

# Programmable Shading



# Lighting, Texturing, Shading

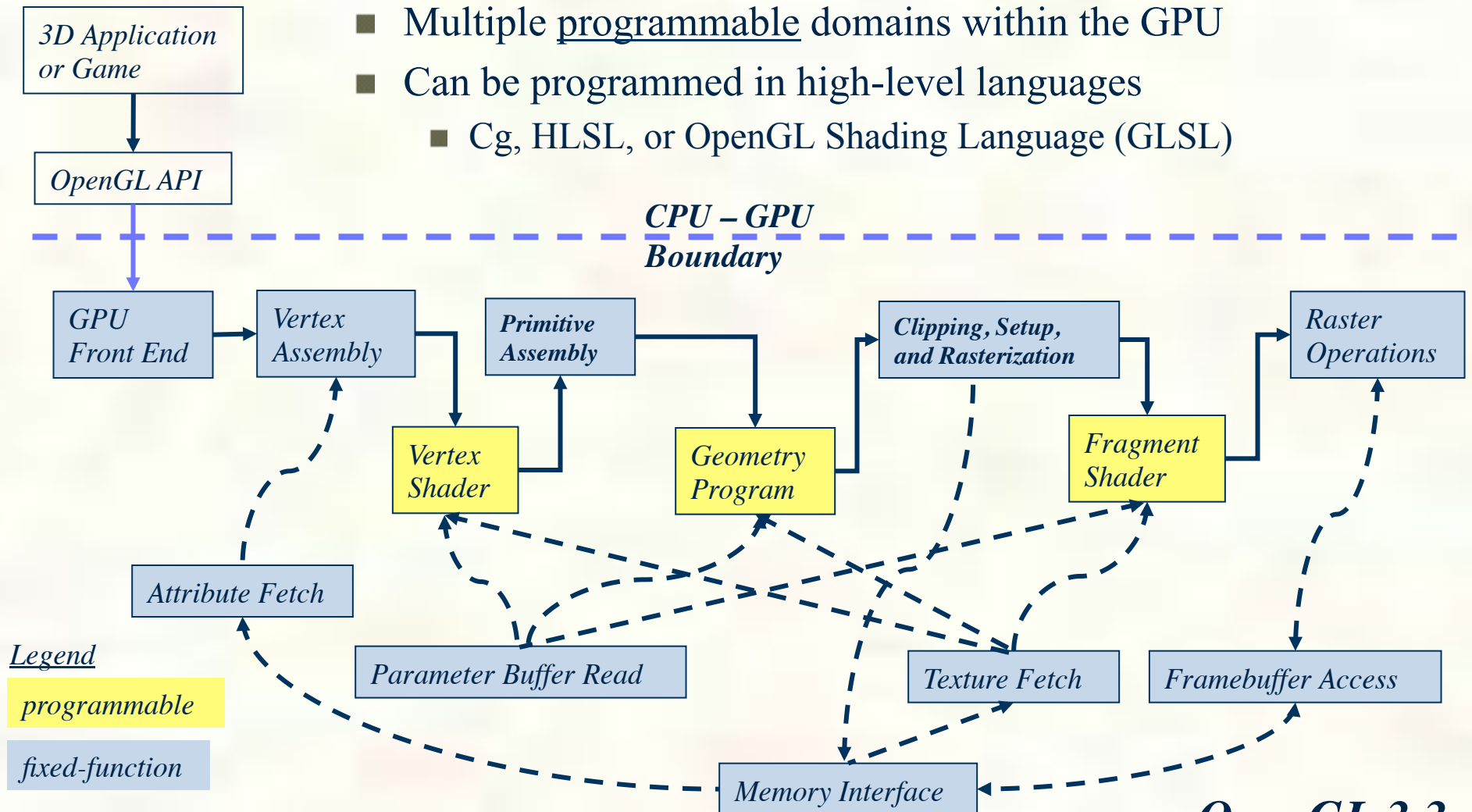
---

- Discussed
  - Transformation, Texturing, and Lighting
- What if...
  - We could write a program that controls these
- What if...
  - These programs could execute in dedicated hardware
- What if...
  - We could write these programs in a high-level language
- That's what shaders are about!



# Programming Shaders inside GPU

- Multiple programmable domains within the GPU
- Can be programmed in high-level languages
  - Cg, HLSL, or OpenGL Shading Language (GLSL)



**OpenGL 3.3**



# Example Simple GLSL Shaders

## ■ Vertex Shader

- Operates on each vertex of geometric primitives

```
void main(void)
{
    gl_FrontColor = gl_Color;
    gl_Position =
    ftransform();
}
```

- Passes through per-vertex color
- Transforms the vertex to match fixed-function processing

## ■ Fragment Shader

- Operates on each fragment (think pixel)

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

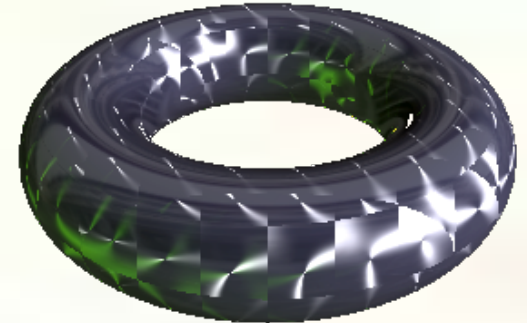
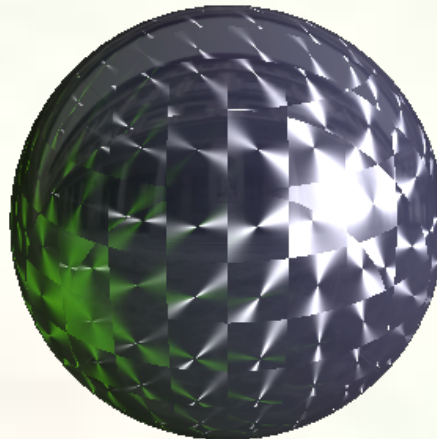
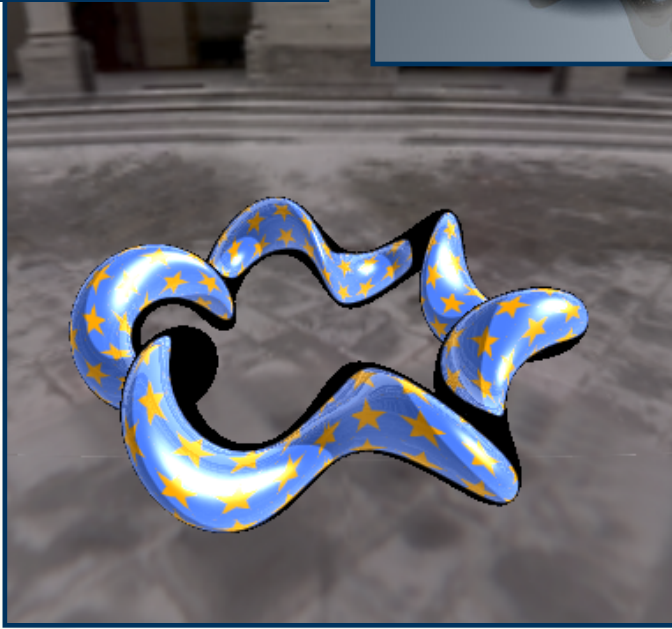
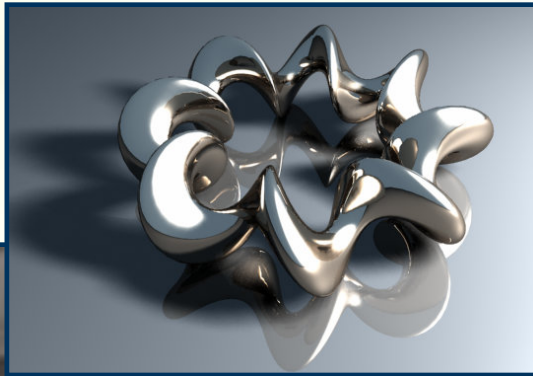
- Outputs the fragment's interpolated color to the framebuffer

*Shaders are way more interesting than these minimal examples*





# Examples of Complex Shaders





# Building Up Shaders

$$\left( \begin{array}{c} \text{Diffuse} \\ \text{Decal} \end{array} \right) \times \left( \begin{array}{c} \text{Specular} \\ \text{Gloss} \end{array} \right) =$$

*Diffuse*      *Decal*      *Specular*      *Gloss*



*Result*



# Shade Trees

- Lucasfilms [Cook '84] proposes using a tree structure to represent shading expressions
  - Proposed a specialized language to represent shading expressions

