CENG 477
Introduction to Computer Graphics

OpenGL and Programmable Shaders

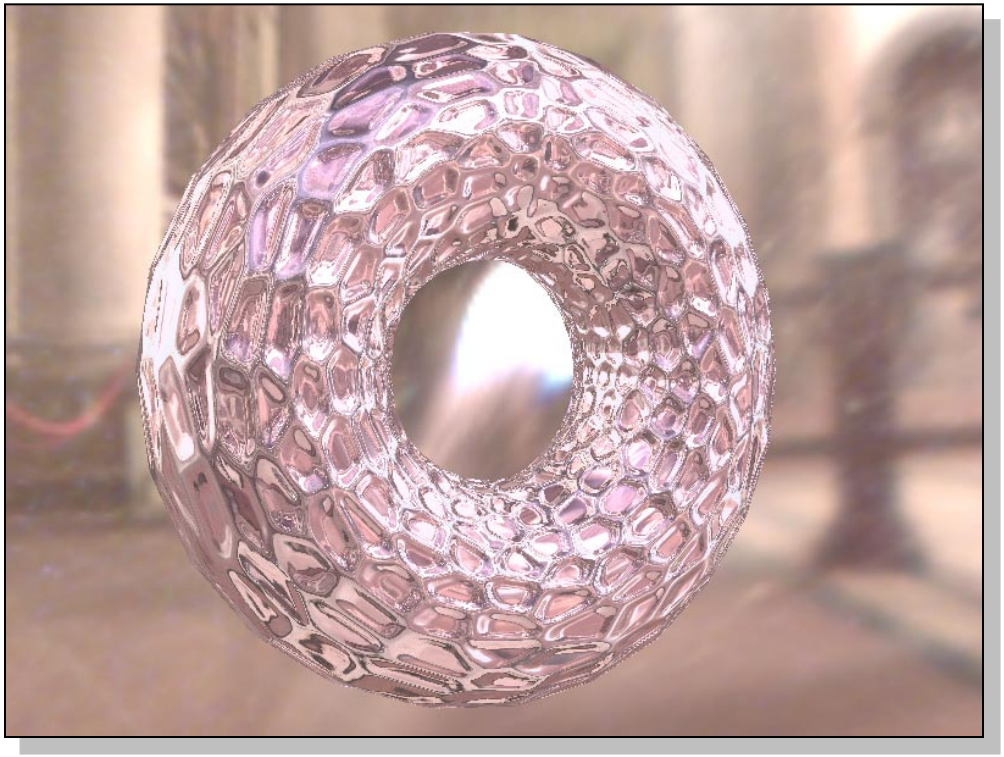
Shaders

- A shader is:
  - Programmable logical unit on the GPU which can replace the "fixed" functionality of OpenGL with user-generated code
- By using custom shaders, the user can now override the existing implementation of core per-vertex and per-pixel behavior
- Some tasks such as clipping and rasterization will still be performed automatically

Shaders

- Written using GPU language
  - Low-level: assembly language
  - High-level: Cg (Nvidia only) or GLSL (general)
- Run on the GPU while the application runs at the same time on the CPU
- This allows CPU for computations other than graphics
- Shaders can create a variety of visual effects

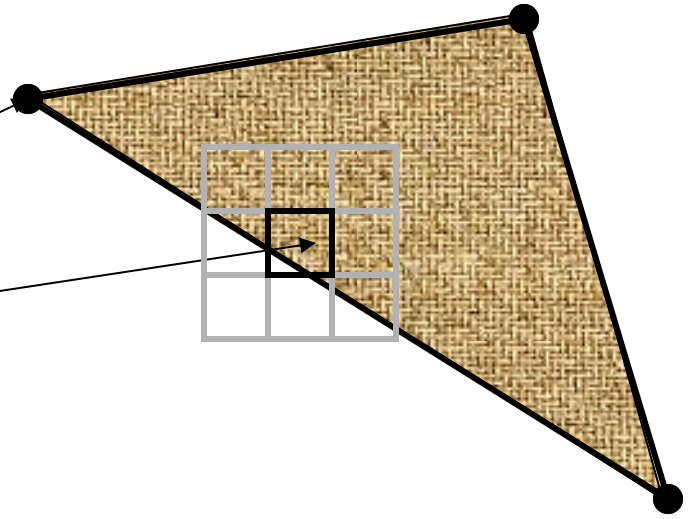
Shader Gallery I



Above: Demo of Microsoft's XNA game platform
Right: Product demos by Nvidia (top) and AMD (bottom)

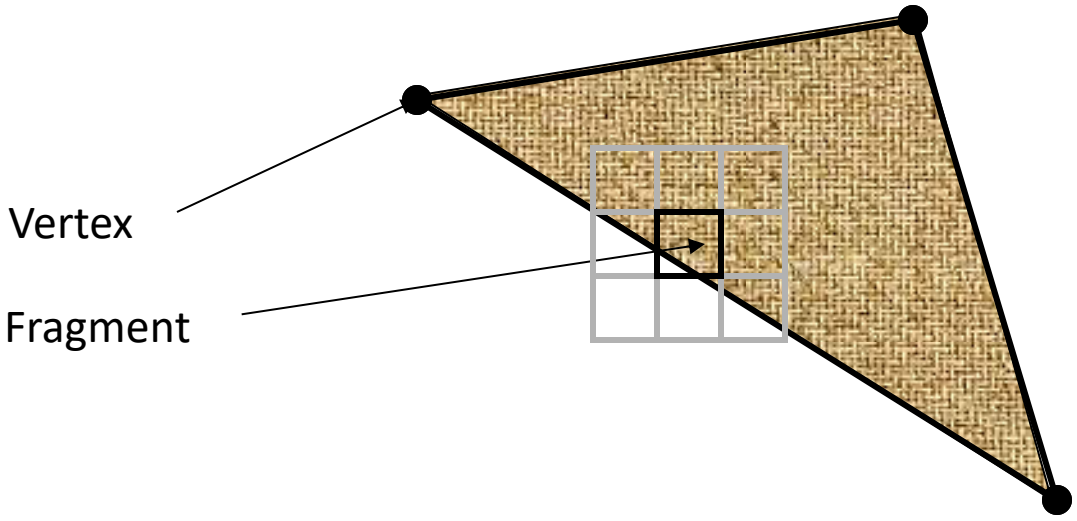
What are we targeting?

