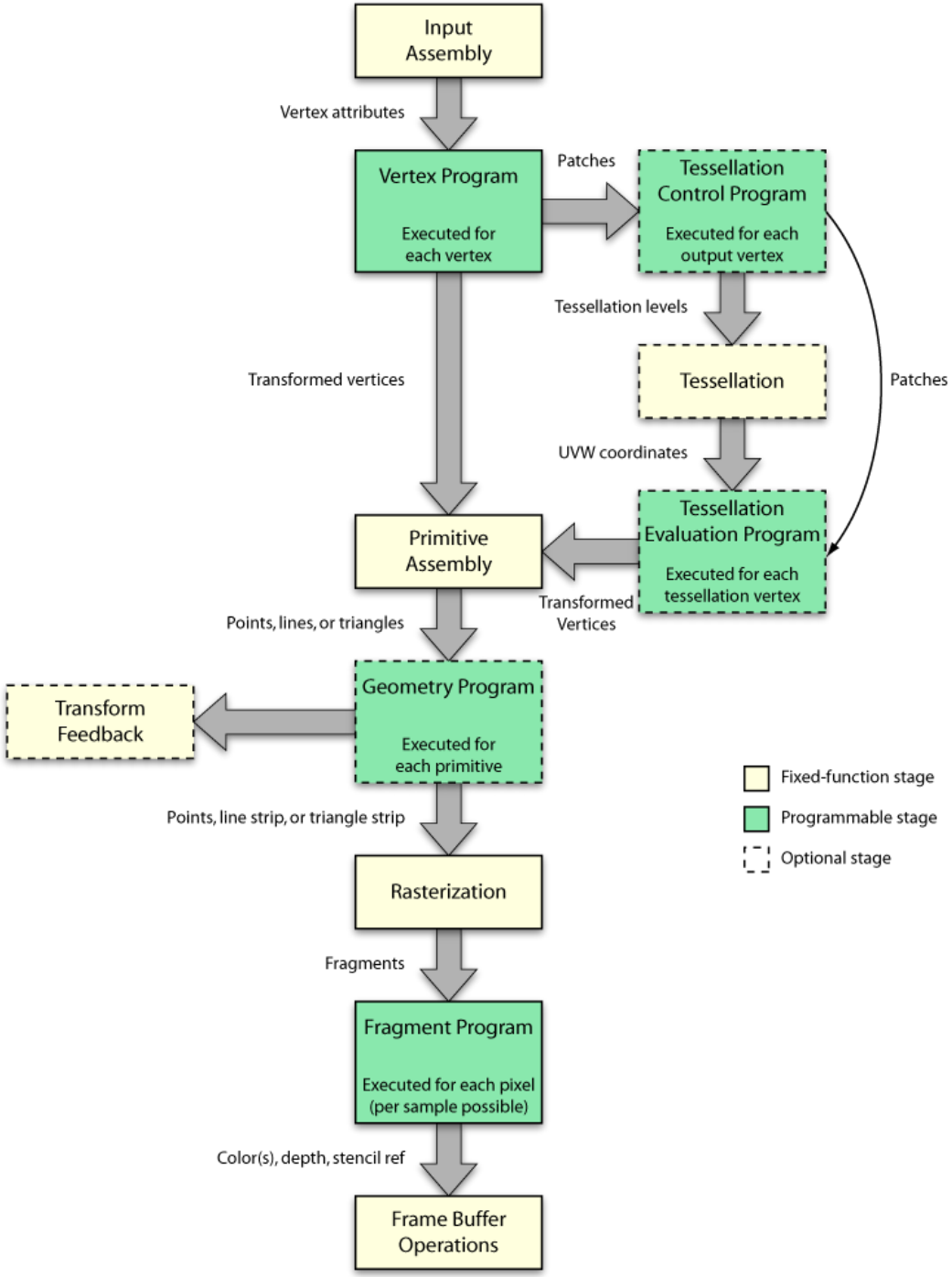
OpenGL pipeline Evolution
and
OpenGL Shading Language (GLSL)

