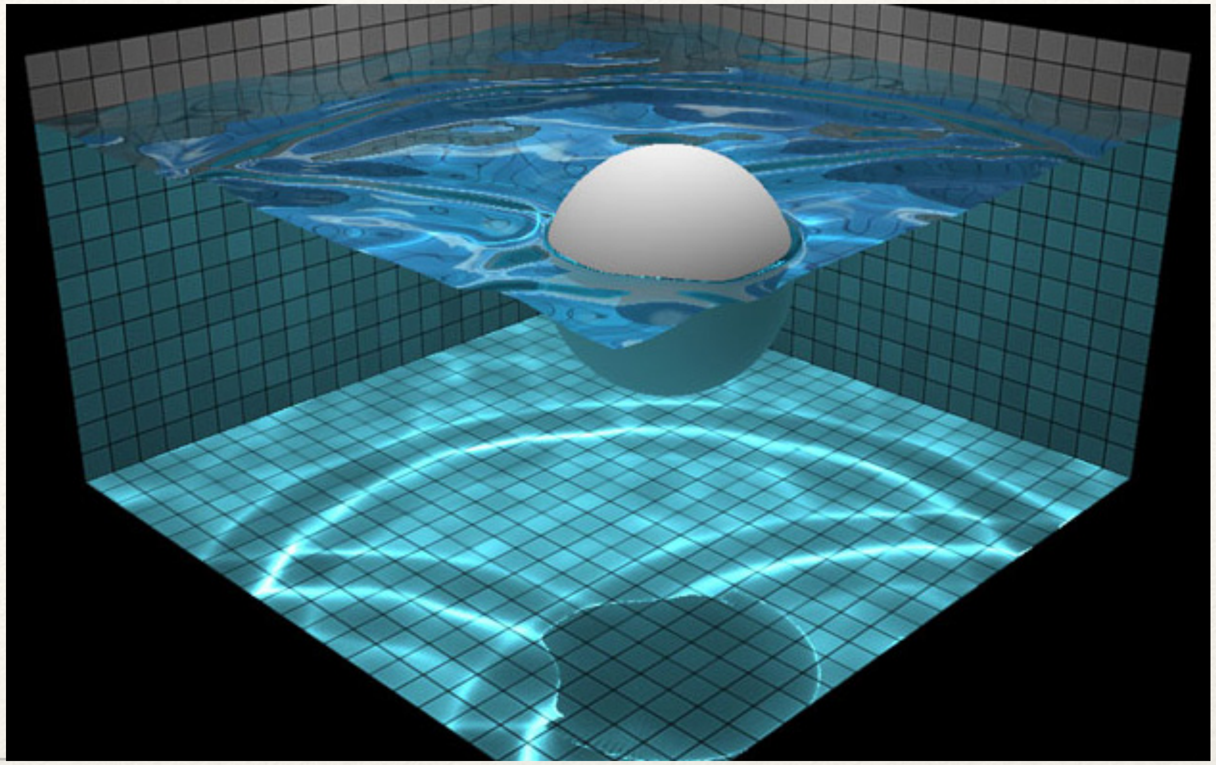


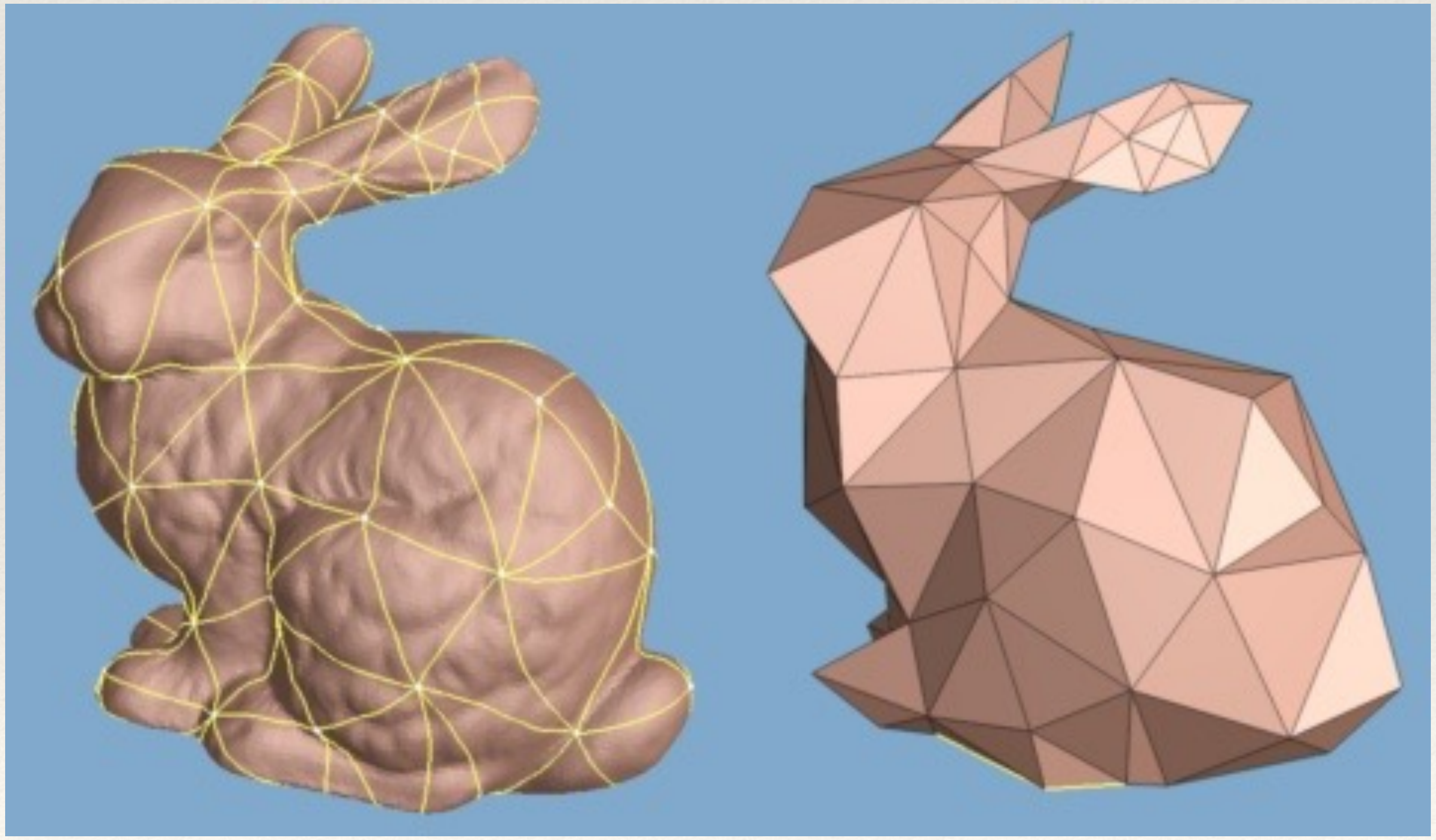
Dr. Sarah Abraham
University of Texas at Austin
Computer Science Department