- OpenGL shaders give the user control over each *vertex* and each *fragment* interpolated between vertices.



What are we targeting?

- Each vertex is passed to vertex shader
- After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and other attributes are interpolated across the polygon. The interpolated values are passed to fragment shader

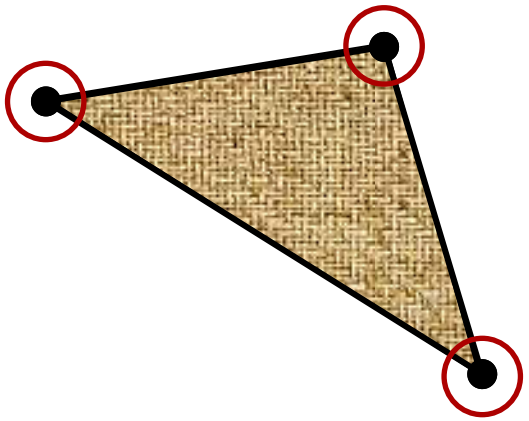


What can you do in a shader?

- Per vertex:
  - Vertex transformation
  - Normal transformation and normalization
  - Texture coordinate assignment/transformation
  - Per-vertex lighting
  - Material application
  - …
- Per fragment:
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
  - Color summation
  - Per-fragment lighting
  - Scale and bias
  - Color table lookup
  - Convolution
  - …

Parallelism

- Shader source codes are read by the CPU program and given to OpenGL for compilation
- They execute on the GPU at draw time
  - No other way to execute a shader by other than drawing something
- A shader executes simultaneously on its data

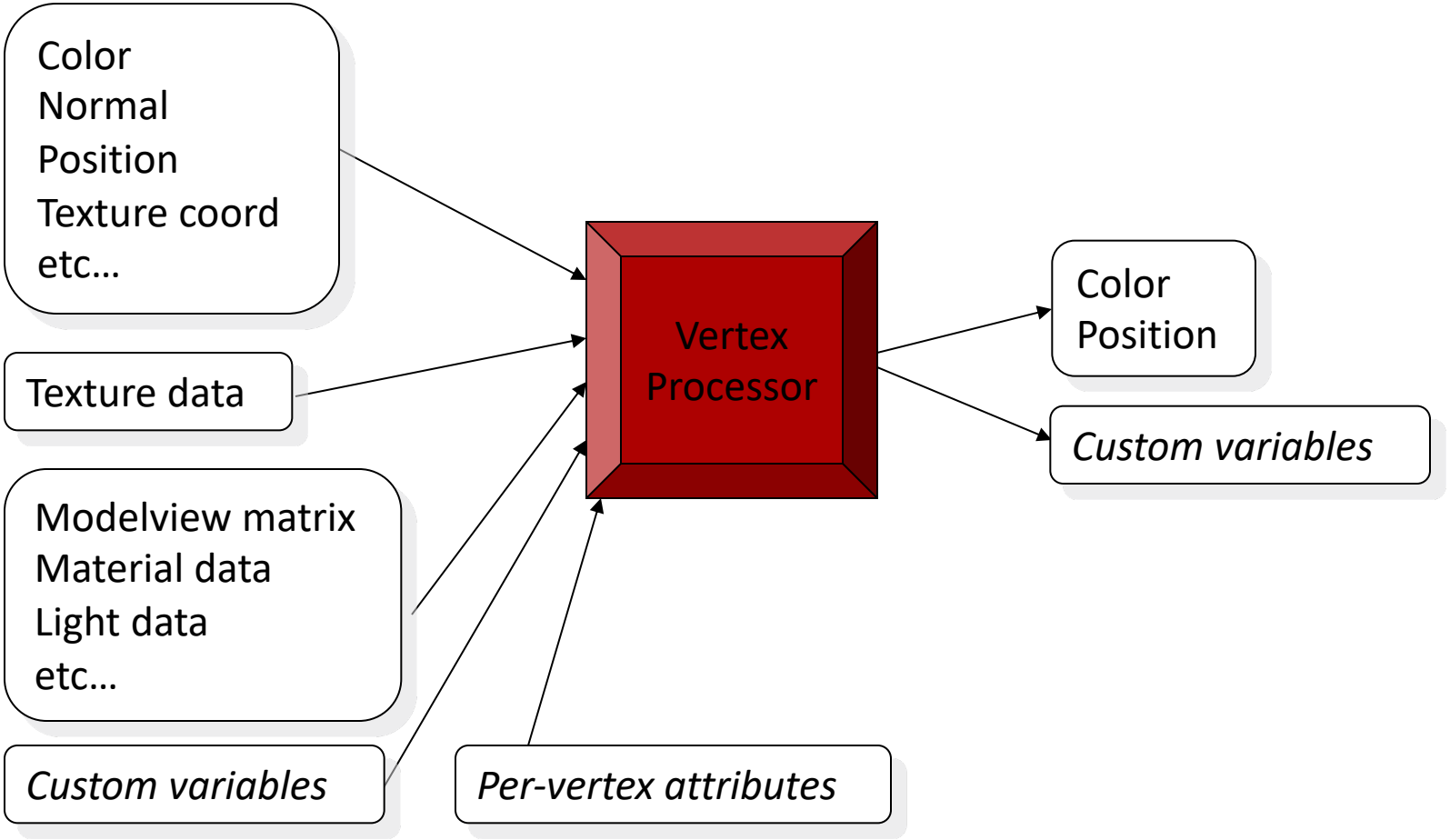


- All processed simultaneously (up to the physical limit allowed by the GPU)
- So we **don't** loop over vertices in a vertex shader
- The same applies for the fragement shader as well

