Shaders and the Graphics Pipeline

CS 385 - Class 2
27 January 2022

Computer Graphics: What's it All About?

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

In computer graphics, the process
of ***determining*** something
is called ***shading***

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

In computer graphics, the process
of ***determining*** something
is called ***shading***
and the thing that
does the ***shading***
is called a
shader

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

It's All About the Colors

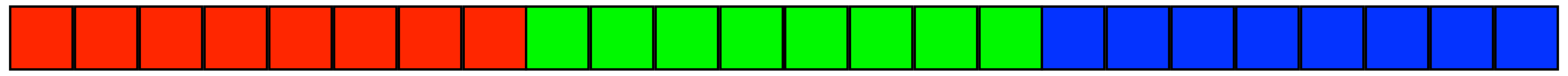
- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

Those dots are,
of course,
pixels

Pixels

- Pixels most commonly store colors
- Recall colors in computer graphics are normally described as RGB triples
  - red, green, and blue
- GPUs normally store colors as RGBA
  - A is alpha, a measure of translucency
- A pixel's size is measured in *bits per pixels*
  - providing the size of each component



5-6-5	five bits red and blue, six bits green
24	eight bits for each component
32	either, eight bits for RGBA, or 11-11-10 for RGB

It's All About the Colors

- Simply put, the principal motive of computer graphics is

to determine the colors of the dots on the screen

Those dots are,
of course,
pixels
which are grouped
together to form
a ***framebuffer***

Framebuffers

- Rectangular collection of pixels that can be read and written from a program
- Framebuffer's size is called its *resolution*
  - defined as width x height
  - e.g., 1080p HD TVs have a framebuffer resolution of 1920 x 1080
- It's the same concept as an image's resolution
  - images are written to a file
  - framebuffers are stored in GPU's memory
    - and only last while the application's using them

* NTSC and PAL/SECAM are analog video standards with no fixed horizontal resolution. The resolutions depicted here for 4:3 aspect ratio assume square pixels, but the actual horizontal resolution (in non-square pixels) ranges from 320 (VHS/Betamax) to 720 (DVD) for both standards.

** Although computer industry defines SIF as 384x288 for PAL countries (for NTSC countries, it is equivalent to QVGA), MPEG-1 defines it as 352x288 (CIF) or 360x288 (for NTSC: 352x240 or 360x240).

† The resolutions depicted here for 16:9 widescreen NTSC and PAL/SECAM assume square pixels, but the actual horizontal resolution (in non-square pixels) ranges from 520 (PALplus) to 720 (DVD).

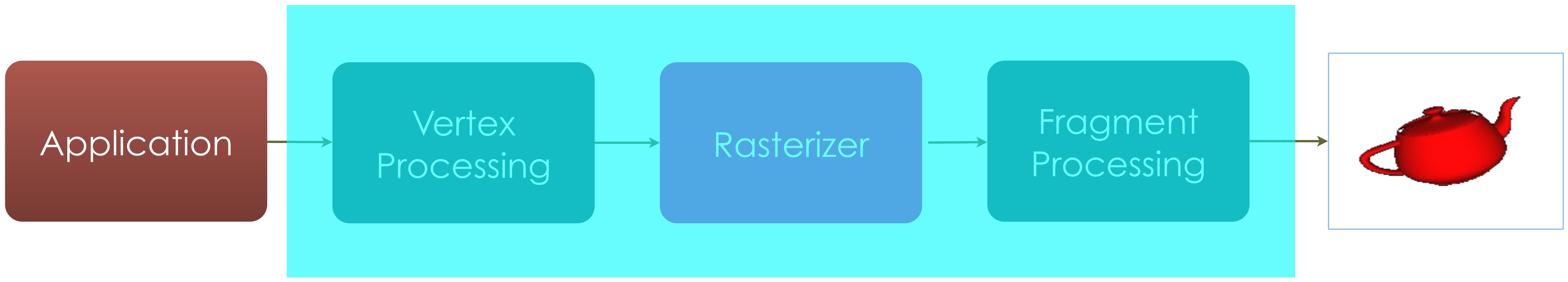
‡ Although Digital Cinema System specifies 2K at the depicted resolution in square pixels, in some situations it can assume non-square pixels and go as high as 2048 x 1536. Aspect ratio 17:9 is approximate.

