INCLUDES OPENGL 4.X

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

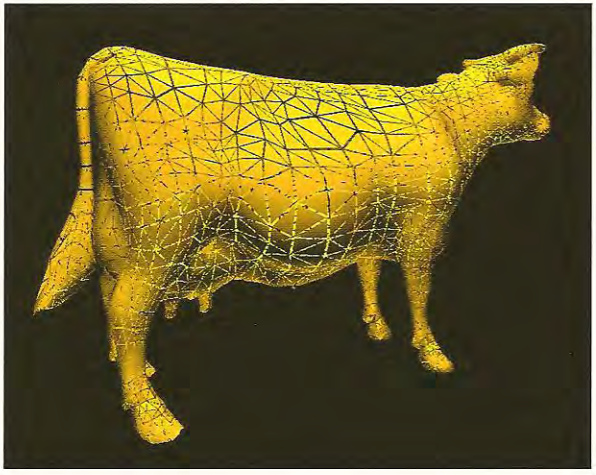
Graphics Shaders

THEORY AND PRACTICE

SECOND EDITION

MIKE BAILEY • STEVE CUNNINGHAM

3 Fundamental Shader Concepts



Shaders in the Graphics Pipeline

Let's have another look at the graphics pipeline, but let's break it out in a little different way than we did in the previous chapter. Let's add into the pipeline the five shaders we are considering in this book: vertex shaders, tessellation control shaders, tessellation evaluation shaders, geometry shaders, and fragment shaders. This expanded view of the pipeline is shown in Figure 3.1, where the positions of the shaders in the pipeline suggest the functions that each provides. While it is not obvious from the diagram, each shader block is in an alternate branch of the pipeline; they are optional capabilities that may or may not be used for any application. You may use any combination of vertex, tessellation, geometry, or fragment shaders in your program; you do not have to use any particular combinations, although, in general, if you use *any* shad-

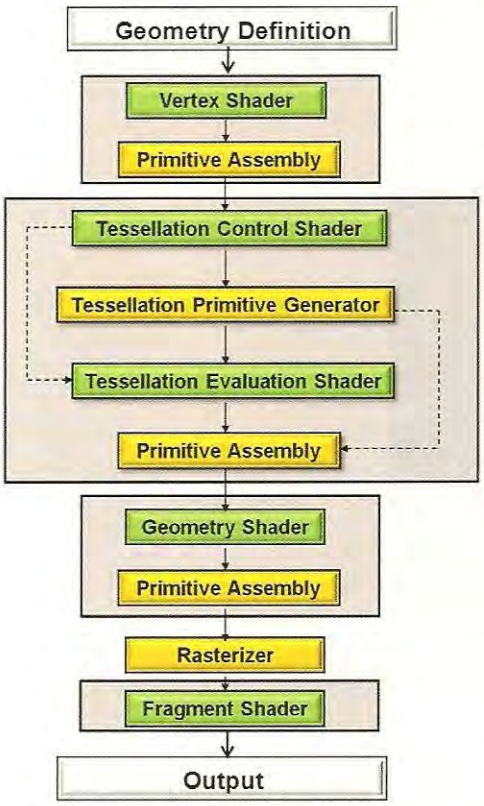


Figure 3.1. The expanded graphics pipeline, with programmable stages shown in green and fixed-function stages shown in orange.

ers, you usually are required to include a vertex shader, too.

When you're developing shaders, however, you don't necessarily need to think of the entire graphics pipeline like this. For each individual shader, it is helpful to understand what data comes into this shader, what this shader can do with it, and what new data gets transmitted to the next stage. For this, it's interesting to consider how the graphics pipeline looks to shaders; this is shown in Figure 3.2, with an emphasis on how data moves among the shader stages. Of course, if you choose not to include any shader stage, the in/out variables from the previous stage simply skip the omitted stage and go on to the subsequent stage.

Notice in Figure 3.2 that all attribute variables are input to the vertex shader, and all uniform variables are input to whatever shader needs them. Uniform variables are written by the application; none of them can be written by any shader. Any computation that needs to pass data on to the next shader must do so through an `out` variable, and that variable must be read (as an `in` variable) and passed along (as an `out` variable) by intermediate variables until it is used.

Let's consider how the separate functionalities of the graphics pipeline might be enhanced by using shaders. To begin, let's look at the modeling functions that begin the geometry pipeline. In the standard pipeline, you define the vertices of your model either by using specific statements, such as `glVertex3f(2.0, -1.0, 3.0)`, or by using a computation to create the vertex coordinates. You can add other geometric information such as normals and texture coordinates as you need them and as they are available. You can also add appearance information. This may be done while the geometry is defined, as you might do with colors through the `glColor*(...)` function. Another approach to appearance defines and enables environments such as lighting, with its associated materials definition, or textures, with their associated texture parameters, texture environment, and texture image.

The geometry operations in the fixed-function pipeline can be replaced and possibly expanded by any (or all) of the GLSL vertex shaders, tessellation shaders, or geometry shaders. A vertex shader only operates on one vertex at a time and can take the initial vertex definition and alter it by changing the values of the position, normal, or texture coordinates. As we will see, the vertex shader must set the transformed position of each vertex. It may also set the color for the vertex, especially if per-vertex lighting is used.

The tessellation shaders take a set of points called a *patch*, which can represent anything, and interpolate the points to create a new geometry. You get to define what meaning these points have. The tessellation shaders will then assist you in creating new geometry from them.