Shading languages

- There are several popular languages for describing shaders, such as:
  - *HLSL*, the *High Level Shading Language*
    - Author: Microsoft
    - DirectX 8+
  - *Cg*
    - Author: Nvidia
  - *GLSL*, the *OpenGL Shading Language*
    - Author: the Khronos Group, a self-sponsored group of industry affiliates from various vendors (AMD, Nvidia, Intel, Apple, …)

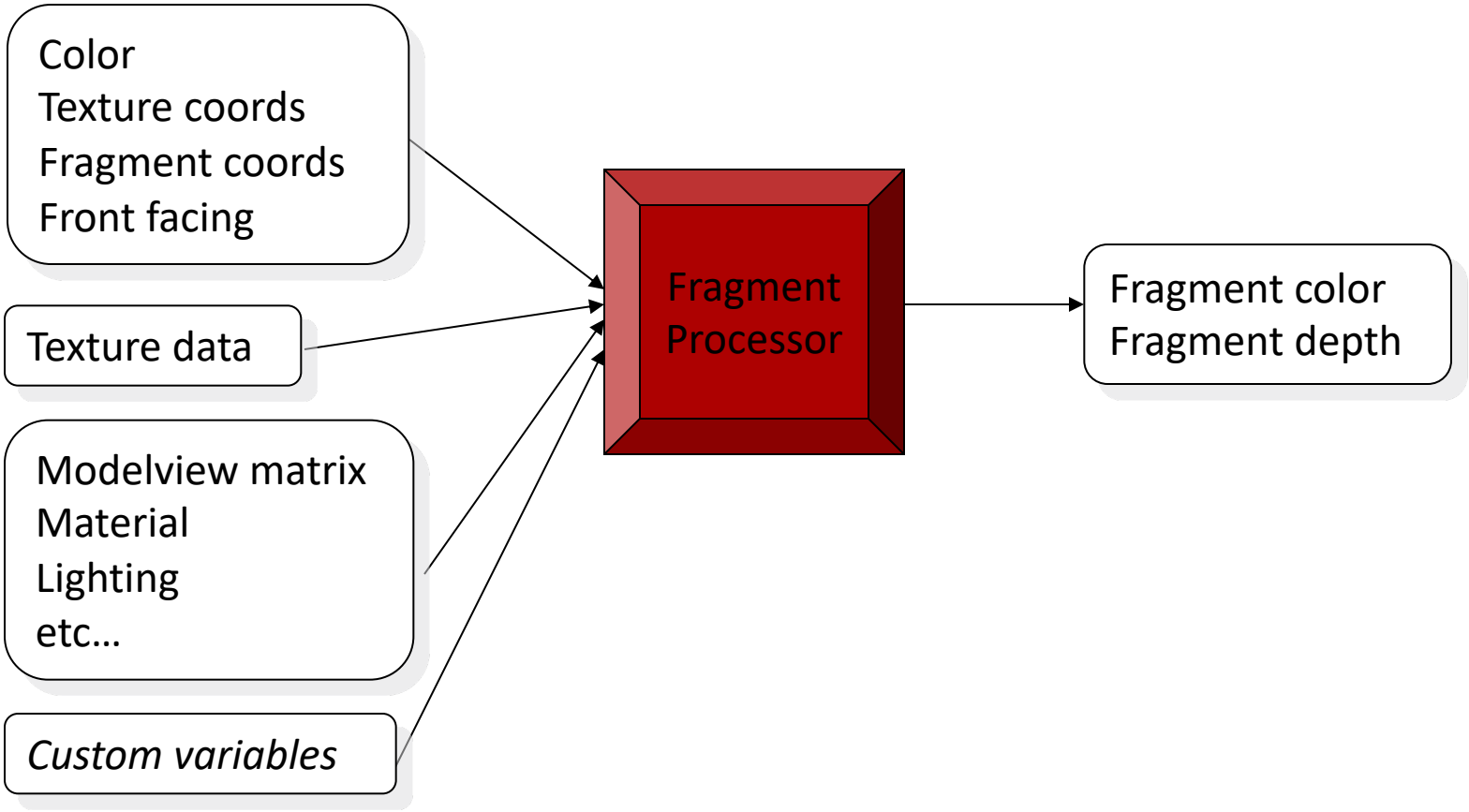
Vertex processor – inputs and outputs



Vertex Processor

- Vertex shader is executed once for each vertex
- Vertex position is usually transformed using the modelview and projection matrices
- Normals are transformed with the appropriate matrix
- Texture coordinates may be generated, passed along, or transformed
- Lighting computations may be done for each vertex
- Vertex positions may be modified based on texture values, etc.

Fragment processor – inputs and outputs



Fragment Processor

- Fragment shader is executed once for each fragment
- Lighting may be computed at each fragment using interpolated normals
- Texture data may be fetched from the texture image
- Effects such as fog, blur, etc. may be added
- Fragments can be killed, their depth values can be altered
- Various post-processing effects can be applied

Activating a Shader

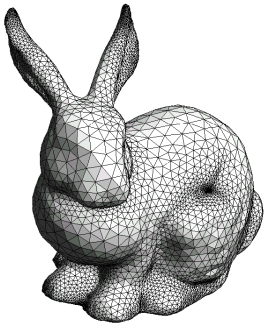
- **Step 1:** Create a shader program
  - (Gluint) glCreateProgram()
- **Step 2:** Create vertex and fragment shaders
  - (Gluint) glCreateShader(GL_VERTEX_SHADER)
  - (Gluint) glCreateShader(GL_FRAGMENT_SHADER)
- **Step 3:** Provide the source code
  - glShaderSource(vertexShaderId, …)
  - glShaderSource(fragmentShaderId, …)
- **Step 4:** Compile the shaders
  - glCompileShader(vertexShaderId)
  - glCompileShader(fragmentShaderId)

