An Introduction to 3D Computer Graphics, Stereoscopic Image, and Animation in OpenGL and C/C++

Fore June

Chapter 19 OpenGL Shading Language (GLSL)

19.1 Extending OpenGL

The OpenGL architecture we have presented in previous chapters is called fixed-pipeline architecture, in which the functionality of each processing stage is fixed. The user can control the parameters to the functions but the underlying processing of the functions are fixed. All OpenGL versions up to 1.5 are based on this fixed-function pipeline. Figure 19-1 below shows this traditional fixed function pipeline architecture:

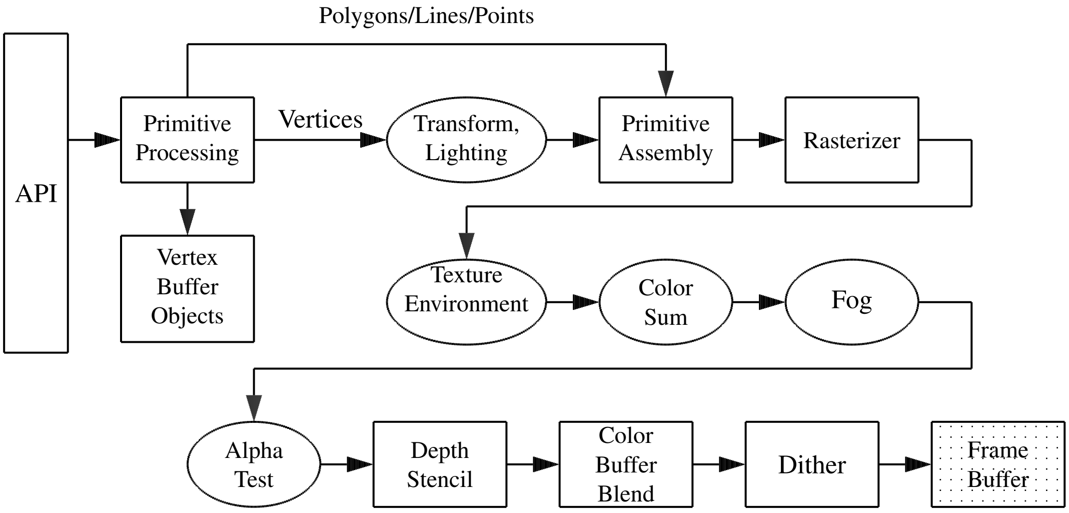


Figure 19-1. Fixed Function Pipeline

Before version 2.0, to modify OpenGL we must define the new features through extensions. Consequently, a lot of OpenGL functionality is available in the form of extensions that expose new hardware functionality. OpenGL has well-defined extensions for hardware vendors to define and implement special graphics features. For complex applications, there is a trend in graphics hardware to replace fixed functionality with programmability, including vertex processing and fragment processing.

Since version 2.0, besides supporting the fixed pipeline architecture, OpenGL also supports the programmable pipeline, which replaces the fixed function transformations and fragment pipeline by programmable shaders as shown in Figure 19-2 below.

The OpenGL Shading Language (glsl) is designed to address the programmable issue and is part of the OpenGL distribution. It allows programmers to write shaders to alter the processing of the graphics attributes along the pipeline. GLSL is an extension of the original OpenGL with the following properties:

1. GLSL is included in OpenGL 2.0, which was released in 2004.
2. Other competing shading languages include Cg (C for Graphics), which is cross platform and HLSL (High Level Shading Language) by Microsoft, which is used in DirectX.
3. GLSL is part of OpenGL. Therefore, it is naturally easily integrated with OpenGL programs.
4. It is a C-like language with some C++ features,
5. It is mainly used for processing numerics, but not for strings or characters.
6. OpenGL 1.5 has fixed function pipeline.
7. OpenGL 2.0 allows processors to be programmed, introducing GLSL, which is approved by OpenGL Architectural Review Board (ARB).

8. With programmed shaders, data flow from the application to the vertex processor, on to the fragment processor and ultimately to the frame buffer. This allows vendors to produce faster graphics hardware with more parallel features.

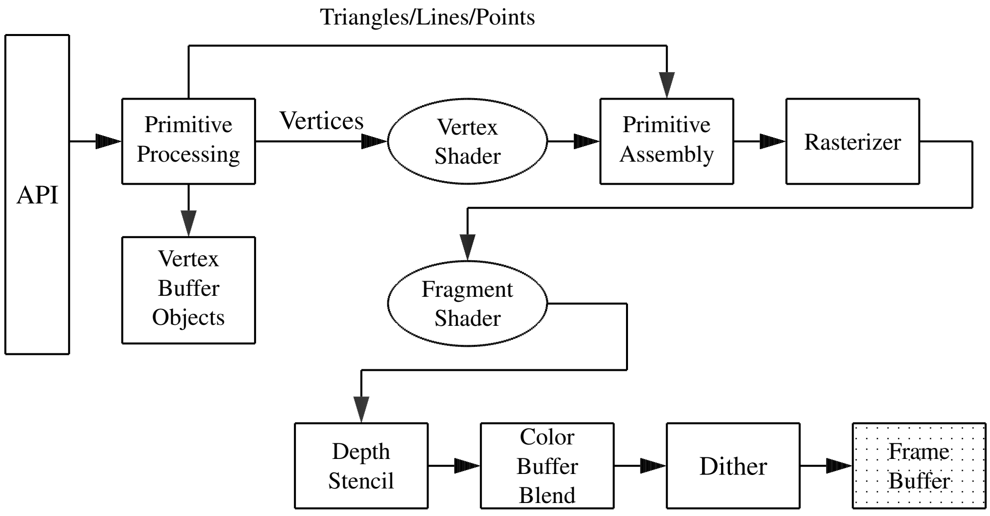


Figure 19-2. Programmable Pipeline

19.2 OpenGL Shaders Execution Model

We can consider a driver as a piece of software that manages the access of a hardware. In this sense, we can view OpenGL libraries as drivers because they manage shared access to the underlying graphics hardware; applications communicate with graphics hardware by calling OpenGL functions. An OpenGL shader is embedded in an OpenGL application and may be viewed as an object in the driver to access the hardware. We use the command **glCreateShader**() to allocate within the OpenGL driver the data structures needed to store an OpenGL shader. The source code of a shader is provided by an application by calling **glShaderSource**() and we have to provide the source code as a null-terminated string to this function.. Figure 19-3 below shows the steps to create a shader program for execution.

There are two kinds of shaders, the vertex shaders and the fragment shaders. A **vertex shader** (program) is a shader running on a **vertex processor**, which is a programmable unit that operates on incoming vertex values. This processor usually performs traditional graphics operations including the following:

1. vertex transformation
2. normal transformation and normalization
3. texture coordinate generation
4. texture coordinate transformation
5. lighting
6. color material application

The following is an example of a simple "pass-through" vertex shader, which does not do anything:

```
//A simple pass-through vertex shader
void main()
{
  gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

A **fragment shader** is a shader running on a **fragment processor**, which is a programmable unit that operates on fragment values. A fragment is a pixel plus its attributes such as color and depth. A fragment shader is executed after the rasterization. Therefore a fragment processor operates on each fragment rather than on each vertex. It usually performs traditional graphics operations including:

1. operations on interpolated values
2. texture access, and application
3. fog effects
4. color sum
5. pixel zoom
6. scaling
7. color table lookup
8. convolution
9. color matrix operations

The following is an example of a simple fragment shader which sets the color of each fragment for rendering.

```
//A simple fragment shader
void main()   {
  gl_FragColor = gl_FrontColor;
}
```

19.3 OpenGL Shading Language API

Figure 19-3 below shows the development steps of a glsl shader program.

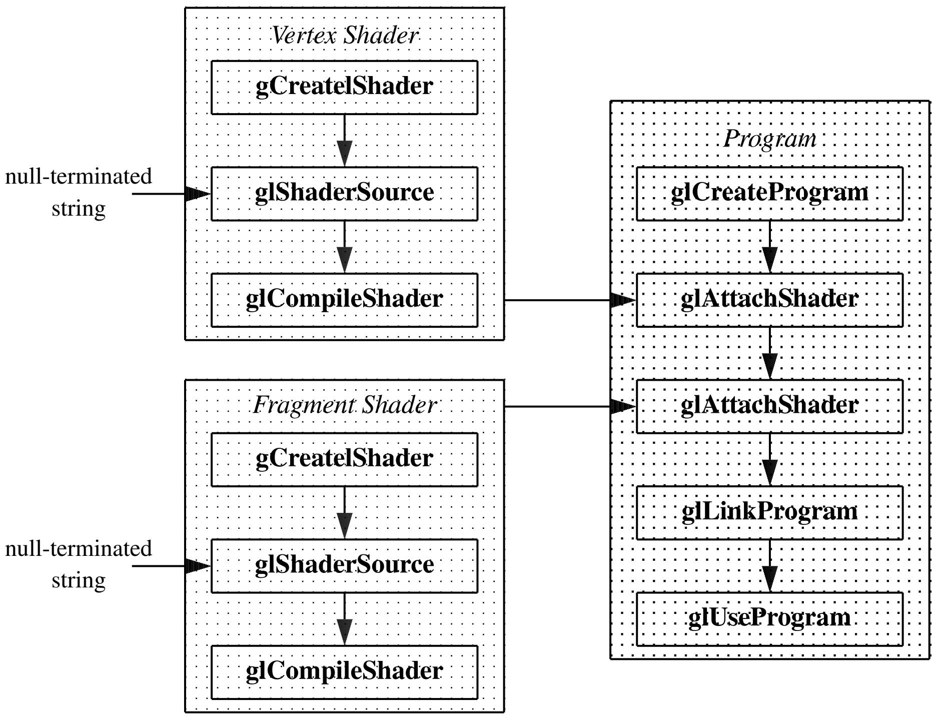


Figure 19-3 Shader Program Development

Table 19-1 below lists the OpenGL functions involved in the process.

Table 19-1 OpenGL Commands for Embedding Shaders

glCreateShader()	Creates one or more shader objects.
glShaderSource()	Provides source codes of shaders.
glCompileShader()	Compiles each of the shaders.
glCreateProgram()	Creates a program object.
glAttachShader()	Attach all shader objects to the program.
glLinkProgram()	Link the program object.
glUseProgram()	Install the shaders as part of the OpenGL program.

The following are the normal steps to develop an OpenGL shader program.

1. **Creating a Shader Object**

   We first create an empty shader object using the function **glCreateShader**, which has the following prototype:

   Gluint **glCreateShader** (GLenum *shaderType*)
   Creates an empty shader.
   shaderType specifies the type of shader to be created. It can be either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.
   Return: A non-zero integer handle for future reference.

2. **Providing Source Code for the Shader**

   We pass the source code to the shader as a null-terminated string using the function **glShaderSource** which has the prototype:

   void **glShaderSource** (GLuint *shader*, GLsizei *count*, const GLchar ***string*, const GLint *lengthp)
   Defines a shader's source code.
   shader is the shader object created by glCreateShader().
   string is the array of strings specifying the source code of the shader.
   count is the number of strings in the array.
   lengthp points to an array specifying the lengths of the strings. If NULL, the strings are NULL-terminated.

   The source code can be hard-coded as a string in the OpenGL program or it can be saved in a separate file and read into an array as a null-terminated string. The following example shows how this is done.

```
struct stat statBuf;
FILE* fp = fopen ( fileName, "r" );
char* buf;

stat ( fileName, &statBuf );
buf = (char*) malloc ( statBuf.st_size + 1 * sizeof(char) );
fread ( buf, 1, statBuf.st_size, fp );
buf[statBuf.st_size] = '\0';
fclose ( fp );
return buf;
```

   In the example, the **stat**() function gives detailed information about a file; from it we obtain the size of the file containing the shader source code and allocate the buffer *buf* to hold the string. At the end of the string we save the null character '\0' to indicate its end.

3. **Compiling Shader Object**

   We use the function **glCompileShader** to compile the shader source code to object code. This function has the following prototype.

   ---

   void **glCompileShader** (GLuint *shader*)
   Compiles the source code strings stored in the shader object *shader*.
   The function **glShaderInfoLog** gives the compilation log.

   ---

4. Linking and Using Shaders

   Each shader object is compiled independently. To create a shader program, we need to link all the shader objects to the OpenGL application. These are done within the C/C++ application using the functions **glCreateProgram, glAttachShader, glLinkProgram**, and **glUseProgram**, which have the prototypes listed below. These are done while we are running the C/C++ application. Performing the steps of compiling and linking shader objects are simply making C function calls.

   GLuint **glCreateProgram** (void)
   Creates an empty program object and returns a non-zero integer handle for future reference.

   void **glAttachShader** (GLuint *program*, GLuint *shader*)
   Attaches the shader object specified by *shader* to the program object specified by *program*.

   void **glLinkProgram** (GLuint *program*)
   Links the program objects specified by *program*.

   void **glUseProgram** (GLuint *program*)
   Installs the program object specified by *program* as part of current rendering state.
   If *program* is 0, the programmable processors are disabled, and fixed functionality is used for both vertex and fragment processing.

   ---

5. **Cleaning Up**

   At the end, we need to release all the resources taken up by the shaders. The clean up is done by the commands,
   void **glDeleteShader** (GLuint *shader*),
   void **glDeleteProgram** (GLuint *program*),
   void **glDetachShader** (GLuint *program*, GLuint *shader*).

Listing 19-1 (a)-(c) below is a complete example of a shader program; the OpenGL application is the C/C++ program **tests.cpp**, which does the shader creation, reading shader source code, shader compilation and shader linking. The shader source code for the vertex shader is saved in the text file **tests.vert**, and the code for the fragment shader is saved in **tests.frag**. In compiling **tests.cpp**, we need to link the GL extension library by "-lGLEW". If we compile "tests.cpp" to the executable "tests", we can run the shader program by typing "./tests" and press 'Enter'. Note that when we change the code of a shader ("tests.vert" or "tests.frag"), we do not need to recompile the C/C++ program "tests.cpp". We just need to execute "./tests" and the changed features will take place. Figure 19-4 below shows the output of executing “tests”.

Program Listing 19-1 Complete Example of a Shader Program

(a) tests.cpp

```
/*
  tests.cpp
  Sample program showing how to write GL shader programs.
  Shader sources are in files "tests.vert" and "tests.frag".
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <GL/glew.h>
#include <GL/glut.h>

using namespace std;

// Global handles for the current program object, with its two shader objects
GLuint programObject = 0;
GLuint vertexShaderObject = 0;
GLuint fragmentShaderObject = 0;
static GLint win = 0;

int readShaderSource(char *fileName, GLchar **shader )
{
    // Allocate memory to hold the source of our shaders.
    FILE *fp;
    int count, pos, shaderSize;

    fp = fopen( fileName, "r");
    if ( !fp )
        return 0;
    struct stat statBuf;
    stat ( fileName, &statBuf );
    shaderSize = statBuf.st_size;
    if ( shaderSize <= 0 ){
        printf("Shader %s empty\n", fileName);
        return 0;
    }

    *shader = (GLchar *) malloc( shaderSize + 1);

    // Read the source code
    count = (int) fread(*shader, 1, shaderSize, fp);
    (*shader)[count] = '\0';

    if (ferror(fp))
        count = 0;

    fclose(fp);

    return 1;
}

int installShaders ( const GLchar *vs, const GLchar *fs )
{
    GLint  vertCompiled, fragCompiled;  // status values
    GLint  linked;
    // Create a vertex shader object and a fragment shader object
    vertexShaderObject = glCreateShader ( GL_VERTEX_SHADER );
    fragmentShaderObject = glCreateShader ( GL_FRAGMENT_SHADER );

    // Load source code strings into shaders, compile and link
    glShaderSource ( vertexShaderObject, 1, &vs, NULL );
    glShaderSource ( fragmentShaderObject, 1, &fs, NULL );

    glCompileShader ( vertexShaderObject );
    glGetShaderiv ( vertexShaderObject, GL_COMPILE_STATUS, &vertCompiled );

    glCompileShader ( fragmentShaderObject );
    glGetShaderiv( fragmentShaderObject, GL_COMPILE_STATUS, &fragCompiled);

    if (!vertCompiled || !fragCompiled)
        return 0;

    // Create a program object and attach the two compiled shaders
    programObject = glCreateProgram();
    glAttachShader( programObject, vertexShaderObject);
    glAttachShader( programObject, fragmentShaderObject);

    // Link the program object
    glLinkProgram(programObject);
    glGetProgramiv(programObject, GL_LINK_STATUS, &linked);

    if (!linked)
        return 0;