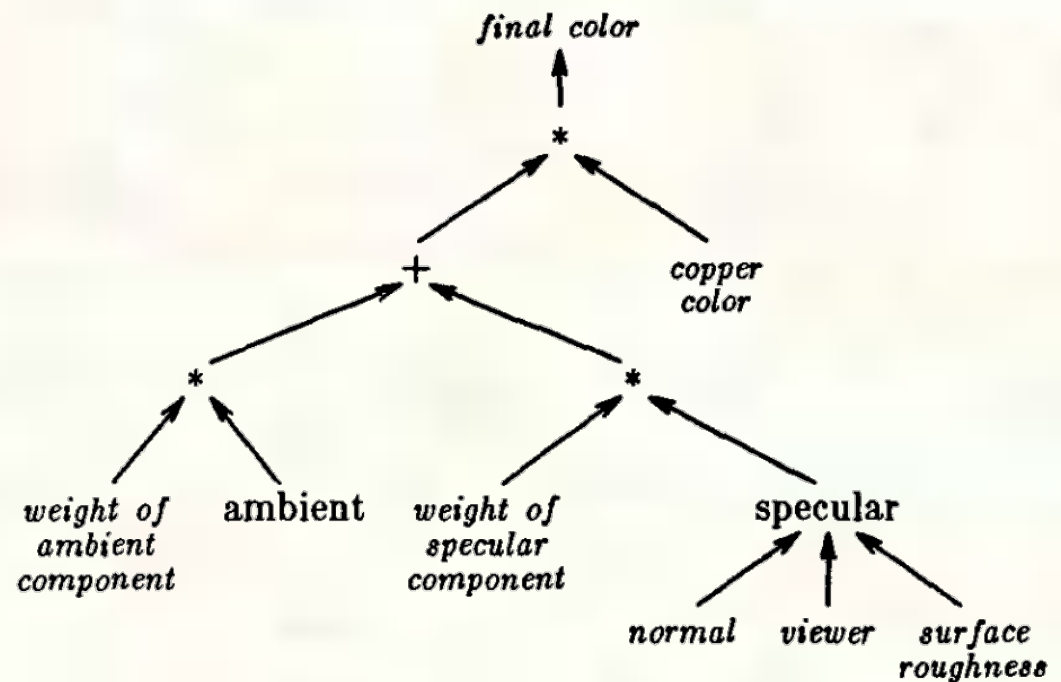


Figure 1a. Shade tree for copper.



# Pioneering Shading Language

- RenderMan Shading Language
  - Developed by Pixar
    - Academy Award winning!
  - For production rendering of animated films
    - Still in use
  - Intended for CPU-based rendering system
    - Not oriented for graphics hardware
    - Assumes a micro-polygon style renderer





# OpenGL Shading Language

---

- Known as GLSL
  - Part of core OpenGL 2.0 and beyond
- Uses a C-like language
  - Authoring of vertex and fragment shaders
    - Later added geometry shaders (GL 3.0) and tessellation shaders (GL 4.0)





# OpenGL Versions, Extensions

- OpenGL is constantly evolving
  - New core version updates:
    - 1.0, 1.1, 1.2,... 2.0, 2.1,... 3.0, ... 4.2
    - Updates happening every year now!
  - OpenGL also supports extensions
- Using new versions and extensions
  - Possible using “GetProcAddress” API to ask driver for entry-point for new commands
    - Avoids probably with static OpenGL library linking
  - Particularly a problem on Windows
    - Because Microsoft has not updated OpenGL since 1.1!
      - Benign neglect sometimes better than other alternative ☺
  - Still Linux and Mac support a “GetProcAddress” mechanism too
- **Solution:** OpenGL Extension Wrangler Library (GLEW)
  - Regularly updated to support all available OpenGL extensions and versions
  - Linking with GLEW keeps you from dealing with “GetProcAddress” hassles
- Details
  - Link with -lGLEW
  - Call glewInit() right after creating an OpenGL context
  - Call OpenGL new version and extension routines
  - That’s basically it
- Open source
  - <http://glew.sourceforge.net/>



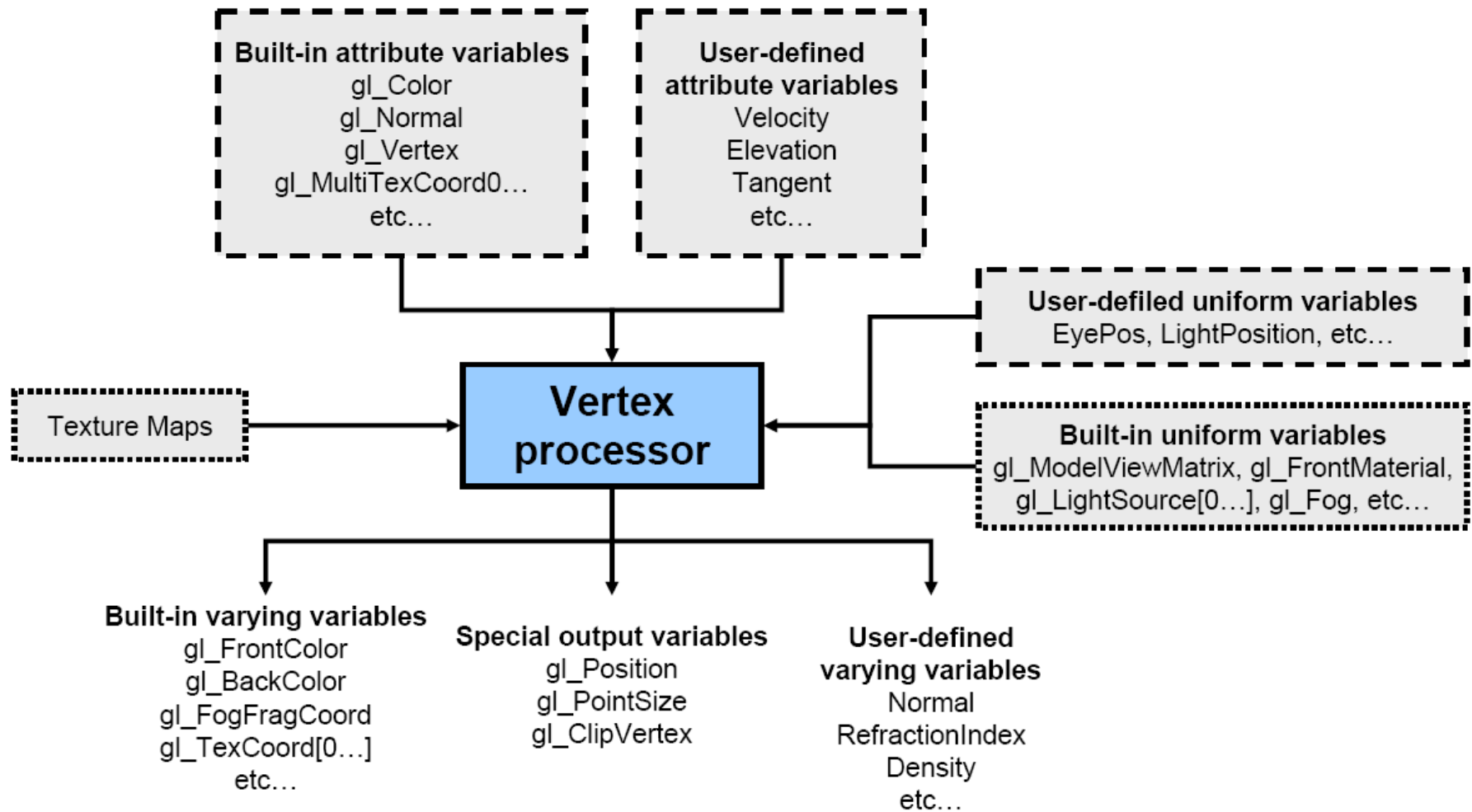
# GLSL Language Overview

---

- C-like
  - C-style syntax and control flow
- Adds graphics features
  - First-class vertex and matrix data types
  - Texture samplers
  - Graphics-oriented standard library
    - Including built-in OpenGL state variables
  - Uniform and varying parameters, interpolation qualifiers
- Various modern language features
  - `bool`, `//` comments, limited overloaded functions
  - `in/out/inout` function parameters
- Minus features of C not suited for GPUs
  - Pointers, unions, multi-dimensional arrays
  - `goto`, string literals, standard C library stuff like `printf` and `malloc`