Activating a Shader

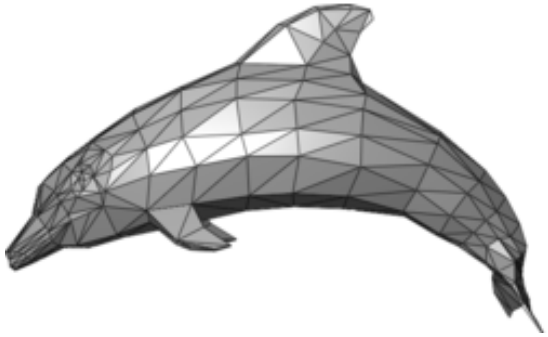
- **Step 5:** Attach the shader to the shader program:
  - glAttachShader(programId, vertexShaderId)
  - glAttachShader(programId, fragmentShaderId)
- **Step 6:** Link the program:
  - glLinkProgram(programId)
- **Step 7:** Activate the program:
  - glUseProgram(programId)

Activating a Shader

- Many shader programs may be linked but only one can be active at a time (i.e. for a single draw)
- It is possible to draw different models using different shader programs



glUseProgram(prg1);
glDrawElements(…); // bunny

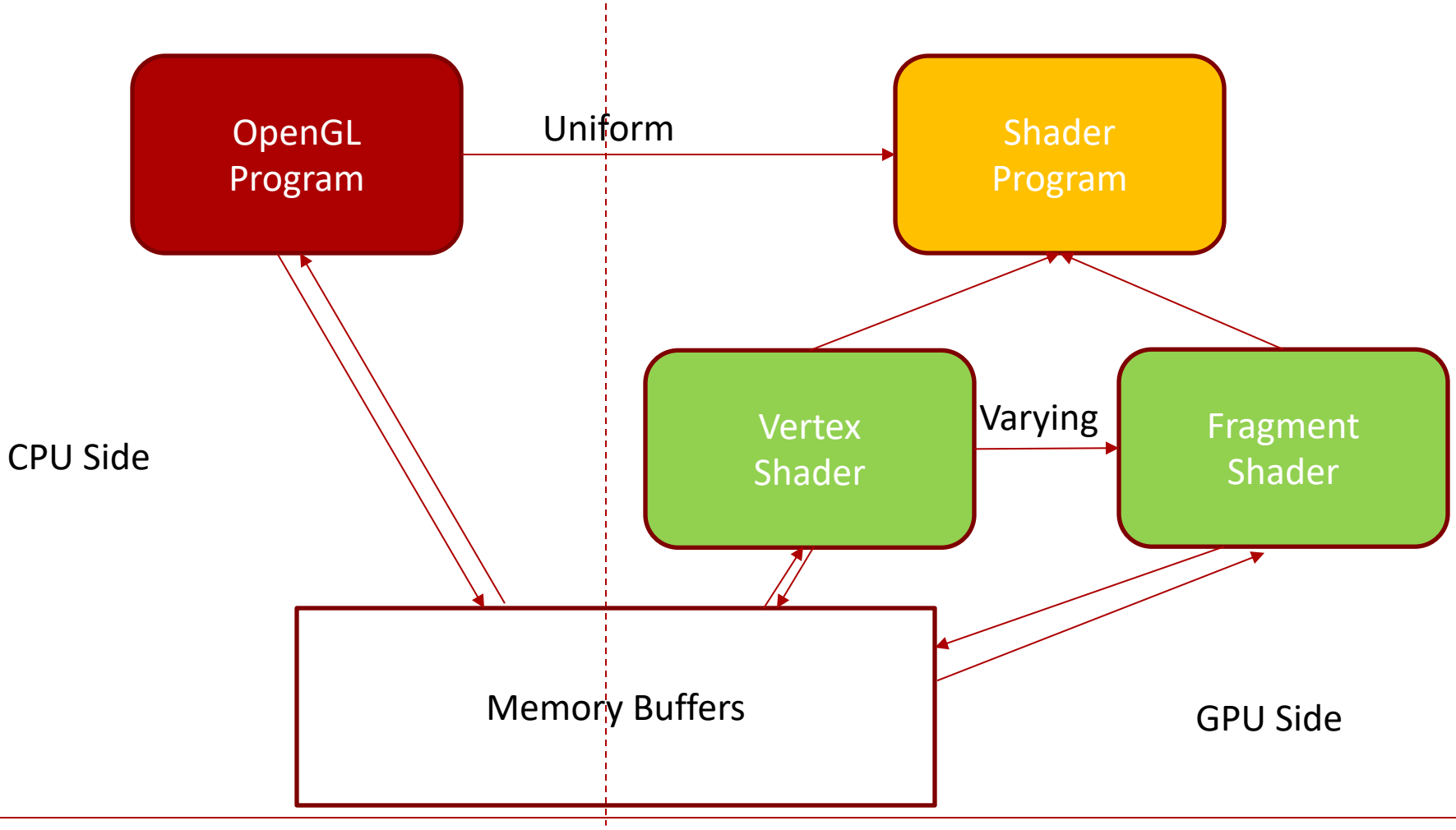


glUseProgram(prg2);
glDrawElements(…); // dolphin

Activating a Shader

- Once a shader program is activated, certain operations will no longer be done for you
- It's up to you to replace it!
  - You'll have to transform each vertex into canonical viewing volume manually (i.e. multiply with modelviewprojection matrix)
  - You'll have to shade each vertex or fragment manually (i.e. perform lighting computations)
- The installed program replaces all OpenGL fixed functionality for all renders until you remove it with glUseProgram(0)