Introduction to WebGL

Elements of Graphics
CS324e

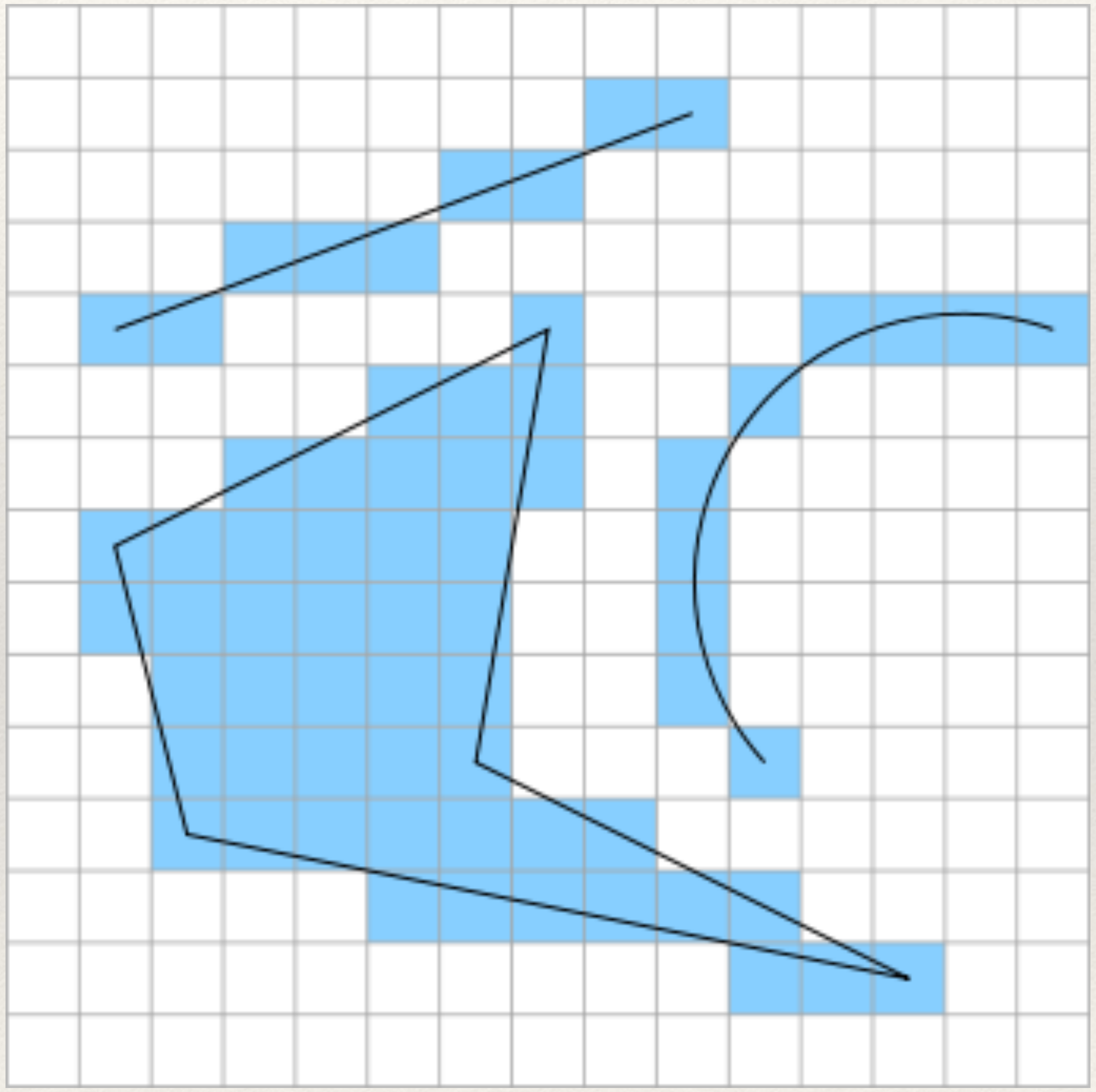
Objects in 3D

- Objects are composed of vertex data
- Vertex data forms "primitives" such as triangles



Rasterization

- Primitives have a color and a position
- Pixels shaded based on these primitive colors and positions



How fast is this process?

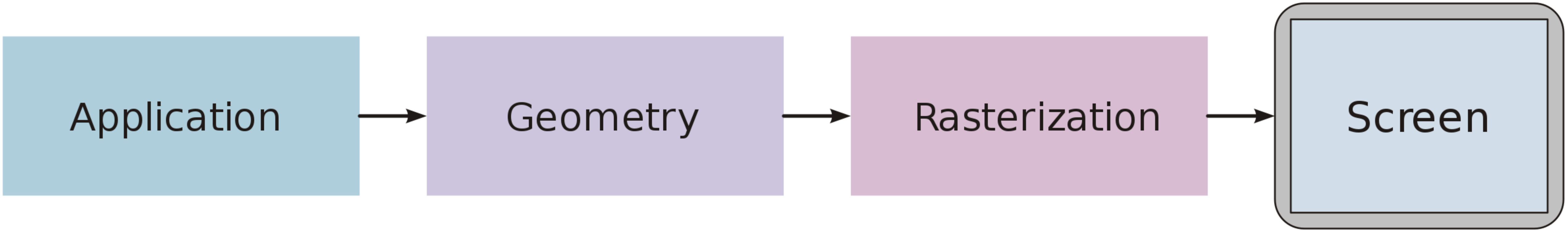
- Highly parallel
    - Each vertex and pixel is not dependent on other vertices and pixels
    - Possible to process all of them at the same time
- Even faster with dedicated hardware support geared toward high parallelization