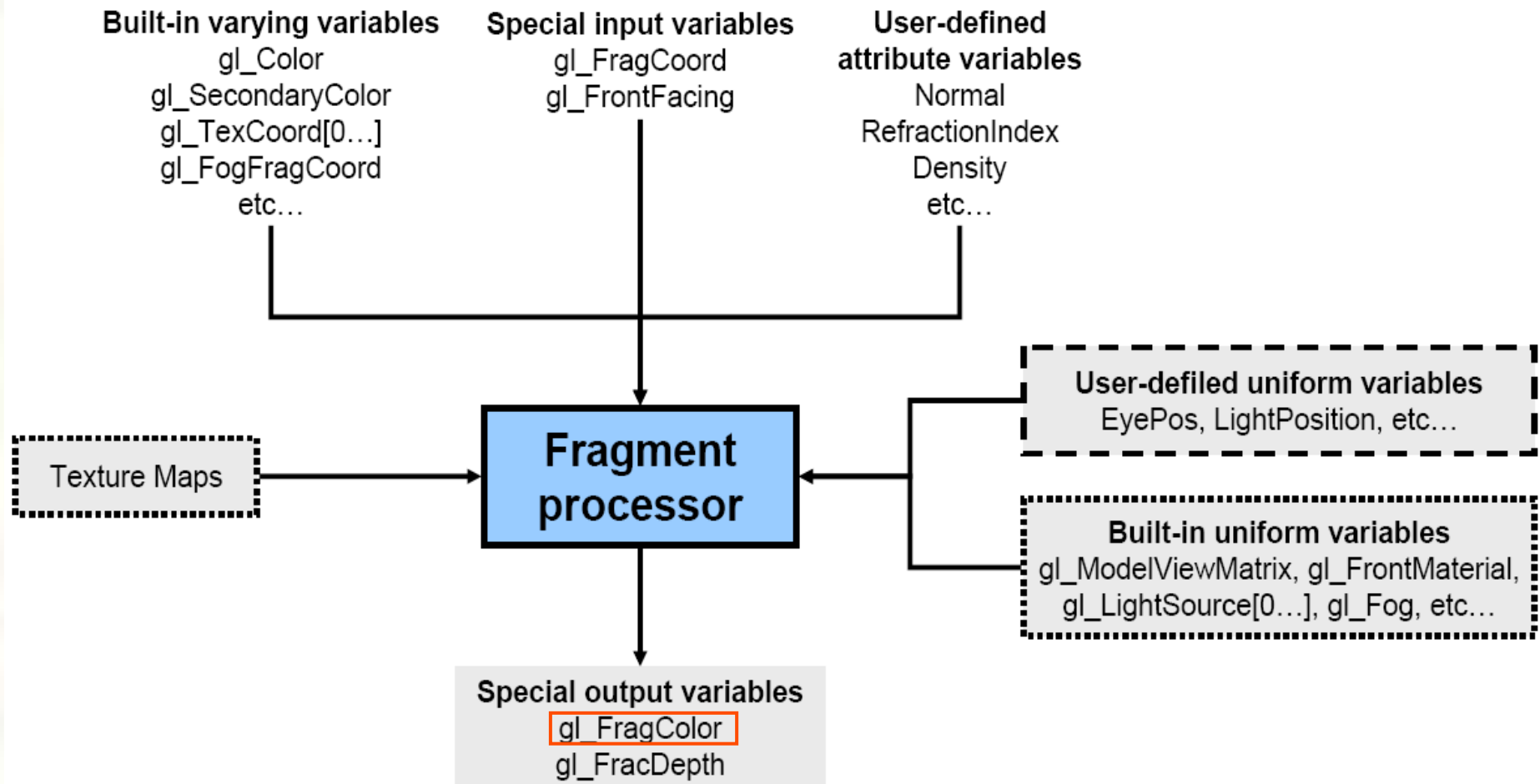
# Vertex Shader Inputs & Outputs





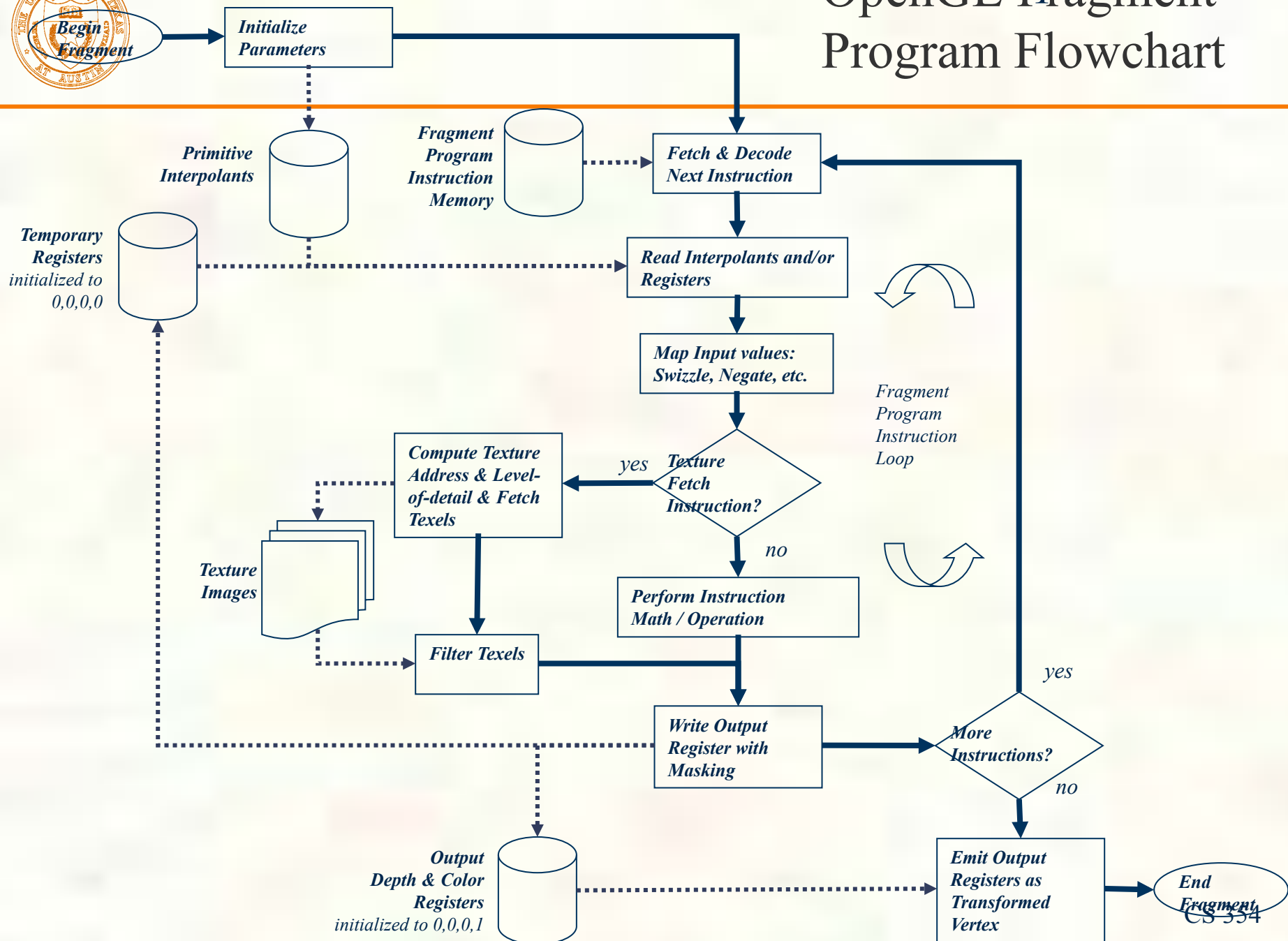


# Fragment Shader Ins & Outs





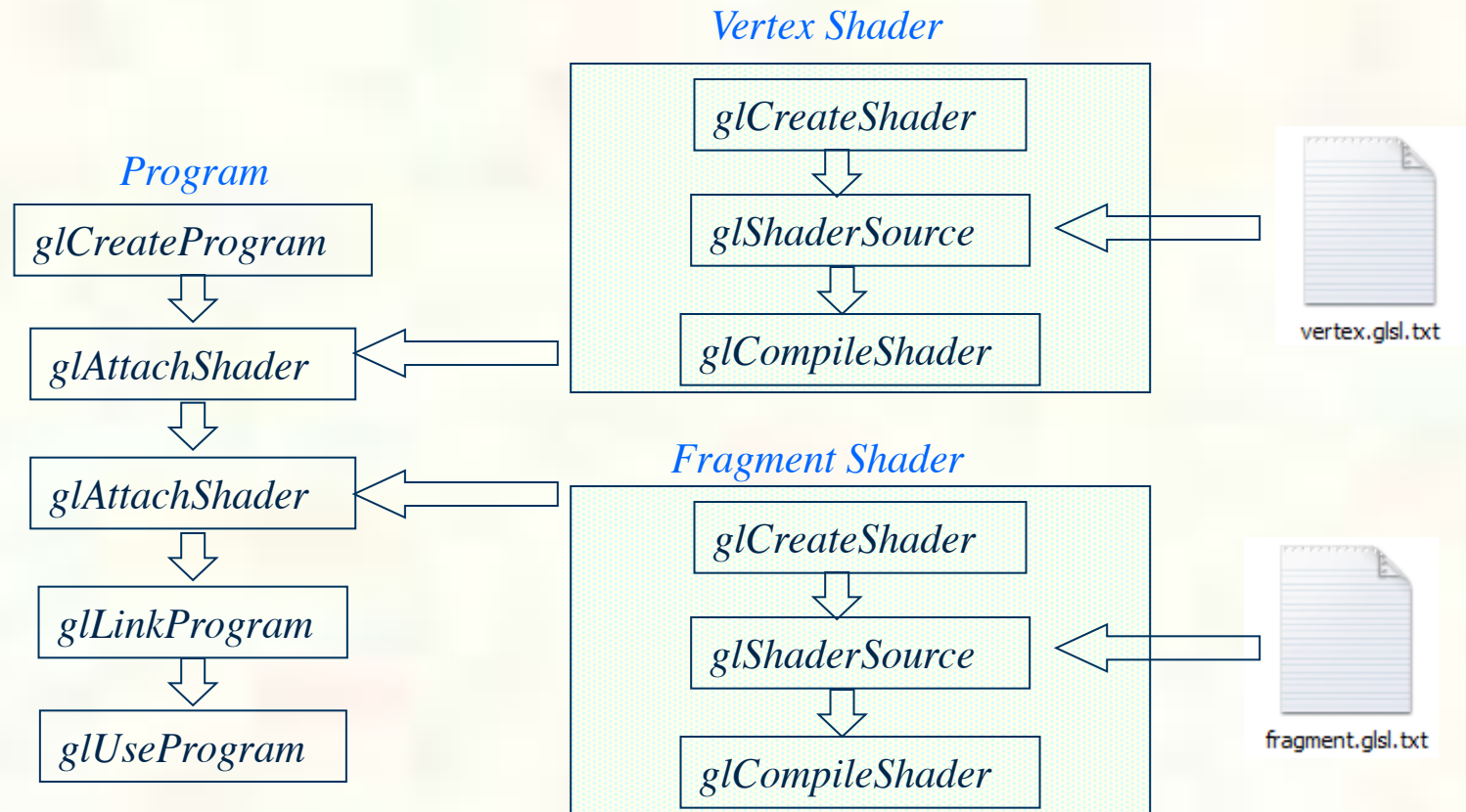
# OpenGL Fragment Program Flowchart





# Creating GLSL Programs

- Create & compile vertex & fragment shader
- Attach shaders & link program





# GLSL Implementation

- GLSL is built into OpenGL drivers
  - Means your graphics driver contains an optimizing compiler for a high-level language
    - Targeting a complex, dedicate processor
    - What could possibly go wrong?
      - Well, you might want to test your shaders on hardware from different vendors (hint, careful about Intel Integrated Graphics)
      - But fairly mature at this point, assuming good drivers
- GLSL shaders compiled to hardware-dependent shader micro-code
  - Details all hidden from the OpenGL application programmer
  - Provides very little visibility into compiled result
    - NVIDIA's Cg Toolkit contains compiler for Cg and GLSL code that can show you an assembly level view of your program
      - Use `-ogls` option to accept GLSL language into `cgc` compiler
- Shaders execution in *Single Program, Multiple Data* (SPMD) model
  - Means (in latest GPUs) that each shader instance can branch differently
    - More powerful than *Single Instruction, Multiple Data* (SIMD) model



# Vector Data Types

- Vector data types built into language
  - First class
  - Supports swizzling, masking, and operators
    - Swizzling/mask example: `foo.zyx = foo.xxy`
    - Operators like `+`, `-`, `*`, and `/` do component-wise operations
      - Also supports vector-with-scalar operations
  - Type names
    - Floating-point vectors: `vec2`, `vec3`, `vec4`
    - Integer vectors: `ivec2`, `ivec3`, `ivec4`
    - Double-precision vectors: `dvec2`, `dvec3`, `dvec4`
- Lots of standard library support
  - `dot`, `length`, `normalize`, `reflect`, etc.
  - `sin`, `cos`, `rsqrt`, etc.



# Vector Details

---

- Swizzle forms
  - `.x`, `.y`, `.z`, `.w`
  - `.r`, `.g`, `.b`, `.a`
  - `.s`, `.t`, `.p`, `.q`
- Create with C++ style constructors
  - **Example:** `vec3 foo = foo(1.0, 2.0, 3.0)`
  - Aggregate initializers aren't allowed
    - So cannot say: `vec3 foo = { 1, 2, 3 }`



# Matrix Data Types

- Matrix data types also built into language
  - **Floating-point matrices:** `mat2`, `mat3`, `mat4`
    - Also `mat2x2`, `mat3x3`, `mat4x4`
  - **Non-square matrices:** `mat4x2`, etc.
- Operator overloading
  - `*` does matrix-by-vector and matrix-by-matrix multiplication
- Matrices are stored **column-major**
  - Defying the convention of C/C++ 2D arrays
- Use matrix constructors
  - **Example:** `mat2x2 foo = mat2x2(1.0, 2.0, 3.0, 4.0)`
- Access matrix elements via swizzles and/or indexing
  - **Example:** `foo[0].y` or `foo[0][1]`



# Samplers

---

- Samplers = opaque objects
  - Provides access textures and fetching texels
  - Type names:
    - **Basic:** `sampler1D`, `sampler2D`, `sample3D`
    - **Cube maps:** `samplerCube`
    - **Shadow maps:** `sampler2DShadow`, etc.
    - **Rectangle:** `sampler2DRect`, etc.
    - **Array textures:** `sampler2DArray`, etc.
- Standard library routines
  - `texture2D`, `textureCube`, `texture3D`, etc.
  - Returns a 4-component vector, typically a color





# Type Qualifiers

- Shaders are excepted to source inputs and write outputs—that's what shaders do
  - Type qualifiers identify these special variables
  - Vertex input qualifier: **attribute**