    // Install program object as part of current state
    glUseProgram ( programObject );

    return 1;
}

int init(void)
{
   GLchar *VertexShaderSource, *FragmentShaderSource;
   int loadstatus = 0;

   const char *version  = (const char *) glGetString ( GL_VERSION );
   if (version[0] < '2' || version[1] != '.') {
      printf("This program requires OpenGL >= 2.x, found %s\n", version);
      return 0;
   }
   readShaderSource("tests.vert", &VertexShaderSource );
   readShaderSource("tests.frag", &FragmentShaderSource );
   loadstatus = installShaders(VertexShaderSource, FragmentShaderSource);
   if ( !loadstatus ) {
     printf("\nCompilation of shaders not successful!\n");
   }

   return loadstatus;
}

static void reshape(int width, int height)
{
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-1.0, 1.0, -1.0, 1.0, 5.0, 25.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0f, 0.0f, -15.0f);
}
void CleanUp(void)
{
   glDeleteShader(vertexShaderObject);
   glDeleteShader(fragmentShaderObject);
   glDeleteProgram(programObject);
   glutDestroyWindow(win);
}

void display(void)
{
   glClearColor ( 1.0, 1.0, 1.0, 0.0 ); //get white background color
   glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
   glColor3f ( 0, 1, 0 );         //green, no effect if shader is loaded
   glLineWidth ( 4 );
   glutWireSphere ( 2.0, 16, 8 );
   glutSwapBuffers();
   glFlush();
}

int main(int argc, char *argv[])
{
   glutInit(&argc, argv);
   glutInitWindowPosition( 0, 0 );
   glutInitWindowSize ( 300, 300 );
   glutInitDisplayMode ( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
   win = glutCreateWindow(argv[0]);
   glutReshapeFunc ( reshape );
   glutDisplayFunc ( display );
   // Initialize the "OpenGL Extension Wrangler" library
   glewInit();

   int successful = init();
   if ( successful )
     glutMainLoop();

   return 0;
}
--------------------------------------------------------------------------------
```

(b) tests.vert

```
// tests.vert : a minimal vertex shader
void main(void)
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
--------------------------------------------------------------------------------
```

(c) tests.frag

```
// tests.frag : a minimal fragment shader
void main(void)
{
   gl_FragColor = vec4( 1, 0, 0, 1);   //red color
}
--------------------------------------------------------------------------------
```

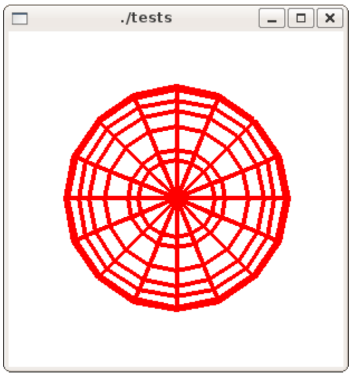


Figure 19-4 Output of Shader Program **tests.cpp**

19.4 Data Types in GLSL

There are four main data types in GLSL: **float, int, bool**, and **sampler**. Vector types are available for the first three types:

vec2, vec3, vec4	2D, 3D and 4D floating point vector
ivec2, ivec3, ivec4	2D, 3D and 4D integer vector
bvec2, bvec3, bvec4	2D, 3D and 4D boolean vectors

For floats there are also matrix types:

mat2, mat3, mat4	$2 \times 2, 3 \times 3, 4 \times 4$ floating point matrix

Samplers are types used for representing textures:

sampler1D, sampler2D, sampler3D	1D, 2D and 3D texture
samplerCube	Cube Map texture
sampler1Dshadow, sampler2Dshadow	1D and 2D depth-component texture

Attributes, Uniforms and Varyings

GLSL shaders have three different input-output data types for passing data between vertex and fragment shaders, and the OpenGL application. The data types are **uniform, attribute** and **varying**. They must be declared as global (visible to the whole shader object). The variables have the following properties:

1. **Uniforms** : These are read-only variables (i.e. A shader object can only read the variables but cannot change them.). Their values do not change during a rendering. Therefore, Uniform variable values are assigned outside the scope of **glBegin/glEnd**. Uniform variables are used for sharing data among an application program, vertex shaders, and fragment shaders.

2. **Attributes**: These are also read-only variables. They are only available in vertex shaders. They are used for variables that change at most once per vertex in a vertex shader. There are two types of attribute variables, user-defined and built-in. The following are examples of user-defined attributes:

```
attribute float x;
attribute vec3 velocity, acceleration;
```

Built-in variables include OpenGL state variables such as color, position, and normal; the following are some examples:

gl_Vertex
gl_Color

3. **Varyings**: These are read/write variables, which are used for passing data from a vertex shader to a fragment shader. They are defined on a per-vertex basis but are interpolated over the primitive by the rasterizer. They can be user-defined or built-in.

Built-in Types

The following tables list some more of the GLSL built-in types.

Table 19-2 Built-in Attributes (for Vertex Shaders)

gl_Vertex	4D vector representing the vertex position
gl_Normal	3D vector representing the vertex normal
gl_Color	4D vector representing the vertex color
gl_MultiTexCoord$n$	4D vector representing the texture coordinate of texture n

Table 19-3 Built-in Uniforms (for Vertex and Fragment Shaders)

gl_ModelViewMatrix	4×4 Matrix representing the model-view matrix
gl_ModelViewProjectionMatrix	4×4 Model-view-projection matrix
gl_NormalMatrix	3×3 Matrix used for normal transformation

Table 19-4 Built-in Varyings (for Data Sharing between Shaders)

gl_FrontColor	4D vector representing the primitives front color
gl_BackColor	4D vector representing the primitives back color
gl_TexCoord[n]	4D vector representing the n-th texture coordinate
gl_Position	4D vector representing the final processed vertex position (vertex shader only)
gl_FragColor	4D vector representing the final color written in the frame buffer (fragment shader only)
gl_FragDepth	float representing the depth written in the depth buffer (fragment shader only)

GLSL has many built in functions, including

1. trigonometric functions: **sin, cos, tan**
2. inverse trigonometric functions: **asin, acos, atan**
3. mathematical functions: **pow, log2, sqrt, abs, max, min**
4. geometrical functions: **length, distance, normalize, reflect**

Built-in types provide users an effective way to access OpenGL variables as they are mapped to the OpenGL states. For example, if we call **glLightfv**(GL_LIGHT0, GL_POSITION, *myLight-Position*), its value is available as a **uniform** using gl_LightSource[0].position in a vertex and/or fragment shader.

The following is an example of using various data types; it consists of a vertex shader and a fragment shader for defining a modified Phong lighting model.

Program Listing 19-2 Shaders for Modified Phong Lighting

(a) Vertex Shader: phong.vert

```
//phong.vert
varying vec3 N;  //normal direction
varying vec3 L;  //light source direction
varying vec3 E;  //eye position

void main(void)
{
  gl_Position =gl_ModelViewMatrix*gl_Vertex;
  vec4 eyePosition = gl_ModelViewProjectionMatrix*gl_Vertex;
  vec4 eyeLightPosition = gl_LightSource[0].position;

  N = normalize( gl_NormalMatrix*gl_Normal );
  L = eyeLightPosition.xyz - eyePosition.xyz;
  E = -eyePosition.xyz;
}
```

(b) Fragment Shader: phong.frag

```
//phong.frag
varying vec3 N;
varying vec3 L;
varying vec3 E;
void main()
{
  vec3 norm = normalize(N);
  vec3 lightv = normalize(L);
  vec3 viewv = normalize(E);
  vec3 halfv = normalize(lightv + viewv);
  float f;
  if(dot(lightv, norm)>= 0.0) f =1.0;
  else f = 0.0;

  float Kd = max(0.0, dot(lightv, norm));
  float Ks = pow(max(0.0, dot(norm, halfv)), gl_FrontMaterial.shininess);
  vec4 diffuse = Kd * gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
  vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
  vec4 specular = f*Ks*gl_FrontMaterial.specular*gl_LightSource[0].specular;
  gl_FragColor = ambient + diffuse + specular;
}
```

19.5 The OpenGL Extension Wrangler Library

To develop applications with glsl, we also need the *OpenGL Extension Wrangler Library* (GLEW), which is a cross-platform open-source C/C++ extension loading library. It is a simple tool providing efficient run-time mechanisms to determine the OpenGL extensions that are supported on the target platform. An application uses the OpenGL core and extension functionality in a single header file. The library has been tested on a various operating systems, including Windows, Linux, Mac OS X, FreeBSD, Irix, and Solaris. Currently, the library supports OpenGL 4.4 and the following extensions:

1. OpenGL extensions
2. WGL extensions
3. GLX extensions

We can download the package from the site:

`http://sourceforge.net/projects/glew/`

The compilation and installation process is simple and is described in the *README.txt* file of the package. The version that we have used to test the programs described here is *glew-1.11.0*.

19.6 Drawing Polygons

To make our code easier to be ported to OpenGL ES platforms, the graphics objects of many of our examples are drawn using triangles. OpenGL ES, where ES stands for *embedded systems*, is a streamlined version of OpenGL for rendering sophisticated 3D graphics on handheld and embedded devices.

19.6.1 Creating a *Shader* Class

We have seen in Section 19.3 that we always need to call the same few functions to create shaders that render graphics objects. It will be more convenient if we create a base class called *Shader* to call these functions and do the miscellaneous initializations such as compiling and linking. When we draw an actual shape such as a triangle or a rectangle, we can always create a corresponding class that extends this base class to draw the actual shape. This concept is shown in Figure 19-5 below.

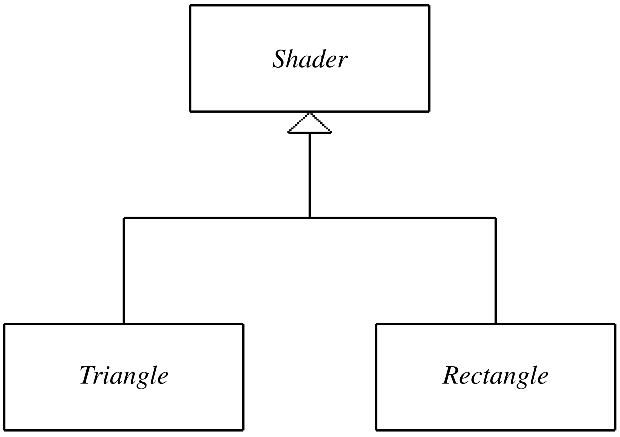


Figure 19-5 *Shader* Class and Typical Subclasses

Subclasses or applications should provide the shader source codes to the functions of the *Shader* class, which does the routine work of creating shader programs and objects, compiling the shader source code, linking the shaders, attaching the shaders, using the shader programs and cleaning up as shown in Figure 19-3 above. The following shows the members of this class:

```
//Shader.h
class Shader
{
  public:
   int program;
   int vertexShader;
   int fragmentShader;
   char *infoLog;
   Shader();
   ~Shader();
   int loadShader (int shaderType, const string *shaderCode );
   bool createShader( const string *vs, const string *fs );
```

```
    void cleanUp();
  };
--------------------------------------------------------------------------
```

Typically, an application (or a subclass) calls the function **createShader** of this class to create a vertex shader and a fragment shader. The application provides pointers as input string parameters, pointing to the source code of the vertex shader (*vs* and that of the fragment shader (*fs*). If the application only needs to create one shader, it can set the other pointer pointing to **NULL**. The following is its implementation:

```
--------------------------------------------------------------------------
 bool  Shader::createShader( const string *vs, const string *fs )
 {
   // create empty OpenGL Program, load, attach, and link shaders
   program =  glCreateProgram();
   if ( vs != NULL ) {
     vertexShader = loadShader( GL_VERTEX_SHADER, vs);
     // add the vertex shader to program
     glAttachShader(program,vertexShader);
   }
   if ( vs != NULL ) {
     fragmentShader = loadShader( GL_FRAGMENT_SHADER, fs);
     // add the fragment shader to program
     glAttachShader(program,fragmentShader);
   }
   glLinkProgram(program); // creates program executables
   int linked;
   glGetProgramiv(program, GL_LINK_STATUS, &linked);

   if (!linked) {
     printf( "Shader not linked!\n" );
     return false;         // mission failed
   }

   glUseProgram( program); // use shader program

   return true;            // mission successful
 }
--------------------------------------------------------------------------
```

This function actually calls the other member function, **loadShader** to create a shader object and compile it. The function **loadShader** also saves the log information in memory, and the data member pointer *infoLog* points at the memory. If necessary, the application can examine the log for debugging or other purposes. The following is its implementation:

```
--------------------------------------------------------------------------
 int Shader::loadShader (int shaderType, const string *shaderCode)
 {
   // create a vertex shader type ( GL_VERTEX_SHADER)
   // or a fragment shader type ( GL_FRAGMENT_SHADER)
   int shader =  glCreateShader( shaderType);

   // pass source code to the shader and compile it
   char *strPointer = (char *) shaderCode->c_str();
   glShaderSource(shader, 1, &strPointer, NULL);
   glCompileShader(shader);
```

```
  int compiled;
  glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled);
  if ( !compiled )
    printf("Compiling %d failed!\n", shaderType );

  int maxLength;
  glGetShaderiv( shaderType, GL_INFO_LOG_LENGTH, &maxLength);

  // maxLength includes NULL character
  infoLog = (char *) malloc ( sizeof( char ) * maxLength );
  glGetShaderInfoLog(vertexShader,maxLength,&maxLength,infoLog);

  return shader;
}
------------------------------------------------------------------------
```

19.6.2 Drawing a Triangle

As a very simple example, we write a class called *Triangle*, which extends the classs *Shader* to create shaders to draw a triangle with a specified color. For simplicity, the shader source codes are hard-coded in the *Triangle* class:

```
------------------------------------------------------------------------
class Triangle : public Shader
{
 private:
  static const string vsCode;   // Source code of vertex shader
  static const string fsCode ;  // Source code of fragment shader
  static const int vertexCount = 3;
  static const int COORDS_PER_VERTEX = 3;
  static const float triangleCoords[];
  static const float color[];
 public:
  Triangle( int &success );
  void draw();
};

const string Triangle::vsCode =
   "attribute vec4 vPosition;   \
      void main() {             \
      gl_Position = vPosition;  \
   }";


const string Triangle::fsCode =
    "uniform vec4 vColor;       \
     void main() {              \
       gl_FragColor = vColor;   \
     }";

const float Triangle::triangleCoords[] =
{   // in counterclockwise order:
   0.0f,  0.9f, 0.0f, // top vertex
  -0.5f, -0.3f, 0.0f, // bottom left
```

```
   0.5f, -0.2f, 0.0f  // bottom right
};

// Set color of displaying object
// with red, green, blue and alpha (opacity) values
const float Triangle::color[] = {0.89f, 0.6f, 0.4f, 1.0f};
---------------------------------------------------------------------------
```

This class makes use of the functions of its parent and provides the source codes to do the initilization of the shaders in the constructor:

```
---------------------------------------------------------------------------
// Create a Triangle object
Triangle::Triangle ( int  &success )
{
 string *vs, *fs;
 vs = (string *) &vsCode;
 fs = (string *) &fsCode;
 success = createShader ( vs, fs );
 if ( !success )
   printf("infoLog: %s\n", infoLog );
}
---------------------------------------------------------------------------
```

The **draw** function of *Triangle* shown below draws the triangle using the OpenGL command **glDrawArrays**. Before calling this command, it must pass the values of the triangle vertices to the vertex shader and the color values to the fragment shader. We pass the vertex data via the **attribute** variable *vPosition* and use the command **glVertexAttribPointer** to point to where the vertex data are located. The color value is passed to the fragment shader via the **uniform** variable *vColor*. The command **glUniform4fv** points to the location of the color data. The function **glUniform** has 1 to 4 dimensional forms and a vector (v) version, and we can use it to set scalar or vector values.

