Shader Programming

Technical Game Development II

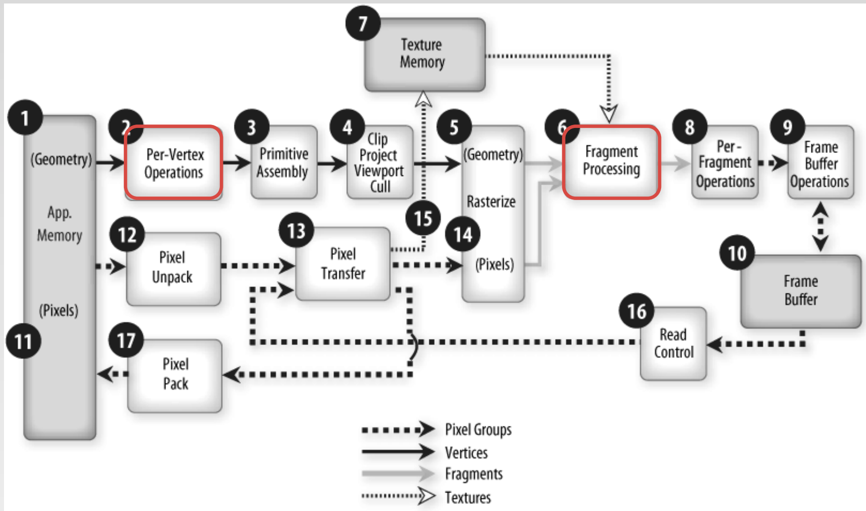
Professor Charles Rich
Computer Science Department
rich@wpi.edu

Reference: Rost, OpenGL Shading Language, 2nd Ed., AW, 2006
"The Orange Book"
Also take CS 4731 – Computer Graphics

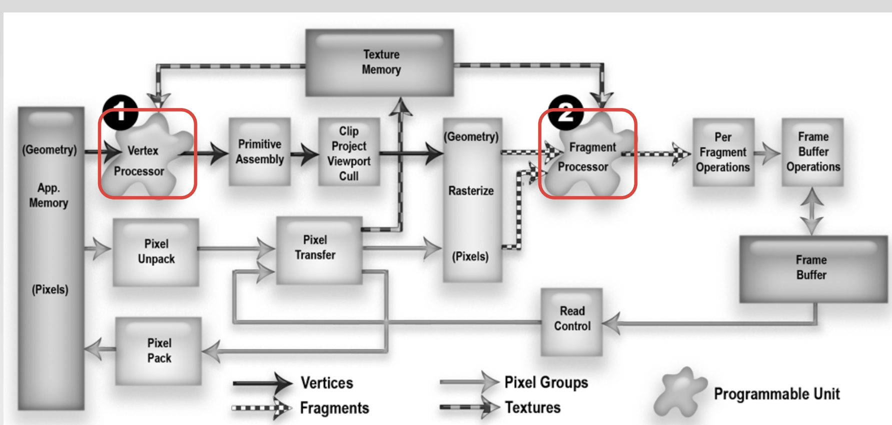
Shader Programming

- graphics hardware has replaced (1st generation) *fixed functionality* with *programmability* in:
  - *vertex* processing (geometry)
    - transformation
    - lighting
  - *fragment* (per-pixel) processing
    - reading from texture memory
    - procedurally computing colors, etc.
- OpenGL Shading Language (GLSL) is a open standard for programming such hardware
    - other languages, e.g., RenderMan, ShaderLab