GPUs

- The Graphics Processing Unit (GPU) is designed for efficient manipulation of 2D and 3D data
- CPUs not effective at processing large numbers of vertices and material information
  - Too slow to render at 60 Hz for large scenes
- Highly parallel for good throughput
- Usually on separate hardware (the graphics card) so data must be bussed from CPU

The Graphics Pipeline

- Application sends scene data from CPU (central processing unit) to GPU (graphics processing unit)
- The GPU transforms the scene information into **geometry**
- The geometry is **rasterized** (converted to image data consisting of color values) based on camera position
- The image data is transformed into the display's **screenspace** based on aspect ratio and display width and height



OpenGL vs WebGL

- Open Graphics Library is API for managing data transfer to the GPU and processing of data on the GPU
    - Low level library in C/C++
    - Microsoft equivalent is DirectX
- WebGL is equivalent API for running in a web browser
    - Library in Javascript
    - Simplified instruction set (similar to OpenGL ES for mobile graphics)
    - Runs in an HTML5 Canvas

How do we tell the GPU what to do?

Shaders

- Programs that run on the GPU
- Used to determine how to render vertices to screen
  - Vertex shader
- Used to determine how to color objects on the screen
  - Fragment shader

Using WebGL

- Create and compile **shaders**
  - Determines how to process vertices of model into pixels
- Create a **canvas**
  - Determines where the program should render out the models into pixels
- Create a WebGL script that uses the shaders to draw to the **canvas context**
  - Defines the model data and which shaders they use

Creating a Canvas

- Canvas element used to draw graphics via Javascript
  - Equivalent to the canvas in Processing
  - Can draw on the canvas in 2D or 3D (WebGL)
- To use WebGL, must embed a Canvas element into the html:

```
<html>
  <body>
     <canvas id=“helloworld” width=“800” height=“600”>
    </canvas>
  </body>
</html>
```

Initializing WebGL

- Access the canvas' WebGL **context**
- The context manages the current **state** of the graphics environment
  - Context issues commands to graphics state and passes values to GPU
- Context hidden by Processing but still present!
  - Where have we seen the context in action in Processing?

GL Context

```
function initGL(canvas) {
  gl = canvas.getContext(“webgl");
  if (!gl) {
   console.log("WebGL not available”);
  }
  gl.viewportWidth = canvas.width;
  gl.viewportHeight = canvas.height;
}
```

Creating a Buffer

- Create a **buffer** using `context.createBuffer()`
- Specify the type of resource the buffer represents using `context.bindBuffer(target, buffer)`
  - `target` is the location of the type of resource
  - `buffer` is the buffer to be associated with `target`
- Provide data to be stored in the buffer as a Javascript array
- `context.bufferData(target, data, usage)` takes the `data`, associates it with the `target` and specifies how the `data` is to be used

Consider

- Where have we seen buffers in Processing?
- What parts of these buffers are hidden from us in Processing and why?
- How does this effect the usability of Processing?