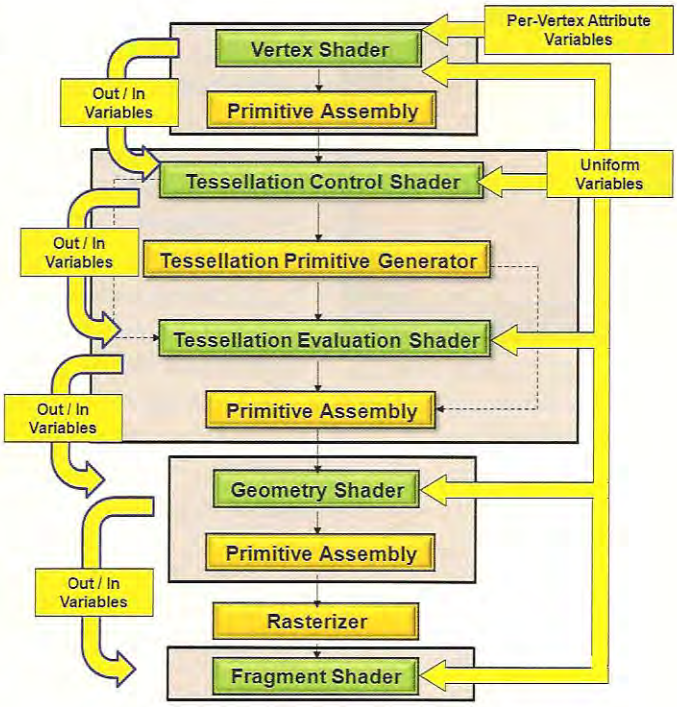


Figure 3.2. The shader's-eye view of the pipeline.

A geometry shader can take a graphics primitive from a vertex shader and create one or more new primitives. Geometry shaders can do the same computation as a vertex shader to compute the full geometry and color of each new vertex. They can also prepare variables for later use by a fragment shader.

The final shader capability is fragment processing, done by the fragment shader. This takes the information developed by vertex processing (vertex shader, tessellation shader, or geometry shader) and expands the traditional fragment operations by letting you operate on each fragment individually to generate the color of its pixel. This is a highly parallel operation that can apply traditional or procedural textures; special coloring, such as pseudocolor transfer functions; and advanced kinds of shading, such as Phong or anisotropic shading. The operation can also determine whether its pixel is to be retained or discarded for the final image. The fragment shader has the strongest impact on the visual effect of your images.

In the next few sections, we will look at the functionality of each shader by looking at simple examples. For reference, a sphere with only standard

Figure 3.3. A sphere with simple color, diffuse lighting, and smooth shading.

fixed-function processing is shown in Figure 3.3. In each section, we will outline the shaders' operations and give a short example of a vertex and a fragment shader that produce the figure; we will then give a brief description of the GLSL shader language, so you can see the language features that we use in the examples. A more complete discussion of GLSL will come in Chapter 5.

In the next chapter, we'll describe the *glman* tool that lets you create and experiment with shaders without having to write a complete application; here, it is useful if you see how you could define this image with *glman*. Here is the GLIB file that sets up the image and specifies the shaders to be used:

```
Vertex Sphere.vert
Fragment Sphere.frag
Program Sphere

Color  1 0.5 0
Sphere 2.0 100 100
```

We will provide the vertex and fragment shader files for this example later in this chapter.

Vertex Shaders

A GLSL vertex shader takes the vertex and environment information that is stored by the OpenGL system and makes it available to you through a set of uniform and attribute variables, so that you can do your own vertex computations. Later in this chapter, we will outline some of the highlights of the GLSL shader language, including these commonly used uniform and attribute variables. Vertex shaders act on geometry that is usually given in model space coordinates and produce geometry that is output in 3D clip space; all projection and clipping is done later in the graphics pipeline. Vertex shaders must do much more than that, however. A GLSL vertex shader replaces these operations in the fixed-function geometry pipeline:

- Vertex transformations.
- Normal transformations.
- Normal normalization (i.e., turn it into a unit vector).
- Handling of per-vertex lighting.
- Handling of texture coordinates.

These are very important operations. Fortunately, the necessary information is readily available, and the operations you need to perform are expressed well in the GLSL language, which handles vector and matrix operations with ease.

However, a GLSL vertex shader does not replace *all* of the operations in the geometry pipeline. In particular, it does not replace the operations that take the clip space to the final pixel space. The specific functions that are still done by the fixed-function pipeline are

- View volume clipping.
- Homogeneous division.
- Viewport mapping.
- Backface culling.
- Polygon mode.
- Polygon offset.

A key function of a vertex shader is to take all attribute variables and either use them or copy them into out variables for later shaders to use.

Vertex shaders have several kinds of output. The most important are the transformed vertices and the color associated with each vertex. Of course, the vertex shader can compute or re-compute normals and texture coordinates as well as vertex coordinates. If you use a fragment shader, the vertex processing can develop variables that let the fragment shader interpolate these properties as each fragment is developed. By setting up color, normals, or textures with variables from vertex processing, the fragment shader can carry out sophisticated operations on each fragment.

The vertex shader for the smooth shading on the simple sphere of Figure 3.3 is shown below. This shader code, and the other shader code examples in this chapter, will be better understood when we have discussed GLSL in more depth later in the book. For now, though, note that this shader calculates per-vertex light intensity by the standard diffuse lighting com-

The shader code in this chapter uses the name prefix conventions we introduced in Chapter 2. Variable names start with a character that indicates who created it:

a	attribute variable
f	variable from a fragment shader
g	variable from a geometry shader
tc	variable from a tessellation control shader
te	variable from a tessellation evaluation shader
v	variable from a vertex shader
u	uniform variable

As in C/C++, constants are generally written in all caps.

putation using the normal, vertex eye coordinates, and light position, and that it sets the required output `gl_Position` from the `uModelViewProjection` matrix and the vertex coordinates.

```
uniform mat4 uModelViewMatrix;
uniform mat4 uModelViewProjectionMatrix;
uniform mat3 uNormalMatrix;

in vec4 aVertex;
in vec4 aNormal;
in vec4 aColor;

out vec4  vColor;
out vec3  vMCposition;
out float vLightIntensity;

const vec3 LIGHTPOS = vec3( 3., 5., 10. );

void main( )
{
  vec3 transNorm   = normalize( uNormalMatrix * aNormal );
  vec3 ECposition  = vec3( uModelViewMatrix * aVertex );
  vLightIntensity  = dot(normalize(LIGHTPOS - ECposition),
                         transNorm);
  vLightIntensity  = abs( vLightIntensity );

  vColor        = aColor;
  vMCposition   = aVertex.xyz;
  gl_Position   = uModelViewProjectionMatrix * aVertex;
}
```