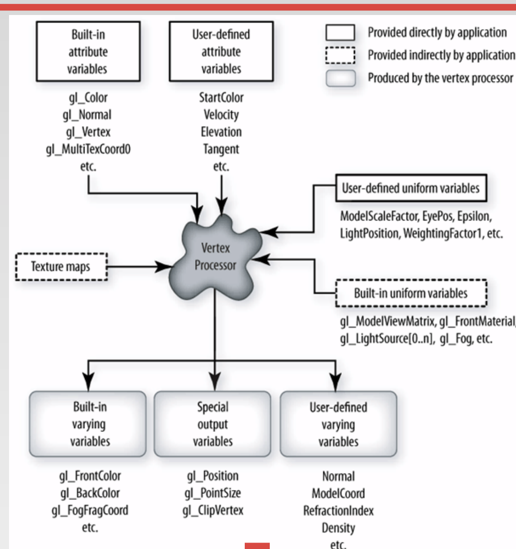
OpenGL "Fixed Functionality" Pipeline



OpenGL Programmable Processors



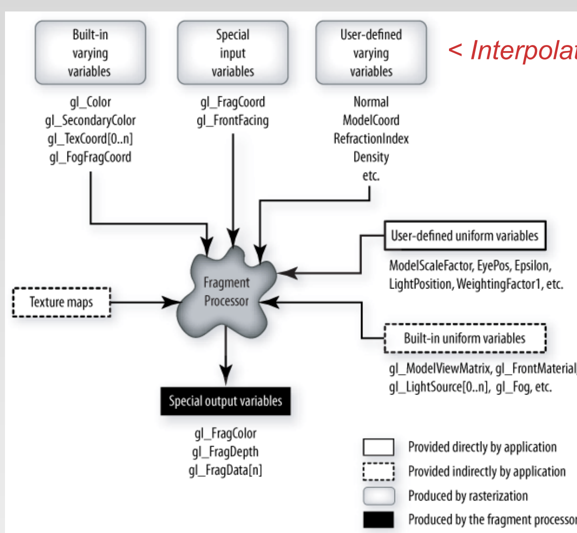
Vertex (Geometry) Processor



in parallel - per vertex only!

to rasterization

Fragment (Pixel) Processor



< Interpolated from vertices

in parallel - per fragment only!

GLSL Language

- Similar to C, C++
- Builtin vector and matrix operations:
  - vec2, vec3, vec4
  - mat2, mat3, mat4
- Texture memory lookup
  - sampler1D, sampler2D, sampler3D

Simple Shader Program Example

- Surface temperature coloring – "false color"
  - Assume *temperature* (user defined variable) is known at each vertex in model
  - smoothly color surface to indicate *temperature* at every pixel (using interpolation)
  - uses both a vertex and a fragment shader program working together (typical)
- Does coloring in *parallel* on GPU (*much* faster than using CPU)

Vertex Shader

```
// global parameters read from application
uniform float CoolestTemp;
uniform float TempRange;

// user-defined incoming property of this vertex
attribute float VertexTemp;

// "output" variable to communicate to the fragment shader
// (via interpolation - see scaling below)
varying float Temperature;

void main()
{
    // communicate this vertex's temperature scaled to [0.0, 1.0]
    Temperature = (VertexTemp - CoolestTemp) / TempRange;

    // don't move this vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```


Fragment Shader

```
// global parameters read from application
uniform vec3 CoolestColor;
uniform vec3 HottestColor;

// interpolated value from vertex shader
varying float Temperature;

void main()
{
    // compute a color using built-in mix() function
    vec3 color = mix(CoolestColor, HottestColor, Temperature);

    // set this pixel's raw color (with alpha blend of 1.0)
    gl_FragColor = vec4(color, 1.0);
}
```

Shader Execution

- Vertex shader is run once per vertex
- Vertex values are *interpolated* to get fragment values
- Fragment shader is run once per pixel
- Many such executions can happen *in parallel*
- *No* communication or ordering between parallel executions
  - no vertex-to-vertex
  - no pixel-to-pixel

Another Example: Adding Noise

- Using shader to change geometry (!)
- Moving vertices randomly a bit to simulate roughness
- More complicated vertex shader
- ”No-op” fragment shader

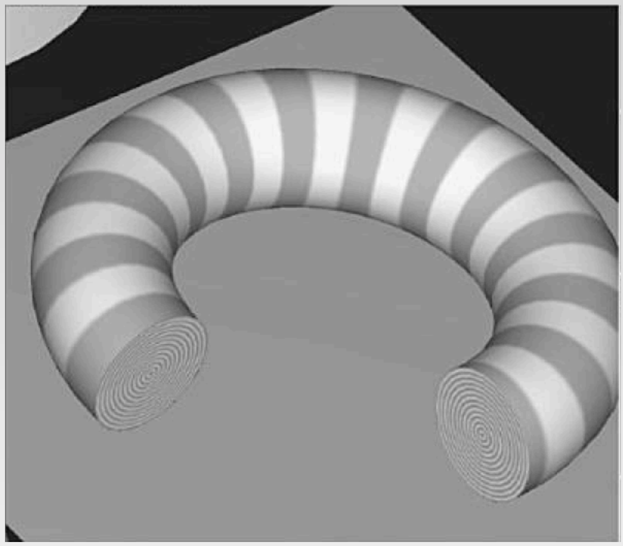
Vertex Shader

```
uniform vec3  LightPosition; // global application parameters
uniform vec3  SurfaceColor;
uniform vec3  Offset;
uniform float ScaleIn;
uniform float ScaleOut;
varying vec4  Color;  // output color for pixel shader

void main()
{
    vec3 normal = gl_Normal;
    vec3 vertex = gl_Vertex.xyz +
                  noise3(Offset + gl_Vertex.xyz * ScaleIn) * ScaleOut;