Part 2/3

Vertex and Fragment Shaders

Prateek Shrivastava
CS12S008
shrvstv@cse.iitm.ac.in


Input Assembly
Vertex attributes
Vertex Program
Executed for each vertex
Patches
Tessellation Control Program
Executed for each output vertex
Tessellation levels
Tessellation
Patches
Transformed vertices
UVW coordinates
Tessellation Evaluation Program
Executed for each tessellation vertex
Primitive Assembly
Transformed Vertices
Points, lines, or triangles
Geometry Program
Executed for each primitive
Transform Feedback
Points, line strip, or triangle strip
Rasterization
Fragments
Fragment Program
Executed for each pixel (per sample possible)
Color(s), depth, stencil ref
Frame Buffer Operations
Fixed-function stage
Programmable stage
Optional stage

GLSL Data types

Scalar types: float, int, bool

Vector types: vec2, vec3, vec4
ivec2, ivec3, ivec4
bvec2, bvec3, bvec4

Matrix types: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D, sampler3D,samplerCube

C++ Style Constructors vec3 a = vec3(1.0, 2.0, 3.0);

- Standard C/C++ arithmetic and logic operators
- **Operators overloaded for matrix and vector operations**

```
Mat4 m;
Vec4 a, b, c;

b = a*m;
c = m*a;
```

GLSL BASICS

- For vectors can use [ ], xyzw, rgba or stpq

  vec3 v;

  v[1], v.y, v.g, v.t

  all refer to the same element

- **Swizzling:**

  vec3 a, b;

  a.xy = b.yx;

Representing geometric objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space

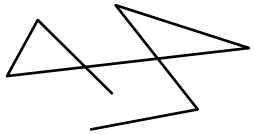
- Vertex data must be stored in *vertex buffer objects* (VBOs)
- VBOs must be stored in *vertex array objects* (VAOs)

OpenGL Geometric Primitives

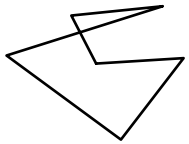
- All primitives are specified by vertices

GL_POINTS

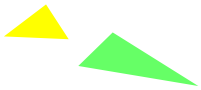
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

First Program

Rendering a cube with colors at each vertex

Example demonstrates:

- initializing and organizing vertex data for rendering.
- simple object modeling

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how must storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a vec4 class (implemented in C++) similar to GLSL's vec4 type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

Initializing the Data

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
point4  points[NumVertices];
color4  colors[NumVertices];
```

Cube Data

```
// Vertices of a unit cube centered at
   origin, sides aligned with axes
point4 vertex_positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating cube face from vertices

```
// quad() generates two triangles for each face and
   assigns colors to the vertices
int Index = 0;  // global variable indexing into VBO arrays

void quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] =vertex_positions[a];
   Index++;
    colors[Index] = vertex_colors[b]; points[Index] =vertex_positions[b];
   Index++;
    colors[Index] = vertex_colors[c]; points[Index] =vertex_positions[c];
   Index++;
    colors[Index] = vertex_colors[a]; points[Index] =vertex_positions[a];
   Index++;
    colors[Index] = vertex_colors[c]; points[Index] =vertex_positions[c];
   Index++;
    colors[Index] = vertex_colors[d]; points[Index] =vertex_positions[d];
   Index++;
}
```

Generating cube from faces

```
// generate 12 triangles: 36 vertices and 36 colors
void
colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Vertex Array Objects (VAOs)

- VAOs store the data of an geometric object.

- Steps in using a VAO
  1. generate VAO names by calling `glGenVertexArrays()`
  2. bind a specific VAO for initialization by calling `glBindVertexArray()`
  3. update VBOs associated with this VAO
  4. bind VAO for use in rendering

- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in code

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```

Storing vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO

- The code-flow is similar to configuring a VAO

  1. generate VBO names by calling `glGenBuffers()`
  2. bind a specific VBO for initialization by calling

     `glBindBuffer( GL_ARRAY_BUFFER, … )`
  3. load data into VBO using

     `glBufferData( GL_ARRAY_BUFFER, … )`
  4. bind VAO for use in rendering `glBindVertexArray()`

VBOs in code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
              sizeof(colors), NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                 sizeof(colors), colors );
```

Connecting Vertex Shaders with Geometric Data

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)

