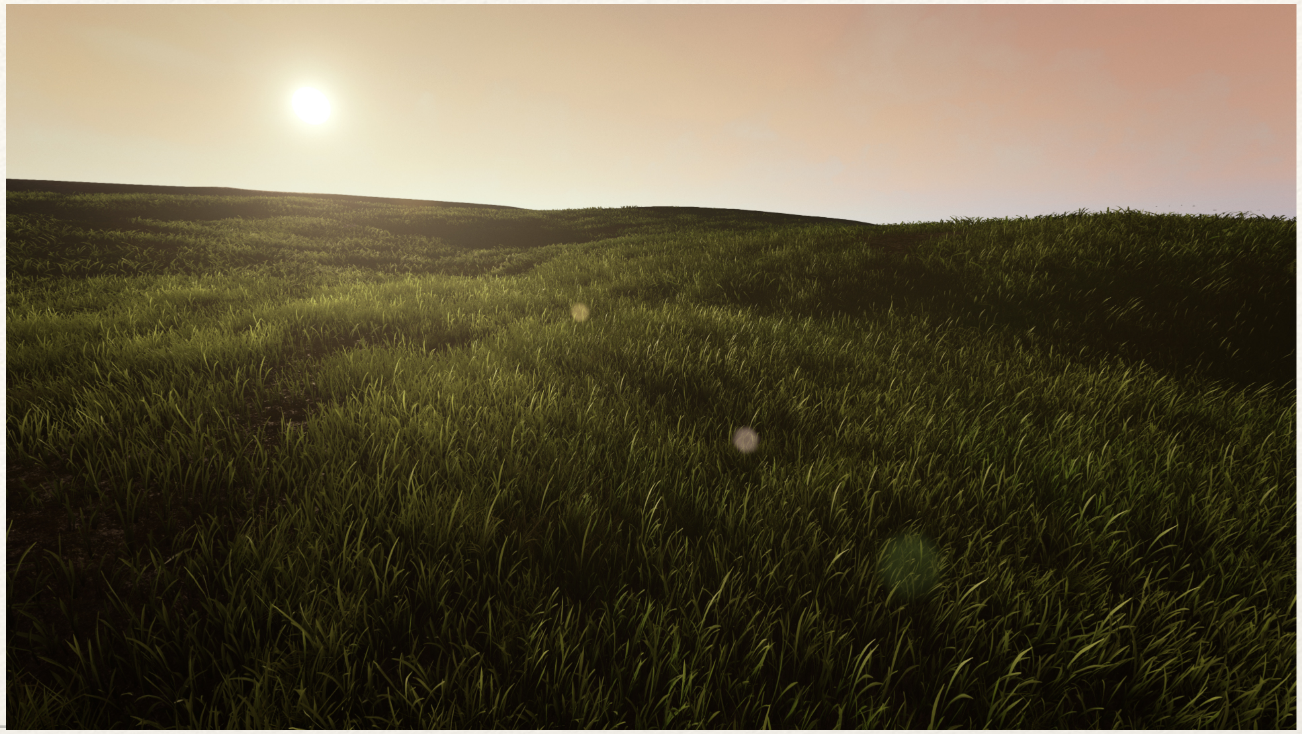
Dr. Sarah Abraham
University of Texas at Austin
Computer Science Department