https://en.wikipedia.org/wiki/Display_resolution#/media/File:Vector_Video_Standards8.svg

The Graphics Pipeline

Graphics Pipeline

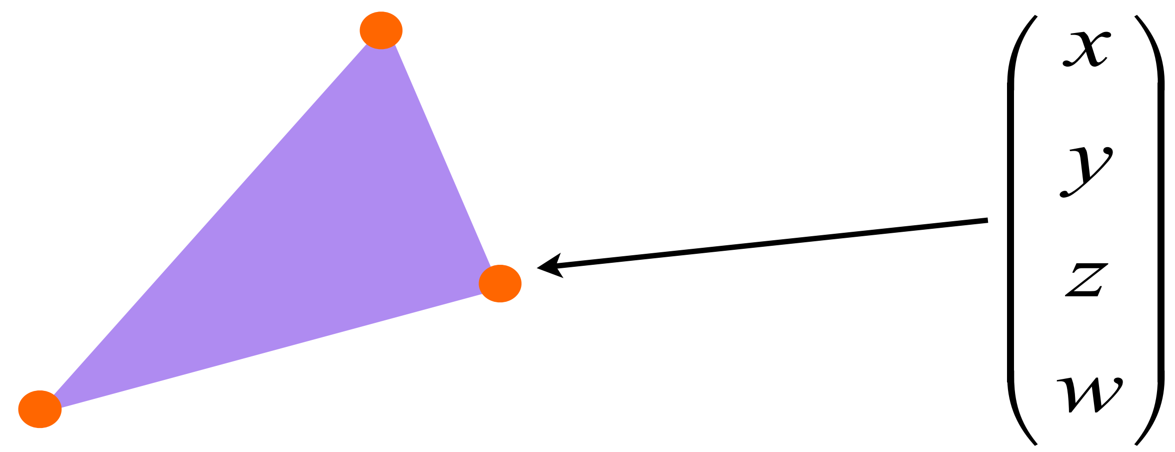
Graphics Pipeline



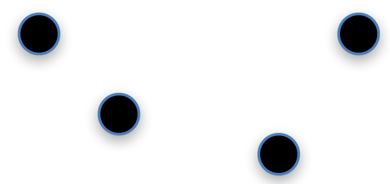
This part happens on the GPU

Vertices and Geometry

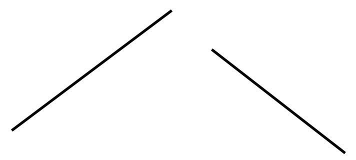
- Geometric objects are represented using *vertices*
- A vertex is a *collection* of generic *attributes*, but must include a position
  - positions are represented as a 4-dimensional homogenous coordinate
  - other attributes can be of any types
- Vertex information must be stored in *vertex buffers*