GLuint vPosition = glGetAttribLocation( program, "vPosition" );

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE,
                       0,BUFFER_OFFSET(0) );

GLuint vColor = glGetAttribLocation( program, "vColor" );

glEnableVertexAttribArray( vColor );

glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(sizeof(points)) );
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

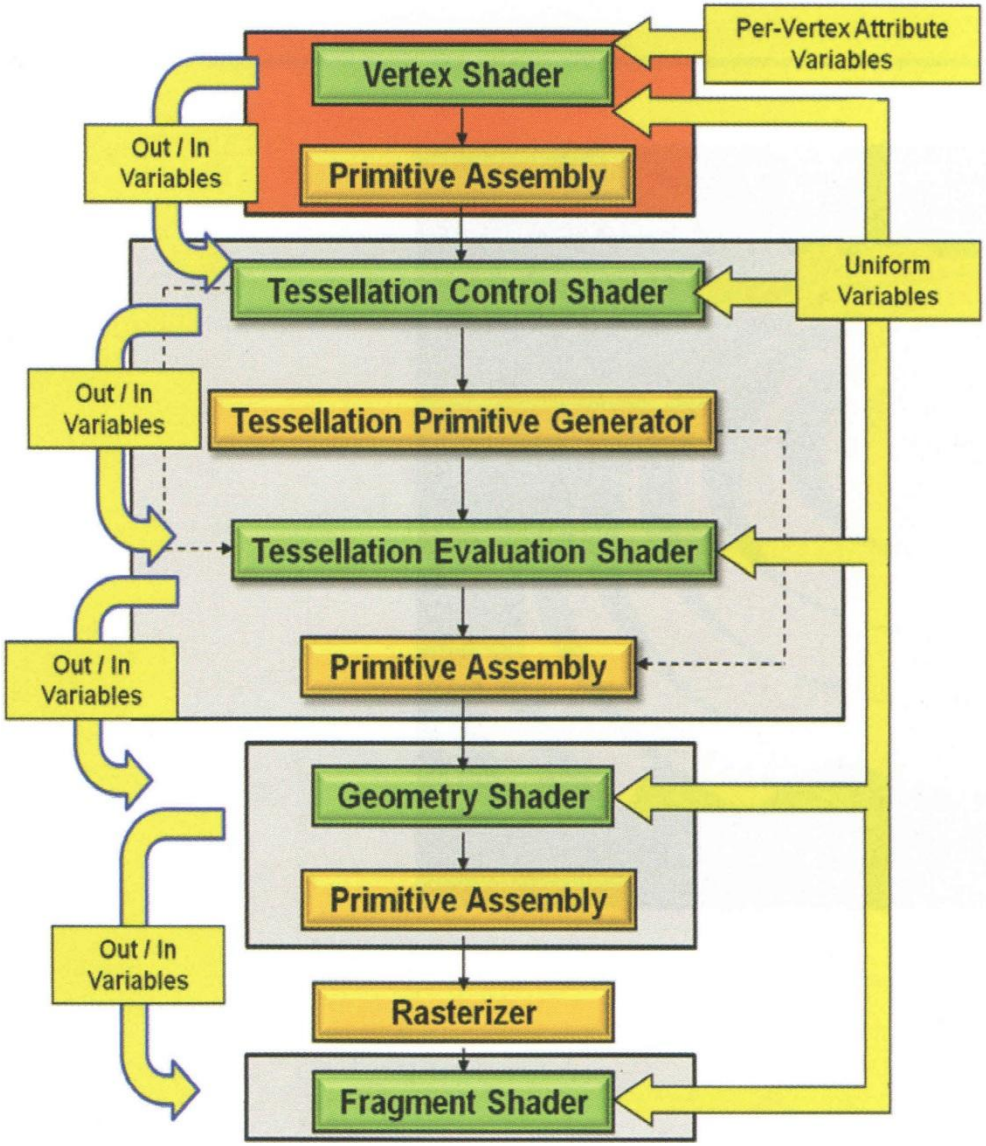
- Usually invoked in display callback
- Initiates vertex shader

