Vertex and fragment/pixel shaders

Part 2

– a CAGD approach based on OpenGL and C++ –

Ágoston Róth

Department of Mathematics and Computer Science, Babeș-Bolyai University, Cluj-Napoca, Romania

(agoston.roth@gmail.com)

Lecture 8 – April 25, 2022

Data types in GLSL

- Simple data types:

```
float
bool
int
```

- Vectors with 2, 3, or 4 components are also available for each of the simple data types:

```
vec{2,3,4}  // a vector of 2, 3, or 4 floats
bvec{2,3,4} // bool vector
ivec{2,3,4} // vector of integers
```

- Square matrices of size 2 × 2, 3 × 3, and 4 × 4 are also provided:

```
mat2
mat3
mat4
```

- The data types for texture sampling are:

```
sampler1D       // for 1D textures
sampler2D       // for 2D textures
sampler3D       // for 3D textures
samplerCube     // for cube map textures
sampler1DShadow // for shadow maps
sampler2DShadow // for shadow maps
```

- Arrays can be declared using the same syntax as in C. However, arrays cannot be initialized when declared. Accessing elements of an array is done as in C.

Data types in GLSL

- Simple data types:

```
float
bool
int
```

- Vectors with 2, 3, or 4 components are also available for each of the simple data types:

```
vec{2,3,4}  // a vector of 2, 3, or 4 floats
bvec{2,3,4} // bool vector
ivec{2,3,4} // vector of integers
```

- Square matrices of size 2 × 2, 3 × 3, and 4 × 4 are also provided:

```
mat2
mat3
mat4
```

- The data types for texture sampling are:

```
sampler1D       // for 1D textures
sampler2D       // for 2D textures
sampler3D       // for 3D textures
samplerCube     // for cube map textures
sampler1DShadow // for shadow maps
sampler2DShadow // for shadow maps
```

- Arrays can be declared using the same syntax as in C. However, arrays cannot be initialized when declared. Accessing elements of an array is done as in C.

Data types in GLSL

- Simple data types:

```
float
bool
int
```

- Vectors with 2, 3, or 4 components are also available for each of the simple data types:

```
vec{2,3,4}  // a vector of 2, 3, or 4 floats
bvec{2,3,4} // bool vector
ivec{2,3,4} // vector of integers
```

- Square matrices of size 2 × 2, 3 × 3, and 4 × 4 are also provided:

```
mat2
mat3
mat4
```

- The data types for texture sampling are:

```
sampler1D       // for 1D textures
sampler2D       // for 2D textures
sampler3D       // for 3D textures
samplerCube     // for cube map textures
sampler1DShadow // for shadow maps
sampler2DShadow // for shadow maps
```

- Arrays can be declared using the same syntax as in C. However, arrays cannot be initialized when declared. Accessing elements of an array is done as in C.

Data types in GLSL

- Simple data types:

```
float
bool
int
```

- Vectors with 2, 3, or 4 components are also available for each of the simple data types:

```
vec{2,3,4}  // a vector of 2, 3, or 4 floats
bvec{2,3,4} // bool vector
ivec{2,3,4} // vector of integers
```

- Square matrices of size 2 × 2, 3 × 3, and 4 × 4 are also provided:

```
mat2
mat3
mat4
```

- The data types for texture sampling are:

```
sampler1D       // for 1D textures
sampler2D       // for 2D textures
sampler3D       // for 3D textures
samplerCube     // for cube map textures
sampler1DShadow // for shadow maps
sampler2DShadow // for shadow maps
```

- Arrays can be declared using the same syntax as in C. However, arrays cannot be initialized when declared. Accessing elements of an array is done as in C.

Data types in GLSL

- Simple data types:

```
float
bool
int
```

- Vectors with 2, 3, or 4 components are also available for each of the simple data types:

```
vec{2,3,4}  // a vector of 2, 3, or 4 floats
bvec{2,3,4} // bool vector
ivec{2,3,4} // vector of integers
```

- Square matrices of size 2 × 2, 3 × 3, and 4 × 4 are also provided:

```
mat2
mat3
mat4
```

- The data types for texture sampling are:

```
sampler1D       // for 1D textures
sampler2D       // for 2D textures
sampler3D       // for 3D textures
samplerCube     // for cube map textures
sampler1DShadow // for shadow maps
sampler2DShadow // for shadow maps
```

- Arrays can be declared using the same syntax as in C. However, arrays cannot be initialized when declared. Accessing elements of an array is done as in C.