```
---------------------------------------------------------------------------
void Triangle::draw()
{
 // get handle to vertex shader's attribute variable vPosition
 int positionHandle= glGetAttribLocation(program,"vPosition");

 // Enable a handle to the triangle vertices
 glEnableVertexAttribArray( positionHandle );

 // Prepare the triangle coordinate data
 glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                         GL_FLOAT, false, 0, triangleCoords);

 // get handle to fragment shader's uniform variable vColor
 int colorHandle =  glGetUniformLocation(program, "vColor");
 if ( colorHandle == -1 )
   printf("No such uniform named vColor\n");

 // Set color for drawing the triangle
 glUniform4fv ( colorHandle, 1, color );

 // Draw the triangle
 glDrawArrays( GL_TRIANGLES , 0, vertexCount);
```

```
  // Disable vertex array
  glDisableVertexAttribArray(positionHandle);
 }
```

We can now render the triangle with a code similar to the following:

```
   .....
   int  loadStatus = 0;
   Triangle *triangle = new Triangle( loadStatus );
   triangle->draw();
   triangle->cleanUp();
   glutSwapBuffers();
   glFlush();
```

The *Triangle* constructor calls the functions of its parent *Shader* to do all the initialization of the shaders and returns the status of loading the shaders via the reference variable *loadStatus*, with a value of 1 meaning successful loading and 0 meaning failure. The shader source codes are hard-coded in the *Triangle* class. The **draw** method passes the necessary information to the shaders to render a triangle. The example here has not setup any world window or viewing volume; default values are used. In the vertex shader, we have not applied any transformation operation to the vertex values. Figure 19-6 below shows the output of this code.

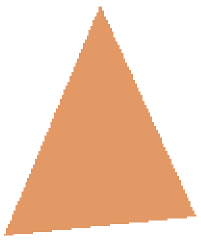


Figure 19-6 Output of Triangle Shader

19.6.2 Temperature Shader

Our next example is a temperature shader where we use colors to represent temperatures with red meaning hot and blue meaning cold. We draw a square with temperature gradient where a warm temperature is a mixture of red and blue. We can imagine that the square is a metallic sheet with each corner connected to a heat or a cooling source. We can express smoothly the surface temperature as a mixture of red and blue. In the example, we assume that the lowest temperature is 0 and the highest is 50.

To accomplish this we pass the temperature value at each square vertex via an **attribute** array variable called *vertexTemp* to the vertex shader. The shader normalizes it to a value between 0 and 1 before passing the value to the fragment shader via the **varying** variable *temperature*. We create a class called *Square*, which is similar to the *Triangle* class above, to load the shaders and pass values to them:

```
 class Square : public Shader
 {
   private:
    static const string vsCode;   //Source code of vertex shader
    static const string fsCode ;  //Source code of fragment shader
    static const int vertexCount = 4;
    static const int COORDS_PER_VERTEX = 3;
    static const float squareCoords[];
    static const float vertexTemp[];
   public:
    Square( int &success );
    void draw();
 };
------------------------------------------------------------------------
```

In this example, instead of hard-coding the shader source codes in the class, we read them from external files. So we add a function called **readShaderFile**() to the parent class *Shader*; it loads a shader source code from a file to a character array:

```
------------------------------------------------------------------------
 int Shader::readShaderFile(char *fileName, char **shader)
 {
    // Allocate memory to hold the source of our shaders.
    FILE *fp;
    int count, pos, shaderSize;

    fp = fopen( fileName, "r");
    if ( !fp )
        return 0;

    pos = (int) ftell ( fp );
    fseek ( fp, 0, SEEK_END );                  //move to end
    shaderSize = ( int ) ftell ( fp ) - pos;    //calculates file size
    fseek ( fp, 0, SEEK_SET );                  //rewind to beginning

    if ( shaderSize <= 0 ){
        printf("Shader %s empty\n", fileName);
        return 0;
    }
    // allocate memory
    *shader = (char *) malloc( shaderSize + 1);

    if ( *shader == NULL )
      printf("memory allocation error\n");
    // Read the source code
    count = (int) fread(*shader, 1, shaderSize, fp);
    (*shader)[count] = '\0';

    if (ferror(fp))
        count = 0;
   fclose(fp);

    return 1;
 }
------------------------------------------------------------------------
```

In this function, *fileName* is the input parameter holding the file name of the shader source and *shader* is the output parameter pointing to the address of a character array that stores the shader source. The function simply opens the file, calculating its length, allocating memory for the shader, and reading the source code into the allocated character array. The function returns 1 for success and 0 for failure.

We can load the shaders and peform the initialization in the constructor of *Square*, assuming that the source codes of the vertex shader and the fragment shader are saved in the files *temp.vert* and *temp.frag* respectively:

```
------------------------------------------------------------------------
 // Create a Square object
 Square::Square ( int  &success )
 {
  string *vs, *fs;
  char *vsSource, *fsSource;

  // Read shader source code.
  readShaderFile( (char *) "temp.vert",  &vsSource);
  readShaderFile( (char *)"temp.frag",  &fsSource);
  vs = new string ( vsSource );
  fs = new string ( fsSource );
  success = createShader ( vs, fs );
  if ( !success ) {
    printf("infoLog: %s\n", infoLog );
    return;
  }
  delete vs;       delete fs;
  delete vsSource; delete fsSource;
 }
------------------------------------------------------------------------
```

On the other hand, we hard-code the coordinates of the square vertices and their associated temperatures in the class *Square*. These data are passed to the vertex shader in the **draw**() function, where the command **glVertexAttribPointer** is used to point to the data array:

```
------------------------------------------------------------------------
 //  Coordinates of a square
 const float Square::squareCoords[] =
 { // in counterclockwise order:
   -0.8f,  0.8f, 0.0f,  // top left vertex
   -0.8f, -0.8f, 0.0f,  // bottom left
    0.8f, -0.8f, 0.0f,  // bottom right
    0.8f,  0.8f, 0.0f   // upper right
 };

 // Temperature at each vertex
 const float Square::vertexTemp[] = {
    5.0f,              // v0 cold (top left)
    12.0f,             // v1 cool
    22.0f,             // v2 warm
    40.0f              // v3 hot (upper right)
 };

 void Square::draw()
 {
  // get handle to vertex shader's attribute variable vPosition
  int positionHandle= glGetAttribLocation(program,"vPosition");
  int vertexTempHandle= glGetAttribLocation(program,"vertexTemp");
  // Enable a handle to the square vertices
  glEnableVertexAttribArray( positionHandle );

  // Prepare the square coordinate data
  glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                           GL_FLOAT, false, 0, squareCoords);
  glEnableVertexAttribArray( positionHandle );

  // Enable a handle to the temperatures at vertices
  glEnableVertexAttribArray( vertexTempHandle );
  // Pointing to the vertex temperatures
  glVertexAttribPointer (vertexTempHandle, 1,
                           GL_FLOAT, false, 0, vertexTemp);

  GLchar names[][20] = { "coldColor", "hotColor", "tempRange" };
  GLint handles[10];
  for ( int i = 0; i < 3; ++i ) {
    handles[i] =  glGetUniformLocation(program, names[i]);
    if (handles[i] == -1)
      printf("No such uniform named %s\n", names[i]);
  }

  // set uniform values
  glUniform3f(handles[0], 0.0, 0.0, 1);   //cold color
  glUniform3f(handles[1], 1.0, 0.0, 0.0); //hot color
  glUniform1f(handles[2], 50.0);          //temperature range

  // Draw the square
  glDrawArrays( GL_QUADS, 0, vertexCount);

  // Disable vertex array
  glDisableVertexAttribArray(positionHandle);
  glDisableVertexAttribArray(vertexTempHandle);
 }
------------------------------------------------------------------------
```

In the **draw** function the integer handles *positionHandle* and *vertexTempHandle* point to the attribute variables, *vPosition*, a **vec4**, and *vertexTemp*, a **float**, defined in the vertex shader. Through these handles, the application assigns data to *vPosition* that specifies the vertex coordinates and to *vertexTemp* that specifies the temperature at each vertex. The command **glVertexAttribPointer**() tells the handle where the actual data are located, and in our case, the vertex data and the temperature data are hard-coded in the arrays *squareCoords* and *vertexTemp* respectively. The command **glDrawArrays** sends these data to the shader on a per-vertex basis. The **draw** function declares another 3 integer handles to pass the color data representing hot and cold temperatures to the fragment shader and the temperature range to the vertex shader. The commands **glUniform***() send the data.

The vertex and fragment shaders, which are saved in the files *temp.vert* and *temp.frag* respectively, are relatively simple:

```
------------------------------------------------------------------------
 // temp.vert  (temperature vertex shader)
 attribute vec4  vPosition;
 attribute float vertexTemp;
 uniform   float tempRange;
 varying   float temperature;
```

```
void main(void)
{
  temperature = ( vertexTemp - 0.0 ) / tempRange;
  gl_Position = vPosition;
}

// temp.frag  (temperature fragment shader)
uniform vec3 coldColor;
uniform vec3 hotColor;
varying float temperature;

void main()
{
  vec3 color = mix ( coldColor, hotColor, temperature );
  gl_FragColor = vec4 ( color, 1 );
}
```

The vertex shader and the fragment shader communicate via the **varying** varialbe *temperature*, which is a **float**. The vertex shader calculates *temperature* from **uniform** variables *vertexTemp* and *tempRange*, whose values are obtained from the application or from interpolation. The *temperature* value, which has a value ranges from 0.0 to 1.0 is passed to the fragment shader to evaluate a color using the glsl **mix** function that interpolates a new color from two provided colors, $color1$ and $color2$, and a fraction f:

$$mix(color1, color2, f) = color1 \times (1 - f) + color2 \times f$$

When we execute this program, we will see a color square like the one shown in Figure 19-7 below. Again, in this example no world window and viewing volume has been setup and no transformation matrix operation has been applied to the vertex values.

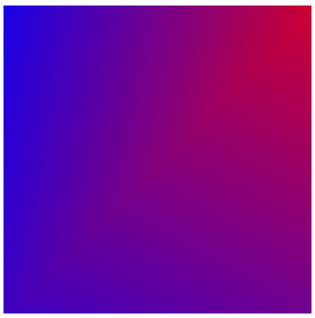


Figure 19-7 Output of Temperature Shader

19.6.3 Drawing a Wireframe Tetrahedron

A tetrahedron is composed of four triangular faces, three of which meet at each vertex and thus it has four vertices. It may be the simplest kind of 3D objects.

A tetrahedron can be considered as a pyramid, which is a polyhedron with a flat polygon base and triangular faces connecting the base to a common point. A tetrahedron simply has a triangular base, so it is also known as a **triangular pyramid**.

A **regular tetrahedron** is one in which all four faces are equilateral triangles. The vertices coordinates of a regular tetrahedron with edge length 2 centered at the origin are

$$(1, 0, \frac{-1}{\sqrt{2}}),\ \ (-1, 0, \frac{-1}{\sqrt{2}}),\ \ (0, 1, \frac{1}{\sqrt{2}}),\ \ (0, -1, \frac{1}{\sqrt{2}}) \tag{19.1}$$

The following example illustrates the basic techniques of drawing 3D objects with glsl. We will setup the viewing and projection paramters in the application and the vertex shader will multiply the model-view projection matrix to the vertex coordinates.

In the application, the class *Tetrahedron*, which is similar to classes *Triangle* and *Square* above, defines the coordinates of the four tetrahedron vertices and the indices of the four faces:

```
const float Tetrahedron::tetraCoords[] =
{
  1, 0, -0.707f,  -1, 0, -0.707f,
  0, 1,  0.707f,   0, -1, 0.707f
};

const int nIndices = 8;  // number of indices
// Order of indices of drawing the tetrahedron
const short indices[nIndices] = {0, 1, 2, 0, 3, 1, 2, 3};
```

It passes the drawing color to the fragment shader through a **uniform** variable named *vColor*. We set the drawing color to cyan and use line stripes to draw the 3D tetrahedron as a wireframe. So the *draw* function can be implemented as:

```
-------------------------------------------------------------------------
 void Tetrahedron::draw()
 {
  // get handle to vertex shader's attribute variable vPosition
  int positionHandle= glGetAttribLocation(program,"vPosition");

  // Enable a handle to the tetrahedron vertices
  glEnableVertexAttribArray( positionHandle );
  // Prepare the tetrahedron coordinate data
  glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                           GL_FLOAT, false, 0, tetraCoords);

  int colorHandle  =  glGetUniformLocation(program, "vColor");
  if (colorHandle == -1)
      printf("No such uniform named %s\n", "vColor" );
  // set drawing color
  glUniform4f ( colorHandle, 0.0, 1.0, 1.0, 1.0 );
  glLineWidth(5);
  // Draw the tetrahedron
  glDrawElements( GL_LINE_STRIP, nIndices,
                              GL_UNSIGNED_SHORT, indices);

  // Disable vertex array
  glDisableVertexAttribArray(positionHandle);
 }
-------------------------------------------------------------------------
```

In this example, the application defines the projection and model-view transformations; we can implement the **reshape** function of the *renderer* as:

```
--------------------------------------------------------------------------
 static void reshape(int width, int height)
 {
   glViewport(0, 0, width, height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt (0.0, 0.0, 6.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
 }
--------------------------------------------------------------------------
```

The vertex shader (*tetra.vert*) and the fragment shader (*tetra.frag*) are very simple:

```
--------------------------------------------------------------------------
// tetra.vert : tetrahedron vertex shader
attribute vec4  vPosition;
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * vPosition;
}

// tetra.frag :  tetrahedron fragment shader
uniform vec4 vColor;
void main()
{
  gl_FragColor = vColor;
}
--------------------------------------------------------------------------
```

The built-in glsl variable **gl_ModelViewProjectionMatrix** is the product of the projection matrix and the model-view matrix, and is multiplied to the vertex coordindates for every vertex. The shader statement is equivalent to

```
  gl_Position=gl_ProjectionMatrix*gl_ModelViewMatrix*vPosition;
```

In principle we should get the same transformation as the fixed-pipeline architecture. However, in practice the order of transforming the vertices in our shader may differ from that of the fixed functionality due to the optimization in a graphic card, that special built-in functions may take advantage of the optimization to speed up calculations. Also, rounding errors may generate different results for different methods of calculations. Therefore, **glsl** provides a function that guarantees the result to be the same as when using the fixed functionality:

vec4 **ftransform** (void);

This function does the matrix calculations in the order as that of the fixed functionality and produces the same results.

Alternatively, we can pass in the transformation matrices from the application as **mat4 uniform** variables rather than using the built-in variables.

The color of drawing is set by the variable *vColor* in the fragment shader. Note that if we declare a variable in a shader and do not actually use it in the shader program, the variable will be removed in the compilation process for optimiazation purposes. Consequently, the application will not be able to find the variable even though it has been declared in the shader.

Figure 19-8 below shows the output of this program.

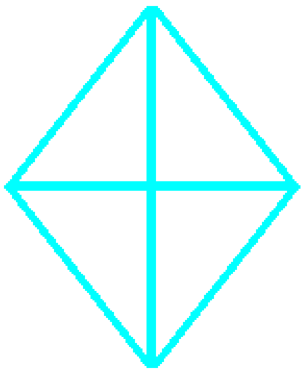


Figure 19-8 Output of Tetrahedron Shader

19.6.4 Drawing a Color Solid Tetrahedron

In this example, we draw a solid 3D tetrahedron, specifying the data inside a class named *Tetrahedrons*. We draw each face independently with a different color. Therefore, we have to specify the vertex indices of each face and the associated colors like the following:

```
//  Coordinates of a tetrahedron
const float Tetrahedrons::tetraCoords[] =
{
   1.0f, 0.0f, -0.707f, // vertex v0
  -1.0f, 0.0f, -0.707f, // v1
   0.0f, 1.0f,  0.707f, // v2
   0.0f, -1.0f, 0.707f  // v3
};

// draw indices for each face
const short Tetrahedrons::drawOrders[][3] = {
  {0, 1, 2},  {0, 2, 3},  {0, 3, 1},  {3, 2, 1}
};

// color for each face
const float Tetrahedrons::colors[][4] = {
  {1.0f, 0.0f, 0.0f, 1.0f},   // 0,1, 2 red
  {0.0f, 1.0f, 0.0f, 1.0f},   // 0, 2, 3 green
  {0.0f, 0.0f, 1.0f, 1.0f},   // 0, 3, 1 blue
  {1.0f, 1.0f, 0.0f, 1.0f}    // 3, 2, 1 yellow
};
```

As a demonstration of the alternative method of transforming vertices, we pass in a 4×4 transformation matrix from the application to the vertex shader instead of using the built-in variable *gl_ModelViewProjectionMatrix* like what we did above.

In this example, we rotate the tetrahedron about an axis by dragging the mouse and perform our own matrix operations. Alternatively, one may use the OpenGL Mathematics (glm), which is a C++ mathematics library for 3D software based on the glsl specification to do matrix operations. The package can be downloaded from:

http://sourceforge.net/projects/ogl-math/

But for simplicity, we have not used this math library in our examples. We calculate the composite transformation matrix in our display call-back function:

```
Tetrahedrons *tetrahedron;
float mvMatrix[4][4];   //model-view matrix
float mvpMatrix[4][4];  //model-view projection matrix
float angle = 0;        //rotation angle
.....
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor( 1.0, 1.0, 1.0, 1.0 );  //set white background
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt (0.0, 0.0, 6.3, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glRotatef ( angle, 1.0f, 0.2f, 0.2f );
  // retrieve model-view matrix
  glGetFloatv(GL_MODELVIEW_MATRIX, &mvMatrix[0][0]);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
  // multiply projection matrix by model-view matrix
  glMultMatrixf ( &mvMatrix[0][0] );
  // retrieve model-view projection matrix
  glGetFloatv(GL_PROJECTION_MATRIX, &mvpMatrix[0][0]);
  // pass transformation matrix to vertex shader
  tetrahedron->draw( mvpMatrix );
  tetrahedron->cleanUp();
  glutSwapBuffers();
  glFlush();
}
```

The code uses the command **glGetFloatv**() to obtain the 4×4 model-view matrix *mvMatrix*. It then multiplies it to the projection matrix, and the product is retrieved into the model-view projection matrix *mvpMatrix*, which is an input parameter to the **draw**() function of the class *Tetrahedrons*, where it passes it to the vertex shader. The code also rotates the object by *angle* degrees around the axis $(1.0, 0.2, 0.2)$. The rotation angle is changed by the mouse-movement callback function, **movedMouse**():

```
const int screenWidth = 200;
const int screenHeight = 200;
float previousX=0,  previousY=0, dx = 0, dy = 0;
void movedMouse(int mouseX, int mouseY)
{
  dx = mouseX - previousX;
  dy = mouseY - previousY;
  // reverse direction of rotation above the mid-line
  if (mouseY > screenHeight / 2)
    dx = dx * -1 ;
  // reverse direction of rotation to left of the mid-line
  if (mouseX < screenWidth / 2)
    dy = dy * -1 ;
  angle = angle + (dx + dy) / 2;  //scale factor of 2
  previousX = mouseX;
  previousY = mouseY;
  glutPostRedisplay();
}
```

The following is the **draw**() function with some non-significant code omitted. The main thing new here is that it passes the 4×4 model-view projection matrix to the vertex shader via the **uniform** variable *mvpMatrix* declared in the vertex shader.