GLSL variable types

- **Attribute**
  - A variable used only in vertex shaders;
  - Is set by the application per vertex and is generally sent from the application to the graphics card by OpenGL functions
  - Eg. color, normal, normal matrix or texture coordinates.
- **const**
  - Compile time constant and can't be accessed outside the shader that defines it.(like c++)
- **in, out, inout**
  - **Copy vertex attributes and other variable to/ from shaders**

```
in vec2 tex_coord;
out vec4 color;
```

GLSL variables



GLSL variable types

- **Uniform: variable from application (outside the shader)**
  - **Can be changed at most once per primitive.**

  uniform float time;

  uniform vec4 rotation;

- **Varying variables:**
  - They provide **communication** from vertex Shaders to Fragment Shaders. Vertex Shaders compute information for each vertex and write them to varying variables to be interpolated across a graphics primitive and then used by a fragment shader.

How often the data they represent changes:

- **Uniform:** **Infrequently** and never within a graphics primitive.
- **Attribute:** **Frequently** , often as frequently as each vertex.
- **Varying:** **Most frequently**, with each fragment as it's interpolated by the rasterizer.

GLSL limitations

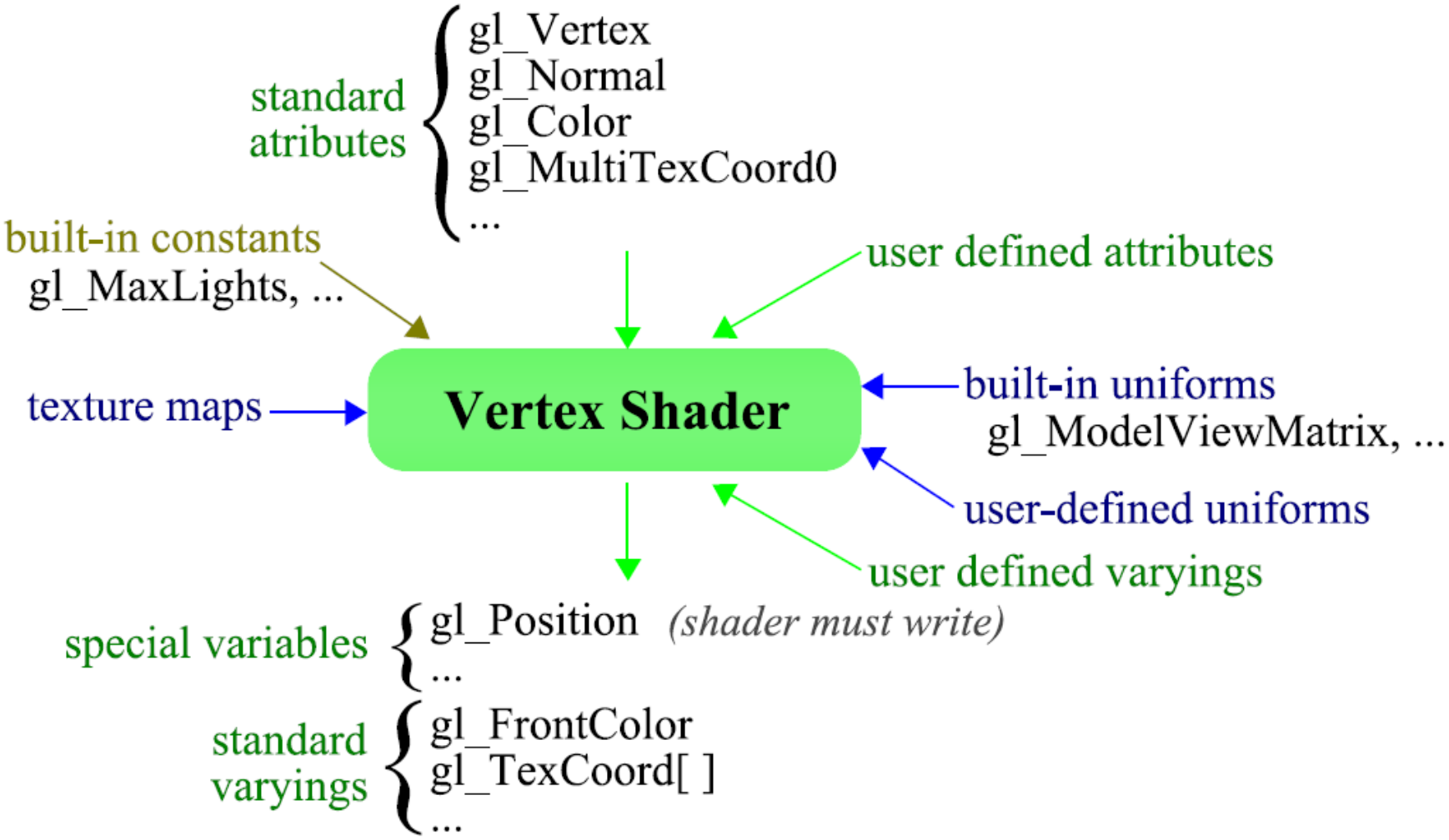
- C-like with some limitations.
  - No typecasts (use constructors instead).
  - No automatic promotion (although some GLSL compilers handle this).
  - No pointers, enums, strings.
  - No file-based pre-processor directives.
  - Only use 1D arrays (no bounds checking).