Data types in GLSL

- Structures are also allowed in GLSL. The syntax is the same as C.

```glsl
struct DirectionalLight
{
    vec4 position;
    vec3 ambient, diffuse, specular;
};
```

Variables in GLSL, part I

```
/*
    Declaring a simple variable is pretty much the same as in C,
    you can even initialize a variable when declaring it.
*/

    vec3 a, b;                      // two 3D vectors
    int  c = 2;                     // c is initialized with 2
    bool d = true;                  // d is true

/*
    Declaring the other types of variables follows the same pattern,
    but there are differences between GLSL and C regarding initialization.
    GLSL relies heavily on constructor for initialization and type casting.
*/
    float e = 2;                    // incorrect, there is no automatic type casting
    float f = (float)2;             // incorrect, requires constructors for type casting

    int   g = 2;
    float h = float(g);             // correct

    vec3  v = vec3(1.0, 0.0, 0.0);  // declaring and initializing v

/*
    GLSL is pretty flexible when initializing variables using other variables.
    All that it requires is that you provide the necessary number of components.
*/
    vec2  x = vec2(1.0,2.0);
    vec2  y = vec2(3.0,4.0);

    vec4  z = vec4(x, y)            // z = vec4(1.0, 2.0, 3.0, 4.0);

    vec2  w = vec2(1.0, 2.0);
    float p = 3.0;

    vec3  q = vec3(w, p);           // q = vec3(1.0, 2.0, 3.0);
```

Variables in GLSL, part II

```
/*
    Matrices also follow this pattern. You have a wide variety of constructors for matrices.
    For instance the following constructors for initializing a matrix are available:
*/
        mat4 M = mat4(1.0)              // initializing the diagonal of the matrix with 1.0

        vec2 x = vec2(1.0, 2.0);
        vec2 y = vec2(3.0, 4.0);

        mat2 N = mat2(x, y);            // matrices are assigned in column major order

        mat2 K = mat2(1.0,0.0,1.0,0.0); // all elements are specified


/*
    The declaration and initialization of structures is demonstrated below.
*/
        struct DirectionalLight
        {
            vec4 position;
            vec3 ambient, diffuse, specular;
        };

        DirectionalLight dl = DirectionalLight(
                vec4(0.0, 0.0, -1.0, 0.0),
                vec3(0.4, 0.4,  0.4),
                vec3(0.8, 0.8,  0.8),
                vec3(1.0, 1.0,  1.0));


/*
    In GLSL a few extras are provided to simplify our lives, and make the code a little bit clearer.
    Accessing a vector can be done using letters as well as standard C selectors.
*/
    vec4  s = vec4(1.0, 2.0, 3.0, 4.0);
```

Variables in GLSL, part III

```
    float x_coordinate = s.x;
    float y_coordinate = s[1];

    vec2  both_xy_coordinates = s.xy;

    float depth = s.w;

/*
    If you are talking about colors then r, g, b, a can be used.
    For texture coordinates the available selectors are s, t, p, q.
    Notice that by convention, texture coordinates are often referred as s, t, r, q.
    However r is already being used as a selector for "red" in RGBA.
    Hence there was a need to find a different letter, and the lucky one was p.

    Matrix selectors can take one or two arguments, for instance m[0], or m[2][3].
    In the first case the first column is selected, whereas in the second a single element is selected.

    As for structures the names of the elements of the structure can be used as in C,
    so assuming the structures described above the following line of code could be written:
*/
    dl.ambient = vec3(0.5, 0.0, 0.0);
```

Variable qualifiers

- const – the declaration is of a compile time constant;
- attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders for which such a variable is read-only);
- uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders such variables are read-only);
- varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

Variable qualifiers

- const – the declaration is of a compile time constant;
- attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders for which such a variable is read-only);
- uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders such variables are read-only);
- varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

Variable qualifiers

- const – the declaration is of a compile time constant;
- attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders for which such a variable is read-only);
- uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders such variables are read-only);
- varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

Variable qualifiers

- const – the declaration is of a compile time constant;
- attribute – global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders (this qualifier can only be used in vertex shaders for which such a variable is read-only);
- uniform – global variables that may change per primitive (may not be set inside glBegin/glEnd), that are passed from the OpenGL application to the shaders (this qualifier can be used in both vertex and fragment shaders; for the shaders such variables are read-only);
- varying – used for interpolated data between a vertex shader and a fragment shader (available for writing in the vertex shader, and read-only in a fragment shader).