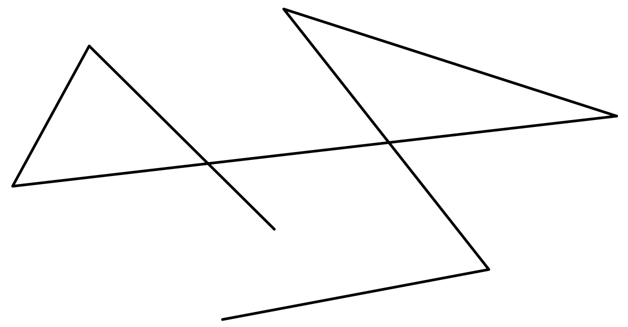
WebGL Geometric Primitives



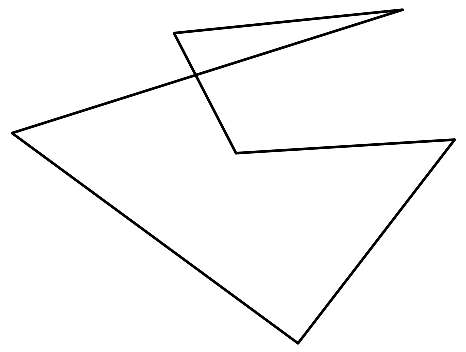
gl.POINTS



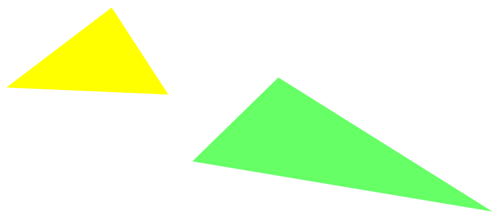
gl.LINES



gl.LINE_STRIP

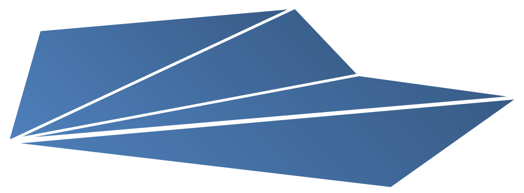


gl.LINE_LOOP



gl.TRIANGLES

gl.TRIANGLE_STRIP

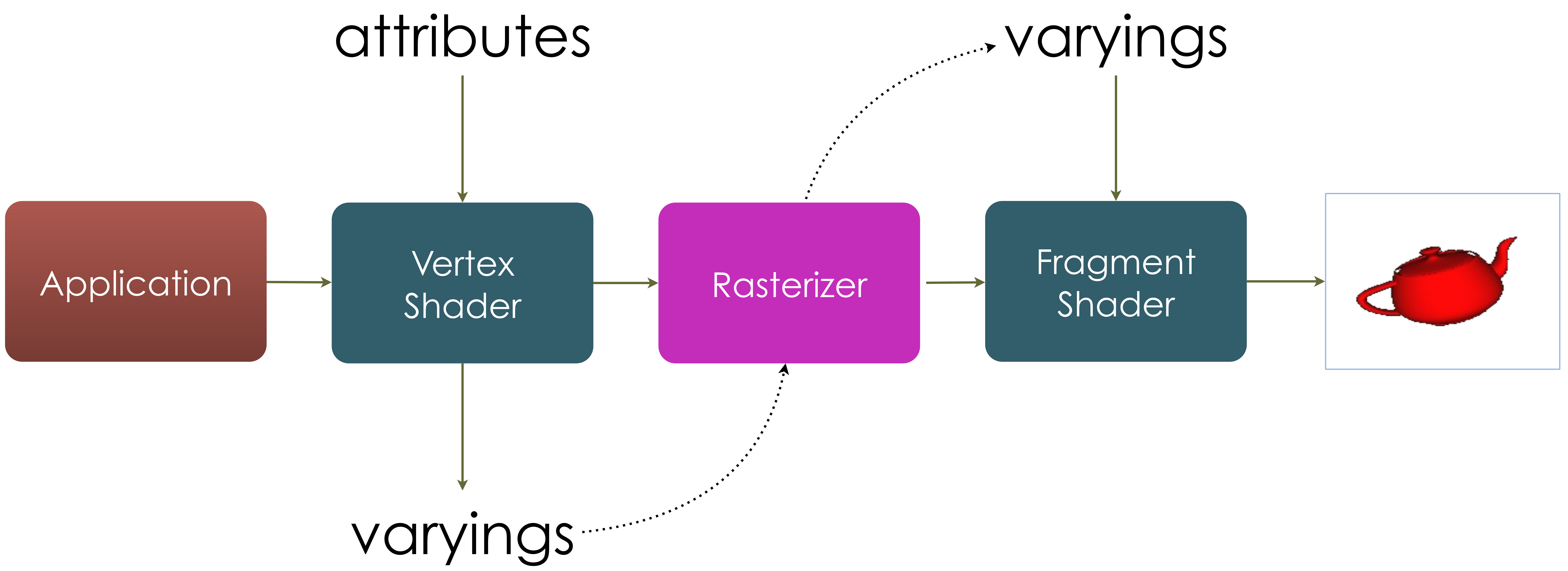


gl.TRIANGLE_FAN

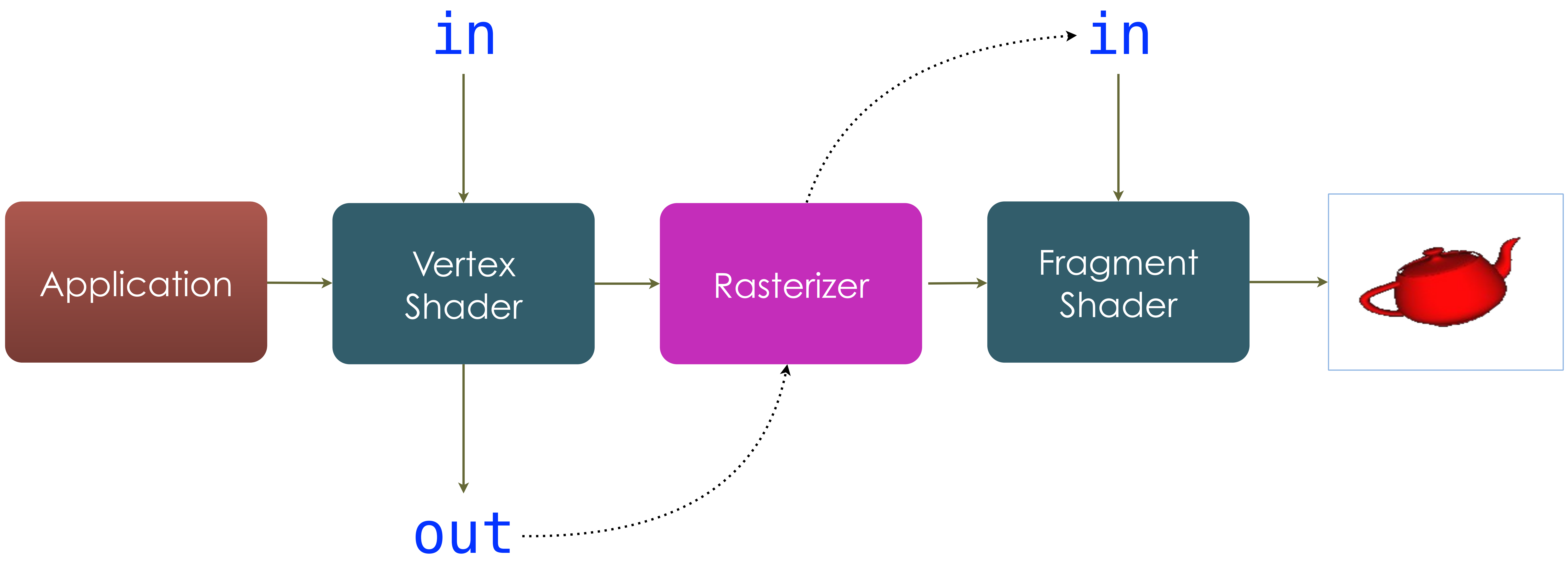
Shaders

Graphics Pipeline

Data Flow Between Shader Stages



Shader Keywords



In WebGL 1.0, `in`s were labeled `attribute`, `out`s were `varyings`

Vertex Shaders

- Every vertex is processed by a vertex shader
  - an application will likely have many vertex shaders
    - only one can be active at a time
- A vertex shader is required for rendering
- Vertex attributes are tagged with the keyword `in`
- Varying values are are tagged with the keyword `out`

```
in   vec4 aPosition;
out  vec4 vColor;

void main()
{
  vColor = vec4(0.0, 0.0, 1.0, 1.0);
  gl_Position = aPosition;
}
```

Vertex Shaders

- `gl_Position` is an implicitly defined varying variable for every vertex shader

```
out vec4 gl_Position;
```