    // redo default color calculation based on new vertex location
    normal = normalize(gl_NormalMatrix * normal);
    vec3 position = vec3(gl_ModelViewMatrix * vec4(vertex,1.0));
    vec3 lightVec = normalize(LightPosition - position);
    float diffuse = max(dot(lightVec, normal), 0.0);
    if (diffuse < 0.125) diffuse = 0.125;
    Color = vec4(SurfaceColor * diffuse, 1.0);

    gl_Position = gl_ModelViewProjectionMatrix * vec4(vertex,1.0);
}
```


"No-Op" Fragment Shader

```
varying vec4 Color;

void main()
{
    gl_FragColor = Color;
}
```

Procedural Textures - Stripes



Fragment Shader for Stripes

```
uniform vec3  StripeColor; // global application parameters
uniform vec3  BackColor;   // that define striping pattern
uniform vec3  Width;
uniform float Fuzz;
uniform float Scale;

varying vec3 DiffuseColor; // inputs from vertex shader
varying vec3 SpecularColor;

void main()
{
    float scaledT = fract(gl_TexCoord[0].t * Scale);

    float frac1 = clamp(scaledT / Fuzz, 0.0, 1.0);
    float frac2 = clamp((scaledT - Width) / Fuzz, 0.0, 1.0);

    vec3 finalColor = mix(BackColor, StripeColor, frac1)
    finalColor = finalColor * DiffuseColor + SpecularColor;

    gl_FragColor = vec4(finalColor, 1.0);