Communicating with Shadrs

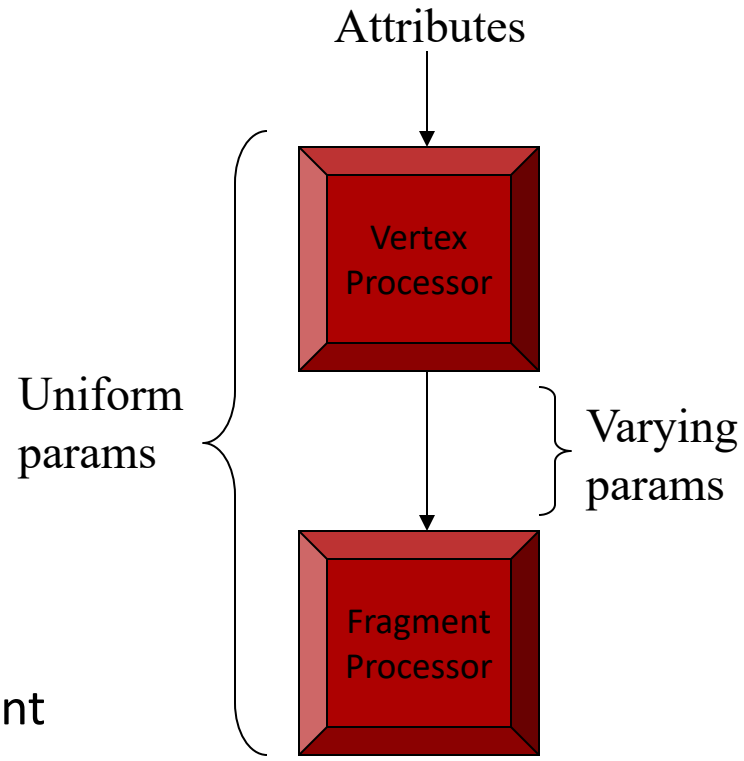


Communicating with Shaders

- OpenGL program can send data to shaders through uniform variables or by using memory buffers
- Vertex shader can send data to a fragment shader but not vice-versa (dataflow is one-way only)
- Vertex and fragment shaders can write data to memory buffers which can be read back by the OpenGL program

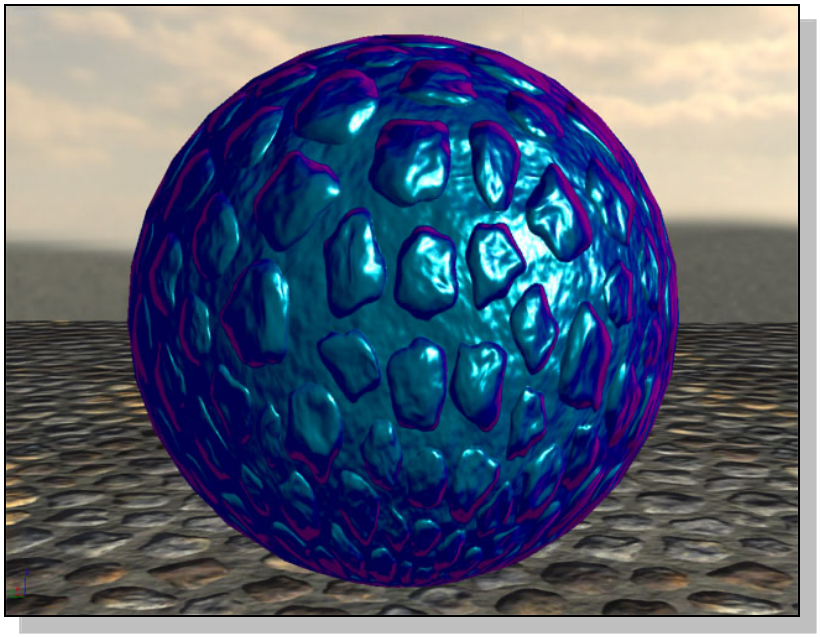
Communicating with Shaders

- There are three types of shader parameter in GLSL
- *Uniform parameters*
  - Set from the CPU program
  - Think of them as global variables
- *Attribute parameters*
  - Set per vertex
  - E.g. position, color, normal, …
- *Varying parameters*
  - Passed from vertex processor to fragment processor
  - E.g. rasterized position, color, …



Shader Gallery II

Above: Kevin Boulanger (PhD thesis, "*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*", 2005)



Above: Ben Cloward ("Car paint shader")

A Sample Vertex Shader

```
#version 330

uniform mat4 modelingMatrix;
uniform mat4 viewingMatrix;
uniform mat4 projectionMatrix;

varying vec4 fragWorldPos;
varying vec3 fragWorldNor;

void main(void)
{
    // Compute the world coordinates of the vertex and its normal.
    // These coordinates will be interpolated during the rasterization
    // stage and the fragment shader will receive the interpolated
    // coordinates.

    fragWorldPos = modelingMatrix * gl_Vertex;
    fragWorldNor = inverse(transpose(mat3x3(modelingMatrix))) * gl_Normal;

    gl_Position = projectionMatrix * viewingMatrix * modelingMatrix * gl_Vertex;
}
```

Version string

Uniform variables that must be set by the OpenGL program

Varying variables that will be rasterized for each fragment

Attribute variables fetched from buffers

A Sample Fragment Shader (1)

```
#version 330

// All of the following variables could be defined in the OpenGL
// program and passed to this shader as uniform variables. This
// would be necessary if their values could change during runtim.
// However, we will not change them and therefore we define them
// here for simplicity.

vec3 I = vec3(1, 1, 1);           // point light intensity
vec3 Iamb = vec3(0.8, 0.8, 0.8); // ambient light intensity
vec3 kd = vec3(1, 0.2, 0.2);     // diffuse reflectance coefficient
vec3 ka = vec3(0.3, 0.3, 0.3);   // ambient reflectance coefficient
vec3 ks = vec3(0.8, 0.8, 0.8);   // specular reflectance coefficient
vec3 lightPos = vec3(5, 5, 5);   // light position in world coordinates

uniform vec3 eyePos;

varying vec4 fragWorldPos;
varying vec3 fragWorldNor;
```