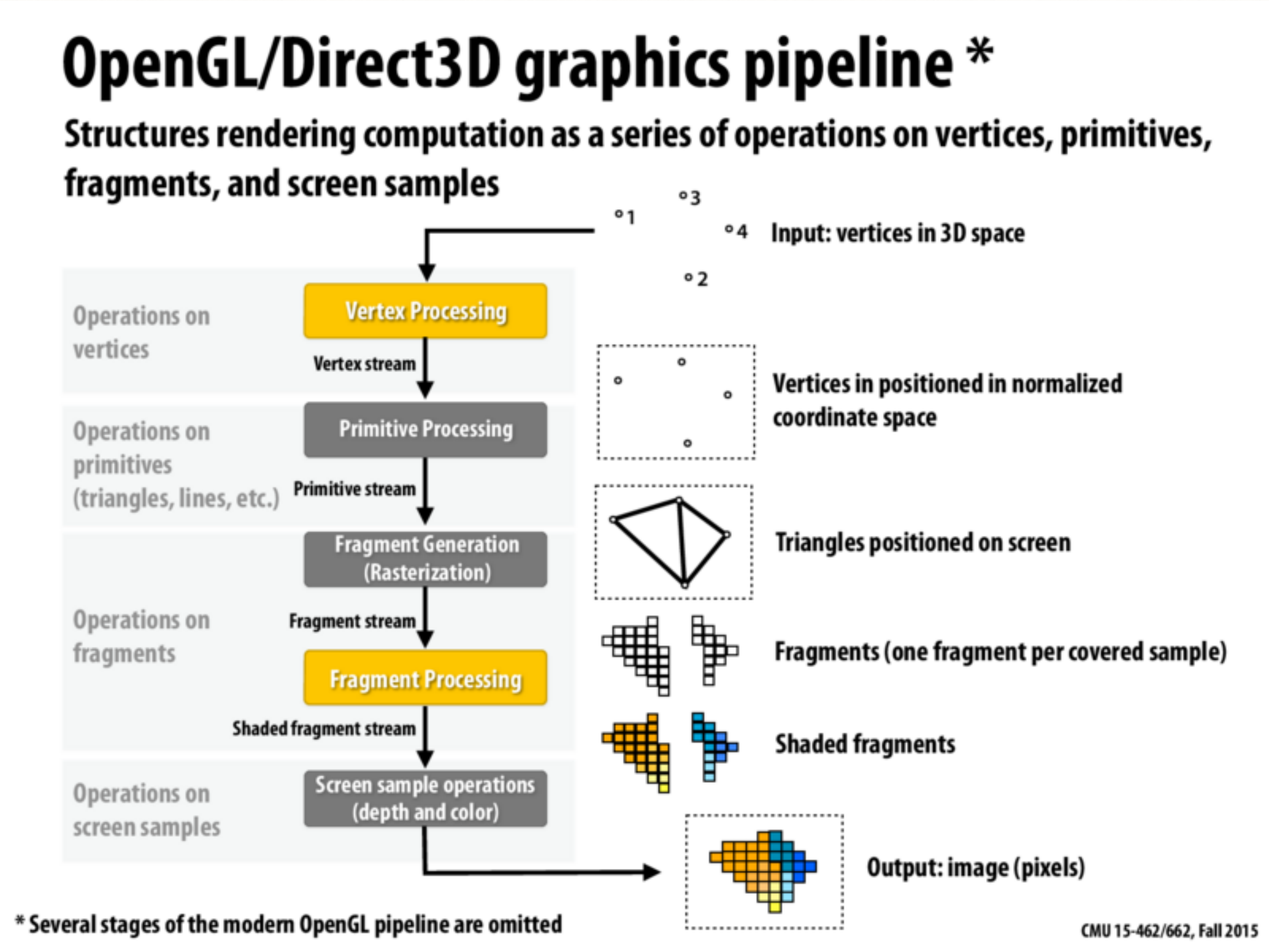
Passing Buffers to Shaders

- Shaders linked to the graphics context using **programs**
  - `program = context.createProgram();`
  - `context.attachShader(program, shaderProgram);`
  - `context.linkProgram(program);`
- When it's time to use a shader on some given data, we then call `context.useProgram(program)`
- `context.drawArrays(mode, first, count)` will run the current shader program on its associated buffer data
  - Must specify type of primitive to process (points, lines, triangles, etc) using mode
  - First defines where in the buffer to start
  - Count tells shaders how many times to execute their code

WebGL and Shaders

- WebGL is primarily the setup to get data to shader programs that run on the GPU
- Initialization phase:
  - Initializes any data that is needed by the shaders
  - Tells shaders where to find that data
- Rendering phase:
  - Sets/updates values needed by the shader
  - Determines what shaders/data to draw every frame

Shader Pipeline



OpenGL/Direct3D graphics pipeline *

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

Input: vertices in 3D space

Operations on vertices

Vertex Processing

Vertex stream

Vertices in positioned in normalized coordinate space

Operations on primitives (triangles, lines, etc.)