Vertex Shaders

In the fixed-function geometry pipeline A GLSL Vertex shader:

Replaces these operations	Doesn't replaces
• Vertex Transformation. • Normal Transformation. • Normal normalization. • Handling per-vertex lighting. • Handling texture Coordinates.	• View volume clipping. • Homogeneous division. • Viewport mapping. • Backface culling. • Polygon mode. • Polygon offset.

Vertex Shader



A BASIC VERTEX SHADER

```
#version 400

layout(location=0) in vec4 in_Position;
layout(location=1) in vec4 in_Color;
out vec4 ex_Color;

void main(void){
	gl_Position = in_Position;
	ex_Color =   in_Color;
}
```

Rotation Vertex Shader for the CUBE

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
 // Remember: these matrices are column-major

    mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                    0.0,  c.x,  s.x, 0.0,
                    0.0, -s.x,  c.x, 0.0,
                    0.0,  0.0,  0.0, 1.0 );
```

```
mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );


mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                s.z,  c.z, 0.0, 0.0,
                0.0,  0.0, 1.0, 0.0,
                0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

Fragment Shaders

In the fixed-function geometry pipeline A GLSL Vertex shader:

Replaces these operations	Doesn't replaces
• Color Computation. • Texturing. • Per-pixel lighting. • Fog. • Discarding Pixels in fragments.	• Blending. • Stencil Test. • Depth Test. • Scissor Test. • Stippling operations. • Raster operations performed as a pixel is being written to the frame buffer.

A simple Fragment Shader

```
#version 400

 in vec4 ex_Color;
 out vec4 out_Color;

 void main(void) {

       out_Color = ex_Color;
 }
```

This data may include, but is not limited to:

- raster position
- depth
- interpolated attributes (color, texture coordinates, etc.)
- stencil
- alpha
- window ID

- **A fragment is the data necessary to generate a single pixel's worth of a drawing primitive in the frame buffer.**
- **In general, a fragment can be thought of as the data needed to shade the pixel, plus the data needed to test whether the fragment survives to become a pixel (depth, alpha, stencil, scissor, window ID, etc.).**