Shaders

Elements of Graphics
CS324e

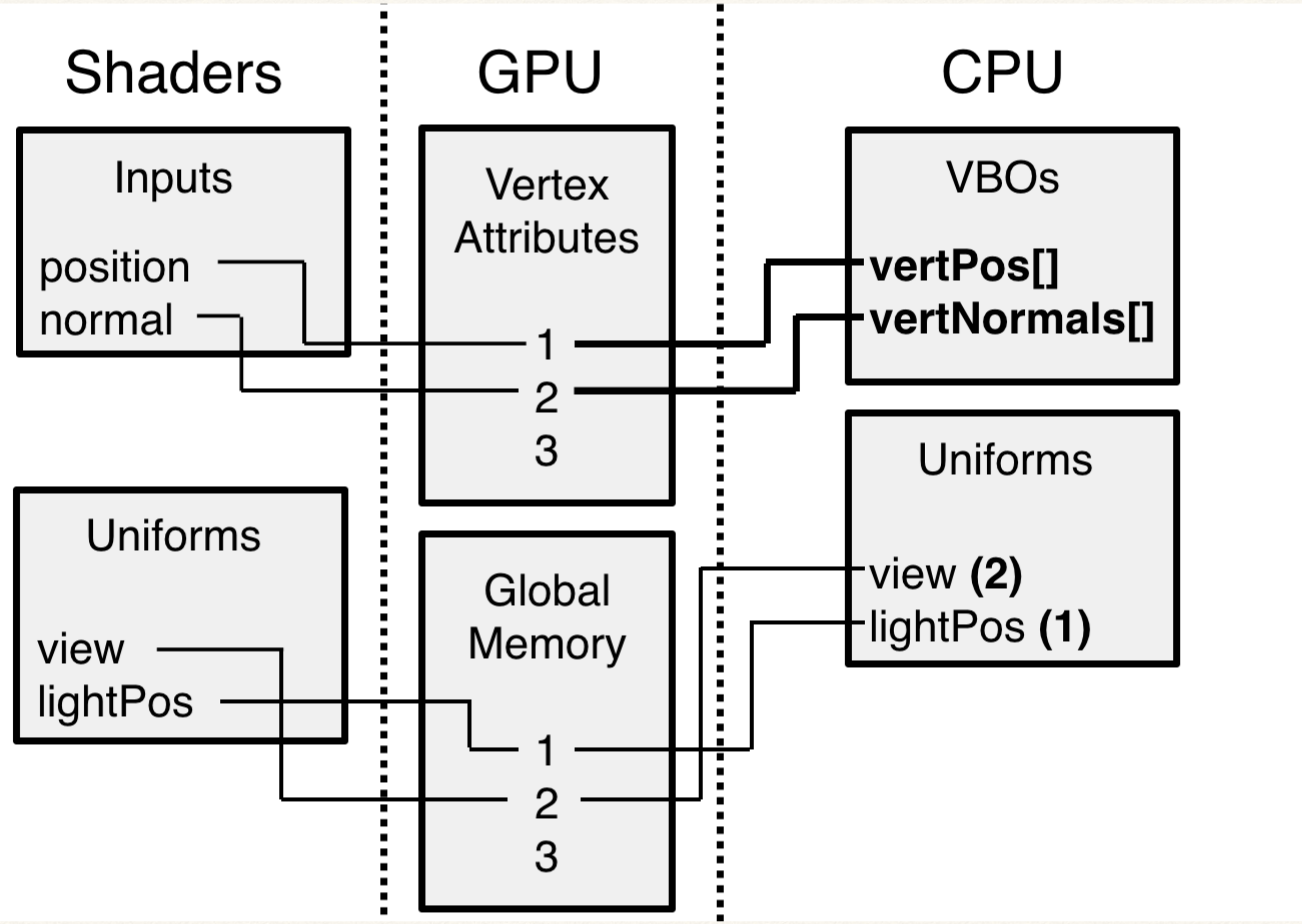
Shaders

- Small programs that are run on the GPU
- Used to process vertices (vertex shader) and pixels (fragment shader)
- OpenGL shaders are written in GLSL (OpenGL Shading Language)
- `main()` function is run for *every* vertex/pixel in parallel

Attributes

- Data stored in buffers on CPU then shared with GPU to run shaders
  - Vertices
  - Normals
  - Color
  - Texture
- Attributes tell the GPU where to look to find that buffer data

Shader Communication



Putting It Together (Concept)

1. Load and compile shaders
2. Attach and link shaders to shader program
3. Set shader program to use on GPU
4. Set attribute location of data
5. Set buffer of data
6. Draw data on GPU

Initialization vs Draw

- Initialization of buffers is expensive!
- Draw loop should only include calls that change per frame
    - e.g. Initialize data once then draw it per frame

Vertex Shader Example

```
attribute vec3 position;


void main() {
  //Must set gl_Position in vertex shader
  for each vertex processed
  gl_Position = vec4(position, 1.0);
}
```