The example for Figure 3.3 did not do one important thing that a vertex shader can do, however: modify the application-supplied vertex coordinates. As an example of geometry modification, let's start with a simple plane (represented by a 200 × 200 mesh of quads) considered as the domain of a function, and let the vertex shader apply that function. The GLIB file is essentially the same as that for the Figure 3.3 example, except that the specified geometry is a 200 × 200 set of quads in the XY-plane, instead of a sphere, specified like this:

```
QuadXY  -2.  1. 200 200
```

The vertex shader will apply the function

$$z(x,y) = 0.3 * \sin\left(x^2 + y^2\right)$$

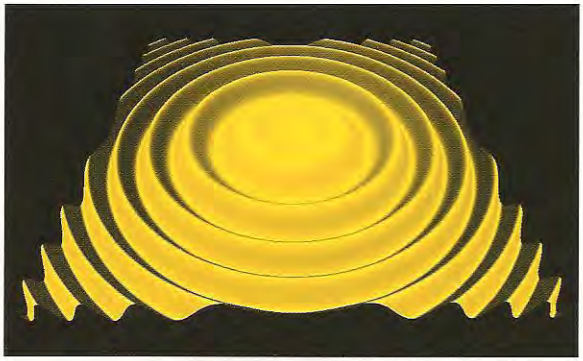


Figure 3.4. A rippled surface generated by a vertex shader; still with simple color, ambient plus diffuse lighting, and smooth shading.

to the x and y coordinates of each vertex to calculate the z-coordinate, and will calculate the normals to each vertex by using an analytic computation, since the derivative is known. This uses the fact that the tangent vectors are given by taking the derivatives of z with respect to x and y:

$$\frac{\partial z}{\partial x} = 2.*0.3*x*\cos\left(x^2+y^2\right),$$
$$\frac{\partial z}{\partial y} = 2.*0.3*y*\cos\left(x^2+y^2\right).$$

After the vertices and normals are set up, the usual computations for eye coordinates (`ECposition`) and light intensity are done. The resulting function surface is shown in Figure 3.4.

The vertex shader for the rippled surface in Figure 3.4 is given below. The operations for the diffuse light intensity are those for standard ambient and diffuse lighting, based on the eye-space coordinates of each vertex (the `ECpos` variable), the normal (`myNorml`) computed from the analytic partial derivatives, and a fixed light position (`LIGHTPOS`) that would ordinarily be passed into the shader from the application as a uniform variable. The actual display coordinates `gl_Position` are set by multiplying by `uModelViewProjectionMatrix` to apply the model, view, and projection transformations. The output of this vertex shader includes two variables: the light intensity and color values defined in the vertex shader. None of this is difficult, but it requires you to work with your objects at a lower level than the usual OpenGL.

```
in vec4 aVertex;
in vec4 aColor;

uniform mat4 uModelViewMatrix;
uniform mat4 uModelViewProjectionMatrix;

out float vLightIntensity;
out vec3  vMyColor;

const vec3 LIGHTPOS = vec3( 0., 10., 0. );

void main( )
{
  vec4 thisPos = aVertex;
  vMyColor     = aColor.rgb;

  // create a new height for this vertex:
  float thisX = thisPos.x;
  float thisY = thisPos.y;
  // the surface is z = 0.3 * sin (x^2 + y^2)
  thisPos.z = 0.3 * sin( thisX*thisX + thisY*thisY );

  // now compute the normal and the light intensity
  vec3 xtangent = vec3( 1., 0., 0. );
  xtangent.z = 2. * 0.3 * thisX * cos( thisX*thisX +
                                       thisY*thisY );
  vec3 ytangent = vec3( 0., 1., 0. );
  ytangent.z = 2. * 0.3 * thisY * cos( thisX*thisX +
                                       thisY*thisY );
  vec3 thisNormal = normalize( cross( xtangent, ytangent ) );

  vec3 ECpos = vec3( uModelViewMatrix * thisPos );
  vLightIntensity  = dot( normalize(LIGHTPOS - ECpos),
                          thisNormal );
  vLightIntensity = 0.3 + abs( vLightIntensity );// 0.3 ambient
  vLightIntensity = clamp( vLightIntensity, 0., 1. );

  gl_Position     = uModelViewProjectionMatrix * thisPos;
}
```

A Comment on Shader Code Efficiency

GLSL gives you some clever ways to make your code execute super efficiently on graphics hardware. As with many such things in computing, however, it often makes the code harder to read. For example, rather than creating two separate variables above, `thisX` and `thisY`, and then squaring each to compute `thisPos.z` as shown previously, it would be more efficient to say

```
vec2 thisXY = thisPos.xy;
thisPos.z   = 0.3 * sin( dot( thisXY, thisXY ) );
```

Similarly, the computation for the tangent vectors could be expressed more efficiently as

```
xtangent.z = 2. * 0.3 * thisX * cos( dot( thisXY, thisXY ) );
ytangent.z = 2. * 0.3 * thisY * cos( dot( thisXY, thisXY ) );
```

But, at least for some, this would make the code less readable. For this book, we have often taken our own code and re-written it to be more readable, even though that may make it less efficient. We're sure you will find lots of examples of this. Don't email us about it—we already know.

Fragment Shaders

Sometimes called *pixel shaders* (e.g., in Cg), fragment shaders operate on a fragment to determine the color of its pixel. We know that rasterization operations interpolate quantities such as colors, depths, and texture coordinates. Fragment shaders use these interpolated values, as well as many other kinds of information, to determine the color of each fragment's pixel.

