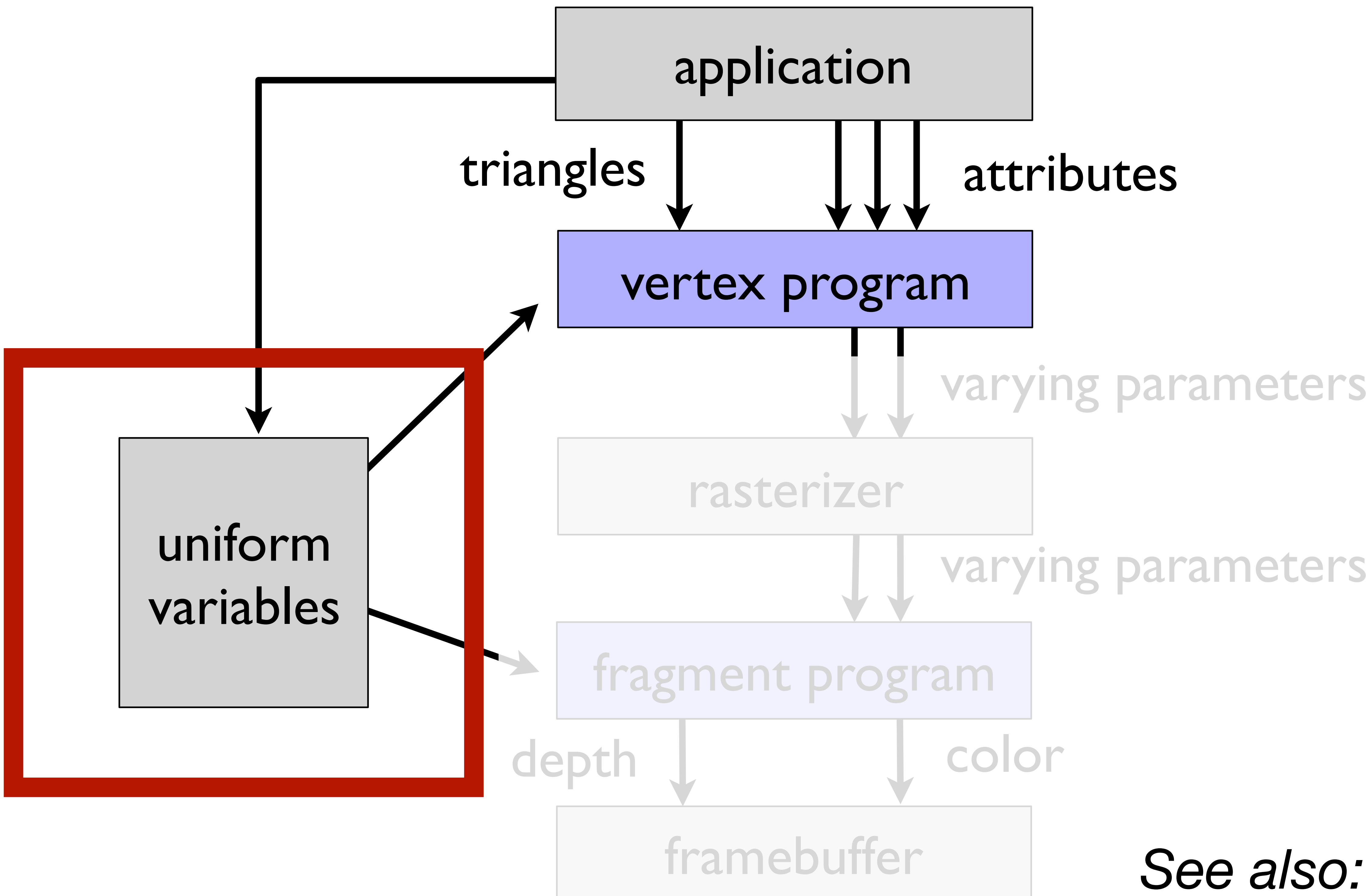
See also: today's lecture notes

WebGL Data Plumbing



See also: today's lecture notes

WebGL Data Plumbing



See also: today's lecture notes

GLSL - GL Shader Language

- Built-in types for small vectors/matrices (e.g., `vec3`, `mat4`)
- Multiplication on the above types does matrix multiplication:

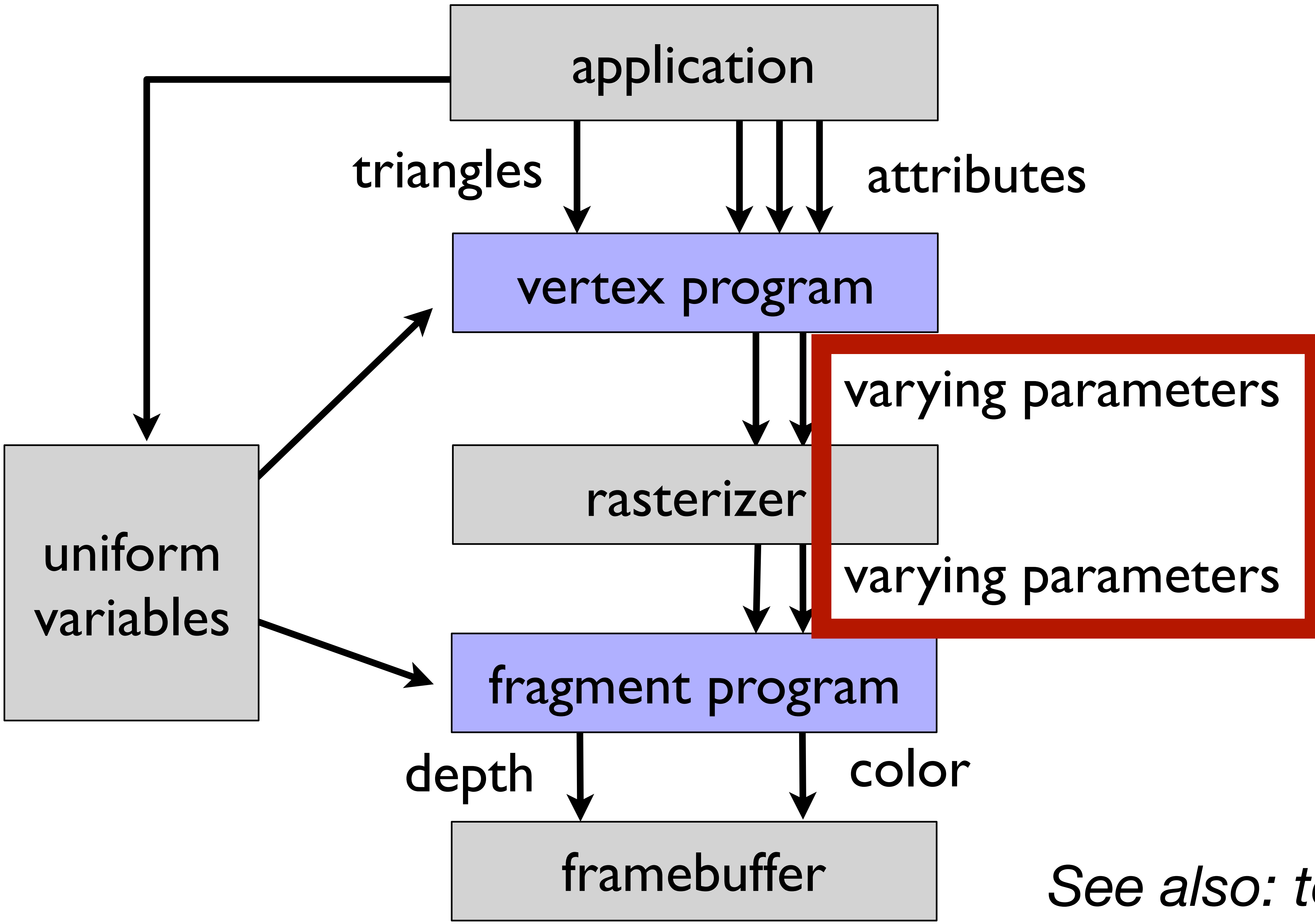
```
// GL matrices are in column-major order
mat2 A = mat2(1.0,2.0,3.0,4.0);
vec2 x = vec2(1.0, 0.0);

vec2 a = A * x; // a = (1,2)
```

Task 2: Add a uniform

- Add a uniform variable called `Matrix` containing a 4x4 matrix

- In the vertex shader, multiply the `Position` attribute of the vertex by the `Matrix` to move the triangle vertices.

Terminology: data plumbing



See also: today's lecture notes

GLSL - GL Shader Language

- `varyings` are declared in both the Vertex shader and in the Fragment shader.
  - The vertex shader sets their values for each vertex, then the rasterizer **interpolates** their values for each fragment and passes to the fragment shader.
- By convention, `varying` names are usually chosen to begin with `v`, such as `vColor` or `vNormal`

Task 3: Add a varying

- Set up a `varying` parameter to set the color at each vertex
- Use the interpolated values in the fragment shader to set each fragment's color.