Fragment Shader Example

```
//Specifies float precision
precision mediump float;


void main() {
  //Must set gl_FragColor in fragment shader
  for each pixel processed
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Going Farther with Fragment Shaders

- All the fragment shader does is output a `gl_FragColor`
- `gl_FragColor` determines the final color value of that pixel to display on screen
- Final color value of pixel usually calculated from mesh material, lighting, textures, etc

Passing Values from Vertex to Fragment

- `varying` values are passed from the **vertex** shader to the **fragment** shader
- To pass a `varying` value:
  1. Declare in vertex shader
  2. Update value in vertex `main` function
  3. Declare in fragment shader

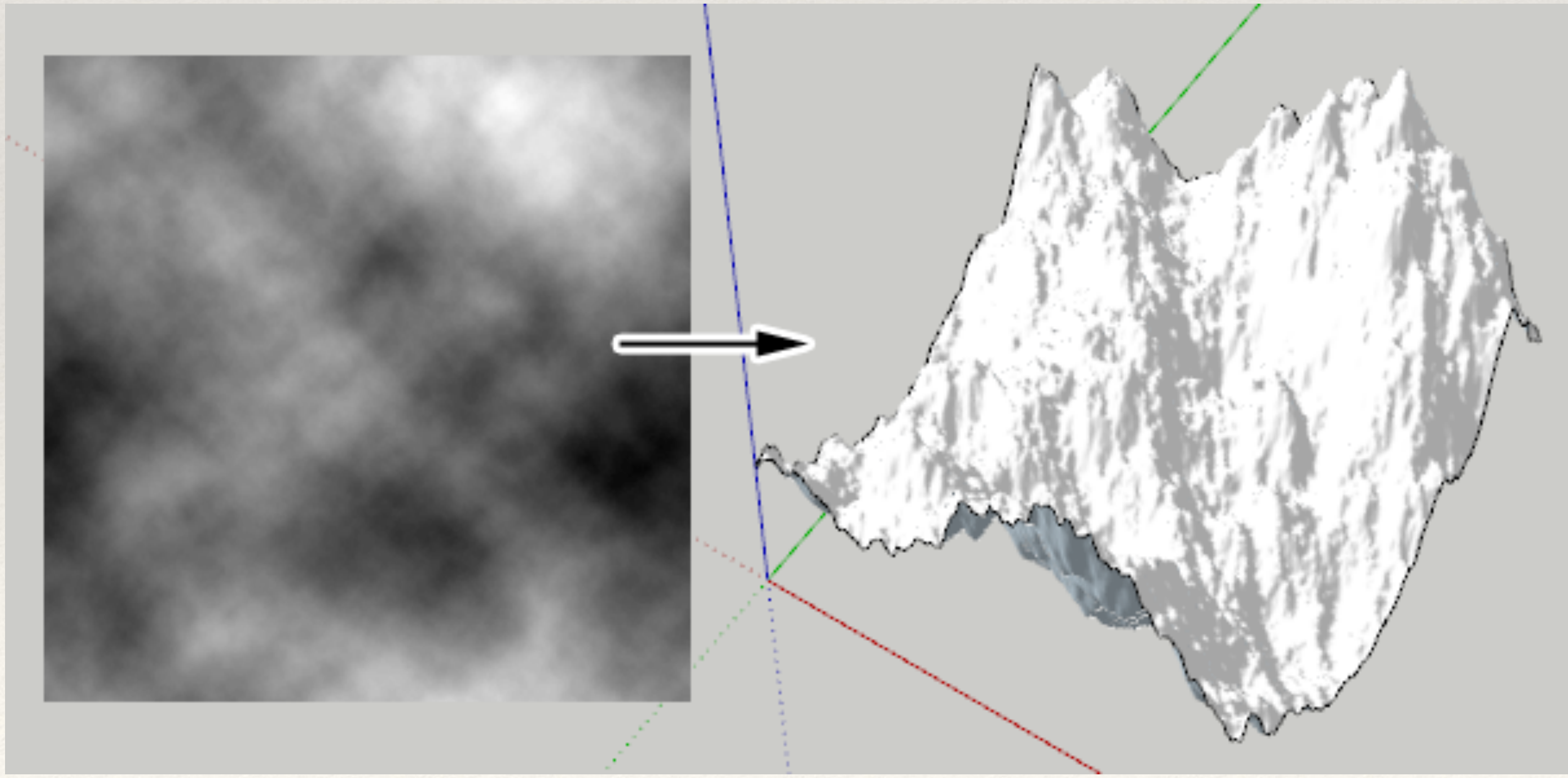
Updated Example

```
//vertex shader
attribute vec3 position;
varying highp vec3 light;
void main() {
  light = vec3(1.0, 1.0,
1.0);
  gl_Position =
  vec4(position, 1.0);
}
```