The rasterizer interpolates any variables that have been defined in the geometry processing stages and passed to the fragment shader. These interpolated values may be used in any kind of fragment computation you want. These computations are performed on several fragments in parallel, with the width of the parallelization depending on the particular graphics card you use. This parallelization lets a fragment shader operate with the same kind of acceleration as graphics cards do for the fixed-function pipeline.

As we saw for vertex shaders, many operations that were automatically handled by the fixed-function pipeline are now the responsibility of the shader programmer. A GLSL fragment shader replaces or adds the following operations:

- Color computation.
- Texturing.
- Per-pixel lighting.
- Fog.
- Discarding pixels in fragments.

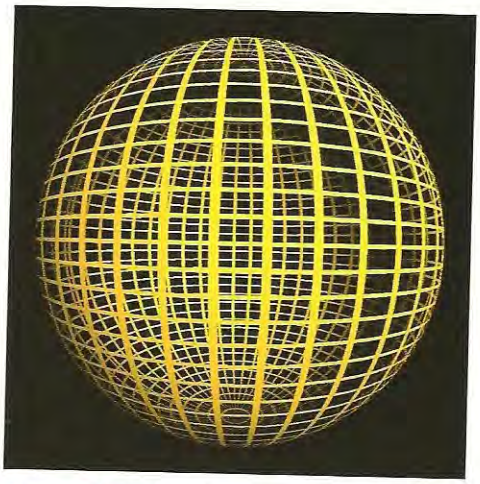


Figure 3.5. A sphere with a positional screen pixel-discard fragment shader.

However, a fragment shader does not replace all the operations in the rasterization process. In particular, a GLSL fragment shader does not replace several raster operations, including

- Blending.
- Stencil test.
- Depth test.
- Scissor test.
- Stippling operations.
- Raster operations performed as a pixel is being written to the framebuffer.

Figure 3.5 shows the sphere with some parts made invisible by discarding pixels in the fragment shader instead of drawing them. Its fragment shader, which is listed after the figure, takes the three input variables for light intensity, color, and model coordinates, as well as three uniform variables that were set externally to the program (in this case, in the GLIB file needed by *glman*). It also receives texture coordinates that were passed from the application. It uses the scaled and truncated texture coordinates in the model to create a screen effect, and pixels that are not within a given distance of the screen lines are discarded. If a pixel is kept, any alpha value in the color is ignored and the pixel is lit with standard diffuse lighting.

The vertex shader for this figure is straightforward. It simply calculates the normal and eye-coordinate position, from which it gets the light intensity, and then passes the attribute variable `aTexCoord0` along to the fragment shader.

```
uniform mat4 uModelViewMatrix;
uniform mat4 uModelViewProjectionMatrix;
uniform mat3 uNormalMatrix;

in vec4   aVertex;
in vec4   aTexCoord0;
in vec4   aColor;
in vec3   aNormal;

out vec4  vColor;
out float vLightIntensity;
out vec2  vST;

const vec3 LIGHTPOS = vec3( 0., 0., 10. );
```

```
void main( )
{
  vec3 transNorm  = normalize( vec3( uNormalMatrix * aNormal )
                               );
  vec3 ECposition = vec3( uModelViewMatrix * aVertex );
  vLightIntensity = dot( normalize(LIGHTPOS-ECposition),
                         transNorm );
  vLightIntensity = clamp( .3 + abs( vLightIntensity ), 0., 1.
                           );

  vST         = aTexCoord0.st;
  vColor      = aColor;
  gl_Position = uModelViewProjectionMatrix * aVertex;
}
```

Below is the fragment shader for Figure 3.5. It takes the s and t coordinates provided by the vertex shader and uses them to decide whether to discard a pixel.

```
uniform float uDensity;
uniform float uFrequency;

in vec4  vColor;
in float vLightIntensity;
in vec2  vST;

out vec4 fFragColor;

void main( )
{
  float sf = vST.s * uFrequency;
  float tf = vST.t * uFrequency;

  if( fract( sf ) >= uDensity && fract( tf ) >= uDensity )
    discard;

  fFragColor = vec4( vLightIntensity*vColor.rgb, 1. );
}
```

Again, a more efficient implementation that takes advantage of the parallelism in graphics hardware would be

```
  vec2 stf = vST * uFrequency;

  if( all( fract(stf) >= vec2(uDensity, uDensity) ) )
    discard;
```

```
vec3 V[3];
vec3 CG;

void
ProduceVertex( int vi )
{
  gLightIntensity = dot( normalize(LIGHTPOS - V[vi]), \
                         vNormal[vi] );
  gLightIntensity = abs( gLightIntensity );

  gl_Position = uModelViewProjectionMatrix *
                vec4( CG + uShrink * ( V[vi] - CG ), 1. );
  EmitVertex( );
}

void
main( )
{
  V[0]  =  gl_PositionIn[0].xyz;
  V[1]  =  gl_PositionIn[1].xyz;
  V[2]  =  gl_PositionIn[2].xyz;
  CG    =  ( V[0] + V[1] + V[2] ) / 3.;
  ProduceVertex( 0 );
  ProduceVertex( 1 );
  ProduceVertex( 2 );
}
```

The GLSL Shading Language

The GLSL shader language is a C-like language with some extensions and some limitations. From a pure language point of view, it has some characteristics that recall features of early programming languages. For example, there are special variables that give you access to the data set by an OpenGL application into on-card registers, several special-purpose operations on vectors and matrices that are designed specifically for graphics, special variable types to reflect the different kinds of operations that will be done with variables, and shared name spaces that provide communication between applications, vertex shaders, and fragment shaders. We will describe the language in full detail in Chapter 5.

One way to think about GLSL, or any computer language, is to consider some of the basic attributes of the language. For GLSL, some of these are given in the following table.