Primitive Processing

Primitive stream

Triangles positioned on screen

Operations on fragments

Fragment Generation (Rasterization)

Fragment stream

Fragments (one fragment per covered sample)

Fragment Processing

Shaded fragment stream

Shaded fragments

Operations on screen samples

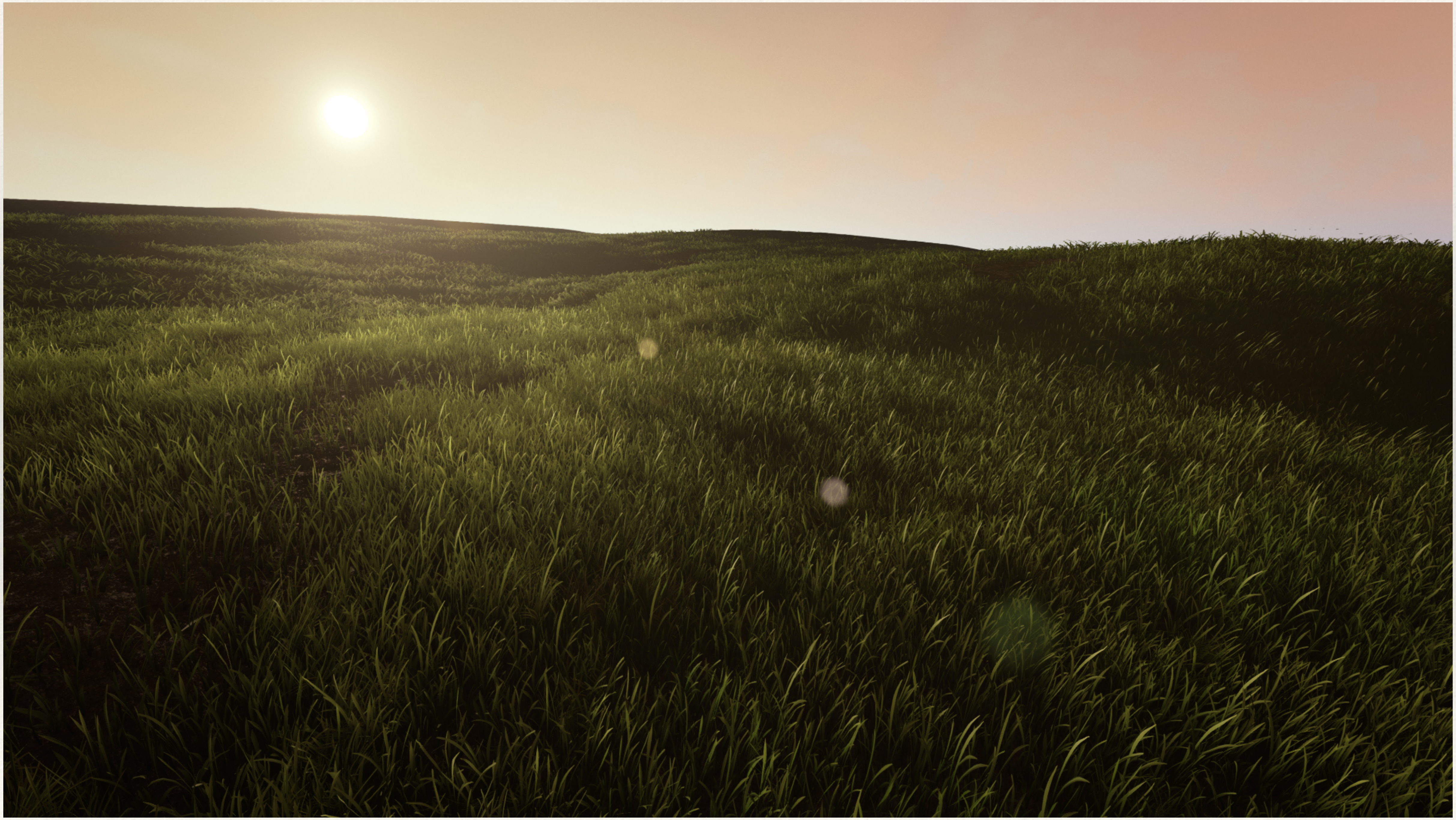
Screen sample operations (depth and color)