```
//fragment shader
precision mediump float;
varying highp vec3 light;
void main() {
  vec3 color = vec3(1.0,
  0.0, 0.0);
  gl_FragColor =
vec4(color * light, 1.0);
}
```

Terrain Generation

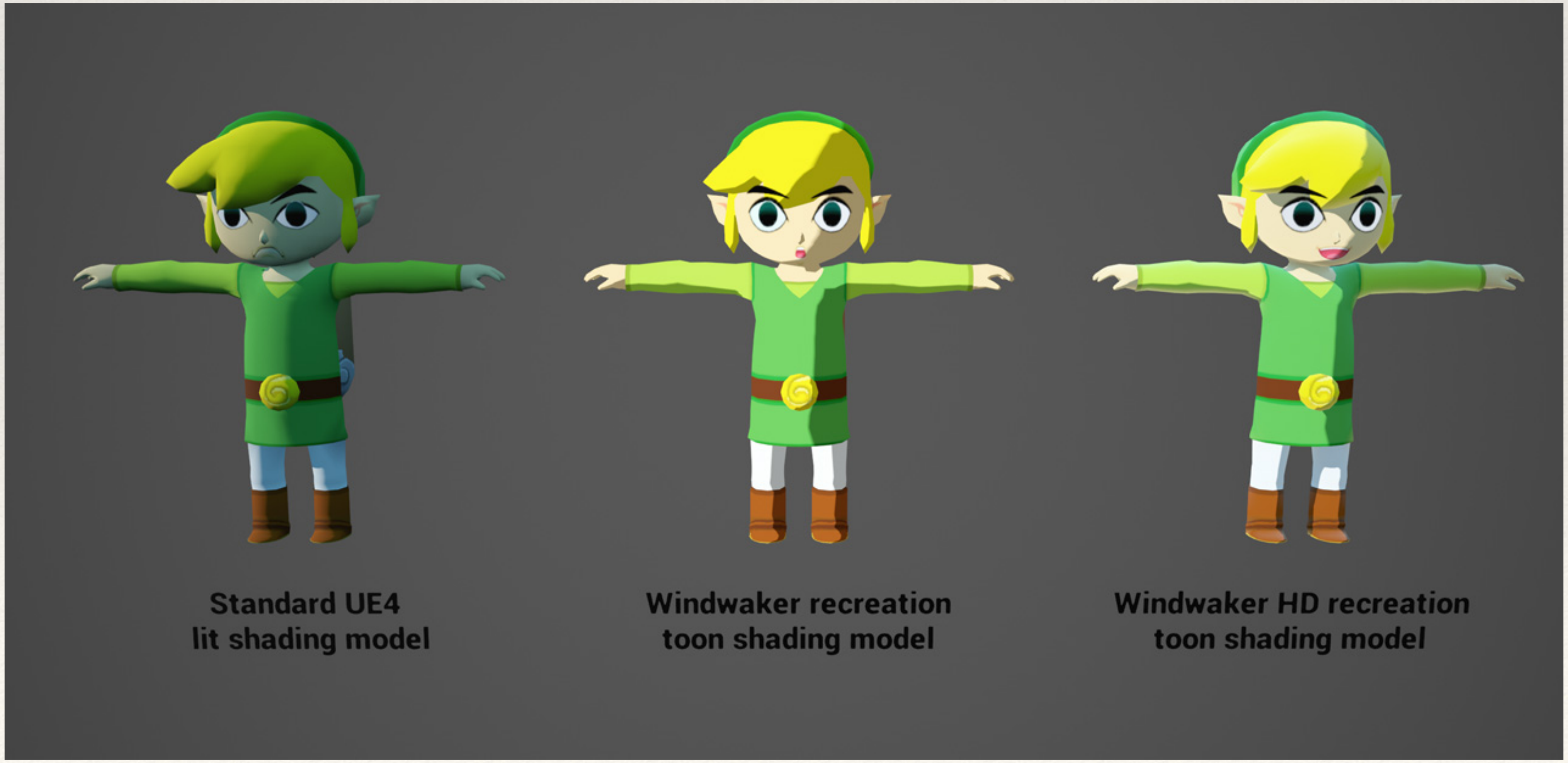
- Use texture data to create heightmap (altitude of cell based on pixel's color value
- Heightmap can generate terrain mesh



(Bitmap to Mesh SketchUp extension)

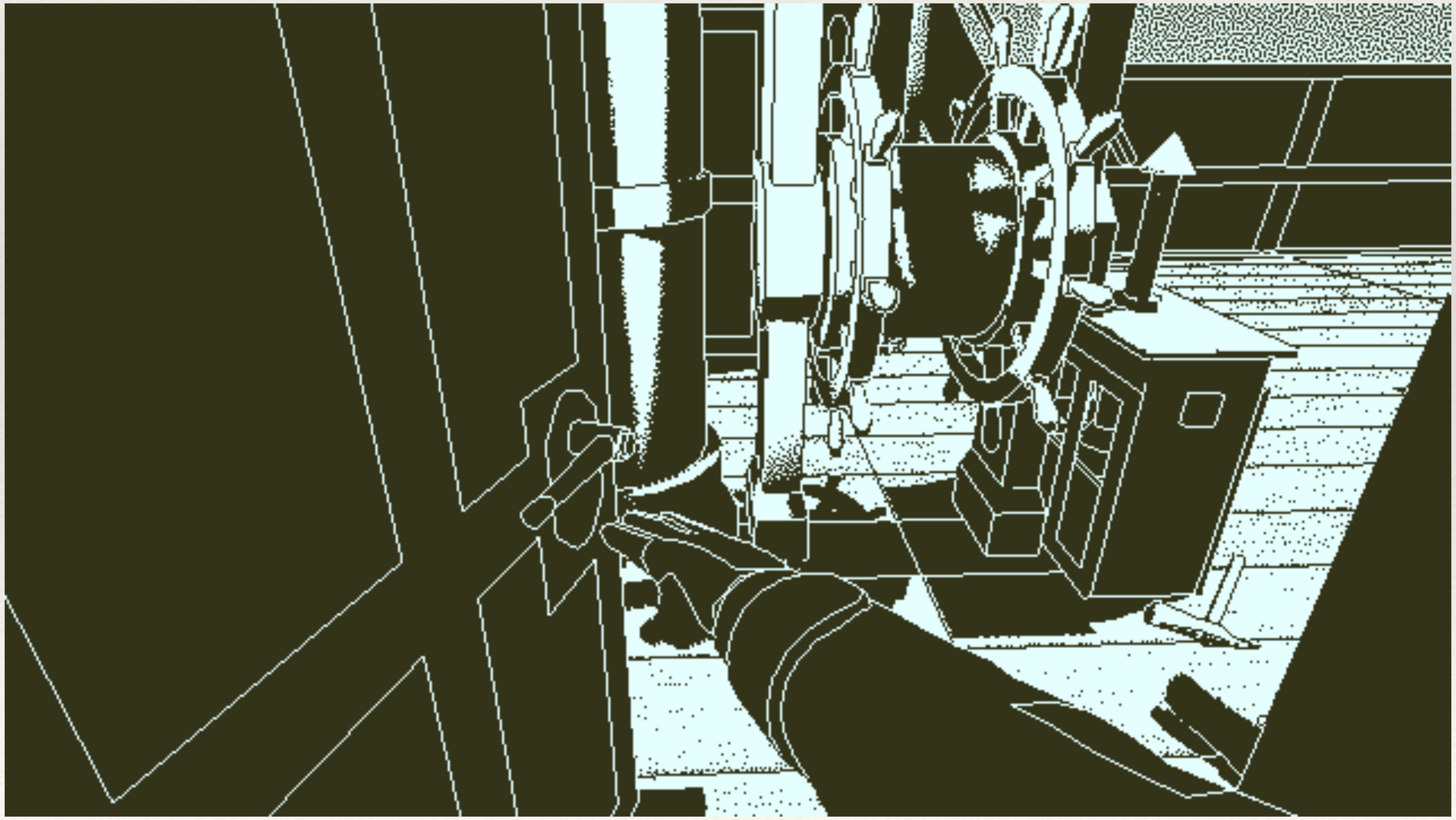