Goals	Primary: speed; secondary: image quality
Shader Types	Vertex, Tessellation Control, Tessellation Evaluation, Geometry, Fragment
Shader Variables	Attribute, Uniform, Constant, Out, In
Coordinate Systems	Model, World, Eye, Clip
Noise	Either as a texture or using the built-in function
Compile Shaders	Done by the driver

GLSL shader code looks much like C, with the usual operators and logic. Preprocessor commands such as `#define`, `#ifdef`, and the like are available. GLSL has some extensions to support graphics operations. These include a number of new types, including some built-in vector and matrix types that are probably new to you, but that make life in graphics much easier.

- Integer scalar and vector types: `int`, `ivec2`, `ivec3`, `ivec4`.
- Real-valued scalar and vector types: `float`, `vec2`, `vec3`, `vec4`.
- Matrix types for square real-valued matrices: `mat2`, `mat3`, `mat4`.
- Matrix types for non-square real-valued matrices: `mat3x2`, etc.
- Boolean scalar and vector types: `bool`, `bvec2`, `bvec3`, `bvec4`.
- A sampler type to access textures.

The new vector and matrix types in GLSL require some new kinds of access and operations. Many familiar operators are overloaded to handle vectors and matrices . The familiar multiplication operator * has some new meanings. For the statement `m*n`, we have four new meanings:

- If `m` is a scalar and `n` is a vector or matrix, then `m*n` is a vector or matrix of the same size as `n` whose entries are the original vector or matrix entries, each multiplied by `m`.
- If `m` and `n` are both vectors of the same size, then `m*n` is the scalar product (component-by-component product) of the vectors, *not* their dot product.
- If `m` is a matrix and `n` is a vector of compatible size, then `m*n` is a vector of the appropriate size that is the usual matrix*vector product.
- If `m` and `n` are both matrices of compatible sizes, then `m*n` is a matrix of the appropriate size that is the usual matrix*matrix product.

A number of other operations have been added, and many operations have been extended to operate on entire vectors or matrices.

Access to components of vectors involves another set of new operations. Vector components may be accessed with the familiar `[index]`, or they may use symbolic names, called *name sets*, that are familiar for the meanings of different

vectors: `.rgba` (for vectors as color), `.xyzw` (for vectors as geometry), and `.stpq` (for vectors as texture coordinates). You can also use any subset of the symbolic names to access parts of a vector. For example, `aVertex.xyz` gets you the first three components of a vertex. `aVertex.rgb` looks wrong, but would get you the same thing. Another new kind of vector access involves rearranging their components, or "swizzling" them. Components can be swizzled by giving the symbolic names of the components in changed order (e.g., `c1.rgba = c2.abgr`) to rearrange their order.

GLSL shaders also extend the normal C functionality in adding new kinds of type qualifiers for variables. The new qualifiers, and their meanings, are

- `const`—a variable that is a compile-time constant and cannot be referenced outside the shader that defines it. These variables cannot be used on the left-hand side of an assignment operation under any circumstances. (This is the same as the C++ const.)
- `attribute`—a variable, only used in a vertex shader, that is set by the application per-vertex and is generally sent from the application to the graphics card by OpenGL functions. Attribute variables may include the traditional per-vertex values of model coordinates, color, normal, normal matrix, or texture coordinates, but an application may define additional attribute variables when needed.
- `uniform`—a variable that is set outside a shader and can be changed at most once per primitive.
- `in` or `out`—variables used to communicate results from one shader to another. An `out` variable is to get its value in the shader where it is defined and be passed from that shader to the next shader further along in the shader pipeline. It is a write-only variable in the shader where it is defined. An `in` variable is to be received from a previous shader in the shader pipeline and used in the shader where it is defined. It is a read-only variable in the shader where it is defined. An `in` variable in a fragment shader will be interpolated across the fragments in a graphics primitive. This interpolation will be done in a perspective-corrected fashion; see [14].

Shaders can create their own functions, just like in C/C++, with their own parameters and local variables. Another set of type qualifiers is used for function parameters for shaders. These are keyed to the role of the parameters in the function, and are

- `in`—a parameter of this type is intended to have a value when it is passed into a function but is not to be changed in the function. It functions much as a const variable would. Such parameters are intended to communicate only from the calling function to the called function.

- `out`—a parameter of this type is not assumed to have an initial value the first time it appears in the function, but it is assumed that a value will be assigned before the function returns. Such parameters are intended to communicate only from the called function to the calling function.
- `inout`—a parameter that is intended to have a value when it is passed into a function and to have a value, possibly different, when the function returns. Such parameters are intended to provide two-way communication between the called function and the calling function.

One final additional capability in fragment shaders that should be mentioned is the *discard* operator. This is used to discard pixels so they will not be passed to the framebuffer. Note that this is quite different from having the pixels made transparent by setting their alpha color value to zero. Pixels with zero alpha still have a depth value and are recorded in the depth buffer, so they mask any pixel that might lie behind them. As you can clearly see in Figure 3.5, discarded pixels do not mask anything.

The GLSL shader language is missing some of the properties of C that you may be used to using. Remember that shaders operate in the graphics processor, not in a general-purpose processor, and that this limits the operations that it makes sense for the language to support. Many of these can be worked around (type casts) and some do not fit the concept of graphics processing (no enums or strings)—and some you simply will need to live without or will need to do outside the shader. Some of the differences include

- No type casts (use constructors instead).
- No automatic promotion (although some GLSL compilers handle this).
- No pointers.
- No strings.
- No enums.
- Can only use 1D arrays (no bounds checking).
- No file-based pre-processor directives.