```
void Tetrahedrons::draw( float mvpMatrix[4][4] )
{
  int positionHandle = glGetAttribLocation(program,"vPosition");
  glEnableVertexAttribArray( positionHandle );
  glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                        GL_FLOAT, false, 0, tetraCoords);
  int mvpMatrixHandle = glGetUniformLocation(program, "mvpMatrix");
  glUniformMatrix4fv(mvpMatrixHandle, 1, GL_FALSE, &mvpMatrix[0][0] );
  int colorHandle  =  glGetUniformLocation(program, "vColor");
  glEnable (GL_CULL_FACE);
  glCullFace(GL_BACK);
  for ( int i = 0; i < N_FACES; i++ ) {
    glUniform4fv(colorHandle, 1, &colors[i][0]);
    glDrawElements( GL_TRIANGLES, 3,
                         GL_UNSIGNED_SHORT, &drawOrders[i][0]);
  }
}
```

The following is the vertex shader, which mulitplies the model-view projection matrix, a **mat4** obtained from the application to the vertex coordinates:

```
// tetra.vert : tetrahedron vertex shader
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;
void main(void) {
  gl_Position = mvpMatrix * vPosition;
}
```

The fragment shader does nothing special. It simply sets the color at a fragment to the corresponding vertex color obtained from the application:

```
// tetra.frag : tetrahedron fragment shader
uniform vec4 vColor;
void main() {
  gl_FragColor = vColor;
}
```

Figure 19-9 below shows a few sample outputs of this program when the mouse is dragged to rotate the tetrahedron.

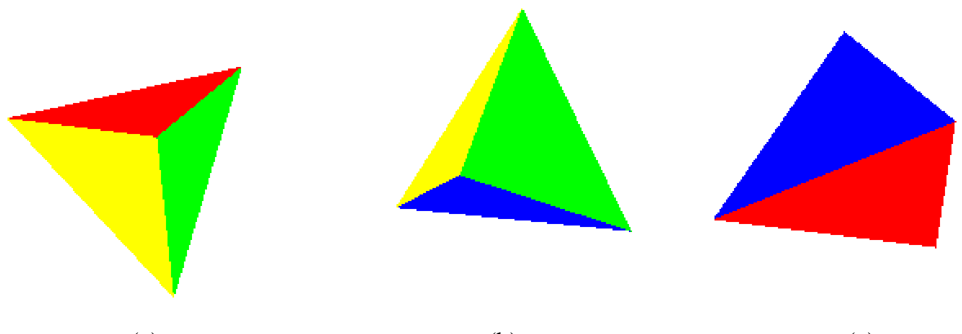


Figure 19-9 Sample Outputs Color Tetrahedron Shader

19.7 Drawing Spheres

19.7.1 Spherical Coordinates

We can use a mesh of triangles to approximate a spherical surface. We have to define the vertices coordinates of each triangle and every vertex is on the surface of the sphere. In practice, it is easier to calculate the position of a point on a sphere using spherical coordinates, where a point is specified by three numbers: the radial distance r of that point from a fixed origin, its polar angle θ (also called inclination) measured from a fixed zenith direction, and the azimuth angle ϕ of its orthogonal projection on a reference plane that passes through the origin as shown in Figure 19-10. So in spherical coordinates, a point is defined by (r, θ, ϕ) with some restrictions:

$$\begin{aligned} &r \geq 0 \\ &0^o \leq \theta \leq 180^o \\ &0^o \leq \phi < 360^o \end{aligned} \tag{19.2}$$

Cartesian coordinates of a point (x, y, z) can be calculated from the spherical coordinates, (radius r, inclination θ, azimuth ϕ), where $r \in [0, \infty)$, $\theta \in [0, \pi]$, $\phi \in [0, 2\pi)$, by:

$$\begin{aligned} &x = r \sin\theta \cos\phi \\ &y = r \sin\theta \sin\phi \\ &z = r \cos\theta \end{aligned} \tag{19.3}$$

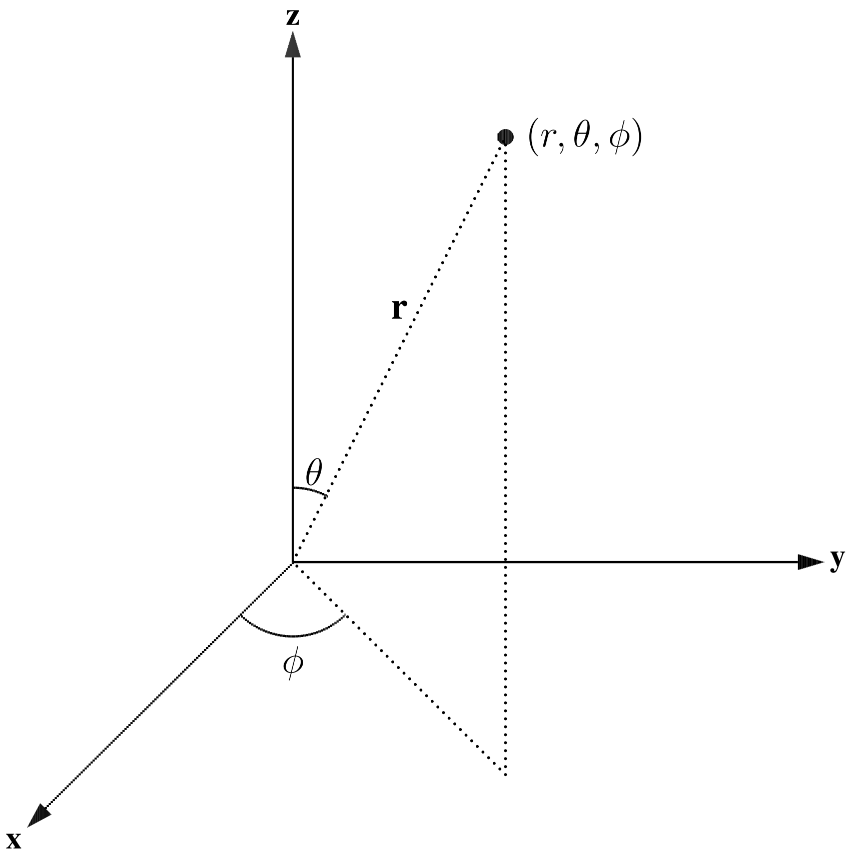


Figure 19-10 Spherical Coordinate System

Conversely, the spherical coordinates can be obtained from Cartesean coordinates by:

$$\begin{aligned} r &= \sqrt{x^2+y^2+z^2} \\ \theta &= \cos^{-1}\left(\frac{z}{r}\right) \\ \phi &= \tan^{-1}\left(\frac{y}{x}\right) \end{aligned} \tag{19.4}$$

19.7.2 Rendering a Wireframe Sphere

To render a sphere centered at the origin, we can divide the sphere into slices around the z-axis (similar to lines of longitude), and stacks along the z-axis (similar to lines of latitude). We simply draw the slices and stacks independently, which will form a sphere. Each slice or stack is formed by line segments joining points together. Conversely, each point is an intersection of a slice and a stack.

Suppose we want to divide the sphere into m stacks and n slices. Since $0 \leq \theta \leq \pi$, the angle between two stacks is $\pi/(m-1)$. On the other hand, $0 \leq \phi < 2\pi$, the angle between two slices is $2\pi/n$ as the angle 2π is not included. That is,

$$\begin{aligned} \delta\theta &= \frac{\pi}{m-1} \\ \delta\phi &= \frac{2\pi}{n} \end{aligned} \tag{19.5}$$

Figure 9-11 below shows a portion of two slices and two stacks, and their intersection points.

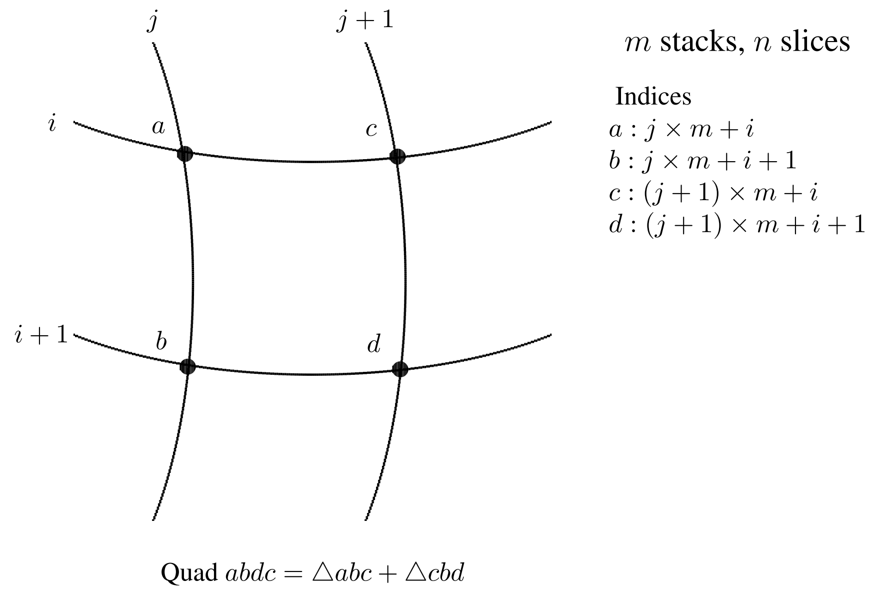


Figure 9 - 11. Spherical Surface Formed by Stacks and Slices

Our task is to calculate the intersection points. Suppose we calculate the points along a slice starting from $\phi = 0$, spanning θ from 0 to π, and then incrementing ϕ to calculate the next slice. We apply equation (19.3) to calculate the x, y, and z coordinates of each point. For convenience, we define a class called *XYZ* that contains the x, y, z coordinates of a point, and save the points in a C++ Standard Template Library (STL) class *vector* called *vertices*. Suppose we have declared a class called *Sphere*, which is similar to the *Tetrahedron* class discussed above. The following code, where **createSphere** can be a member function of the class *Sphere*, shows an implementation of such a task, assuming that r is the radius of the sphere:

```
class XYZ{
public:
  float x, y, z;
};

void Sphere::createSphere ( float r, int nSlices, int nStacks )
{
   double phi,  theta;
   XYZ *p = new XYZ();
   const double PI = 3.1415926;
   const double TWOPI = 2 * PI;

   for ( int j = 0; j < nSlices; j++ ) {
     phi = j * TWOPI / nSlices;
     for ( int i = 0; i < nStacks; i++ ) {
       theta = i * PI / (nStacks-1);  //0 to pi
       p->x = r * (float) ( sin ( theta ) *  cos ( phi ));
       p->y = r * (float) ( sin ( theta ) *  sin ( phi ));
       p->z = r * (float)  cos ( theta );
       vertices.push_back ( *p  );
     }
   }
}
```

In the code, the **push_back**() function of *vector* simply inserts an item into the *vector* object at its back. So the *vector vertices* contains the coordinates of all the points on the sphere.

Now we have obtained all the intersection points. The remaining task is to define a draw order list that tells us how to connect the points. We use a **short** array, named *drawOrderw* to hold the indices of the vertices in the order we want to connect them. Suppose we first draw the slices. The following code shows how to calculate the indices for the points of the slices:

```
int k = 0;
for ( int j = 0; j < n; j++ ) {
 for ( int i = 0; i < m-1; i++ ) {
   drawOrderw[k++] = (short) (j * m + i);
   drawOrderw[k++] = (short)( j* m + i + 1 );
 }
}
```

The two indices $(j * m + i)$ and $(j * m + i + 1)$ define two points of a line segment of a slice. Each slice is composed of $m - 1$ line segments. The following code shows the calculations for the stacks:

```
for ( int i = 1; i < m - 1; i++) {
  for ( int j = 0; j < n; j++){
    drawOrderw[k++] = (short) (j * m + i);
    if ( j == n - 1)  //wrap around: j + 1 --> 0
      drawOrderw[k++] = (short) ( i);
    else
      drawOrderw[k++] = (short) ((j+1)*m + i);
  }
}
```

Each pair of indices defines two end points of a line segment of a stack. When j equals $n - 1$, the next point wraps around so that the last point of the stack joins its first point to form a full circle. So each stack is composed of n segments. Also we do not need to draw the poles, and there are only $m - 2$ stacks. Therefore, the total number of indices in *drawOrderw* is

$$2 \times n \times (m - 1) + 2 \times (m - 2) \times n = 4 \times m \times n - 6 \times n$$

So we can calculate the *drawOrderw* array size and allocate the approximate memory for the array before using it:

```
short *drawOrderw;
int drawOrderwSize = 4 * m * n - 6 * n;
drawOrderw = new short[drawOrderwSize];
```

In order to use the OpenGL command **glDrawElements**, we put all the vertex coordinates in a float array called *sphereCoords*:

```
int nVertices = vertices.size();
float *sphereCoords = new float[3*nVertices];
int k = 0;
for ( int i = 0; i < nVertices; i++ ) {
  XYZ v = vertices[i];
  sphereCoords[k++] = v.x;
  sphereCoords[k++] = v.y;
  sphereCoords[k++] = v.z;
}
```

(The tasks of calculating the *drawOrderw* and *sphereCoords* arrays can be done in the constructor of *Sphere*, after calling the member function **createSphere**.)

To draw the sphere, we simply join these points together. When we draw all the slices and stacks, we get a wireframe sphere:

```
void Sphere::draw( float mvpMatrix[4][4] )
{
   int positionHandle= glGetAttribLocation(program,"vPosition");
   glEnableVertexAttribArray( positionHandle );
   glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                         GL_FLOAT, false, 0, sphereCoords);
   int mvpMatrixHandle = glGetUniformLocation(program, "mvpMatrix");
   glUniformMatrix4fv(mvpMatrixHandle, 1, GL_FALSE, &mvpMatrix[0][0] );
   ......
   glDrawElements(GL_LINES,drawOrderwSize,GL_UNSIGNED_SHORT,drawOrderw);
   ......
}
```

Since the first index in *drawOrderw* references the point that is the north pole of the sphere, we can draw the pole using the statement:

```
glDrawElements(GL_POINTS,1,GL_UNSIGNED_SHORT, drawOrderw);
```

The shaders are very simple and similar to those described in previous examples:

```
// sphere.vert: Source code of vertex shader
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;
void main(void)
{
  gl_Position = mvpMatrix * vPosition;
}
```

```
// sphere.frag: Source code of fragment shader
uniform vec4 vColor;
void main()
{
  gl_FragColor = vColor;
}
```

Suppose we set the number of slices to 24 and the number of stacks to 16. When we run the program, we will see a wireframe sphere like the one shown in Figure 9-12 below. The point near the top of the sphere is its north pole. It has been rotated by the mouse dragging movement.

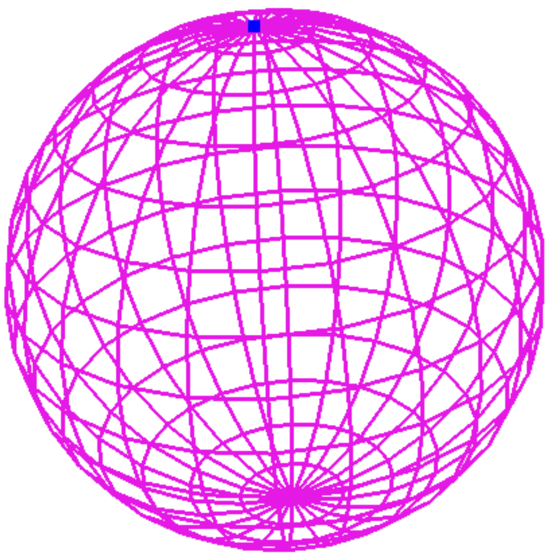


Figure 9-12 A Rendered Wireframe Sphere

19.7.3 Rendering a Color Solid Sphere

Rendering a color solid sphere is similar to rendering a color solid tetrahedron except that we have to calculate the vertices of the triangle mesh. We have already learned how to decompose a sphere into slices and stacks in the previous section. Suppose we have saved all the vertices in the *vector* variable *vertices* as we did in the previous example. The shaders are the same as those in the previous example, which are very simple.

As shown in Figure 9-11, the intersection points of two latitudes and two longitudes form a quadrilateral, which can be decomposed into two triangles. Since all the vertices coordinates have been calculated, we just need to find out the order of drawing them in the form of triangles.

As shown in the figure, to draw the quad abcd, we first draw the triangle abc and then draw the other triangle cbd, both in a counter-clockwise direction. That means the drawing order of the vertices is a, b, c, c, b, d. The following code shows the implementation of this procedure:

```
// 2n(m-1) slices + 2(m-2)n stacks
int nTriangles = 2 * n * (m - 1);    //number of triangles
short *drawOrders = new short[3*nTriangles];
for ( int j = 0; j < n; j++ )
```

```
{
   for ( int i = 0; i < m-1; i++ ) {
     short j1 = (short)(j + 1);
     if ( j == n - 1 ) j1 = 0;   //wrap around
     short ia = (short)( j * m + i ) ;
     short ib = (short)( j * m + i + 1);
     short ic = (short) (j1 * m + i );
     short id = (short)( j1 * m + i + 1 );
     drawOrders[k++] = ia;
     drawOrders[k++] = ib;
     drawOrders[k++] = ic;

     drawOrders[k++] = ic;
     drawOrders[k++] = ib;
     drawOrders[k++] = id;
   }
}
```

Suppose we just use the four colors, red, green, blue, and yellow to draw the whole sphere, alternating the colors between adjacent triangles. We can define a **float** array to hold the four colors:

```
const float Spheres::colors[][nColors] = {
  {1.0f, 0.0f, 0.0f, 1.0f},   // red
  {0.0f, 1.0f, 0.0f, 1.0f},   // green
  {0.0f, 0.0f, 1.0f, 1.0f},   // blue
  {1.0f, 1.0f, 0.0f, 1.0f}    // yellow
};
```

To draw the sphere, we simply draw all the triangles, each of which is defined by three vertices and a color. In this example, only four colors, red, green, blue, and yellow have been used for coloring the triangles. The following is the **draw** function of the class *Sphere* that draws a color solid sphere composed of triangles; the user can rotate the sphere by dragging the mouse:

```
void Spheres::draw( float mvpMatrix[4][4] )
{
  int positionHandle= glGetAttribLocation(program,"vPosition");
  glEnableVertexAttribArray( positionHandle );
  glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                             GL_FLOAT, false, 0, sphereCoords);

  int mvpMatrixHandle = glGetUniformLocation(program,"mvpMatrix");
  glUniformMatrix4fv(mvpMatrixHandle,1,GL_FALSE,&mvpMatrix[0][0]);

  int colorHandle =  glGetUniformLocation(program, "vColor");
  for ( int i = 0; i < nTriangles; i++ ) {
    int j = i % 4;
    glUniform4fv(colorHandle, 1, colors[j]);

    glDrawElements( GL_TRIANGLES, 3,
                           GL_UNSIGNED_SHORT, (drawOrders + i*3 ));
  }
  glDisableVertexAttribArray(positionHandle);
}
```

In the code, the array pointer argument of the OpenGL function **glDrawElements** is

*drawOrders + i * 3*

which points to the three indices of the vertices of the *i-th* triangle to be drawn. For instance, if the first index value is *j* (i.e. *j = *(drawOrders+i*3)*), then the coordinates of the triangle vertices are given by

sphereCoords[j], sphereCoords[j+1], sphereCoords[j+2]

These values are passed to the vertex shader via the attribute variable *vPosition*.

The vertex and fragment shaders are the same as those presented in the previous section, *Rendering a Wireframe Sphere*. When we run the program, we will see an output similar to the one shown in Figure 19-13 below, where the sphere has been rotated by the mouse.

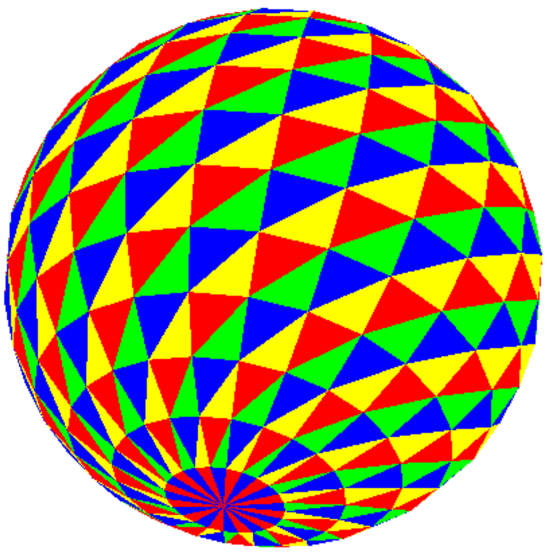


Figure 19-13 A Rendered Color Solid Sphere

19.7.4 Lighting a Sphere

Lighting is an important feature in graphics for making a scene appear more realistic and more understandable. It provides crucial visual cues about the curvature and orientation of surfaces, and helps viewers perceive a graphics scene having three-dimensionality. Using the sphere we have constructed in previous sections, we discuss briefly here how to add lighting effect to it.

To create lighting effect that looks realistic, we need to first design a lighting model. In graphics, however, such a lighting model does not need to follow physical laws though the laws can be used as guidelines. The model is usually designed empirically. In our discussion, we more or less follow the simple and popular Phong lighting model that we discussed in Chapter 7 to create lighting effect. In the model we only consider the effects of a light source shining directly on a surface and then being reflected directly to the viewpoint; second bounces are ignored. Such a model is referred to as a local lighting model, which only considers the light property and direction, the viewer's position, and the object material properties. It considers only the first bounce of the light ray but ignores any secondary reflections, which are light rays that are reflected for more than once by surfaces before reaching the viewpoint. Nor does a basic local model consider shadows created by light. In the model, we consider the following features:

1. All light sources are modeled as point light sources.

2. Light is composed of red (R), green (G), and blue (B) colors.
3. Light reflection intensities can be calculated independently using the principle of superposition for each light source and for each of the 3 color components (R, G, B). Therefore, we describe a source through a three-component intensity or illumination vector

$$\mathbf{I} = \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{19.6}$$

   Each of the components of **I** in (19.6) is the intensity of the independent red, green, and blue components.
4. There are three distinct kinds of light or illumination that contribute to the computation of the final illumination of an object:
   - **Ambient Light**: light that arrives equally from all directions. We use this to model the kind of light that has been scattered so much by its environment that we cannot tell its original source direction. Therefore, ambient light shines uniformly on a surface regardless of its orientation. The position of an ambient light source is meaningless.
   - **Diffuse Light**: light from a point source that will be reflected diffusely. We use this to model the kind of light that is reflected evenly in all directions away from the surface. (Of course, in reality this depends on the surface, not the light itself. As we mentioned earlier, this model is not based on real physics but on graphical experience.)
   - **Specular Light**: light from a point source that will be reflected specularly. We use this to model the kind of light that is reflected in a mirror-like fashion, the way that a light ray reflected from a shinny surface.
5. The model also assigns each surface material properties, which can be one of the four kinds:
   - Materials with **ambient reflection properties** reflect ambient light.
   - Materials with **diffuse reflection properties** reflect diffuse light.
   - Materials with **specular reflection properties** reflect specular light.

In the model, ambient light only interacts with materials that possess ambient property; specular and diffuse light only interact with specular and diffuse materials respectively.

Figure 19-14 below shows the vectors that are needed to calculate the illumination at a point. In the figure, the labels *vPosition*, *lightPosition*, and *eyePosition* denote points at the vertex, the light source, and the viewing position respectively. The labels **L, N, R**, and **V** are vectors derived from these points (recall that the difference between two points is a vector), representing the light vector, the normal, the reflection vector, and the viewing vector respectively. The reflection vector **R** is the direction along which a light from **L** will be reflected if the the surface at the point is mirror-like. Assuming that the center of the sphere is at the origin $O = (0, 0, 0)$, some of them can be expressed as

$$\begin{aligned} \text{light vector } \mathbf{L} &= lightPosition - vPosition \\ \text{normal } \mathbf{N} &= vPosition - O \\ \text{view vector } \mathbf{V} &= eyePosition - vPosition \end{aligned} \tag{19.7}$$

We can normalize a vector by dividing it by its magnitude:

$$\mathbf{l} = \frac{\mathbf{L}}{|\mathbf{L}|}, \quad \mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|}, \quad \mathbf{v} = \frac{\mathbf{V}}{|\mathbf{V}|}, \quad \mathbf{r} = \frac{\mathbf{R}}{|\mathbf{R}|} \tag{19.8}$$

One can easily show that the normalized reflection vector **r** can be calculated from **l** and **n** by the formula,

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l} \tag{19.9}$$

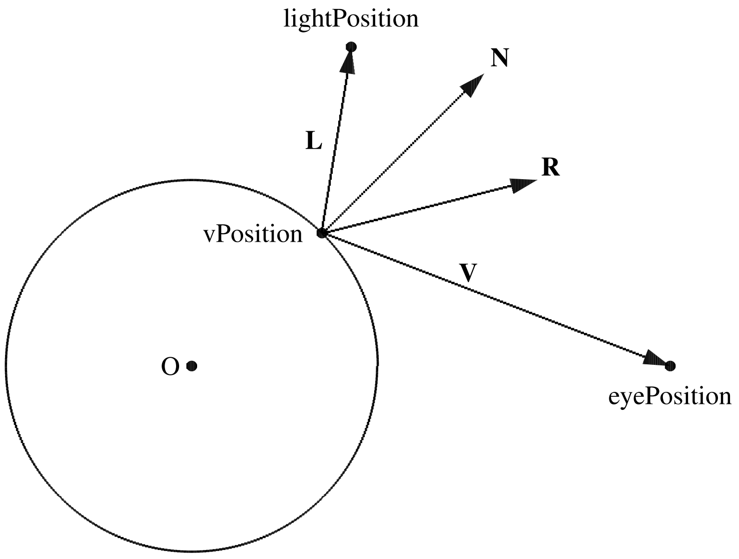


Figure 19-14 Lighting Vectors

Suppose I^{in} denotes the incident illumination from the light source in the direction **l**. The ambient, diffuse, and specular illumination on the point *vPosition* can be calculated according to the following formulas.

The ambient illumination is given by

$$I_a = c_a I_a^{in} \tag{19.10}$$

where I_a^{in} is the incident ambient light intensity and c_a is a constant called the *ambient reflectivity coefficient.*

The diffuse illumination is

$$I_d = c_d I_d^{in} \mathbf{l} \cdot \mathbf{n} \tag{19.11}$$

where $\mathbf{l} \cdot \mathbf{n} = \mathbf{r} \cdot \mathbf{n}$, I_d^{in} is the incident diffuse light intensity and c_d is a constant called the *diffuse reflectivity coefficient.*

The specular illumination can be calculated by

$$I_s \quad = c_s I_s^{in} (\mathbf{r} \cdot \mathbf{v})^f \tag{19.12}$$

where c_s is a constant called the *specular reflectivity coefficient* and the exponent f is a value that can be adjusted empirically on an ad hoc basis to achieve desired lighting effect. The exponent f is ≥ 0, and values in the range 50 to 100 are typically used for shinny surfaces. The larger the exponent factor f, the narrower the beam of specularly reflected light becomes.

The total illumination is the sum of all the above components:

$$\begin{aligned} I &= I_a + I_d + I_s \\ &= c_a I_a^{in} + c_d I_d^{in} (\mathbf{l} \cdot \mathbf{n}) + c_s I_s^{in} (\mathbf{r} \cdot \mathbf{v})^f \end{aligned} \tag{19.13}$$

This model can be easily implemented in the glsl shader language. In our example, where the illuminated object is a sphere, the shader code is further simplified. The positions of the light source, the vertex, and the eye position (viewing point) are passed from the the application to the vertex shader as **uniform** variables. The vertex shader calculates the vectors **L**, **N**, and **V** from the positions and pass them to the fragment shader as **varying** variables:

```
// Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
uniform vec4 eyePosition;
uniform vec4 lightPosition;
varying vec3 N; //normal direction
varying vec3 L; //light source direction
varying vec3 V; //view vector
void main() {
   gl_Position = mvpMatrix * vPosition;
   N = vPosition.xyz;   //normal of  a point on sphere
   L = lightPosition.xyz - vPosition.xyz;
   V = eyePosition.xyz - vPosition.xyz;
}
```

In the code, we have used the **swizzling** operator to access the x, y, and z components of the **vec4** and **vec3** variables. In glsl, a swizzling operator is a variant of the C selection operator (.). It allows us to read and write multiple components of the matrix and vector variables. For example,

N = vPosition.xyz;

is to assign the x, y, and z components of *vPosition* to the corresponding components of *N*. (We may even use it to swap elements like *a.xy = a.yx*.)

The fragment shader obtains the vectors **L**, **N**, and **V** from the vertex shader, normalizes them, and calculates the reflection vector **r**. It then uses formulas (19.10) to (19.12) to calculate the illumination at the vertex. The following is the fragment shader code that calculates the illumination at each pixel:

```
// Source code of fragment shader
varying vec3 N;
varying vec3 L;
varying vec3 V;
uniform vec4 lightAmbient;
uniform vec4 lightDiffuse;
uniform vec4 lightSpecular;
//in this example, material color same for ambient, diffuse, specular
uniform vec4 materialColor;
uniform float shininess;

void main() {
  vec3 norm = normalize(N);
  vec3 lightv = normalize(L);
  vec3 viewv = normalize(V);
  // diffuse coefficient
  float Kd = max(0.0, dot(lightv, norm));

  // calculating specular coefficient
  // consider only specular light in same direction as normal
  float cs;
  if(dot(lightv, norm)>= 0.0) cs =1.0;
  else cs = 0.0;
  //reflection vector
  vec3 r = 2.0 *  dot (norm, lightv) * norm  - lightv;
  float Ks = pow(max(0.0, dot(r, viewv)), shininess);
  vec4 ambient = materialColor * lightAmbient;
  vec4 specular = cs * Ks * materialColor *lightSpecular;
  vec4 diffuse = Kd * materialColor *  lightDiffuse;
```

```
    gl_FragColor = ambient + diffuse + specular;
  }
```

One can modify the code or juggle with it to obtain various lighting effects empirically.

Similar to previous examples, the OpenGL application has to provide the actual values of the **uniform** and **attribute** parameters. The sphere is constructed in the same way that we did in the previous example. However, we do not need to pass in the colors for each triangle as the appearance of the sphere is now determined by its material color and the light colors, and the color at each pixel is calculated by the fragment shader using the lighting model. Suppose we call the class of the drawn sphere *Spherel*. We define the light and material properties as follows:

```
  const float Spherel::eyePos[] = {0, 0, 6.3, 1};
  const float Spherel::lightPos[] = {5, 10, 5, 1};
  const float Spherel::lightAmbi[] = {0.1, 0.1, 0.1, 1};
  const float Spherel::lightDiff[] = {1, 0.8, 0.6, 1};
  const float Spherel::lightSpec[] = {0.3, 0.2, 0.1, 1};
  //material same for ambient, diffuse, and specular
  const float Spherel::materialColor[] = {1, 1, 1, 1};
```

The **draw** function of the class passes the variable values to the shaders and draws the lit sphere:

```
------------------------------------------------------------------------
 void Spherel::draw( float mvpMatrix[4][4] )
 {
   int positionHandle= glGetAttribLocation(program,"vPosition");
   glEnableVertexAttribArray( positionHandle );
   glVertexAttribPointer(positionHandle, COORDS_PER_VERTEX,
                          GL_FLOAT, false, 0, sphereCoords);
  int mvpMatrixHandle = glGetUniformLocation(program, "mvpMatrix");
  glUniformMatrix4fv(mvpMatrixHandle,1,GL_FALSE,&mvpMatrix[0][0]);
  glUniformMatrix4fv(mvpMatrixHandle,1,GL_FALSE,&mvpMatrix[0][0]);
  int eyePosHandle = glGetUniformLocation(program, "eyePosition");
  int lightPosHandle=glGetUniformLocation(program,"lightPosition");
  int lightAmbiHandle=glGetUniformLocation(program,"lightAmbient");
  int lightDiffHandle=glGetUniformLocation(program,"lightDiffuse");
  int lightSpecHandle=glGetUniformLocation(program,"lightSpecular");
  int materialColorHandle=glGetUniformLocation(program,"materialColor");
  int shininessHandle =  glGetUniformLocation(program, "shininess");
  glUniform4fv(eyePosHandle, 1, eyePos);
  glUniform4fv(lightPosHandle, 1, lightPos);
  glUniform4fv(lightAmbiHandle, 1, lightAmbi);
  glUniform4fv(lightDiffHandle, 1, lightDiff);
  glUniform4fv(lightSpecHandle, 1, lightSpec);
  glUniform1f(shininessHandle, shininess);
  glUniform4fv(materialColorHandle, 1, materialColor);