  - Vertex-fragment interface qualifier: **varying**
    - Also interpolation modifiers: **centroid**, **flat**, **noperspective**
  - Shader parameters initialized by driver: **uniform**
- Newer usage is **in** for attribute; **out** for vertex shader varying; **in** for fragment shader varying
  - One problem with GLSL is deprecation
  - GLSL designers don't respect compatibility
    - Hard to write a single shader that works on multiple GLSL versions



# Other Details

---

- C preprocessor functionality available
- Has its own extension and version mechanism
  - `#version`
  - `#extension`
    - `require`, `enable`, `warn`, `disable`
- Type qualifiers for controlling precision
  - `lowp`, `mediump`, `highp`—mainly for embedded GPUs
- Entry function must be named `main`



# Consider a Light Map Shader



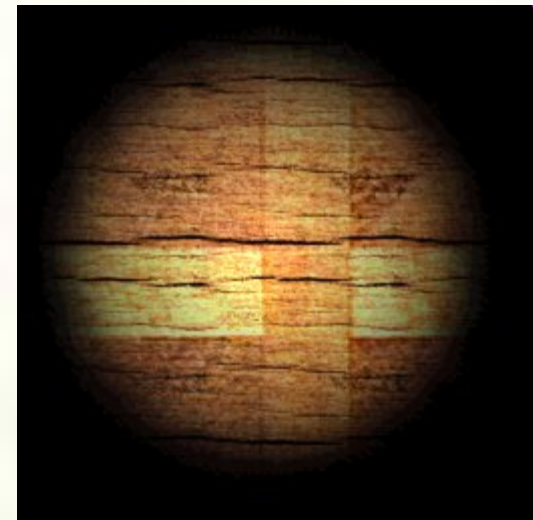
*Precomputed light*

$\times$



*Surface color*

$=$



*"lit" surface*

*Images from: <http://zanir.wz.cz/?p=56&lang=en>*

- Multiply two textures component-wise



# Light Map - Fixed Function OpenGL

## ■ *Application code making OpenGL calls*

```
GLuint lightMap;  
GLuint surfaceMap;  
  
glActiveTexture(GL_TEXTURE0);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, lightMap);  
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
glActiveTexture(GL_TEXTURE1);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, surfaceMap);  
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
glDrawArrays(...);
```



# Light Map - Fixed Function OpenGL

## ■ *Application code making OpenGL calls*

```
GLuint lightMap;  
GLuint surfaceMap;
```

```
glActiveTexture(GL_TEXTURE0);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, lightMap);  
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
glActiveTexture(GL_TEXTURE1);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, surfaceMap);  
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
glDrawArrays(...);
```

*Tell fixed function we are  
using texture mapping*

*Tell fixed function how to  
combine textures*



# Light Map Shader in GLSL

---

## ■ Write a fragment shader in GLSL

```
#version 330

uniform sampler2D lightMap;
uniform sampler2D surfaceMap;

in vec2 fs_TxCoord;
out vec3 out_Color;

void main(void)
{
    float intensity = texture2D(lightMap, fs_TxCoord).r;
    vec3 color = texture2D(surfaceMap, fs_TxCoord).rgb;
    out_Color = intensity * color;
}
```



# Light Map Shader in GLSL

## ■ Write a fragment shader in GLSL

```
#version 330
```

*GLSL version 3.3*

```
uniform sampler2D lightMap;  
uniform sampler2D surfaceMap;
```

*Textures (input)*

```
in vec2 fs_TxCoord;  
out vec3 out_Color;
```

*Per-fragment input*

*shader output*

```
void main(void)
```

```
{
```

```
    float intensity = texture2D(lightMap, fs_TxCoord).r;
```

```
    vec3 color = texture2D(surfaceMap, fs_TxCoord).rgb;
```

```
    out_Color = intensity * color;
```

```
}
```

*one channel intensity*

*three channel color*

*modulate*



# Switching to GLSL Shaders

- *Recall the fixed function light map in C/C++*
  - *What code can be eliminated?*

```
GLuint lightMap;  
GLuint surfaceMap;
```

```
✗ glActiveTexture(GL_TEXTURE0);  
✗ glEnable(GL_TEXTURE_2D);  
✗ glBindTexture(GL_TEXTURE_2D, lightMap);  
✗ glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
✗ glActiveTexture(GL_TEXTURE1);  
✗ glEnable(GL_TEXTURE_2D);  
✗ glBindTexture(GL_TEXTURE_2D, surfaceMap);  
✗ glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
  
glDrawArrays(...);
```