There are several attribute variables that you will use a lot in your vertex shaders. These variables are defined in your application and give you access to per-vertex OpenGL state information for your shader. In the examples above, you saw some key values taken from these attribute variables, such as model coordinates, normals, and color, and these values (possibly modified) were turned into `out` variables so they could be used by tessellation or geometry shaders or interpolated later by a fragment shader. Using our variable name convention, and noting that you may use other names instead of those we chose here, these variables include

- `vec4 aVertex`—the coordinates of the current vertex in model coordinates.
- `vec3 aNormal`—the coordinates of the current vertex normal in the original coordinates.
- `vec4 aColor`—the color defined for the current vertex, if one has been defined.
- `vec4 aTexCoordi` (`i` = 0, 1, 2, ...)—the level `i` texture coordinates associated with the vertex.

There are also some uniform variables that you will use a lot. These variables are also defined in your application and are available to all your shaders. In the examples above you saw some of these variables involved in the coordinate computations. Again, these use our name convention and, noting that these names are chosen for clarity of presentation, we have

- `mat4 uModelViewMatrix`—the ModelView matrix, the product of the viewing and modeling transformation matrices, that is active for the particular vertex.
- `mat4 uProjectionMatrix`—the matrix of the projection transformation that is active for the particular vertex.
- `mat4 uModelViewProjectionMatrix`—the product of the ModelView matrix and the Projection matrix.
- `mat3 uNormalMatrix`—the normal matrix that is active for the particular vertex (as we will see, this is the inverse transpose of the ModelView matrix).

Other important uniform variables you will define in your application define lights and materials. These are described in the discussion of uniform variables below.

The built-in vertex shader output variable `gl_Position` is a particularly key variable, because you set it as the final vertex position for the remaining geometry processing. Another vertex shader output variable you may use is `gl_PointSize`.

Technically, none of the coordinate systems are part of GLSL, but they are available by applying GLSL operations. World space is not available with fixed-function OpenGL but requires the ability to define your own transformations, which, of course, shaders let you do.

There are two fragment shader variables you will use a lot. These are, in a sense, the primary output variables from a fragment shader; you give them values to set the properties of each pixel as the fragment is processed. They let you set the color and depth for a pixel, respectively.

All the operations of a fragment shader—color computation, texturing, color arithmetic, and fog—come together to set these variables. They are

- `vec4 fFragColor`—the color of the pixels.
- `float gl_FragDepth`—the depth of the pixels.

Passing Data from Your Application into Shaders

As you write any program with the OpenGL API, even if you don't intend that program to use GLSL shaders, you create data that the system will use in creating a scene. This is generally graphical data that describes the scene. For example, you can specify the color for each vertex, or you can create an array of vertices and a parallel array with data such as elevations, temperature, or any measured data. The data could be used in fixed-function operations by manipulating primitives based on your data, or with shader-based operations by putting the data into user-defined attribute or uniform data that you can access within the shader function(s). In these sections, we describe how you can create attribute or uniform data for shaders, and we give some examples that show these in action. In Chapter 9, we describe how you can create sampler data for shaders.

Defining Attribute Variables in Your Application

Attribute variables are a way to provide per-vertex data to a vertex shader. These are only available to a vertex shader. If any vertex-specific attribute data needs to be used by a later shader, the vertex shader must first convert it to an `out` variable so the later shader can take it as an `in` variable. Here we describe the general approach to defining variables that describe properties of an individual vertex in your model.

Besides the usual attribute data such as the coordinates, normal, color, or texture coordinates of a vertex, you may also need to define other data to associate with a vertex. OpenGL lets applications define custom attributes to pass to a vertex shader. Each vertex attribute has an indexed location and can contain up to four values.

As with uniform variables, you need to determine the symbol table location of an attribute variable before you can set it:

```
GLuint glGetAttribLocation( GLuint program,
                            GLchar * attribName );
```

where attribName is a character string of the name of the variable.

An application can set a per-vertex attribute using one of the functions:

```
void glVertexAttrib{i}{t}{v}(GLuint index, TYPE val)
```

The value of i can be 1, 2, 3, or 4, depending on the dimension of the data to be given to that attribute. The value of t specifies the data type for the data to be given to the attribute; this can be b (byte), s (short), i (int), f (float), d (double), ub (unsigned byte), us (unsigned short), or ui (unsigned int). The suffix v means that the data is in vector form rather than as a list of scalars. These are consistent with the format of the glVertex* functions.

Notice that the glVertexAttrib* routines do not take a program handle as one of their arguments. Since you set the attribute variables as you do the drawing, it is assumed that the intended shader program has already been made active when glVertexAttrib* is called.

The parameter index is the particular symbol table index of the attribute variable you are setting, and the parameter or parameters val are the value(s) to be written to the attribute variable at that index. All the glVertexAttrib functions are expected to be used between glBegin and glEnd, just as the built-in attribute setting functions are.

The type of the data val is expected to match the type specified in the function name. However, since the vertex attributes are always stored in an array of type vec4, any byte, short, int, unsigned byte, unsigned short, or unsigned int will be converted into a standard GLfloat before it is actually stored.

In the short application code fragment below, which uses compatibility mode for clarity, we assume that the attribute named abArray has been defined in the vertex shader as, say,

```
vec3 aMyArray[N];
```

and we want to set the values of that attribute for each vertex of a triangle. The values to be assigned to that attribute for the vertices are the values of a0, b0, and c0 (respectively a1, b1, and c1, or a2, b2, and c2). The role of the glVertexAttrib3f( ) function is to set these values for the attribute.

```
GLint myArrayLoc = glGetAttribLocation( program, "aMyArray" );
if(myArrayLoc < 0 )
  fprintf( stderr, "Cannot find Attribute variable
                    'aMyArray'\n" );
else
{
```

```
    glBegin( GL_TRIANGLES );
      glVertexAttrib3f( myArrayLoc, a0, b0, c0 );
      glVertex3f( x0, y0, z0 );
      glVertexAttrib3f( myArrayLoc, a1, b1, c1 );
      glVertex3f( x1, y1, z1 );
      glVertexAttrib3f( myArrayLoc, a2, b2, c2 );
      glVertex3f( x2, y2, z2 );
    glEnd( );
}
```

A very simple visualization per-vertex attribute example would display pressure data on a surface. The usual way this would be programmed with the fixed-function OpenGL would be to use the pressure to define the color at each vertex in the surface, and then—assuming a continuous pressure function on the surface—to send the surface's graphics primitives into the rendering stages, to be drawn with smooth shading color interpolation. However, we could also define pressure to be an attribute variable with each vertex, and use that directly for drawing the surface, giving us more options in using color to present the pressure data.