Local variables

Uniform variable

Variables declared as varying in VS must also be declared as varying in FS

A Sample Fragment Shader (2)

```
void main(void)
{
    // Compute lighting. We assume lightPos and eyePos are in world
    // coordinates. fragWorldPos and fragWorldNor are the interpolated
    // coordinates by the rasterizer.

    vec3 L = normalize(lightPos - vec3(fragWorldPos));
    vec3 V = normalize(eyePos - vec3(fragWorldPos));
    vec3 H = normalize(L + V);
    vec3 N = normalize(fragWorldNor);

    float NdotL = dot(N, L); // for diffuse component
    float NdotH = dot(N, H); // for specular component

    vec3 diffuseColor = I * kd * max(0, NdotL);
    vec3 specularColor = I * ks * pow(max(0, NdotH), 100);
    vec3 ambientColor = Iamb * ka;

    gl_FragColor = vec4(diffuseColor + specularColor + ambientColor, 1);
}
```

Special value indicating pixel color

Setting Values of Uniforms

- Uniform variables must be set by the main program
- **Step 1:** Query their location
  - (Glint) glGetUniformLocation(programId, “variableName”)
- **Step 2:** Make the program current:
  - glUseProgram(programId)
- **Step 3:** Set the variable using the proper command based on the type of the variable:
  - glUniform1f(location, v1); // v1 is a float
  - glUniform3fv(location, 1, v2); // v2 is float v2[3]
  - glUniformMatrix4fv(location, 1, GL_FALSE, v3); // upload one 4x4 matrix in column major order (16 values are taken from v3)

Setting Values of Uniforms

- Uniform variables are sticky
- The last set value remains valid until you change it
  - No need to reupload at each draw (or frame) if the value does not change
- **GLM library** facilitates setting matrix values:

```
float angleRad = (float) (angle / 180.0) * M_PI;

// Compute the modeling matrix

modelingMatrix = glm::translate(glm::mat4(1.0), glm::vec3(0.0f, 0.0f, -5.0f));
modelingMatrix = glm::rotate(modelingMatrix, angleRad, glm::vec3(0.0, 1.0, 0.0));

// Set the active program and the values of its uniform variables

glUseProgram(pId);
glUniformMatrix4fv(pId, 1, GL_FALSE, glm::value_ptr(projectionMatrix));
```

Phong Shading

- The previous shaders implement a shading scheme what is known as Phong shading (do not confuse it with Phong Exponent in ray tracing)
- **Basic idea:**
  - Interpolate per-vertex normal
  - Perform shading per-fragment using interpolated normals

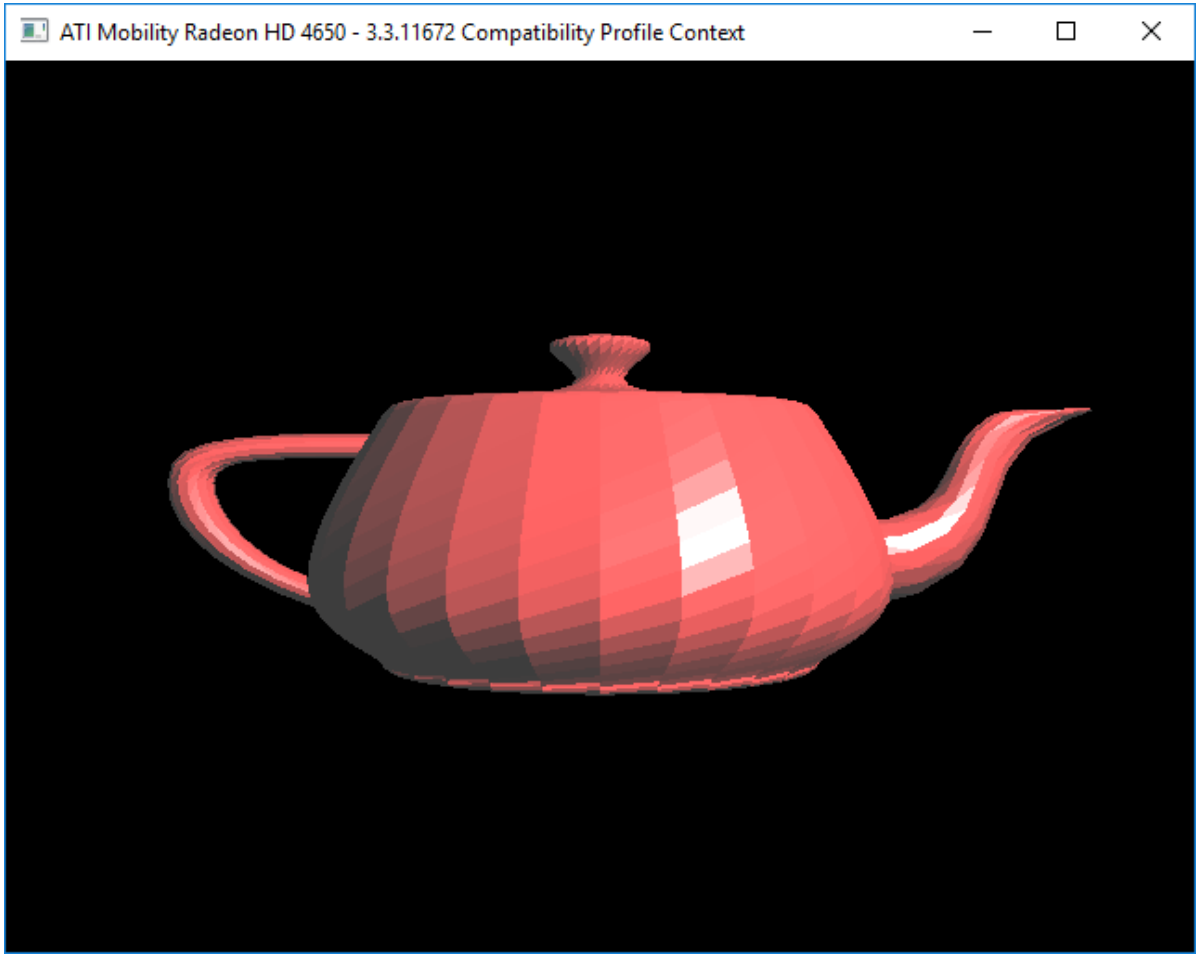
Gouraud Shading

- An alternative method is known as Gouraud shading
- **Basic idea:**
  - Compute shading per-vertex using vertex normal
  - Interpolate the resulting color