# Switching to GLSL Shaders

---

## ■ *Added code to use GLSL shaders*

```
GLuint lightMap;  
GLuint surfaceMap;  
→ GLuint program;
```

```
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_2D, lightMap);
```

```
glActiveTexture(GL_TEXTURE1);  
glBindTexture(GL_TEXTURE_2D, surfaceMap);
```

```
→ glUseProgram(program);  
glDrawArray(...);
```



# Careful: What's not shown

---

- This example cuts a number of corners
- The example leaves out code for
  - Initializing and loading the image data for the two textures
  - The vertex shader that outputs the varying `fs_TxCoord` texture coordinate set
  - GLSL shader compilation and linking code to create the `program` object
  - Setting the sampler units of `lightMap` and `surfaceMap` to point at texture units 1 and 2
    - Use `glUniform1i` for this



# Geometry Shaders

## Vertex Color

```
gl_FrontColorIn[gl_VerticesIn];  
gl_BackColorIn[gl_VerticesIn];  
gl_FrontSecondaryColorIn[gl_VerticesIn];  
gl_BackSecondaryColorIn[gl_VerticesIn];  
gl_FogFragCoordIn[gl_VerticesIn];
```

## Vertex Coord.

```
gl_TexCoordIn[gl_VerticesIn][];  
gl_PositionIn[gl_VerticesIn];
```

## Rasterization Info.

```
gl_PointSizeIn[gl_VerticesIn];  
gl_ClipVertexIn[gl_VerticesIn];
```

**Number of Vertices**  
`gl_VerticesIn`

**Geometry  
processor**

## Color

```
gl_FrontColor;  
gl_BackColor;  
gl_FrontSecondaryColor;  
gl_BackSecondaryColor;  
gl_FogFragCoord;
```

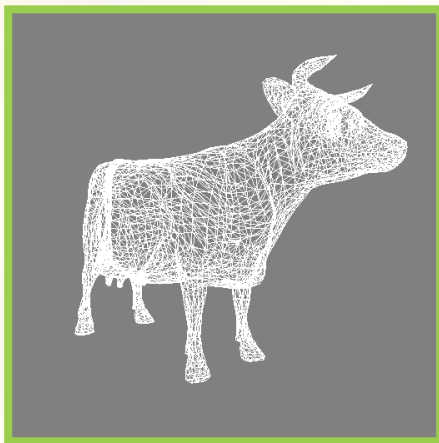
## Coord.

```
gl_Position  
gl_TexCoord[];
```



# Silhouette Rendering

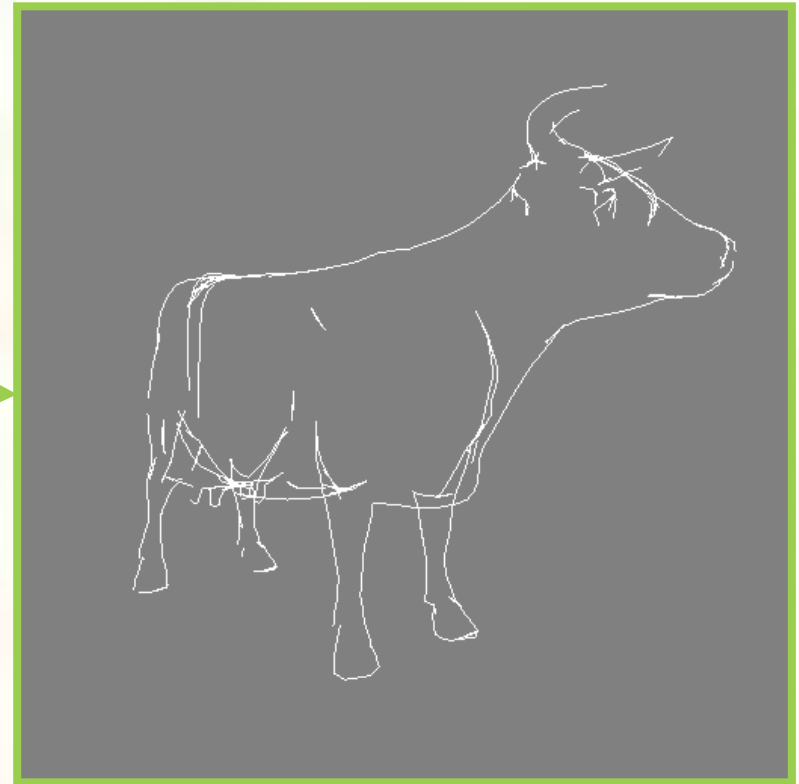
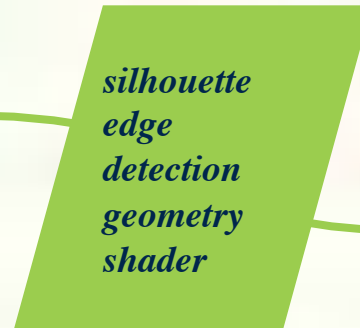
- *Uses geometry shader*



*Complete mesh*

*Useful for non-photorealistic rendering*

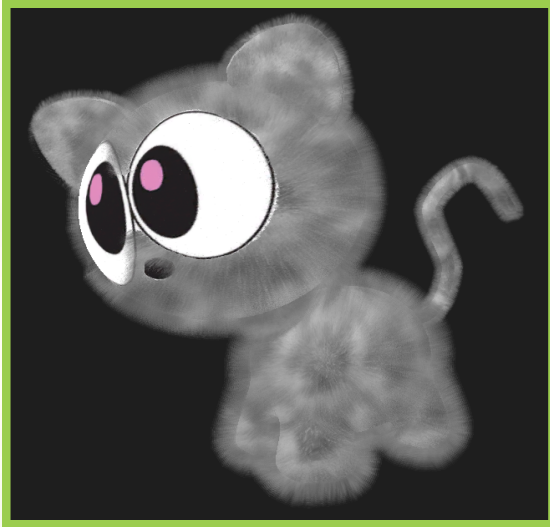
*Mimics artistic sketching*



*Silhouette edges*



# More Geometry Shader Examples



*Generate  
shells for  
fur  
rendering*



*Generate  
fins for  
lines*



*Shimmering  
point sprites*



# Geometry shader setup with and without per-vertex normals

34



*Setup without per-vertex normals, relying on normalized gradients only; faceted look*