}
```

Shaders in Unity

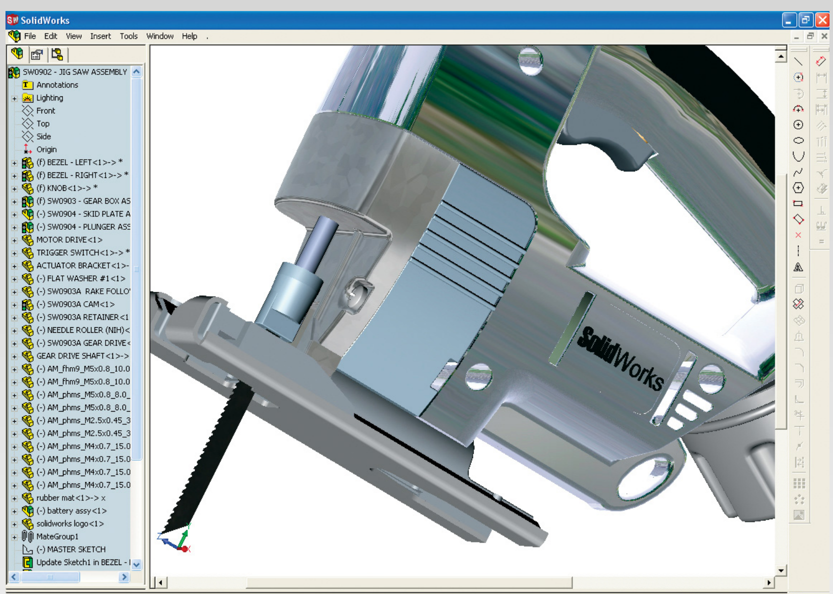
- predefined shaders (via GUI's)
- can write your own in shader in ShaderLab Cg/HLSL (very similar to GLSL)
- coding your own very simple shader counts as an optional tech element
- see http://docs.unity3d.com/Documentation/Manual/Shaders.html

Lots More You Can Do With Shaders

- Procedural Textures
  - patterns (stripes, etc.)
  - bump mapping
- Lighting Effects
- Shadows
- Surface Effects
  - refraction, diffraction
- Animation
  - morphing
  - particles

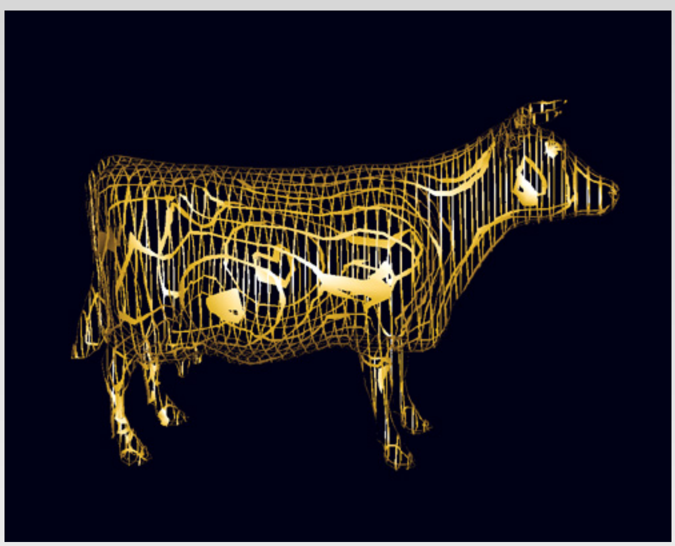
Lots More ...

- Anti-aliasing
- Non-photorealistic effects
  - hatching, meshes
  - technical illustration
- Imaging
  - sharpen, smooth, etc.
- Environmental effects (RealWorldz)
  - terrain
  - sky
  - ocean

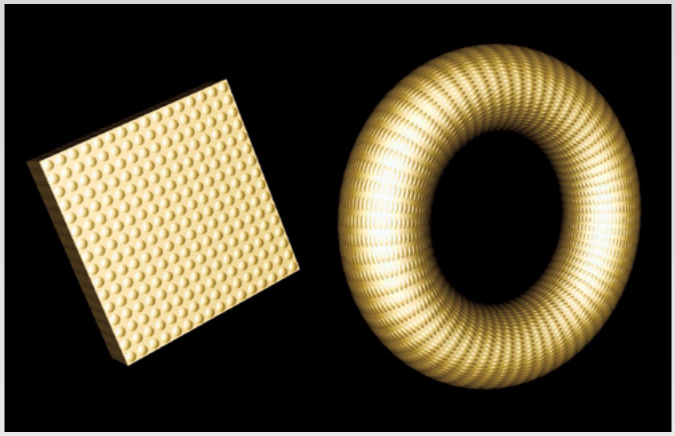


Screen shot of the SolidWorks application, showing a jigsaw rendered with OpenGL shaders to simulate a chrome body, galvanized steel housing, and cast iron blade. (Courtesy of SolidWorks Corporation)

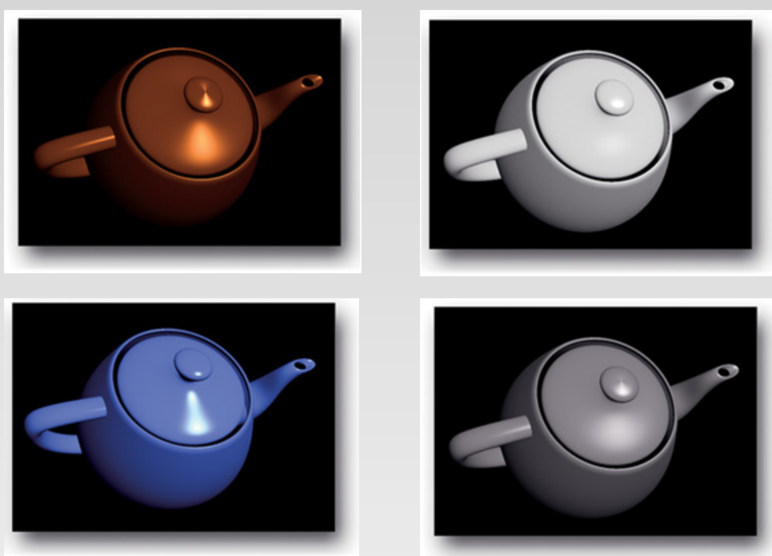
Different glyphs applied to a cube using the glyph bombing shader described in Section 10.6. (3Dlabs, Inc.)



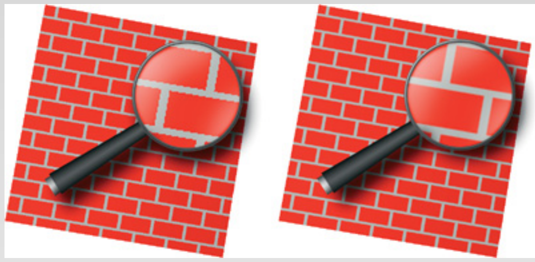
The lattice shader presented in Section 11.3 is applied to the cow model. (3Dlabs, Inc.)



A simple box and a torus that have been bump-mapped using the procedural method described in Section 11.4. (3Dlabs, Inc.)



A variety of materials rendered with Ward's BRDF model (see Section 14.3) and his measured/fitted material parameters.



Brick shader with and without antialiasing. On the left, the results of the brick shader presented in Chapter 6. On the right, results of antialiasing by analytic integration using the brick shader described in Section 17.4.5. (3Dlabs, Inc.)

IMGD 4000 (B 12) 25

A variety of screen shots from the 3Dlabs RealWorldz demo. Everything in this demo is generated procedurally using shaders written in the OpenGL Shading Language. This includes the planets themselves, the terrain, atmosphere, clouds, plants, oceans, and rock formations. Planets are modeled as mathematical spheres, not height fields. These scenes are all rendered at interactive rates on current generation graphics hardware

IMGD 4000 (B 12) 26

Shader Programming - Summary

- Seems to lie on the boundary between art and tech
- programming is hard-core (parallel algorithms)
- but intended result is often mostly aesthetic