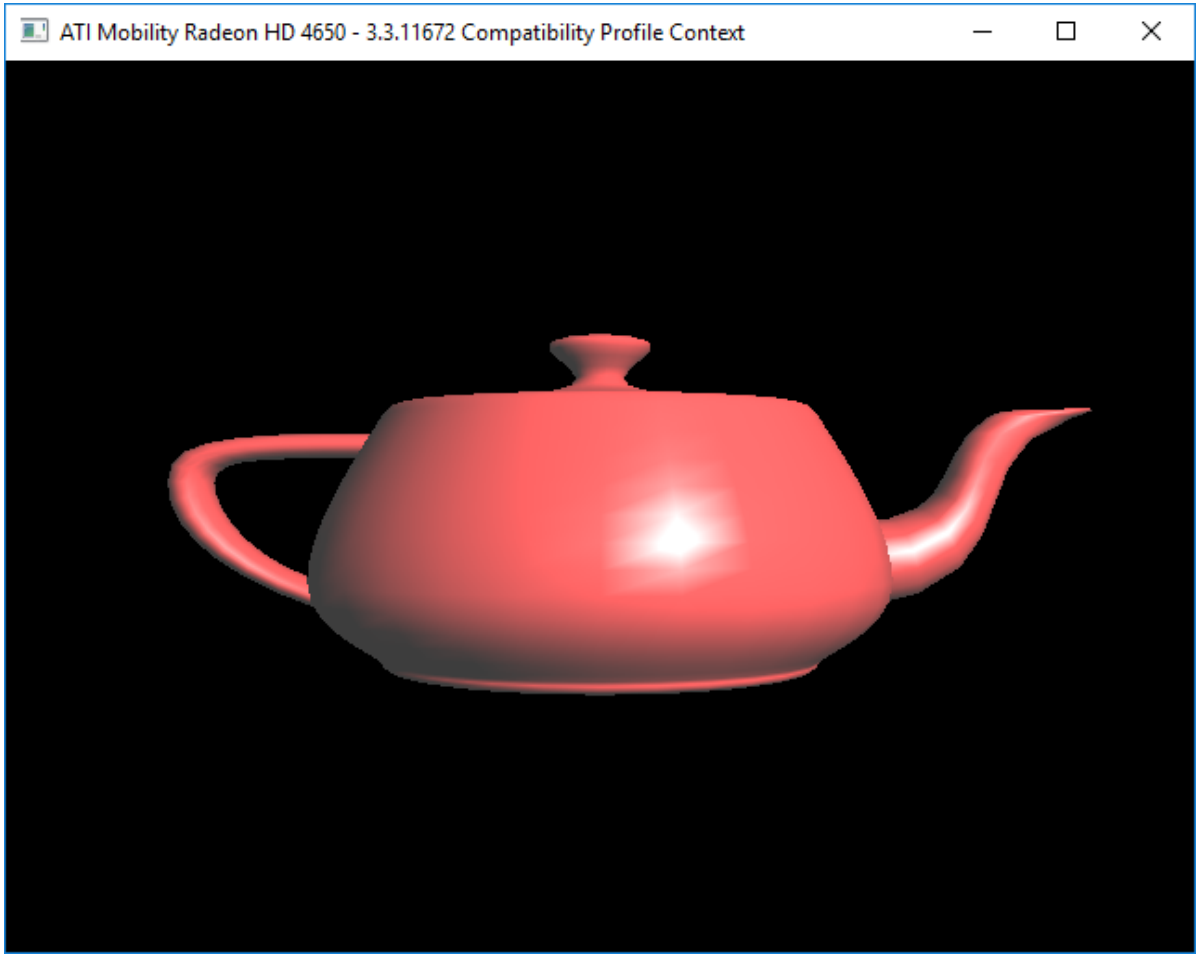
Flat Shading

- An even simpler method is known as flat shading
- **Basic idea:**
  - Compute shading at a single vertex of a primitive
  - Set the entire primitive color to the resulting value