  // Draw all triangles
  for ( int i = 0; i < nTriangles; i++ ) {
    glDrawElements( GL_TRIANGLES, 3,
                          GL_UNSIGNED_SHORT, (drawOrders + i*3 ));
  }
  glDisableVertexAttribArray(positionHandle);
 }
------------------------------------------------------------------------
```

Figure 19-15 below shows an output of this application, where the same sphere of Figure 19-13 has been used. Note that in this example, the light source is rotated together with the sphere. That is, the light source position relative to the sphere is fixed.

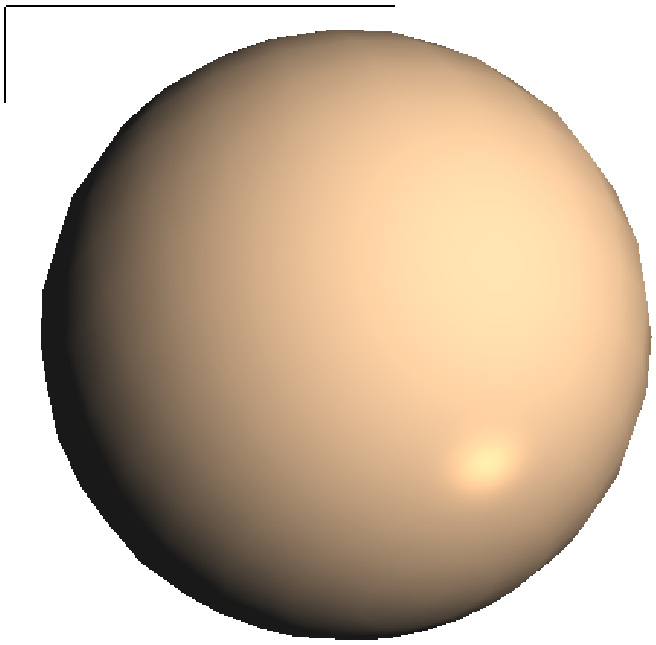


Figure 19-15 Example of Rendered Lit Sphere

We can obtain similar outputs if we draw the sphere using the functions **glutSolidSphere** or **gluSphere** instead of drawing the triangles (i.e. replacing the for-loop of **glDrawElements** by **glutSolidSphere** or **gluSphere**) in the above code. In our example, we calculate the normal of the object at a vertex in the vertex shader. This could be inconvenient if we need to render several different kinds of objects using the same shader program. In general we calculate the normal in the application and specify it using the command **glNormal***(); the normal is passed to a shader via the the built-in variable *gl_Normal*, which has been transformed by the model-view matrix. To recover the normal value in the original world coordinates, we need to perform an inverse transformation on *gl_Normal* by multiplying it by the ineverse of the model-view matrix, which is gl_NormalMatrix:

gl_NormalMatrix * gl_Normal ;

19.8 Animation

We can animate objects using glsl by changing their vertex positions at specified time intervals, which means that we need a variable to keep track of time. However, a vertex shader does not have built-in features to keep track of elapsed time. Therefore, we have to define a time variable in the OpenGL application, and pass its value to the shader via a uniform variable. We can calculate the lapsed time of running the program in the **idle** callback function of the application. We first consider an example of animating the color sphere presented above.

19.8.1 Animating a Color Sphere

In this example, we animate the color solid sphere discussed in Section 19.7.3. We need to first include in the **main** function of the application the idle function callback command

glutIdleFunc (idle);

which sets the global idle callback to be the function **idle** so that a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. The idle callback is continuously called when there are no events. So we make use of this function to find the elapsed time for animation. We define in the renderer the idle function as follows (with some minor details omitted):

```
void idle(void) {
  float t = glutGet ( GLUT_ELAPSED_TIME );
  sphere->setTime ( t );
  glutPostRedisplay();
}
```

In this function, the command **glutGet (GLUT_ELAPSED_TIME)** returns the number of milliseconds since **glutInit** was called (or first call to **glutGet(GLUT_ELAPSED_TIME)**). The function **setTime** is a new function that we add to the class *Spheres* discussed above. It passes the elapsed time to the vertex shader using the function **glUniform1f**:

```
void Spheres::setTime ( float t ) {
  glUniform1f ( timeHandle, t );   // send the lapsed time to vertex shader
}
```

The variable *timeHandle* is a data member of the class *Spheres* and its value is set in the **draw** function:

```
void Spheres::draw( float mvpMatrix[4][4] ) {
  .....
  timeHandle =  glGetUniformLocation(program, "timeLapsed");
  .....
}
```

The variable *timeLapsed* is defined in the vertex shader for animating the sphere:

```
// sphere.vert
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;
uniform  float  timeLapsed;   // time in milliseconds
void main(void)
{
  float t = timeLapsed / 1000.0;  // time t in seconds
  float s = sin ( t );     // s varies between -1 and 1
  mat4 m = mat4( 1.0 );    // 4 x 4 identity matrix
  m = s * m;               // try to create scale matrix
  m[3][3] = 1.0;           // set last matrix element to 1
                           //  to form a proper scale matrix
  gl_Position = m * mvpMatrix * vPosition;
}
```

The vertex shader simply receives the time parameter *timeLapsed* value from the application and divides it by 1000 to convert it to seconds. It then makes use of the **sin** function to obtain the value s, which varies between -1 and 1. The s value is multiplied to an identity matrix to obtain the scaling matrix m, which is multiplied to the model-view projection matrix. Therefore, the vertex values will change sign and vary. This gives us an animated sphere, which shrinks and expands repetitively, and the poles may flip over depending on the orientation of the sphere. The fragment shader is the same as before. Figure 19-16 below shows a few frames of the output of the program.

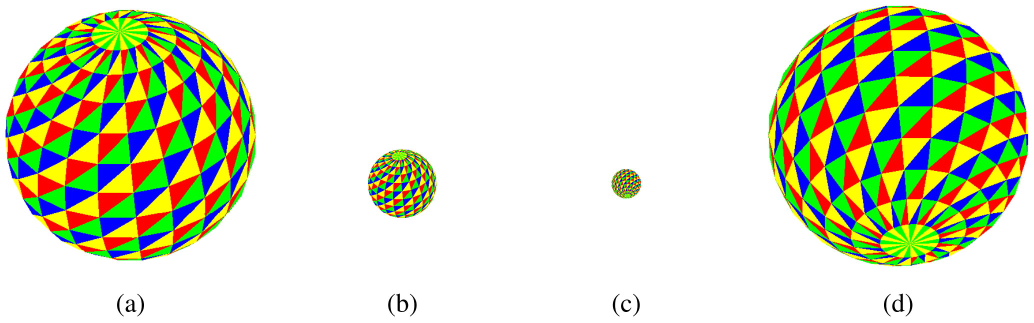


Figure 19-16 Four Frames of Animated Sphere

Note that the constructor **mat4** *(c)* constructs a 4×4 array, setting the diagonal elements to the parameter *c* and other elements to 0. Thus **mat4** (1.0) creates an identity matrix.

In the above discussion, we have used a default frame rate, which depends on how the OpenGL library calls the **idle** function. If we need to animate the sphere at a specified frame rate, we can declare a **float** variable called *previousTime* that records the time at which the most recent frame was rendered. We render a new frame only if the time lapsed has exceeded the specified period between two consecutive frames. For example, if we want the frame rate to be 5 frames per second, the period is 100 ms and we can modify the code of the **idle** function to:

```
void idle(void)
{
  float t = glutGet ( GLUT_ELAPSED_TIME );
  if ( t - previousTime  > 100.0 ){
    sphere->setTime ( t );
    previousTime = t;
    glutPostRedisplay();
  }
}
```

In the above example, the center of the sphere is fixed. If we need to move the sphere in space as a solid object, we can set the translational matrix according to the desired object movements. Recall that the transformation matrix of translation is given by:

$$T = \begin{pmatrix} 1 & 0 & 0 & \mathbf{d}_x \\ 0 & 1 & 0 & \mathbf{d}_y \\ 0 & 0 & 1 & \mathbf{d}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19.14}$$

This matrix translates an object by the values d_x, d_y, and d_z in the x, y, and z directions respectively. Therefore, to translate an object, we just need to set the appropriate entries of the transformation matrix m[][] in the vertex shader to the translation values. Since in OpenGL, a matrix is column-major, where m[i][j] is the element of the i-th column and the j-th row. That is, m[3] represents column 3 of the matrix. So we set the entries in the following way:

$$\text{m[3][0]} = d_x, \quad \text{m[3][1]} = d_y, \quad \text{m[3][2]} = d_z$$

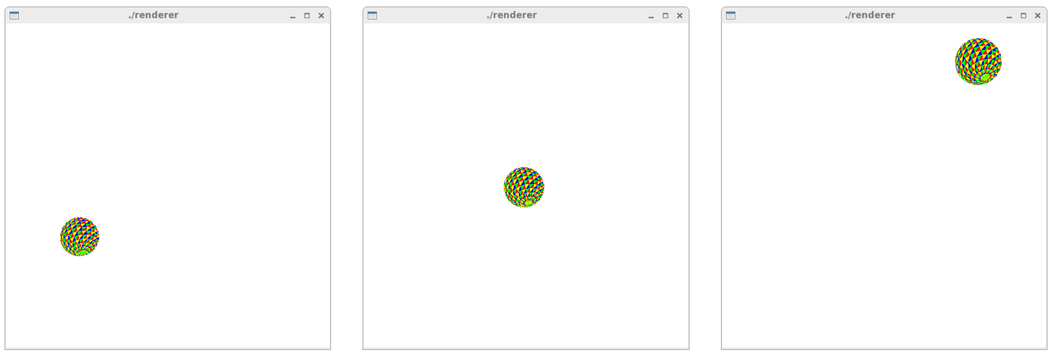
Suppose we want the sphere to move under gravity in the y direction and assume that the sphere's initial velocity in the z direction is 0. Then the translation values are given by:

$$\begin{aligned} d_x &= v_x t \\ d_y &= v_y t + \tfrac{1}{2} g t^2 \end{aligned} \tag{19.15}$$

where v_x, and v_y are initial velocities in the x and y directions respectively and are constants, and g is the gravitational constant. If we set all the constants to 1 (i.e. $v_x = v_y = g = 1$), we can modify our above vertex shader to simulate the motion:

```
// spheres.vert
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;
uniform  float  timeLapsed;   // time in milliseconds
void main(void)
{
  float t1 = timeLapsed / 1000.0;  // time t1 in seconds
  vec3 d;

  float t = sin ( t1 );      // t varies between -1 and 1
  mat4 m = mat4( 1.0 );      // 4 x 4 identity matrix
  m = 0.2 * m;               // shrink sphere by a factor of 5
  d.x = t;                   // translation in x-direction
  d.y = t + 0.5*1.0*t*t;     // translation in y-direction
  m[3][0] = d.x;
  m[3][1] = d.y;
  m[3][3] = 1.0;             // reset last matrix element to 1
  gl_Position =   mvpMatrix * m *  vPosition;
}
```

In the code, the time t is sinusoidal, varying between -1 and 1. This makes the sphere oscillates along a parabolic curve. Figure 19-17 below shows a few frames of this shader output.



(a) (b) (c)

Figure 19-17 Three Frames of Sphere Motion

19.8.2 Animated Particle Systems

We have mentioned in Chapter 11 about the creation and applications of a particle system, which uses a large number of very small sprites or graphic objects to simulate certain kinds of *fuzzy* phenomena or objects such as clouds, dust, explosions, falling leaves,fire, flowing water, fog smoke, snow, sparks, meteor tails, stars and galaxies, or abstract visual effects like glowing trails, and magic spells.

The basic idea in a particle system is that we model the motion of points in space using physical laws or empirical experience. For each time step, we calculate the new position of each particle for rendering. This works well with a vertex shader where it receives a time parameter from the application. As a simple example, we use the technique in animating a sphere discussed above to animate a cloud of randomly formed color particles, each of which is a point. We declare a class called *Particles*, which is similar to the *Spheres* class discussed above except that now we do not need to create any sphere. In addition, we also declare a class called *ColorPoint* that holds the position and color of a "particle":

```
class ColorPoint{
public:
  float position[4];
  float color[4];
  ColorPoint (float position0[], float color0[]){ //constructor
    for ( int i = 0; i < 4; i++ ) {
      position[i] = position0[i];
      color[i] = color0[i];
    }
  }
};
```

In the constructor of the class *Particles*, we create *Npoints ColorPoint* objects with random positions and colors, and save them in an array (a *vector* object) named *pointArray* (both *Npoints* and *pointArray* are data members of *Particles*):

```
//create Npoints particles
float position[4];
float color[4];
for ( int i = 0; i < Npoints; ++i ){
  for ( int j = 0; j < 3; j++ ) {
    position[j] = (float)(rand()%32000)/16000.0-1.0; //-1 --> 1
    color[j] = (float) ( rand() %32000 ) / 32000.0;  // 0 --> 1
  }
  color[3] = position[3] = 1;             //not used
  ColorPoint cp = ColorPoint ( position, color );
  pointArray.push_back ( cp );            //save in vector
}
```

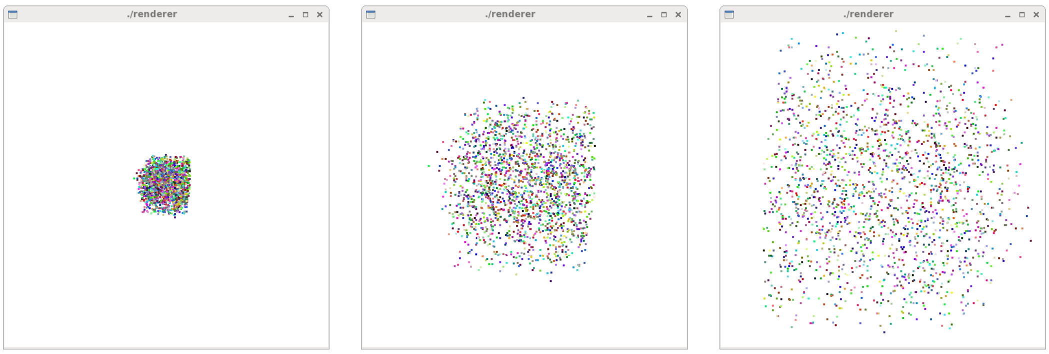
The **draw** function of *Particles* send the position and the color of each particle to the vertex shader:

```
void Particles::draw( float mvpMatrix[4][4] )
{
  int positionHandle= glGetAttribLocation(program,"vPosition");
  int mvpMatrixHandle=glGetUniformLocation(program,"mvpMatrix");
  // pass model-view projection matrix to vertex shader
  glUniformMatrix4fv(mvpMatrixHandle,1,GL_FALSE,&mvpMatrix[0][0]);

  int colorHandle =  glGetUniformLocation(program, "vColor");
  timeHandle =  glGetUniformLocation(program, "timeLapsed");

  ColorPoint *cp;
  for ( int i = 0; i < Npoints; i++ ) {
    cp = &pointArray[i];
    glUniform4fv(colorHandle, 1, cp->color);
    glBegin ( GL_POINTS );
      glVertex4fv ( cp->position );
    glEnd();
  }
}
```

The vertex shader and the fragment shader are the same as those of the *animated color sphere* presented in the previous section:

```
// particles.vert : vertex shader
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;
uniform  float  timeLapsed;    // time in milliseconds
void main(void)
{
  float t = timeLapsed / 5000.0;  // adjust time
  float s = sin ( t );        // s varies between -1 and 1
  mat4 m = mat4( 1.0 );       // 4 x 4 identity matrix
  m = s * m;                  // try to create scale matrix
  m[3][3] = 1.0;              // set last matrix element to 1
                              //  to form a proper scale matrix
  gl_Position = m * mvpMatrix * vPosition;
}

// particles.frag : fragment shader
uniform vec4 vColor;

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

Figure 19-18 below shows three frames of the output of this shader; two thousand particles are randomly palced on the screen.



(a) (b) (c)

Figure 19-18 Three Frames of the Particle System Shader

19.8.3 Morphing

Morphing (shortened term of *metamorphosing*) is a technique that changes an object smoothly from one object to another, a generalization of the tweening technique we discussed in Chapter 11. The points of an intermediate shapes are calculated using interpolation techniques from the initial and final objects. We assume that the two objects have the same number of vertices, which can be saved in arrays. Suppose we use linear interpolation, and we want to morph objects A to B. Suppose A_i and B_i are the corresponding vertices of the objects. A corresponding point C_i of an intermediate shape is given by the affine combination of A_i and B_i:

$$C_i = (1-t)A_i + tB_i \qquad 0 \leq t \leq 1 \tag{19.16}$$

Initially, when $t = 0$, $C_i = A_i$. As t increases to 1, C_i changes to B_i. Equation (19.16) can be easily evaluated using the glsl function **mix**. For instance, suppose $vPostion$ and $vPosition1$ are two **vec4** that have the coordinate values of vertices A_i and B_i. Then the interpolated vertex can be calculated by

vec4 inPosition = mix(vPosition, vPosition1, t);

where, for example, the first component of *inPostion* is calculated according to:

inPosition.x = (1 - t) * vPosition.x + t * vPosition1.x

The implementation of this technique is straight forward. As an example, we declare a class called *Morph*, which is similar to the classes discussed above such as *Spheres* and *Particles*. In the class, we declare two static **float** constant arrays, *figureA* and *figureB*, to store the vertex values of the initial and final objects respectively:

```
const float Morph::figureA[][3] =
   {{0,0,0}, {3,3,0}, {6,0,0}, {6,-6,0}, {4,-6,0},
    {4,-4,0}, {2,-4,0}, {2,-6,0}, {0,-6,0}};
const float Morph::figureB[][3] =
   {{0,0,0}, {3,0,0}, {6,0,0}, {6,-2,0}, {4,-2,0},
    {4,-6,0}, {2,-6,0}, {2,-2,0}, {0,-2,0}};
```

In the example, *figureA* defines a wedge-like shape and *figureB* defines a T-shape. Both of them have 9 vertices.