Toon Shaders

- Check normal of vertex against direction of light
- Pick a “highlight”, “normal”, or “shadow color based on the angle between vertex and light direction



Edge Detection

- Can do per-pixel and pixel neighborhood operations using a texture of screen space in the fragment shader

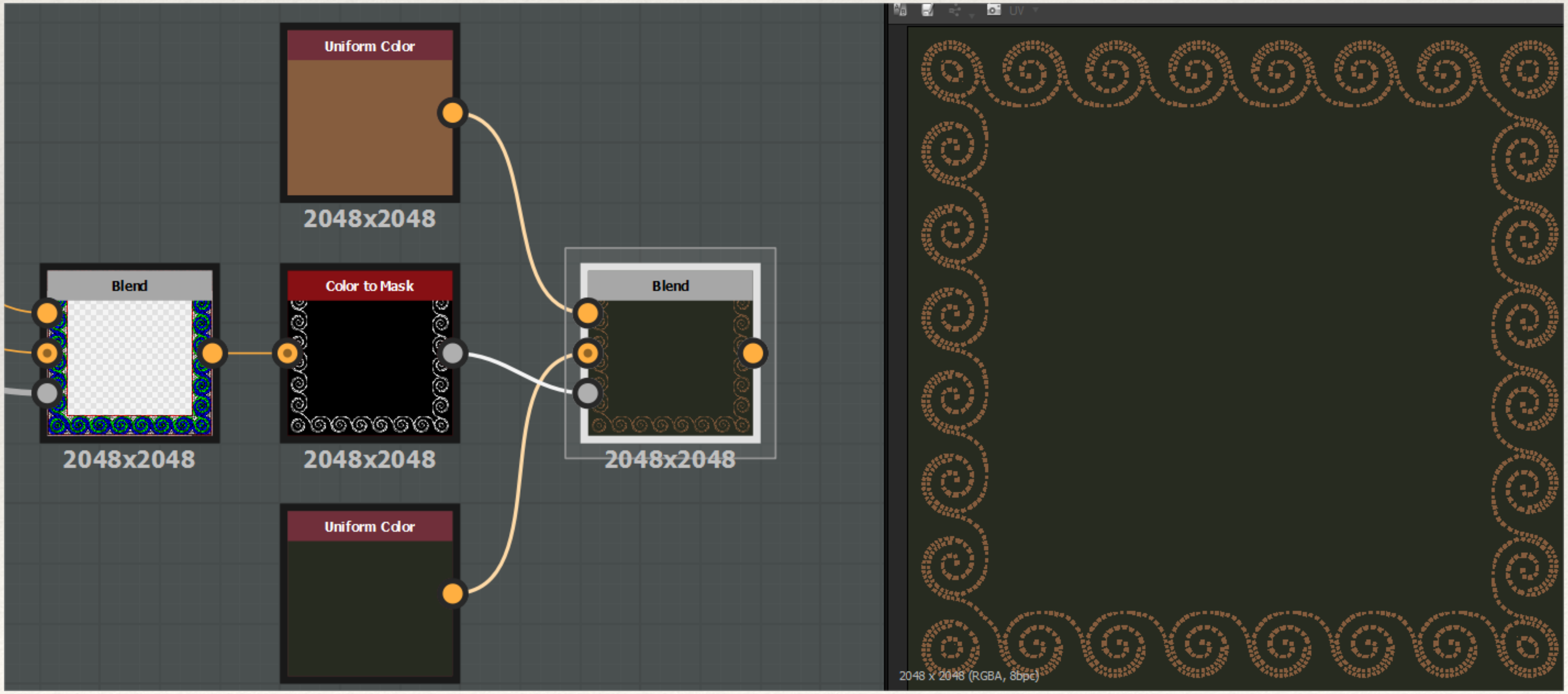


Programming Shaders Visually

- Shaders are can be difficult to program