Output: image (pixels)

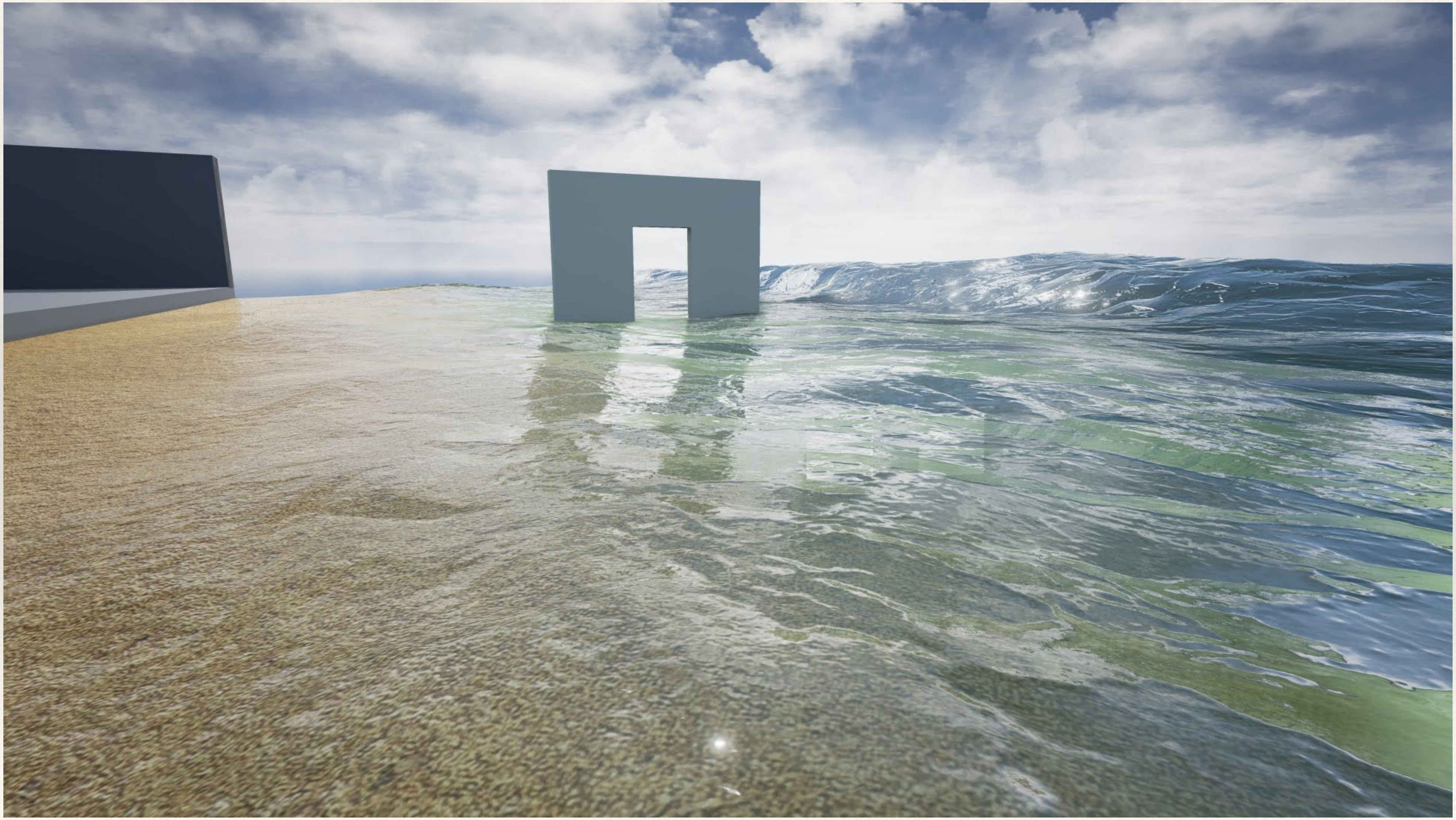
* Several stages of the modern OpenGL pipeline are omitted