We pass the blending parameter t, which we call *timeStep* from the *Morph* class to the vertex shader. This parameter, varying between 0 and 1, is incremented or decremented in the **idle** callback function of the renderer:

```
static void idle(void)
{
  static float timeStep = 0;
  static bool  increase = true;
  float t = glutGet ( GLUT_ELAPSED_TIME );
  if ( t - previousTime  > 500.0 ){
    if ( timeStep > 0.91 )
       increase = false;
    else if ( timeStep < 0.09 )
       increase = true;
    if ( increase )
     timeStep += 0.1;
    else
     timeStep -= 0.1;
    morph->setTime ( timeStep );
    previousTime = t;
    glutPostRedisplay();
  }
}
```

In the code, we have used a frame rate of 2 fps (frames per second). That is, the period between two frames is 500 ms. The variable *timeStep* increases from 0 to 1 with step 0.1 and then decreases from 1 to 0 and so on. So the animation morphs A to B and reverses direction, morphing the figure from B to A and so on.

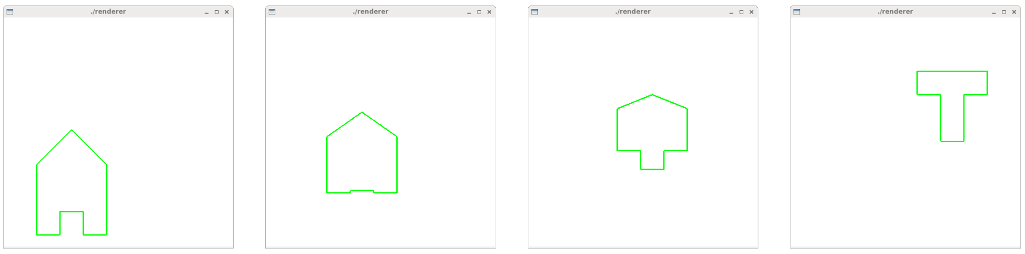
Like before, the **draw** member function passes the vertex data and other parameters to the shaders:

```
void Morph::draw( float mvpMatrix[4][4] )
{
  int posHandle = glGetAttribLocation(program,"vPosition");
  int posHandle1 = glGetAttribLocation(program,"vPosition1");
  glEnableVertexAttribArray( posHandle );
  glEnableVertexAttribArray( posHandle1 );
  glVertexAttribPointer(posHandle,3,GL_FLOAT,false,0,figureA);
  glVertexAttribPointer(posHandle1,3,GL_FLOAT,false,0,figureB);
  //timeHandle and Npoints are data members of class
  timeHandle =  glGetUniformLocation(program, "timeStep");
  .....
  glDrawArrays( GL_LINE_LOOP, 0, Npoints );
}
void Morph::setTime ( float t )
{
  glUniform1f(timeHandle, t); //send timeStep to vertex shader
}
```

The following is the corresponding vertex shader, which is similar to those discussed above for animation:

```
 // morph.vert
 attribute vec4  vPosition;   //vertex of figureA
 attribute vec4  vPosition1;  //vertex of figureB
 uniform   mat4  mvpMatrix;
 uniform  float  timeStep;    //interpolation time parameter
 void main(void)
 {
   vec3 d;                    //displacement
   float t = timeStep;
   mat4 m = mat4( 1.0 );     // 4 x 4 identity matrix
   //interpolated figure move along diagonal of screen
   d.x = 8.0 * t;            // translation in x-direction
   d.y = 8.0 * t;            // translation in y-direction
   m[3][0] = d.x;
   m[3][1] = d.y;
   m[3][3] = 1.0;            // reset last matrix element to 1
   gl_Position = mvpMatrix * m * mix(vPosition, vPosition1, t);
 }
}
```

Figure 19-19 below shows four frames of this shader's output.



(a) (b) (c) (d)

Figure 19-19 Four Frames of the Morphing Shader

19.9 Texture Shaders

One method to perform texture operations in glsl is to access the texture coordinates for each vertex through built in attribute variables, one for each texture unit. The following are the available built in variables that a shader can use:

```
attribute vec4 gl_MultiTexCoord0;
attribute vec4 gl_MultiTexCoord1;
attribute vec4 gl_MultiTexCoord2;
attribute vec4 gl_MultiTexCoord3;
attribute vec4 gl_MultiTexCoord4;
attribute vec4 gl_MultiTexCoord5;
attribute vec4 gl_MultiTexCoord6;
attribute vec4 gl_MultiTexCoord7;
```

The texture matrix of each texture unit can be accessed through a uniform array:

```
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
```

The vertex shader has to compute the texture coordinates for the vertex and store the values in the pre-defined varying variable *gl_TexCoord*[*i*]. For example, we need to declare in the fragment shader a special type of variable called **sampler**. Each sampler variable in a program represents a single texture of a particular texture type. It provides access to a particular texture object, including all its parameters. OpenGL supports a variety of sampler types for accessing one-dimensional (**sampler1D**, two-dimensional (**sampler2D**), three-dimensional (**sampler3D**), and cube-map (**samplerCube**) textures. The following table lists the sampler types that OpenGL supports; the prefix *t* preceding *sampler* in a sampler name represents any of the 3 possible prefixes (nothing for **float**, i for **signed int**, and u for **unsigned int**). The rest of the sampler's name refers to the texture type of the sampler:

Table 19-3 Names Map of GLSL Samplers

GLSL Sampler	OpenGL Texture enum	Texture Type
*t*sampler1D	GL_TEXTURE_1D	1D texture
*t*sampler2D	GL_TEXTURE_2D	2D texture
*t*sampler3D	GL_TEXTURE_3D	3D texture
*t*samplerCube	GL_TEXTURE_CUBE_MAP	Cubemap Texture
*t*sampler2DRect	GL_TEXTURE_RECTANGLE	Rectangle Texture
*t*sampler1DArray	GL_TEXTURE_1D_ARRAY	1D Array Texture
*t*sampler2DArray	GL_TEXTURE_2D_ARRAY	2D Array Texture
*t*samplerCubeArray	GL_TEXTURE_CUBE_MAP_ARRAY	Cubemap Array Texture (requires GL 4.0 or ARB _texture_cube_map_array)
*t*samplerBuffer	GL_TEXTURE_BUFFER	Buffer Texture
*t*sampler2DMS	GL_TEXTURE_2D_MULTISAMPLE	Multisample Texture
*t*sampler2DMSArray	GL_TEXTURE_2D _MULTISAMPLE_ARRAY	Multisample Array Texture

For example,

```
uniform sampler2D texHandle;
```

declares a two-dimensional sampler with **float** data type.

The function **texture2D**() gives us a texel (texture element). It takes a sampler2D and texture coordinates as inputs and returns the texel value:

```
    vec4  texture2D(sampler2D texHandle, vec2 gl_TexCoord[0].st );
```

We can pass the texture coordinates to the fragment shader using gl_TexCoord[*i*], for example,

```
     gl_TexCoord[0]  = gl_MultiTexCoord0;
```

The simple example presented in the next section shows how to use these features to render an object with texture.

19.9.1 A Simple Texture Example

In this example, we apply a two-dimensional texture, which is a black-and-white checkerboard pattern to a graphics object created by the **glu** utilities. The vertex shader is simple enough, receiving the texture coordinates with the built-in variable *gl_MultiTexCoord0* and passing them to the fragment shader using the built-in array variable *gl_TexCoord*:

```
//simpletex.vert
attribute vec4  vPosition;
uniform   mat4  mvpMatrix;

void main(void)
{
  gl_TexCoord[0]  = gl_MultiTexCoord0;
  gl_Position = mvpMatrix * vPosition;
}
```

The fragment shader is also very simple. It obtains the texture color from a sampler and blends it with another color specified by the application using the **mix** function:

```
// simpletex.frag
uniform sampler2D texHandle;
uniform vec4 vColor;

void main (void)
{
  vec3 texColor = vec3 (texture2D(texHandle, gl_TexCoord[0].st));
  vec4 color = mix ( vColor, vec4 (texColor, 1.0), 0.6 );
  gl_FragColor = color;
}
```

In the application, we make use of the **glu** utilites to create some quadric objects and a teapot that support texturing. We do the usual initialization for texture as we discussed in Chapter 7. Suppose we declare a class called *Simpletex*, which is similar to classes such as *Morph* or *Spheres* discussed above, and suppose the texture image data is stored in a two-dimensional array named *checkimage*. The following code shows the specification of texture features:

```
void Simpletex::init2DTexture()
{
 glGenTextures(1, &texName);
 glBindTexture(GL_TEXTURE_2D,texName); //now we work on texName
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, iWidth,
              iHeight,0,GL_RGB,GL_UNSIGNED_BYTE,checkImage);
 glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texName);
}
```

In the code, *iwidth* and *iwidth* are the image width and height of the checkerboard texture image; they are set to 64 in the example.

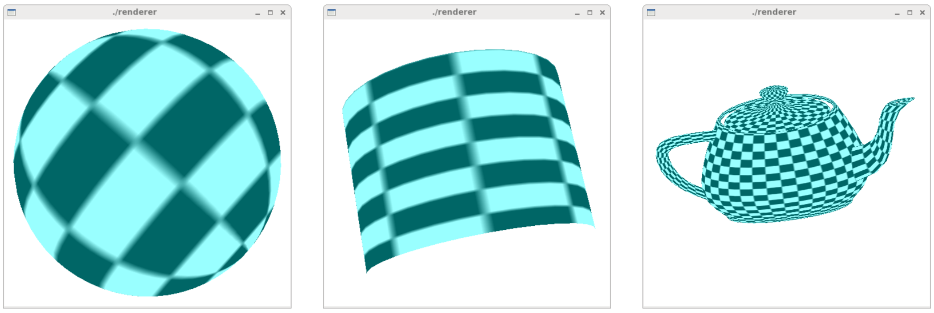
The texture data and other parameters are passed to the shaders in the **draw** routine for rendering:

```
void Simpletex::draw( float mvpMatrix[4][4] )
{
  int posHandle = glGetAttribLocation(program,"vPosition");
  glEnableVertexAttribArray( posHandle );

  int mvpMatrixHandle=glGetUniformLocation(program,"mvpMatrix");
  glUniformMatrix4fv(mvpMatrixHandle,1,GL_FALSE,&mvpMatrix[0][0]);

  int colorHandle =  glGetUniformLocation(program, "vColor");
  glUniform4f(colorHandle,  0, 1, 1, 1);      //set color of object
  GLUquadric *qobj=gluNewQuadric(); //create quadric objects
  gluQuadricTexture(qobj,GL_TRUE);  //requires texture coordinates
  if ( objType == 0 )
    gluSphere(qobj,0.8,32,32);
  else if ( objType == 1 )
    gluCylinder(qobj, 0.8, 0.8, 1.2, 32, 32);//top, base height
  else
     glutSolidTeapot(0.6f);            //has texture coordinates
  gluDeleteQuadric(qobj);
}
```

The code allows us to choose an object to be rendered, which can be a sphere, a cylinder or a teapot, depending on the value of the **int** variable *objType*, which represents object types. The function **gluNewQuadric** creates and returns a pointer to a new quadric object. A quadric surface is described by a polynomial with each term in the form $x^m y^n z^k$, with $m + n + k \leq 2$. Any quadric can be expressed in the form

$$q(x, y, z) = Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J = 0 \quad (19.17)$$

wehre $A, ..., J$ are constants. The function **gluQuadricTexture** specifies whether texture coordinates should be generated for the quadric for rendering with the quadric object. A value of GL_TRUE indicates that texture coordinates should be generated. The *renderer*, which is not shown here, allows a user to press the key 't' to toggle the object types, and to press other keys to rotate, translate or magnify the object. Figure 19-20 below shows a sample output of each textured object in the example.



(a) (b) (c))

Figure 19-20 Sample Outputs of Textured Objects

Actually in this example, we make use of the command **gluQuadricTexture** to map the texture coordinates and have not shown you how the texture coordinates are set. We have already discussed the texture coordinate mapping principles in Chapter 7. Here, again we use the *Spherel* shader example discussed in Section 19.7 above to review how this is done.

We have presented in Equation (7.29) of Chapter 7 that the texture coordinates of a spherical surface is given by:

$$\begin{aligned} s &= \frac{\phi}{2\pi} \\ t &= \frac{\theta}{\pi} \end{aligned} \tag{19.18}$$

To implement the texture mapping for our *Spherel*, we declare another *XYZ* **vector** named *texPoints*, which saves the texture coordinates at all the vertices. The **createSphere** function becomes (we call our new class *Spherelt*):

```
void Spherelt::createSphere ( float r, int nSlices, int nStacks )
{
   double phi,  theta;
   XYZ *p = new XYZ();
   XYZ *t = new XYZ();   //for texture coordinates
   const double PI = 3.1415926;
   const double TWOPI = 2 * PI;

   for ( int j = 0; j < nSlices; j++ ) {
     //phi: 0 to 2pi  (modified for texture coordinates)
     phi = j * TWOPI / (nSlices-1);
     for ( int i = 0; i < nStacks; i++ ) {
       theta = i * PI / (nStacks-1);  //0 to pi
       p->x = r * (float) ( sin ( theta ) *  cos ( phi ));
       p->y = r * (float) ( sin ( theta ) *  sin ( phi ));
       p->z = r * (float)  cos ( theta );
       t->x = phi / TWOPI;      // s (column)
       t->y = 1 - theta / PI;  // t (row)
       vertices.push_back ( *p  );
       texPoints.push_back ( *t );
     }
   }
}
```

The texture coordinates are then saved in an array named *texCoords* along with the saving of the spherical coordinates in the *Spherelt* class constructor:

```
  sphereCoords = new float[3*nVertices];
  texCoords = new float[2*nVertices];
  int k = 0;
  int kk = 0;
  for ( int i = 0; i < nVertices; i++ ) {
    XYZ v = vertices[i];
    sphereCoords[k++] = v.x;
    sphereCoords[k++] = v.y;
    sphereCoords[k++] = v.z;
    v = texPoints[i];
    texCoords[kk++] = v.x;
    texCoords[kk++] = v.y;
  }
```

After we have also incorporated the texture features of *Simpletex* in *Spherelt*, we only need to add two statements in the **draw** function to access the texture coordinates:

```
void Spherelt::draw( float mvpMatrix[4][4] )
{
  ..... //Same as draw() of Spherel
  glEnableClientState (GL_TEXTURE_COORD_ARRAY);
  glTexCoordPointer ( 2, GL_FLOAT, 0, texCoords );
  // Draw all triangles
  for ( int i = 0; i < nTriangles; i++ ) {
    glDrawElements( GL_TRIANGLES, 3,
                         GL_UNSIGNED_SHORT, (drawOrders + i*3 ));
}
```

For the vertex shader, we just need to add the statement in **main**():

gl_TexCoord[0] = gl_MultiTexCoord0;

which passes the texture coordinates to the fragment shader, where we mix the light intensity obtained from the Phong model to the texture color:

```
//fragment shader
void main()
{
  ..... //same as fragment shader of Spherel
  vec4 intensity =  ambient + diffuse + specular;
  vec4 color = mix ( intensity, vec4 (texColor, 1.0), 0.2 );
  gl_FragColor = color;
}
```

Figure 19-21 below shows an output of this shader program.

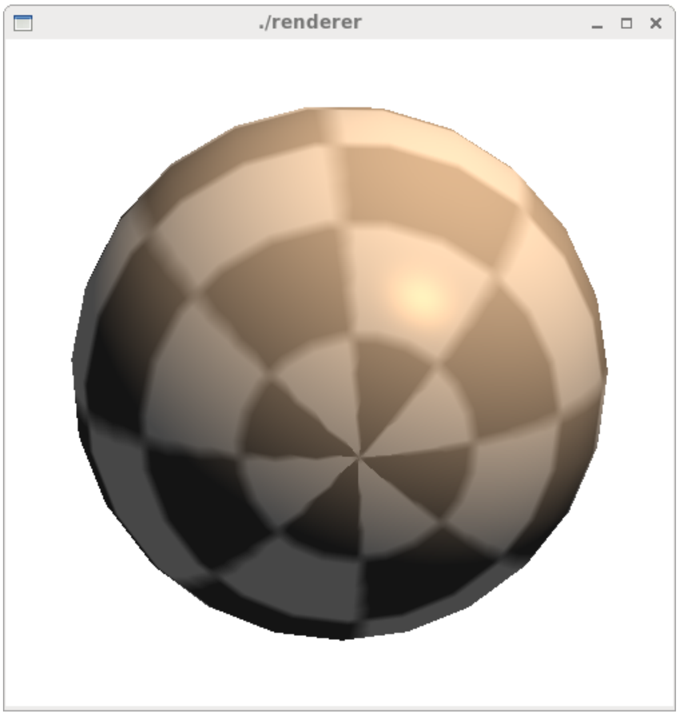


Figure 19-21 Sample Output of Textured Lit Sphere

19.9.2 Bump Mapping with Shaders

Bump mapping is a texture mapping technique that can make a smooth surface appear rough, looking more realistic without increasing the geometry complexity. For example, we can apply bump mapping to the surface of a sphere to make it look like an orange, which has bumpy skin.