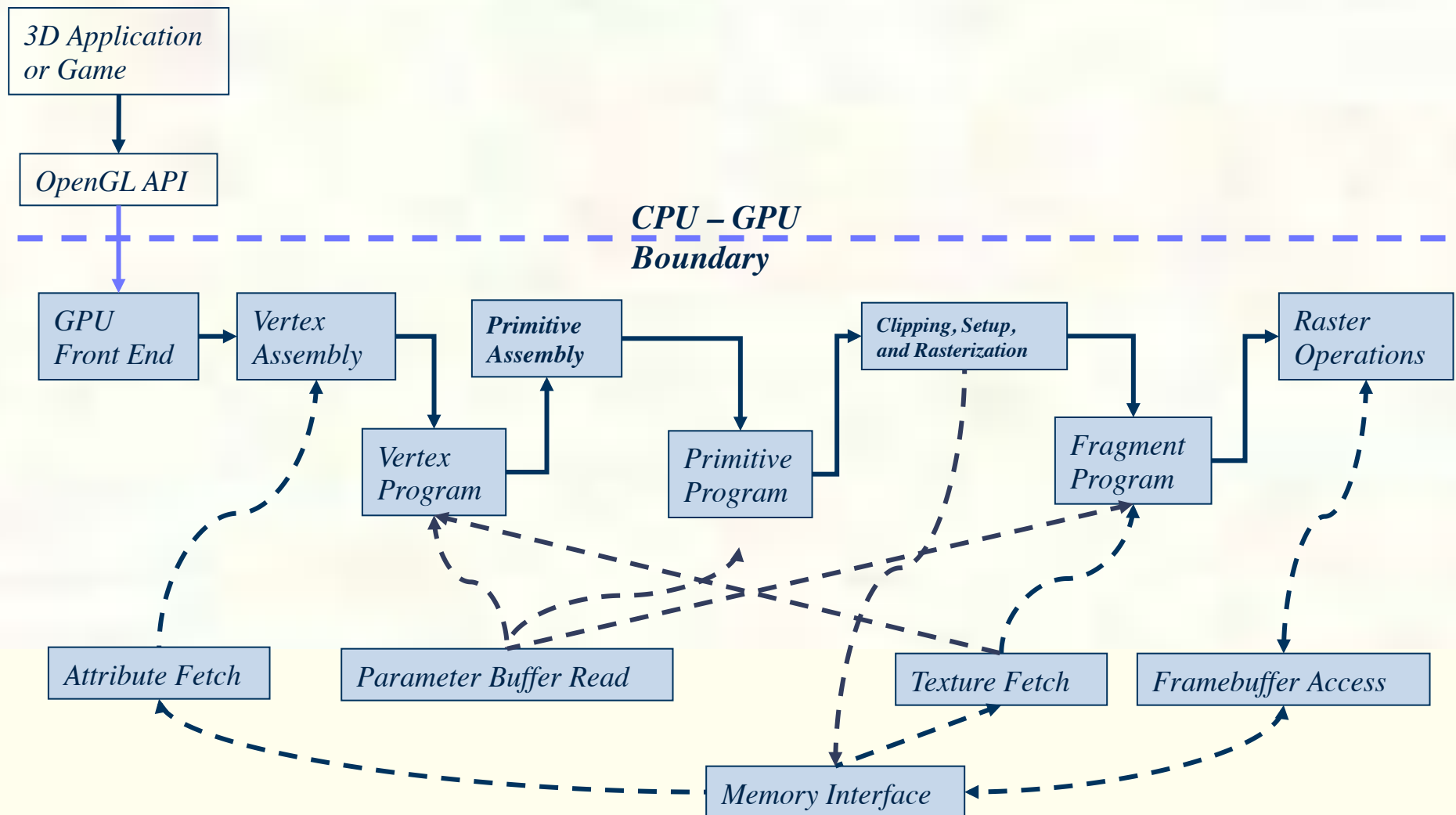


*Setup with per-vertex normals; smoother lighting appearance*

CS 354



# OpenGL 3.0 View of Hardware





# OpenGL 4.0 Double-Precision

---

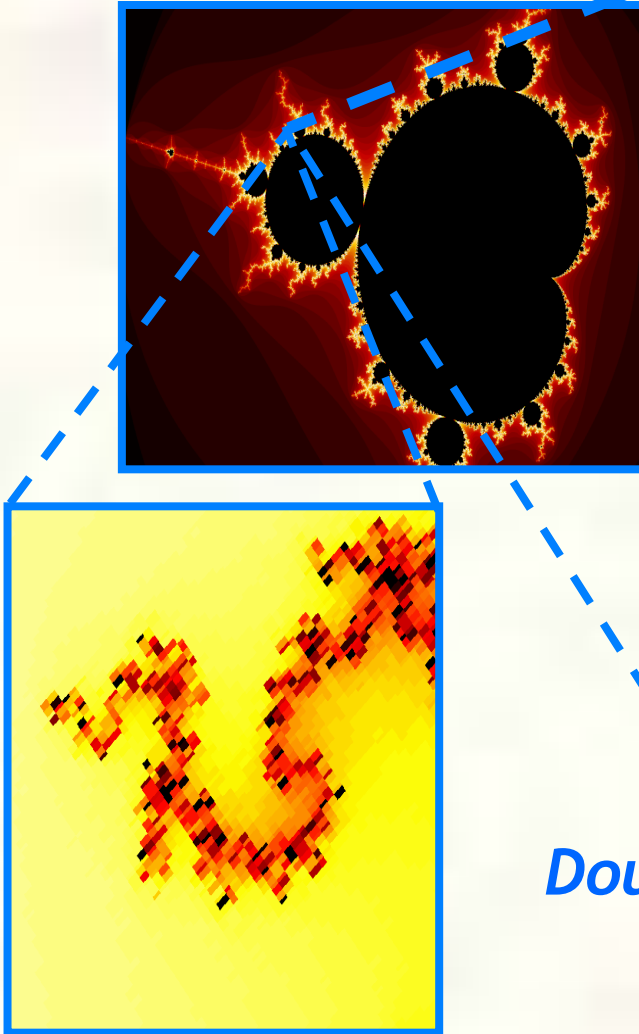
- Double-precision data types
  - Scalar `double`
  - Vector `dvec4`, etc
  - Matrix `dmat3` (3x3), `dmat4x2`, etc.
  - Doubles can reside in buffer objects
- Double-precision operations
  - Multiply, add, multiply-add (MAD)
  - Relational operators, including vector comparisons
  - Absolute value
  - Minimum, maximum, clamping, etc.
  - Packing and unpacking
- No support for double-precision angle, trigonometric, or logarithmic functions



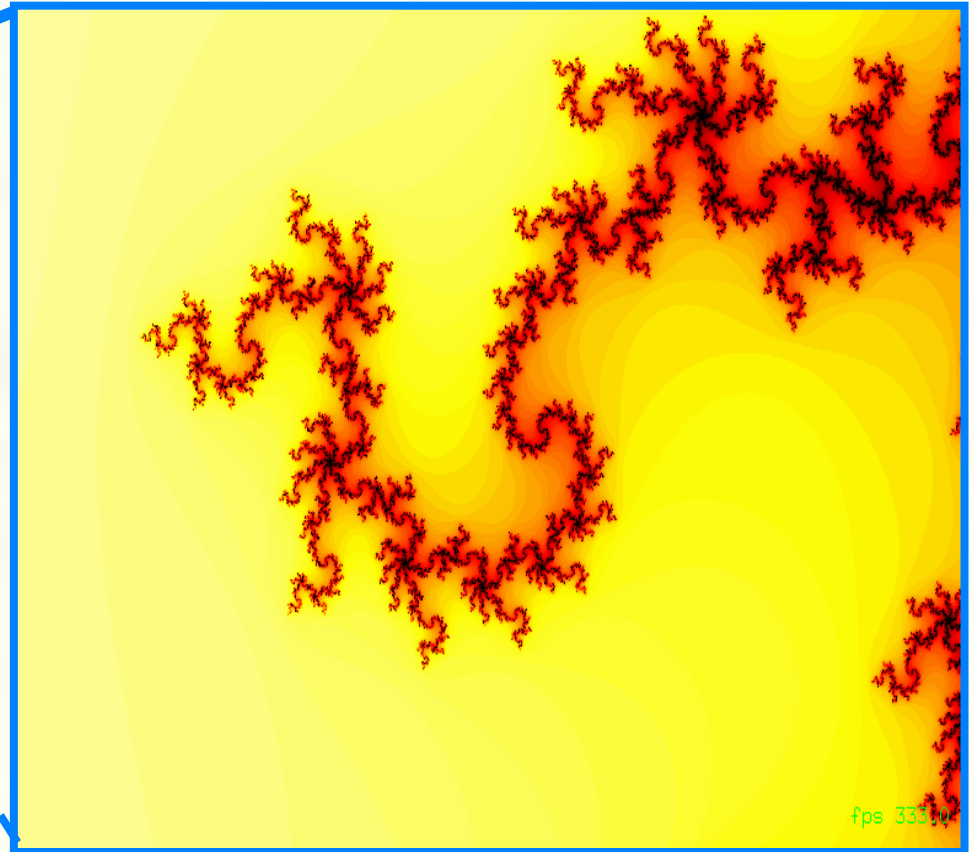


# Double Precision in OpenGL Shaders

*Mandelbrot set  
interactively visualized*



*Single-precision magnified 970,000x*



*Double-precision magnified 970,000x*



# Mandelbrot Shaders Compared

#version 400 compatibility

in vec2 position; Single-precision  
out vec4 color;

uniform float max\_iterations;  
uniform sampler1D lut;  
uniform mat2x3 matrix;

**mat2x3**  
**vec2**  
**vec3**

```
float mandel(vec2 c)
{
    vec2 z = vec2(0.0);
    float iterations = 0.0;
    while(iterations < max_iterations) {
        vec2 z2 = z*z;
        if (z2.x + z2.y > 4.0)
            break;
        z = vec2(z2.x - z2.y, 2.0 * z.x * z.y) + c;
        iterations++;
    }
    return iterations;
}

void main()
{
    vec2 pos = vec2(dot(matrix[0],vec3(position,1)),
                    dot(matrix[1],vec3(position,1)));

    float iterations = mandel(pos);

    // False-color pixel based on iteration
    // count in look-up table
    float s = iterations / max_iterations;
    color = texture(lut, s);
}
```

#version 400 compatibility

in vec2 position; Double-precision  
out vec4 color;

uniform float max\_iterations;  
uniform sampler1D lut;  
uniform dmat2x3 matrix;