Attribute Variables in Compatibility Mode

In compatibility mode, GLSL defines a number of built-in attribute variables for a vertex shader to use directly or to pass along to other shaders. Each of the standard OpenGL functions that define a vertex (those you can call within a `glBegin-glEnd` pair) defines a built-in attribute variable that can be used by a vertex shader. Each time one of these functions is invoked, the corresponding attribute variable's value is updated. These variables are defined fully in Chapter 5 on the GLSL language, and are shown in Table 3.1.

```
attribute vec4 gl_Color;;
attribute vec3 gl_Normal;
attribute vec4 gl_Vertex;
attribute vec4 gl_MultiTexCoord0;
```

Standard OpenGL Function	Built-in Attribute Variable	Our Name
`glVertex*(...)`	`gl_Vertex`	`aVertex`
`glColor*(...)`	`gl_Color`	`aColor`
`glNormal*(...)`	`gl_Normal`	`aNormal`
`glMultiTexCoord*(i, ...)`	`gl_MultiTexCoordi, i=1..N`	`aTexCoord0`

Table 3.1. Attribute variables defined by compatibility-mode OpenGL vertex functions.

The steps in doing this are as follows:

- Define the attribute variable in the application and set the variable to its appropriate value for each vertex as you define the vertex geometry.
- Pick up the value of the attribute variable in the vertex shader and write it to an `out` variable so it can be interpolated smoothly across each graphics primitive.
- Use the variable as an `in` variable to any shader that needs it and, if appropriate, use its value to determine the color to be used in filling pixels.

This could let us add pressure contour lines, or could let us color different pressure regimes in distinct colors, or create other displays as needed. This idea will be explored more fully in Chapter 15.

Defining Uniform Variables in Your Application

GLSL uniform variables contain information that can change at most with each graphics primitive. You can think of these uniform variables as a sort of "global variables" that are available to all the shaders currently being used. If you want a shader to have data and that data isn't directly available from OpenGL, you can define your own uniform variables to give that data to a shader. Uniform variables are used within a shader, and their values are set by the application. Uniform variables can hold any kind of data, including structs and arrays, as we saw with the built-in uniform variables.

The mechanism for defining and using your own uniform variables is indirect and somewhat unusual. When you define a uniform variable in your shader program, you simply declare the variable in the usual way:

```
uniform type name;
```

This associates a name and a type with the variable, but does not associate an address. An address is only assigned when the shader program is linked. Once linking has been done, an address is available for each variable. You query the address and then use it to set the variable from your application.

But how does the application get the address for a variable it does not know about? The application must know the name of the uniform variable in a linked shader program. It can then get the location (or address) with the function

```
GLint glGetUniformLocation(GLuint program, const GLchar *name);
```

Here `program` is the value returned from the `glCreateProgram( )` function, and `name` is the name (a text string) of the uniform variable. This function

returns the address of the named variable within the named program object, so it can be used in the application. The uniform variable must be a simple variable, not an array or struct; these are handled differently. A uniform variable (either built-in or user-defined) is called *active* if the link operation finds that it can be accessed during program execution; a link operation must have been done (though it might not have succeeded) before the uniform variables in the shader program can be active.

You can think of this as creating a pipe from your application to the shader. The location you get from `glGetUniformLocation( )` is the place the pipe goes. You then use one of the `glUniform*( )` functions to put data into the pipe to get it to the shader.

The application can set the value of a uniform variable whose location is known in three ways. The first way sets scalar or simple vector data with the function

```
glUniform{i}{t}(GLint location, TYPE val)
```

where `i` can be 1, 2, 3, or 4, depending on the dimension of the variable, and `t` can be either `f` or `i`, depending on whether the type's base is floating-point or integer. The function causes the value of the parameter `val` to be loaded into the location indicated. This parameter can be a simple `vec1`, `vec2`, `vec3`, `vec4`, `ivec1`, `ivec2`, `ivec3`, or `ivec4`, but not an array of these types.

The second way sets array (vector) data with

```
glUniform{i}{t}v( GLint location, GLuint length, const TYPE
                  *val )
```

where the meanings `i` and `t` are the same, but the data in `val` is a vector of the specified type (including `vec*` and `ivec*`) whose length is `length`.

Finally, the third way sets matrices, and is

```
glUniformMatrix{i}fv( GLint location, GLuint count,
                      GLboolean transpose, const GLfloat *val )
```

If `i` has the value 2, `*val` must be a 2 × 2 matrix; if 3, a 3 × 3 matrix; and if 4, a 4 × 4 matrix. If `transpose` has value `GLfalse`, the matrix is taken to be in standard OpenGL column matrix order, while if `transpose` has value `GLtrue`, the matrix is taken to be in row-major order. The value of `count` is the number of matrices that are being passed, so if you are only passing a single matrix, that value is 1.

When you develop vertex shaders, it is sometimes nice to be able to separate the Model and the Viewing matrices, instead of having them precombined into one ModelView matrix, as OpenGL does. If you are willing

Notice that none of these `glUniform*` routines take a program handle as one of its arguments. Those routines set uniform variables in the currently active shader program. So be sure that you call `glUseProgram( )` on the correct program before setting that program's variables.