  - Lots of context-specific keywords
  - Parallel nature does not translate directly from standard CPU-style programming
  - Usually must debug visually
- Visual scripting languages make shaders more accessible

Visually Scripting Materials

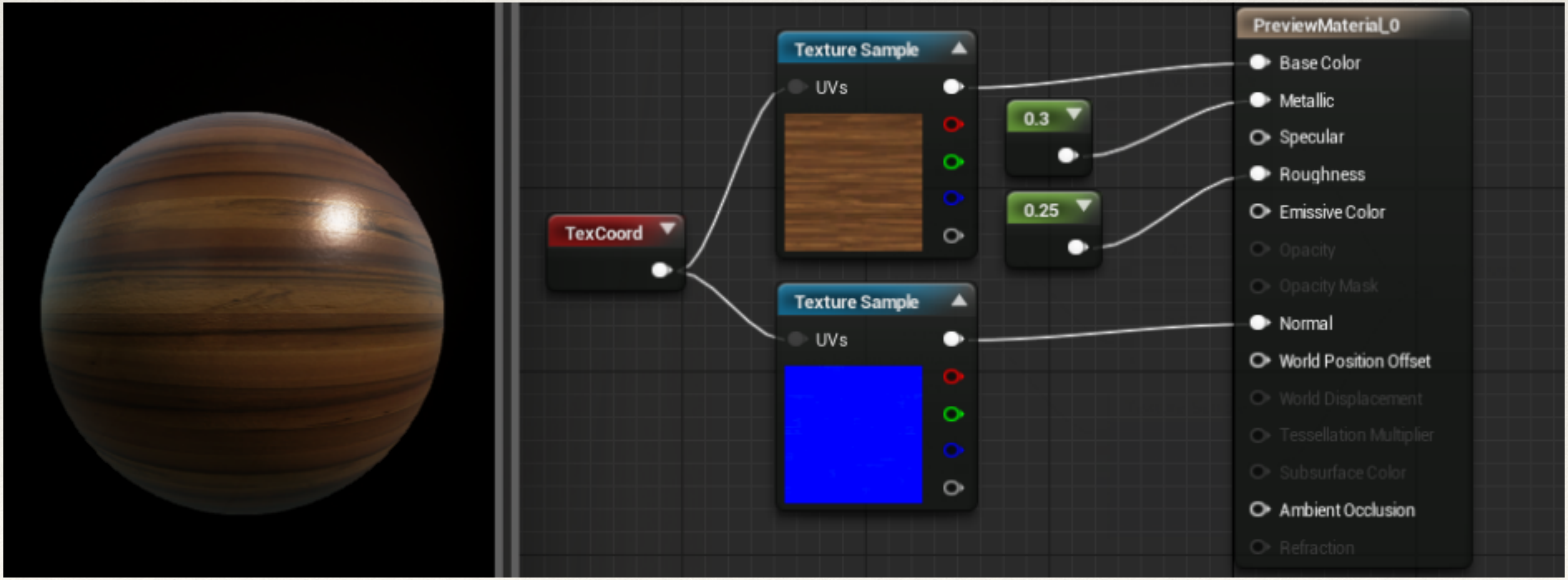
- Programs like Substance allow artists to set material properties in a node-based way
- https://www.youtube.com/watch?v=y8q6-tgQjZc



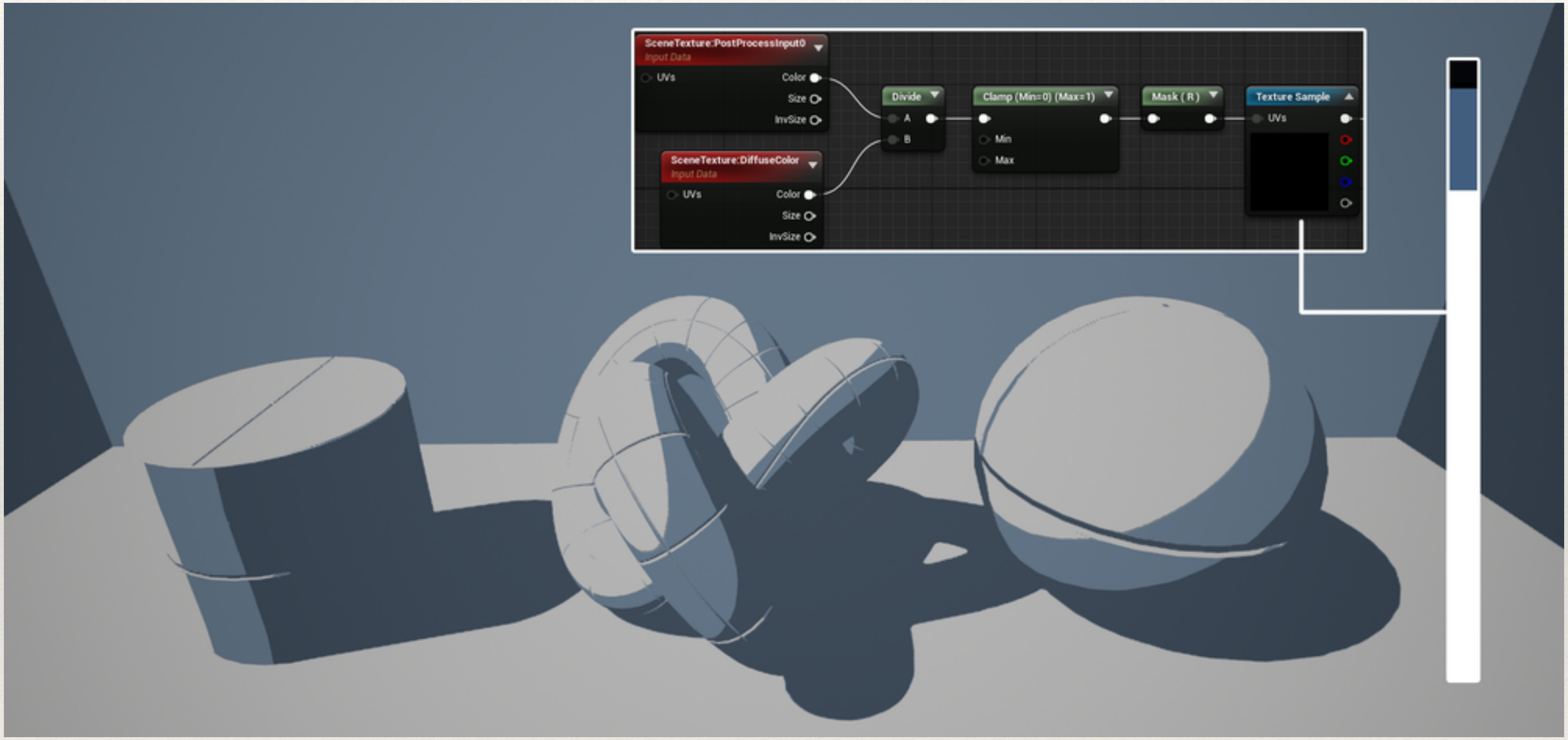
(Pete Sekula)

Visually Scripting Shaders

- Engines like Unreal 4 allow artists to access to materials used in the shader pipeline in a node-based way



Visually Scripting Shaders



- https://www.youtube.com/watch?v=TEmsqez2YQI
-

References

- https://learnopengl.com/Getting-started/Hello-Triangle
- https://medium.com/social-tables-tech/hello-world-webgl-79f430446b5c

Appendices

Initialization of Buffer Data