The idea of the technique is to perturb the normal for each fragment. The perturbations can be generated from an algorithm or saved as textures (called normal-map or bump-map) in memory. The normal vector contained in a texel of a normal-map represents the vector perpendicular to the surface of an object at the considered texel. Without deformation, the proper normal vector is $(0, 0, 1)$ as its XYZ components. We can modulate the vector to make it differ slightly from the proper one. For example, a modulated normal with components $(0.1, 0.2, 0.975)$ indicates a modification of the surface. A normal vector considered here is always normalized, having unit length, i.e. $(X^2 + Y^2 + Z^2) = 1$.

In the technique, we form an orthonormal frame of reference using vectors normal, tangent and binormal (tangent space) at each vertex of the surface:

X-axis = (1, 0, 0) (tangent vector **t**)
Y-axis = (0, 1, 0) (binormal vector **b**)
Z-axis = (0, 0, 1) (normal vector **n**)

This is similar to the Frenet frame we discussed in Chapter 14. To work in this orthonormal frame whose origin is at a vertex of the surface, we need a transformation to project a vector **a** = (x, y, z) expressed in our world coordinate system (**i, j, k**) to **a**‘ in the new coordinate system with basis (**t, n, b**). This can be done by first translating the origin of our coordinate system to the surface vertex and make an appropriate rotation. If all vectors and points are represented in homogeneous coordinates as 4×1 matrices, the 4×4 transformation matrix that brings the basis (**i, j, k**) to the orthonormal frame at a vertex $p = (p_x, p_y, p_z)$ and performs a rotation is given by:

$$M = \begin{pmatrix} t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & p_x \\ 0 & 0 & 0 & p_y \\ 0 & 0 & 0 & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} t_x & t_y & t_z & p_x \\ b_x & b_y & b_z & p_y \\ n_x & n_y & n_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19.19}$$

Note that there is a subtle difference between the Frenet frame transformation matrix of (14.11) in Chapter 14 and the transformation matrix M here. The former transformation rotates the basis (**i, j, k**) to the new basis (**T, N, B**) so that all cross-sections will be rotated accordingly. Here, the transformation matrix M lets us express the components of a vector (or a point) in the new coordinate system (**t, n, b**).

For some parametric surfaces, a tangent at a point can be calculated conveniently by differentiating each coordinate with respect to a surface parameter. For example, a unit sphere centered at the origin O is described by the parametric equations of (19.3) with $r = 1$. At a point $P = (x, y, z)$ on the sphere, we can calculate the unit normal **n** and a unit tangent **t** easily:

$$\mathbf{n} = P - O = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \mathbf{t} = \frac{\partial \mathbf{n}}{\partial \phi} = \begin{pmatrix} -sin\theta sin\phi \\ sin\theta cos\phi \\ 0 \end{pmatrix} = \begin{pmatrix} -y \\ x \\ 0 \end{pmatrix}, \tag{19.20}$$

However, the tangent vector of (19.20) is undefined at the poles when $\theta = 0$, or 180^o, which makes all three components of **t** in (19.20) become 0. For these special situations, we can set the vector to point along the y-axis, i.e.,

$$\mathbf{t} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \text{when } z = 1, -1 \tag{19.21}$$

A binormal vector **b** is given by the cross product of **t** and **n** (i.e. $\mathbf{b} = \mathbf{t} \times \mathbf{n}$). So the transformation matrix M ($z \neq \pm 1$) is

$$M = \begin{pmatrix} -y & x & 0 & x \\ b_x & b_y & b_z & y \\ x & y & z & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19.22}$$

In glsl, the transformation matrix can be easily implemented like the following:

```
    attribute vec4 vPosition;
    .....
    vec3 t, b, n;
    .....
    mat4 M = mat4(vec4 (t, vPosition.x), vec4(b, vPosition.y),
                 vec4( n, vPosition.z), vec4(0.0, 0.0, 0.0, 1.0));

  }
```

Any vector or vertex can be transformed to this surface-local coordinate system by multiplying it by M. (Note that OpenGL matrices are column-major; their columns are our rows.)

In the following, we use the textured sphere (i.e. *Spherelt*) we discussed above as an example to illustrate the principle of a bump mapping shader. We call our new class *Bump*, which is a slight modification of *Spherelt*. For simplicity and clarity of presentation, some parameters are hard-coded in the shaders and we discard many fancy lightling features such as diffuse and ambient lights and material properties. The light intensity is 'made up' inside the fragment shader. We only pass to the shaders the light postion and eye position (viewing point). However, we do need the texture features for reasons explained later. The following is the vertex shader for the example:

```
// bump.vert: Source code of vertex shader
uniform mat4 mvpMatrix;
attribute vec4 vPosition;
uniform vec4 eyePosition;
uniform vec4 lightPosition;

varying vec4 lightVec;  //light direction vector
varying vec4 eyeVec;    //eye direction vector
void main()
{
  vec4 O = vec4 ( 0.0, 0.0, 0.0, 1 );   //origin

  // Normalized normal in object coordinates
  // Normal can be also obtained by:
  //  vec3 n = normalize ( gl_NormalMatrix * gl_Normal );
  vec3 n = normalize(vPosition.xyz - O.xyz);
  vec3 t;               //tangent
  //tangent for sphere (usually from application)
  if ( n.z < 1.0 && n.z > -1.0  ) {
     t.x = -n.y;         //t perpendicular to n;
     t.y = n.x;
  } else {
     t.x = 0.0;          //special treatment at poles
     t.y = 1.0;
  }
  t.z = 0.0;
  vec3 b = cross(n, t); //binormal
  //Since n is normalized, so b and t are also normalized
```

```
   //Transformation matrix, which transforms objects
   //  to surface-local system (t, b, n)
   mat4 M = mat4(vec4(t,vPosition.x), vec4(b,vPosition.y),
                 vec4(n,vPosition.z), vec4(0.0,0.0,0.0,1.0));

   //vectors in surface-local system
   lightVec = normalize ( M * (lightPosition - vPosition) );
   eyeVec = normalize( M * ( eyePosition - vPosition ) );

   //texture coordinates for fragment shader
   gl_TexCoord[0] = gl_MultiTexCoord0;

   gl_Position = mvpMatrix * vPosition;
 }
```

In the code, the **varying** variables *lightVec* and *eyeVec* are the light direction vector and the eye direction vector, corresponding to **L** and **V** at the surface point *vPosition*, of Figure 19-14 respectively. They are calculated, transformed to the surface-local system and normalized and are passed to the fragment shader.

On the spherical surface we need to select some bump centers, where the normals of the regions around them will be perturbed. However, our application has only passed in the coordinates and attributes of a limited number of points, the vertices of the triangles that form the sphere. The attributes of all other pixels are obtained by interpolation. How do we know the coordinates of a point on the surface? The trick is to make use of the texture coordinates, which can specify any point on the surface as the whole spherical surface has been mapped to the 1×1 *st* domain with $0 \leq s \leq 1$ and $0 \leq t \leq 1$. That's why we need the statement

gl_TexCoord[0] = gl_MultiTexCoord0;

in the vertex shader.

The following is a corresponding fragment shader, a simplied version of one used by many authors. We explain this code below.

```
 // bump.frag: Source code of fragment shader

 varying vec4 lightVec;//light dir vector in surface-local system
 varying vec4 eyeVec;  //eye direction vector in surface-local system
 void main()
 {
   float bumpDensity = 10.0;
   float bumpSize = 0.1;
   vec3 litColor;
   vec2 c = bumpDensity * gl_TexCoord[0].st;
   vec2 b = fract( c ) - vec2(0.5, 0.5 );  //a texsel on surface
   vec3 n1;                 //modulated normal

   float r2;                //square of radius of bump
   r2 = b.x * b.x + b.y * b.y;

   if ( r2 <= bumpSize ) { //modulate normal
     n1 = vec3 ( b, 1.0 ); //n1.z = 1
     n1 = normalize ( n1 );
   } else                  //do not perturb normal
     n1 = vec3 ( 0, 0, 1 );
```

```
  vec3 surfaceColor = vec3 ( 1.0, 0.6, 0.2 );
  litColor = surfaceColor * max(dot(n1, lightVec.xyz), 0.0 );
  vec3 reflectVec = reflect ( lightVec.xyz, n1 );
  float specularLight = max( dot(eyeVec.xyz, reflectVec), 0.0);
  float specularFactor = 0.5;
  specularLight = pow ( specularLight, 6.0 ) * specularFactor;
  litColor = min ( litColor + specularLight, vec3(1.0) );

  gl_FragColor = vec4 ( litColor, 1.0 )
}
```

This fragment shader defines the bump locations implicitly through the variable *bumpDensity*, which is the one-dimensional density of bumps, and the fucntion **fract**. The **fract** (x) function returns the fractional part of x, i.e. x minus **floor** (x). The input parameter can be a **float** or a **float** vector. In the latter case, the operation is done component-wise. After the multiplication of *bumpDensity*, the 1×1 st domain is magnified as shown in Figure 19-22 below, where *bumpDensity* is 5. The total number of bumps on the surface is the square of *bumpDensity* and is equal to $5^2 = 25$ in the example of Figure 19-22. This is because the new domain can be divided into equally spaced grid lines in the s and the t directions. Each intersection point of the grid lines is described by two integer coordinates like point b in Figure 19-22(b) that has coordinates $(1, 1)$. In Figure 19-22(b), the integer values of the components of any point in the shaded region are always $(1, 1)$. Therefore, the unit shaded square represent the group of points where the integer parts of their coordinates is $(1, 1)$. Their fractional parts represents how far the point is from $(1, 1)$, an intersection point. If we shift the fraction components by $(0.5, 0.5)$, then the center of the grid square can be regarded as a bump center. This is done by the statement:

vec2 b = fract(c) - vec2(0.5, 0.5); //a texsel on surface

We 'bump' a point (modulate the unit normal *n1* at the point) only if it lies within the radius of a circle as shown in the figure, otherwise the normal is left unchanged (i.e. $n1 = (0, 0, 1)$). In the code the **float** variable *r2* is the square of the radius of the circle, which is inside the square.

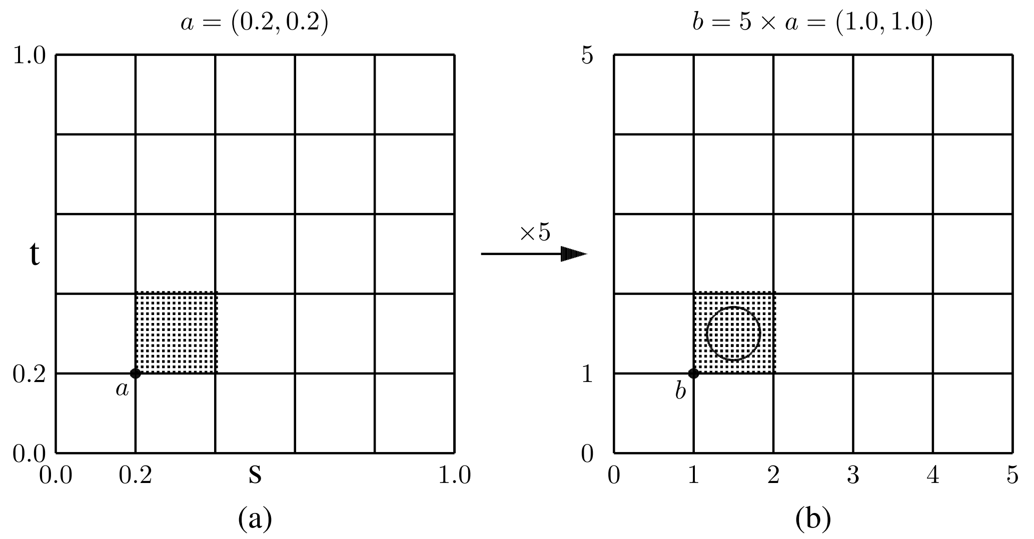


Figure 19-22 Texture Coordinates Multiplied by *bumpDensity* (=5)

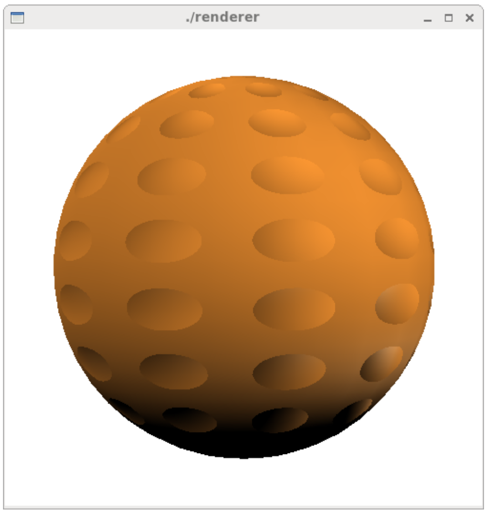
The **vec3** variable *reflectVec* is the ray reflection direction vector, which is the vector **R** of Figure 19-14. Its value is calculated using the function **reflect**.

We hard-code the surface-color and compute the diffuse and specular reflection intensity values in the usual way except that vector values are in surface-local systems. Figure 19-23(a) below shows an output of this shader.

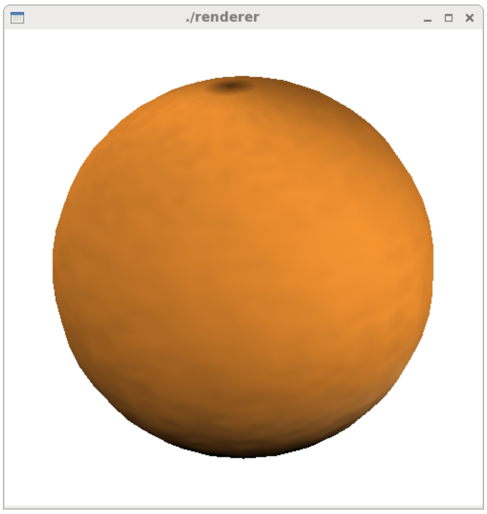
In the above example, we modulate normals using an algorithm. Alternatively, we can store the normal perturbations as a texture, which is referred to as a **bump map** or a **normal map**. Using our textured sphere as an example, we can create a bump map with minor modifications to our checkerboard texture data:

```
void Bump::makeCheckImage(void)
{
  int i, j, r, g;
  for (i = 0; i < iWidth; i++) {
     for (j = 0; j < iHeight; j++) {
       r = rand() % 32;
       g = rand() % 32;
       checkImage[i][j][0] = (GLubyte) r;
       checkImage[i][j][1] = (GLubyte) g;
       checkImage[i][j][2] = (GLubyte) 255;
     }
  }
}
```

The red and green components of the texture are randomly assigned an integer value betwenn 0 and 32, and the blue component is assigned a value of 255. When the RGB components are normalized to $[0, 1]$, the red and green values are in $[0, 0.125]$ and the blue value is always 1. The RGB components are mapped to the XYZ values of a normal at the surface. This means that we only perturb the X and Y components with small values and always keep the Z component, which is perpendicular to the surface, to be 1. (Of course, after we have normalized the normal vector, the Z value also changes.) The following is the code for the fragment shader that uses this normal map. An output of it is shown in Figure 19-23(b).



(a)



(b)

Figure 19-23 Outputs of Bump Mapping Shaders

```
// bump.frag: Source code of fragment shader
```

```
varying vec4 lightVec;  //light dir vector in surface-local system
varying vec4 eyeVec;    //eye direction vector in surface-local system
uniform sampler2D texHandle;
void main()
{
  vec3 litColor;
  vec3 n1;                //modulated normal
  n1 = texture2D(texHandle, gl_TexCoord[0].st).rgb;
  n1 = normalize ( n1 );
  vec3 surfaceColor = vec3 ( 1.0, 0.6, 0.2 );
  litColor = surfaceColor * max(dot(n1, lightVec.xyz), 0.0 );
  vec3 reflectVec = reflect ( lightVec.xyz, n1 );
  float specularLight = max ( dot(eyeVec.xyz, reflectVec), 0.0 );
  float specularFactor = 0.5;
  specularLight = pow ( specularLight, 6.0 ) * specularFactor;
  litColor = min ( litColor + specularLight, vec3(1.0) );

  gl_FragColor = vec4 ( litColor, 1.0 );
}
```

Complete related programs of this chapter can be downloaded from the web site of this book.