However, there is another set of GLSL API routines that let you specify the program. They look like this:

```
glProgramUniform*( program,
  loc, count, value(s) );
```

to manipulate the contents of those matrices yourself, then you can accomplish this using matrix uniform variables.

If you have defined a struct as a uniform variable, you cannot set the entire struct at once; you must use the functions above to set each field individually.

As an example, suppose you wanted to pass a light location into your shaders. The following very short code fragment, to be used in your application, wants to store a value in your shader's `vec3` uniform variable named `uLightPos`.

The `glGetUniformLocation` function lets you find the location of the uniform variable in the shader program's symbol table. The `glUniform3fv` function lets you set that uniform variable. Note also how location is checked to ensure that the variable is actually found.

```
float LightPos[3] = { 0., 100., 0. }; // values to store

GLint lightPosLoc = glGetUniformLocation( program,
                                          "uLightPos" );
            // where in the shader symbol table to store them
if( lightPosLoc < 0 )
  fprintf(stderr, "Uniform variable 'uLightPos' not found\n");

  . . .

glUseProgram( program );

if( lightPosLoc >= 0 )
  glUniform3fv( lightPosLoc, 3, lightPos );
```

A Convenient Way to Transition to the Newer Versions of GLSL

The GLSL specification has been in transition. Many (most) of the built-in GLSL variables have been deprecated in favor of defining and using your own variable names. Although it is not clear if the GLSL deprecated features will completely go away, it is clear that they might. We believe that graphics programmers should start transitioning to the new way of doing things. This is further supported by that fact that OpenGL ES 2.0 *requires* the transition

Uniform Variables in Compatibility Mode

In compatibility mode, GLSL defines a number of built-in uniform variables that give you access to OpenGL states for primitives, as we describe fully in Chapter 5 on the GLSL language. There are a number of built-in uniform variables, including the ModelView, Projection, and Normal matrices and all texture, light, and materials data. Your applications set these values through standard OpenGL functions and can use the associated uniform variables in your shaders. These give you access to all the OpenGL state values or values derived from these states. When a program object is made current, the built-in uniform variables that track the OpenGL state are initialized to the current value of those states, and any later OpenGL calls that modify state values update the built-in uniform variable that tracks those states. The most commonly-used of these are shown in Table 3.1.

Standard OpenGL Function	Built-in Uniform Variable
transformations	mat4 gl_ModelViewMatrix mat4 gl_ModelViewProjectionMatrix mat4 gl_ProjectionMatrix mat3 gl_NormalMatrix
materials	struct gl_MaterialParameters { vec4 emission; vec4 ambient; vec4 diffuse; vec4 specular; float shininess; } gl_Frontmaterial; gl_BackMaterial;
lights	struct gl_LightSourceParameters { vec4 ambient; vec4 diffuse; vec4 specular; vec4 position; vec4 halfVector; vec3 spotDirection; float spotExponent; float spotCutoff; float spotCosCutoff; } gl_LightSource[gl_MaxLights];

Table 3.2. Some common uniform variables defined by OpenGL functions in compatibility mode.

to the new approach. Compare the installed base for OpenGL desktop to the installed base for OpenGL ES (mobile), and you realize that developing applications that run only on OpenGL desktop is short-sighted. As the standard continues to evolve, you will have a huge advantage if you develop applications that can run both on the desktop and on the ubiquitous mobile devices.

For our own work, we have developed a way to start a smooth transition to the new approach through the use of a set of `#defines` in a file called *gstap.h*, shown here and also available at this book's website:

```
#ifndef GSTAP_H
#define GSTAP_H


// gstap.h -- useful for glsl migration
// from:
//   Mike Bailey and Steve Cunningham
//   "Graphics Shaders: Theory and Practice",
//   Second Edition, AK Peters, 2011.


// we are assuming that the compatibility #version line
// is given in the source file, for example:
// #version 400 compatibility


// for OpenGL-ES compatibility:

precision highp         float;
precision highp         int;


// uniform variables:

#define uModelViewMatrix          gl_ModelViewMatrix
#define uProjectionMatrix         gl_ProjectionMatrix
#define uModelViewProjectionMatrix
        gl_ModelViewProjectionMatrix
#define uNormalMatrix             gl_NormalMatrix
#define uModelViewMatrixInverse   gl_ModelViewMatrixInverse
```

2. "gstap" stands for the book title, *Graphics Shaders: Theory and Practice*.

```
// attribute variables:

#define aColor          gl_Color
#define aNormal         gl_Normal
#define aVertex         gl_Vertex

#define aTexCoord0        gl_MultiTexCoord0
#define aTexCoord1        gl_MultiTexCoord1
#define aTexCoord2        gl_MultiTexCoord2
#define aTexCoord3        gl_MultiTexCoord3
#define aTexCoord4        gl_MultiTexCoord4
#define aTexCoord5        gl_MultiTexCoord5
#define aTexCoord6        gl_MultiTexCoord6
#define aTexCoord7        gl_MultiTexCoord7


#endif   // #ifndef GSTAP_H
```

These #defines allow you to use new names for things, without having to (yet) define them and pass them in yourself. Then, when the time comes to complete your migration to the new approach, you don't need to make massive code changes to your shaders. Note that these names use our variable naming standard descsribed earlier in this chapter.

To make life even easier for you, the `gstap.h` code has been built-in to the *glman* software, so that every shader source that you load automatically has it included. Just include a line in your .glib file with the word *gstap.h* on it. If you use *glman*, there is no reason not to transition away from the deprecated built-in variables right away.

Exercises

The code for all the shaders discussed in this chapter is available on the book's website, and this chapter's exercises are mostly concerned with experiments on this code using the *glman* application. Details on *glman* are discussed in the next chapter, so you may want to use it as a reference while you work on these exercises.

1. Experiment with shape: in this chapter we only used spheres for our examples, but *glman* allows you to use a number of other kinds of shapes. In the GLIB file for any of these examples, replace the sphere by other