- Every vertex shader must assign a value to `gl_Position`
- You pass additional information from the vertex shader to the fragment shader using *user-defined varyings*

```
in   vec4 aPosition;
out  vec4 vColor;

void main()
{
  vColor = vec4(0.0, 0.0, 1.0, 1.0);
  gl_Position = aPosition;
}
```

Variable Naming Conventions

- Our examples will use a naming convention for shader variables
- Specify the source and consumer of vertex and fragment data

Prefix	Data Producer	Data Consumer	Example
`a`	Application	Vertex Shader	`aPosition`
`v`	Vertex Shader	Fragment Shader	`vColor`
`f`	Fragment Shader	Framebuffer	`fColor`

Vertex Shaders and HTML

- For HTML, a vertex shader is an additional type of script
- Just "wrap" your shader code in a pair of `<script>` tags
- Name the shader with its *id* attribute
  - we'll use this name later to load the shader
- Specify its type using the *type* attribute
  - use `x-shader/x-vertex` for vertex shaders

```
<script id="vertex-shader"
   type="x-shader/x-vertex">
in  vec4 aPosition;
out vec4 vColor;

void main()
{
  vColor = vec4(0.0, 0.0, 1.0, 1.0);
  gl_Position = aPosition;
}
</script>
```

Fragment Shaders

- Every fragment is processed by a fragment shader
- A fragment shader is required for rendering
  - however, there are a few exceptions (for advanced uses)
- The current fragment shader must write to an `out` tagged variable

```
out vec4 fColor
```

- Fragment shaders also require information about variable precision

```
precision highp float;

in   vec4 vColor;
out  vec4 fColor;

void main()
{
  fColor = vColor;
}
```

Fragment Shaders

- Fragment shaders also require information about variable precision
- The highlighted line indicates that all *floating-point values* are represented using a particular precision
  - there are three precisions availble:
    - `lowp` - usually 8-bit floating point
    - `mediump` - usually 16-bit floating-point
    - `highp` - usually 32-bit floating point
- It's boilerplate that's required in every fragment shader

```
precision highp float;

in   vec4 vColor;
out  vec4 fColor;

void main()
{
  fColor = vColor;
}
```

Fragment Shaders and HTML

- Virtually the same idea as declaring vertex shaders
- Specify its type using the *type* attribute as `x-shader/x-fragment`

```
<script id="fragment-shader"
    type="x-shader/x-fragment">
 precision highp float;

 in   vec4 vColor;
 out  vec4 fColor;

 void main()
 {
   fColor = vColor;
 }
</script>
```

Shader Programs

- In WebGL, a *shader program* is a compiled collection of shaders
- A *shader* is a (potentially complete) WebGL SL function
- A *program* is a collection of shaders linked together
- We’ll write shaders, but we’ll use programs

Getting Your Shaders into WebGL

- Shaders need to be compiled and linked to form an executable shader program
- WebGL provides the compiler and linker
- A program must contain both a vertex and a fragment shader

CREATE PROGRAM	`gl.createProgram()`
CREATE SHADER	`gl.createShader()`
LOAD SHADER SOURCE	`gl.shaderSource()`
COMPILE SHADER	`gl.compileShader()`
ATTACH SHADER TO PROGRAM	`gl.attachShader()`
LINK PROGRAM	`gl.linkProgram()`
USE PROGRAM	`gl.useProgram()`

These steps need to be repeated for each type of shader in the shader program

That’s a lot of work

- We have a helper JavaScript function `initShaders()` to help
  - provided in the `initShaders.js`
- It does all the nastiness shown in the previous slide
- It takes the *id* names of the vertex and fragment shaders
- After compiling the shaders into a program, we'll use it to control rendering
  - we need to use the program
  - call `gl.useProgram()`

```
var program = initShaders(
    gl,                   // our WebGL context
    "vertex-shader",      // vertex shader id
    "fragment-shader"); // fragment shader id

gl.useProgram(program);
```

Assignment

Lab Activities

1. Set up GitHub account and repository
2. Explore shadertoy.com and chromeexperiments.com