```
function initBuffer() {
    vertexdata = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexdata);
    let vertices = [
    0.0, 1.0, 0.0,
    -1.0, -1.0, 0.0,
    1.0, -1.0, 0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new
    Float32Array(vertices), gl.STATIC_DRAW);
}
```

Initialization of Shader and Attributes

```
function initShader() {
  program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);
  gl.linkProgram(program);
  gl.useProgram(program);
  position = gl.getAttribLocation(program, “position");
  gl.enableVertexAttribArray(position);
}
```

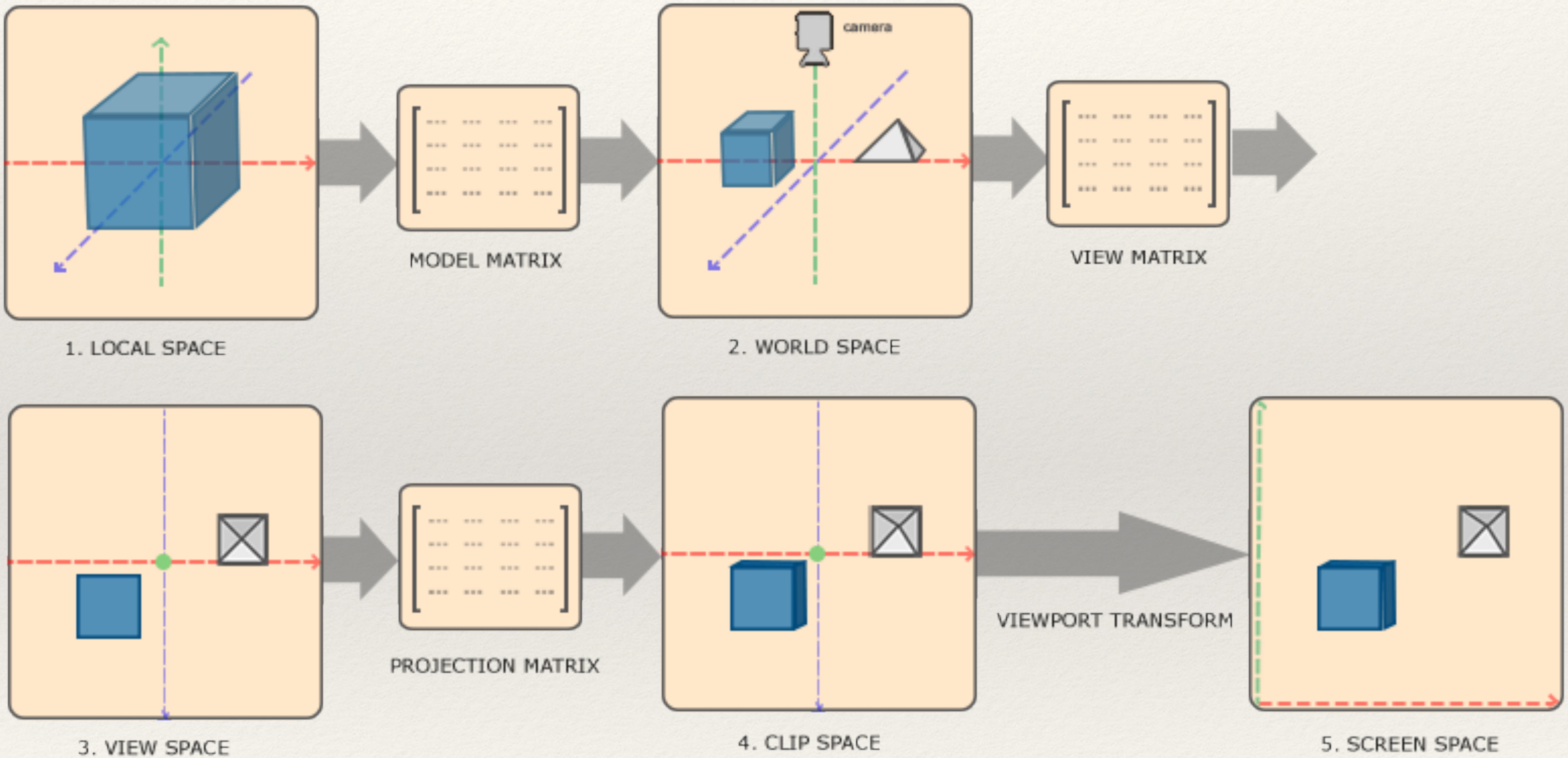
Putting It Together (Example)

```
//Calls from draw loop

gl.useProgram(program);

gl.bindBuffer(gl.ARRAY_BUFFER, vertexdata);

gl.vertexAttribPointer(position, 3,
gl.FLOAT, false, 0, 0);

gl.drawArrays(gl.TRIANGLES, 0,
numVertices);
```

Going Farther with Vertex Shaders

- All the vertex shader does is output a `gl_Position`
- `gl_Position` determines where the vertex is in *clip space*
- Clip space is normalized coordinate system that can be output to any screen resolution and aspect ratio

Transforming Coordinate Systems



(learnopengl.com)

Transforming to Clip Space

- Often vertex shader takes a Model-View-Projection matrix (composed by multiplying M*V*P) as a uniform (same value is used for every draw call)
- Matrices for local space, world space, camera space, and projection space all managed within program

Updated Vertex Shader

```
attribute vec3 position;

uniform mat4 mvp_matrix; //4x4 matrix
representing model (local-world), view
(camera) and projection of camera


void main() {

  gl_Position = mvp_matrix * vec4(position,
  1.0);

}
```