CMU 15-462/662, Fall 2015

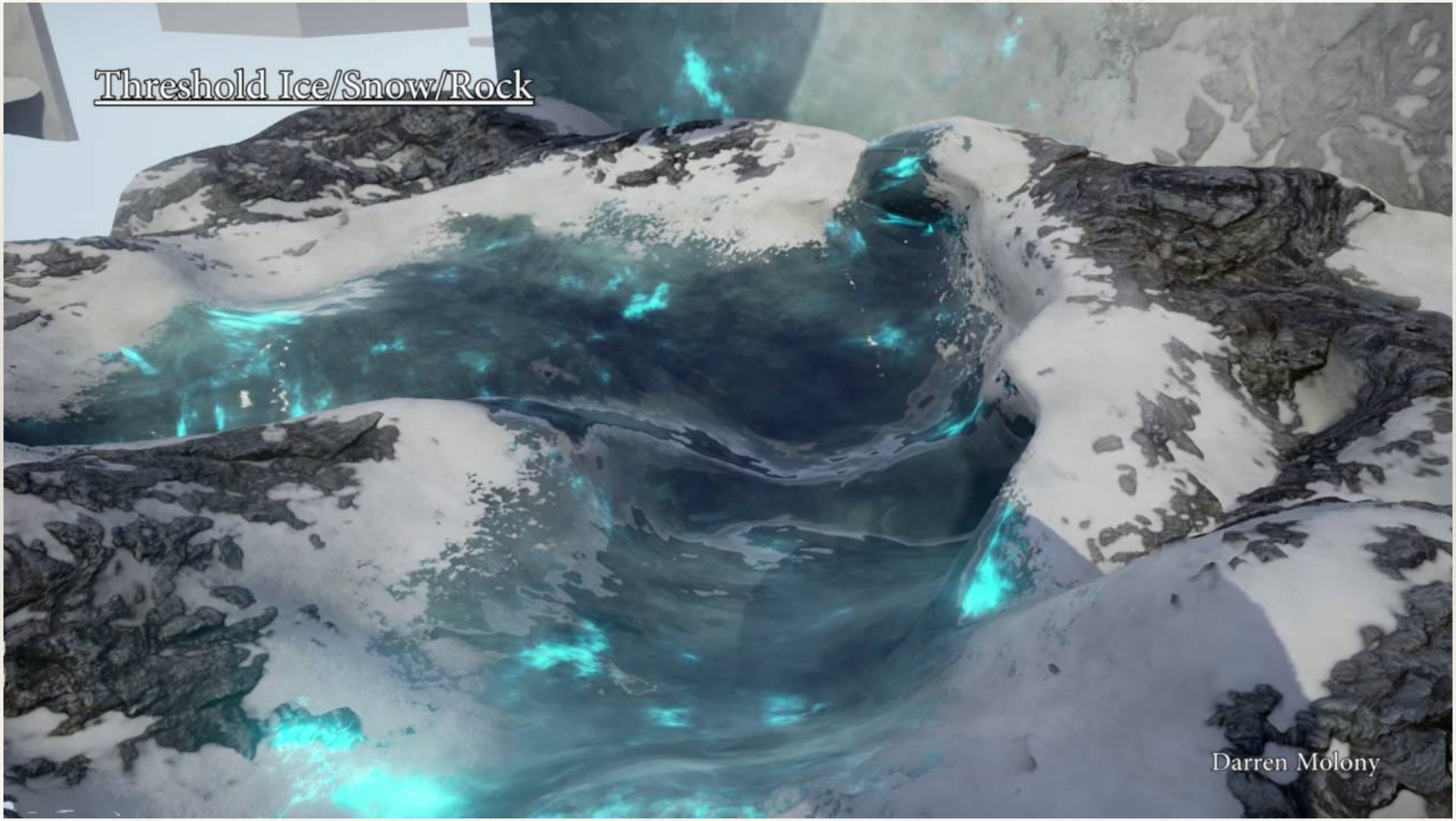
Shader Example



Shaders Example



Shaders Example


Threshold Ice/Snow/Rock
Darren Molony