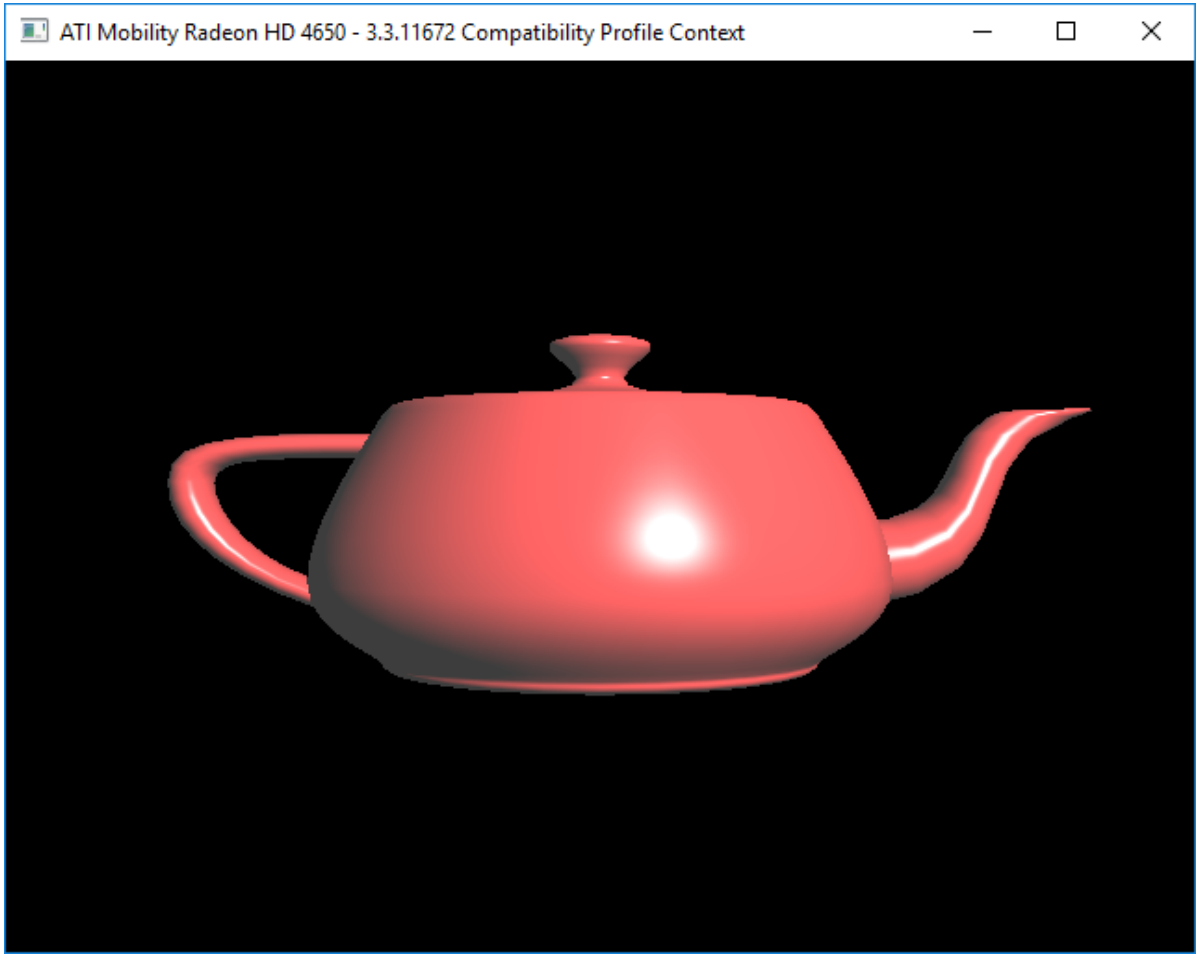
Comparison - Flat



Comparison - Gouraud

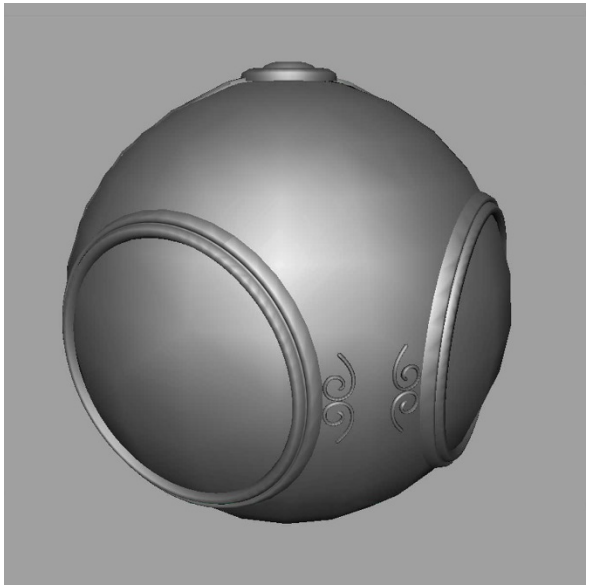


Comparison - Phong

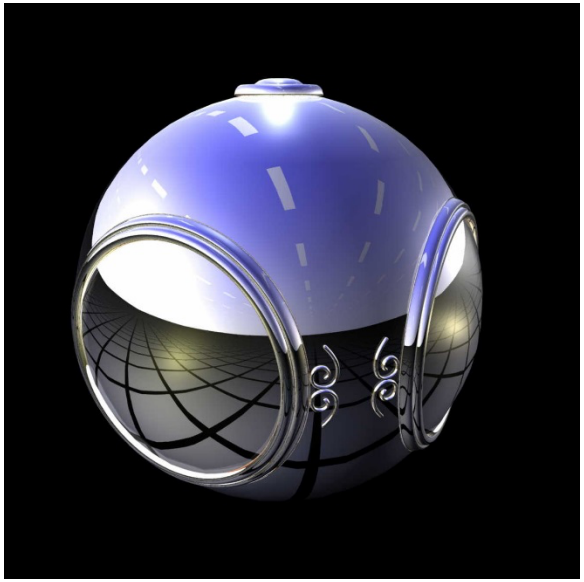


Shader Gallery III

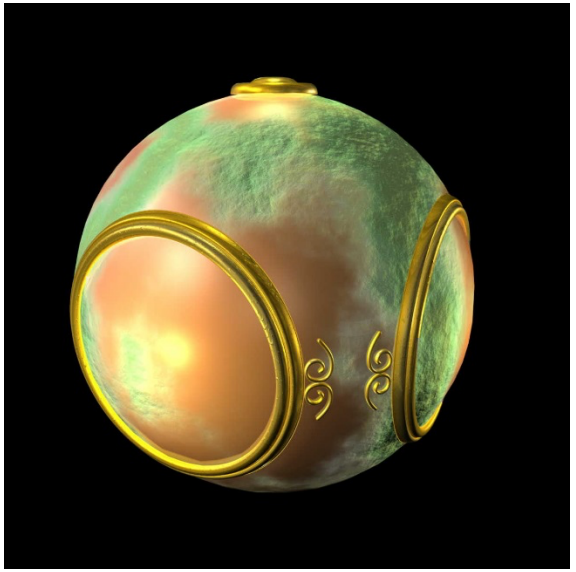
- Various fragment shading results



smooth shading



environment mapping



bump mapping

Learning GLSL

- GLSL looks like C in terms of syntax
- There are many special vector algebra functions
- There is no print function!
- Debugging typically involves converting the value of interest to a color and visually looking at it
- There are many good GLSL tutorials online:
  - E.g. http://www.lighthouse3d.com/tutorials/glsl-tutorial/
- In general, we only use a subset of the language
- Therefore, learning it in entirety is not required for this course