**dmat2x3**  
**dvec2**  
**dvec3**

```
float mandel(dvec2 c)
{
    dvec2 z = dvec2(0.0);
    float iterations = 0.0;
    while(iterations < max_iterations) {
        dvec2 z2 = z*z;
        if (z2.x + z2.y > 4.0)
            break;
        z = dvec2(z2.x - z2.y, 2.0 * z.x * z.y) + c;
        iterations++;
    }
    return iterations;
}

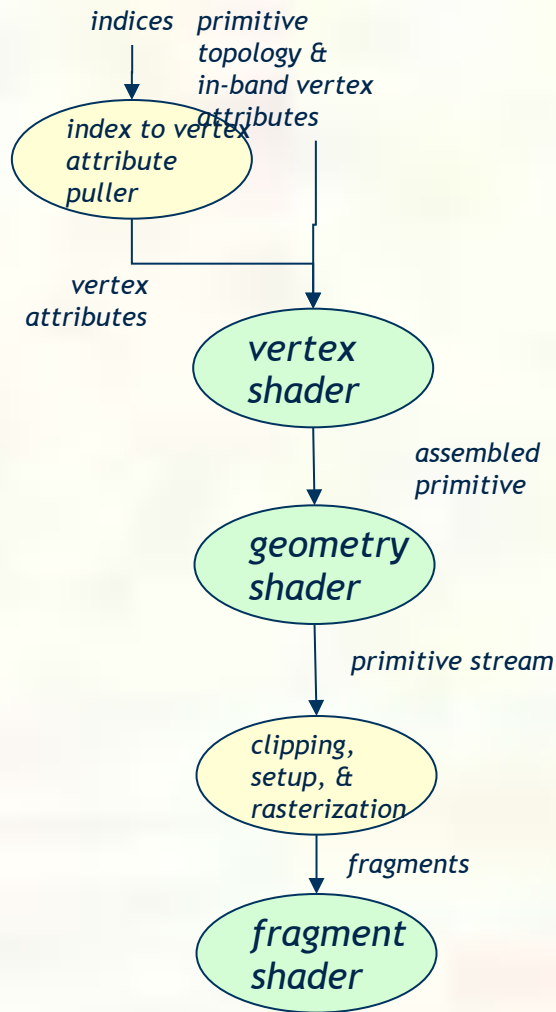
void main()
{
    dvec2 pos = dvec2(dot(matrix[0],dvec3(position,1)),
                      dot(matrix[1],dvec3(position,1)));

    float iterations = mandel(pos);

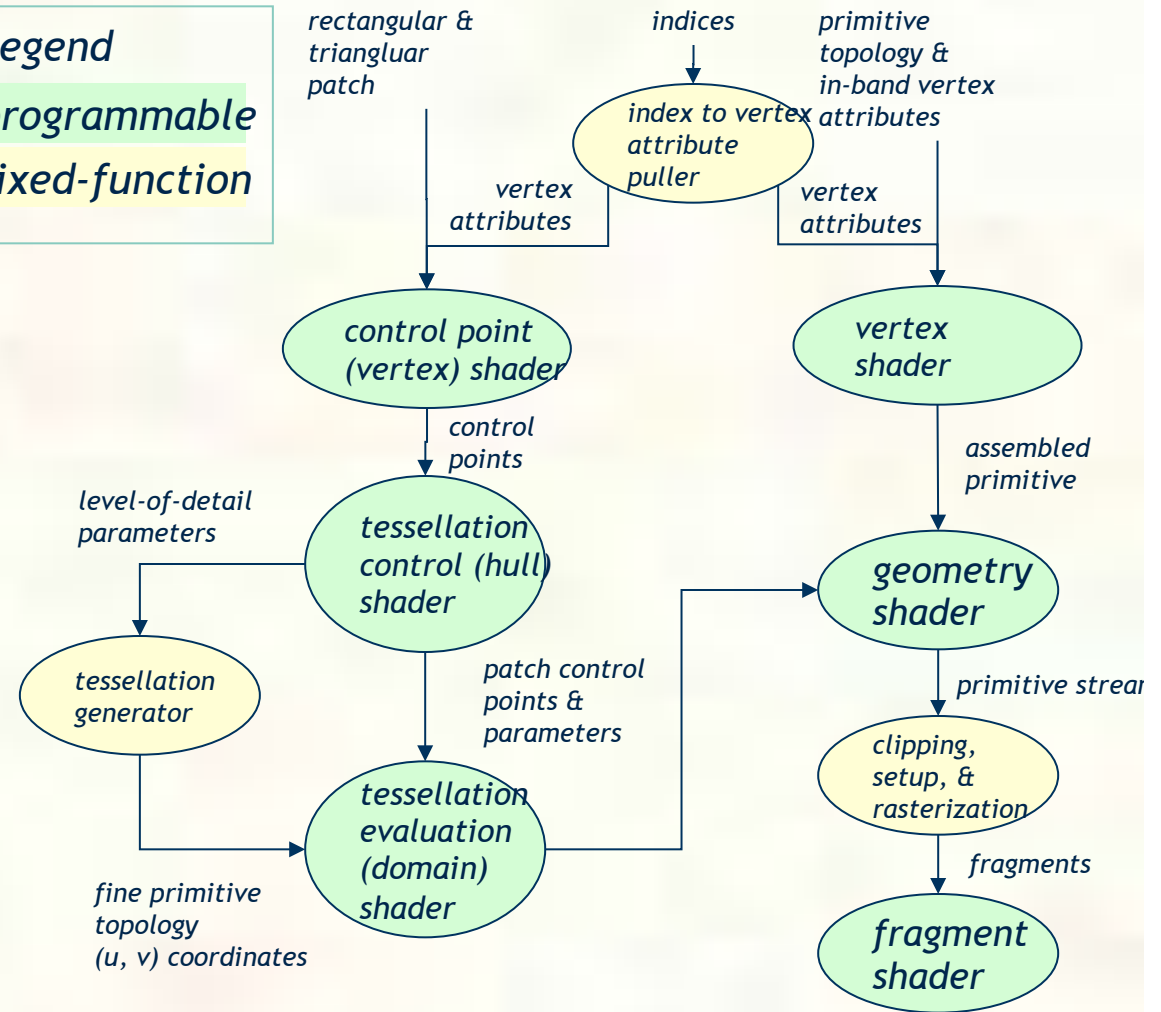
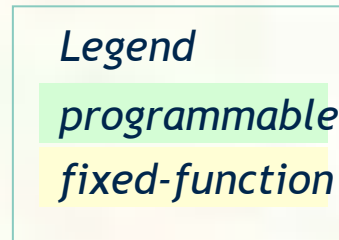
    // False-color pixel based on iteration
    // count in look-up table
    float s = iterations / max_iterations;
    color = texture(lut, s);
}
```



# Programmable Tessellation Data Flow



OpenGL 3.2

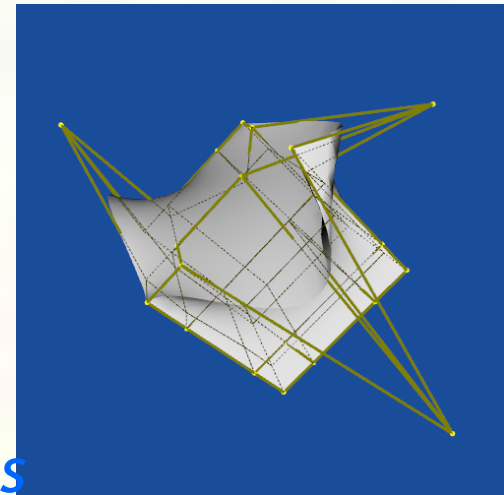
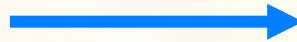
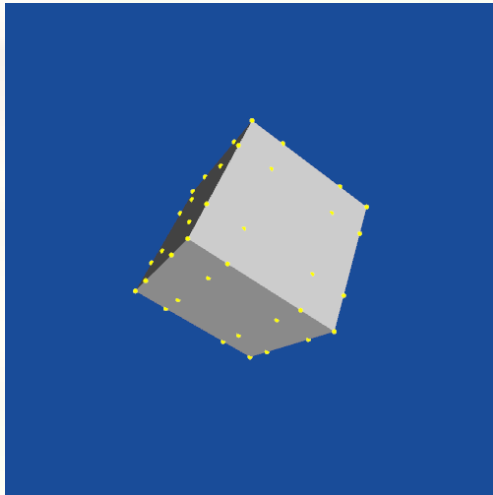


OpenGL 4.0 added tessellation shaders

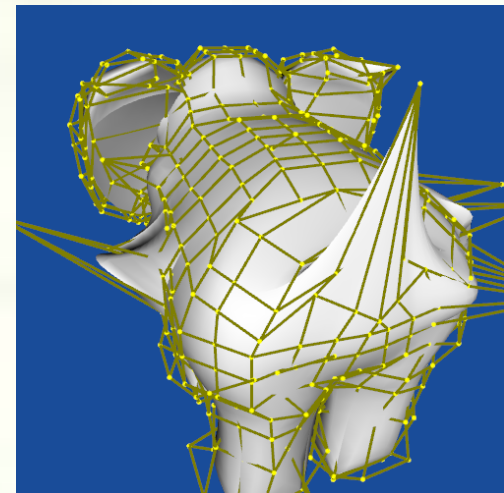
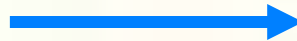
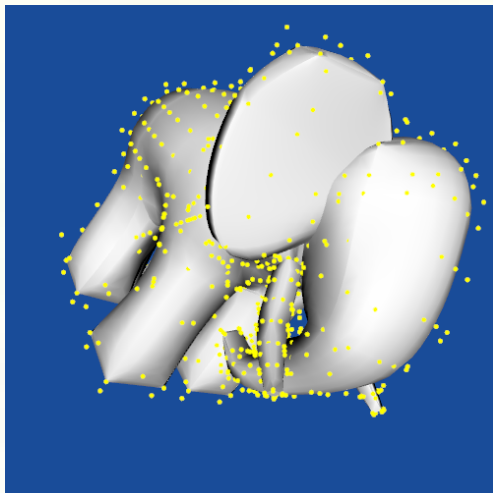