Vertex Shader for textures

```
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```

Fragment Shader for texture

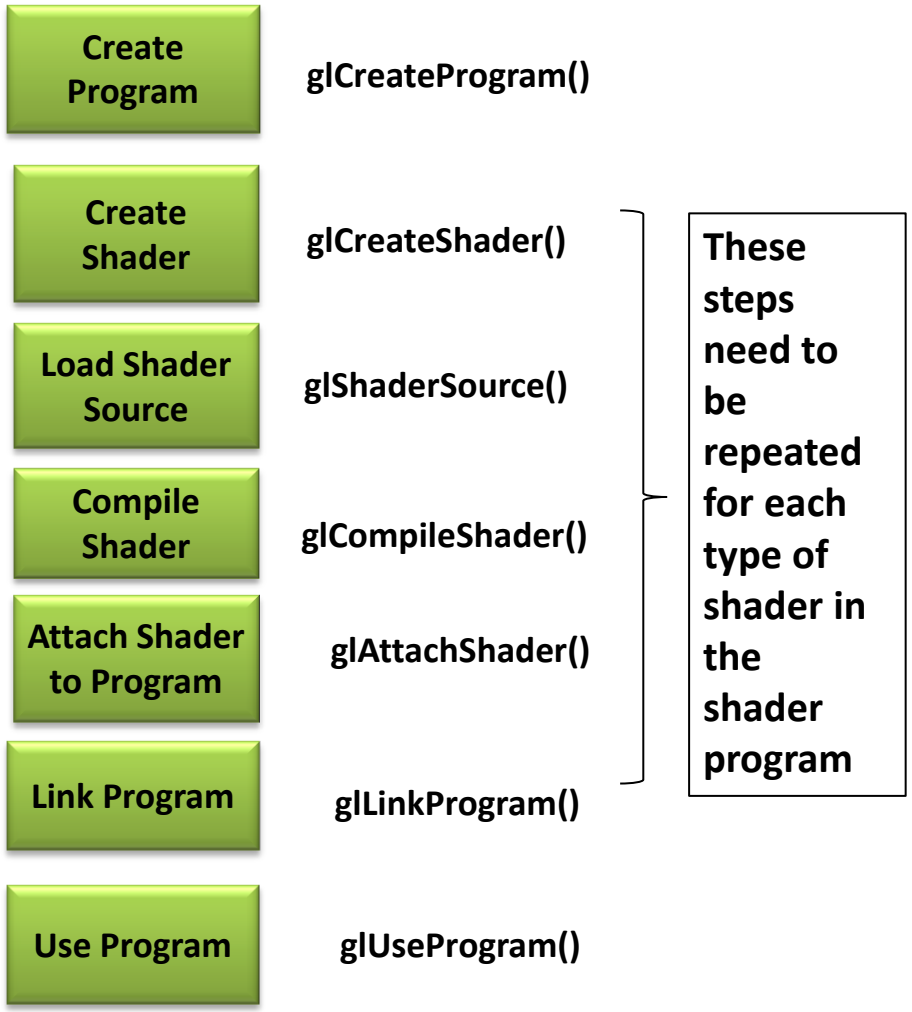
```
in vec4 color;
in vec2 texCoord;

uniform sampler2D texture;

void main()
{
  gl_FragColor = color * texture( texture, texCoord );
}
```

Getting shader to OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



```
void CreateShaders(void)
{
   GLenum ErrorCheckValue = glGetError();

   VertexShaderId = glCreateShader(GL_VERTEX_SHADER);
   glShaderSource(VertexShaderId, 1, &VertexShader, NULL);
   glCompileShader(VertexShaderId);

   FragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
   glShaderSource(FragmentShaderId, 1, &FragmentShader, NULL);
   glCompileShader(FragmentShaderId);

   ProgramId = glCreateProgram();
   glAttachShader(ProgramId, VertexShaderId);
   glAttachShader(ProgramId, FragmentShaderId);
   glLinkProgram(ProgramId);
   glUseProgram(ProgramId);

   ErrorCheckValue = glGetError();
   if (ErrorCheckValue != GL_NO_ERROR)
   {
        fprintf(stderr,"ERROR: Could not create the shaders: %s
   \n", gluErrorString(ErrorCheckValue));
        exit(-1);
   }
}
```

In the program

```
const GLchar* FragmentShader =
{
   "#version 400\n"\

   "in vec4 ex_Color;\n"\
   "out vec4 out_Color;\n"\

   "void main(void)\n"\
   "{\n"\
   "    out_Color = ex_Color;\n"\
   "}\n"
};
```

Finishing the Cube Program

```
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Color Cube" );
    glewInit();
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    glutMainLoop();
    return 0;
}
```

Cube Program GLUT Callbacks

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT |
             GL_DEPTH_BUFFER_BIT
   );
    glDrawArrays( GL_TRIANGLES, 0,
                  NumVertices );
    glutSwapBuffers();
}
```

```
void keyboard( unsigned char key,
               int x, int y )
{
    switch( key ) {
     case 033: case 'q': case Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

Questions…?