# Surfaces are Determined by Their<sup>40</sup> Control Points

---

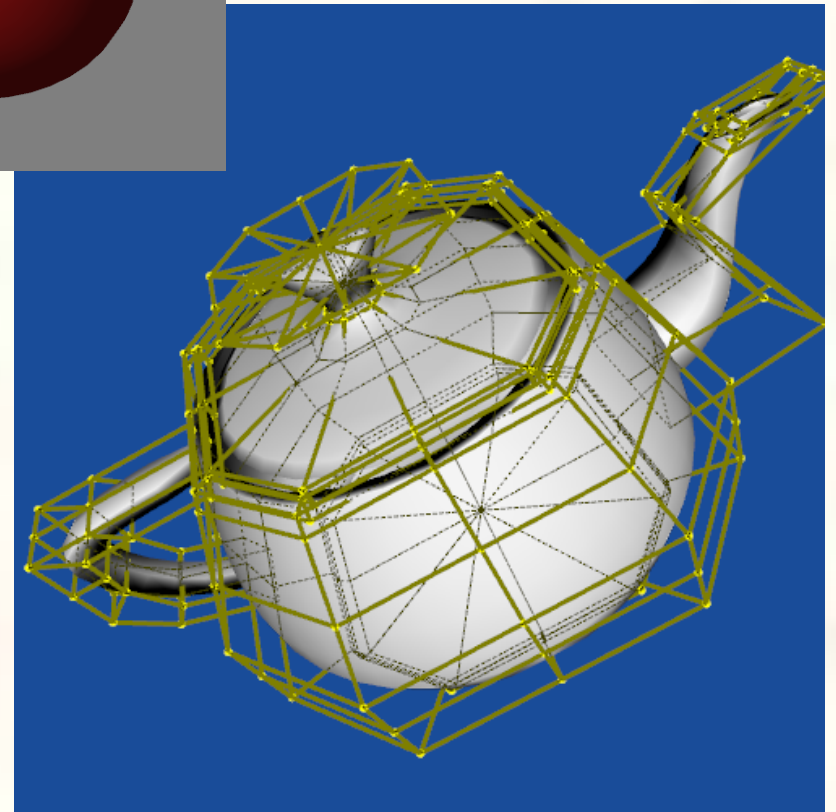
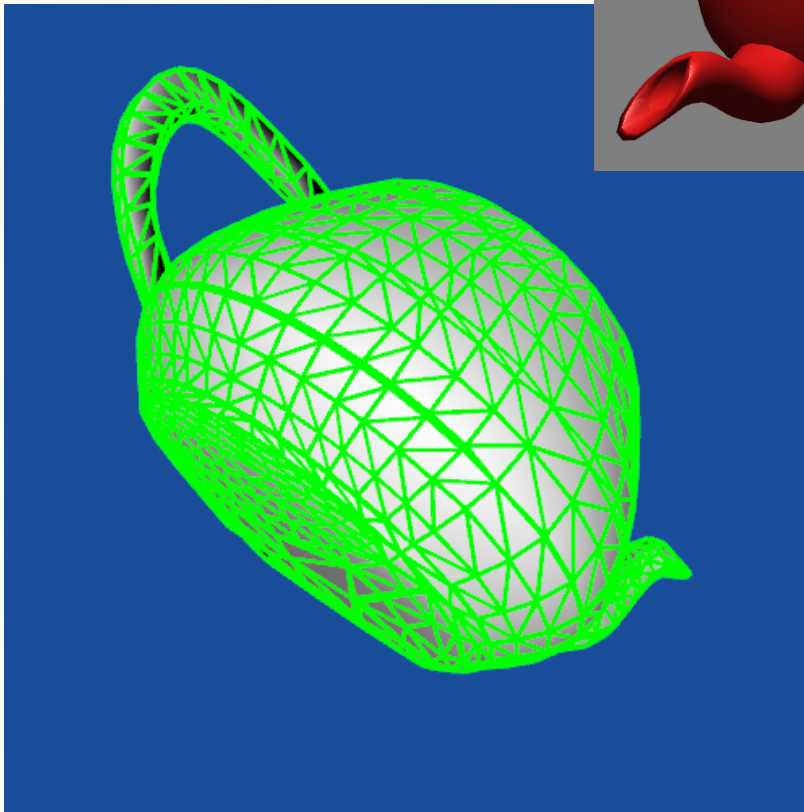
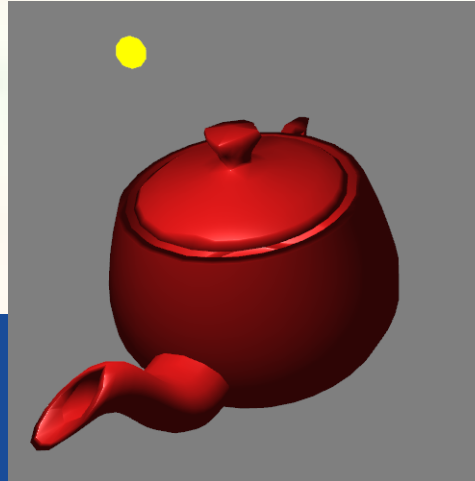


*Moving control  
points displaces  
the evaluated  
surfaces*





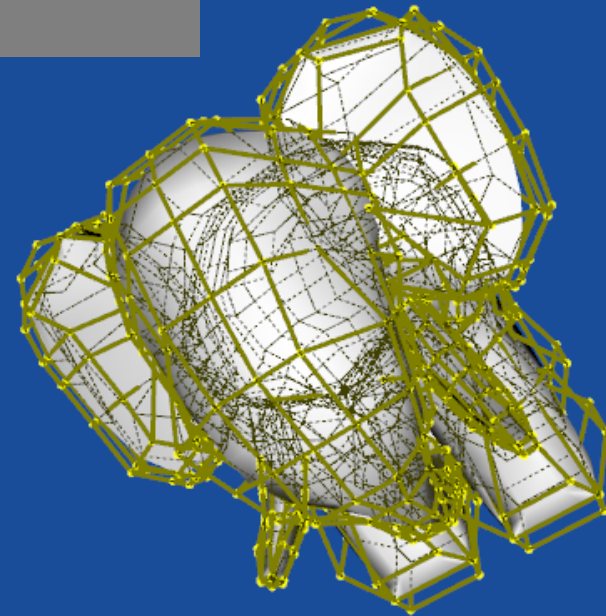
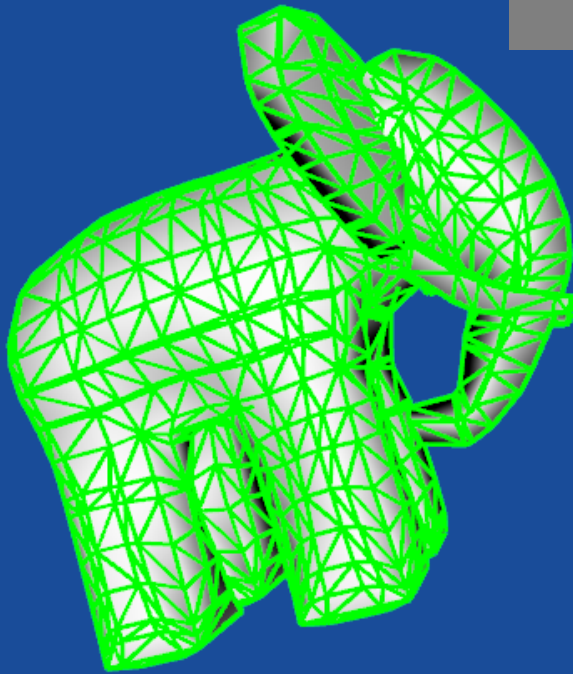
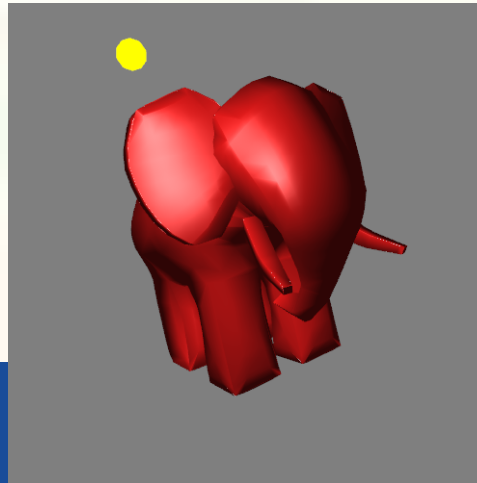
# Utah Teapot: Bi-cubic Patch<sup>4</sup> Mesh







# Ed Catmull's Gumbo





# Take Away Information

---

- Shading gets complicated
  - High-level shading languages have completely obviated the need for the assembly level shading
  - GLSL is the OpenGL Shading Language
  - But Cg and HLSL are also widely used
- Multiple programmable domains in pipeline
  - Vertex and fragment domains
  - More recently: primitive (geometry) and tessellation
- You need to learn GLSL for Project #3
  - Read textbook and web tutorials