Control flow statements in GLSL

- The available options are pretty much the same as in C. There are conditional statements, like if-else, iteration statements like for, while and do-while.
- A few jumps are also defined:

```
continue    // available in loops, causes a jump to the next iteration of the loop

break       // available in loops, causes an exit of the loop

discard     // can only be used in fragment shaders: it causes the termination of the shader
            // for the current fragment without writing to the frame buffer, or depth
```

Control flow statements in GLSL

- The available options are pretty much the same as in C. There are conditional statements, like if-else, iteration statements like for, while and do-while.
- A few jumps are also defined:

```
continue    // available in loops, causes a jump to the next iteration of the loop

break       // available in loops, causes an exit of the loop

discard     // can only be used in fragment shaders: it causes the termination of the shader
            // for the current fragment without writing to the frame buffer, or depth
```

Functions in GLSL

- As in C a shader is structured in functions.
- At least each type of shader must have a main function declared with the following syntax:

```
void main()
{
    /*
        code
    */
}
```

- User defined functions may be defined. As in C a function may have a return value, and should use the return statement to pass out its result. A function can be void of course. The return type can have any type, but it cannot be an array.
- The parameters of a function have the following qualifiers available:

```
in       // for input parameters

out      // for outputs of the function (the return statement is also an option
         // for sending the result of a function)

inout    // for parameters that are both input and output of a function
```

  If no qualifier is specified, by default it is considered to be in.
- A function can be overloaded as long as the list of parameters is different. Recursion behavior is undefined by specification.

Functions in GLSL

- As in C a shader is structured in functions.
- At least each type of shader must have a main function declared with the following syntax:

```
void main()
{
    /*
        code
    */
}
```

- User defined functions may be defined. As in C a function may have a return value, and should use the return statement to pass out its result. A function can be void of course. The return type can have any type, but it cannot be an array.
- The parameters of a function have the following qualifiers available:

```
in      // for input parameters

out     // for outputs of the function (the return statement is also an option
        // for sending the result of a function)

inout   // for parameters that are both input and output of a function
```

  If no qualifier is specified, by default it is considered to be in.
- A function can be overloaded as long as the list of parameters is different. Recursion behavior is undefined by specification.

Functions in GLSL

- As in C a shader is structured in functions.
- At least each type of shader must have a main function declared with the following syntax:

```
void main()
{
    /*
        code
    */
}
```

- User defined functions may be defined. As in C a function may have a return value, and should use the return statement to pass out its result. A function can be void of course. The return type can have any type, but it cannot be an array.
- The parameters of a function have the following qualifiers available:

```
in      // for input parameters

out     // for outputs of the function (the return statement is also an option
        // for sending the result of a function)

inout   // for parameters that are both input and output of a function
```

  If no qualifier is specified, by default it is considered to be in.
- A function can be overloaded as long as the list of parameters is different. Recursion behavior is undefined by specification.

Functions in GLSL

- As in C a shader is structured in functions.
- At least each type of shader must have a main function declared with the following syntax:

```
void main()
{
    /*
        code
    */
}
```

- User defined functions may be defined. As in C a function may have a return value, and should use the return statement to pass out its result. A function can be void of course. The return type can have any type, but it cannot be an array.
- The parameters of a function have the following qualifiers available:

```
in      // for input parameters

out     // for outputs of the function (the return statement is also an option
        // for sending the result of a function)

inout   // for parameters that are both input and output of a function
```

  If no qualifier is specified, by default it is considered to be in.

- A function can be overloaded as long as the list of parameters is different. Recursion behavior is undefined by specification.

Functions in GLSL

- As in C a shader is structured in functions.
- At least each type of shader must have a main function declared with the following syntax:

```
void main()
{
    /*
        code
    */
}
```

- User defined functions may be defined. As in C a function may have a return value, and should use the return statement to pass out its result. A function can be void of course. The return type can have any type, but it cannot be an array.
- The parameters of a function have the following qualifiers available:

```
in      // for input parameters

out     // for outputs of the function (the return statement is also an option
        // for sending the result of a function)

inout   // for parameters that are both input and output of a function
```

  If no qualifier is specified, by default it is considered to be in.

- A function can be overloaded as long as the list of parameters is different. Recursion behavior is undefined by specification.

Two sided lighting

Vertex shader: two_sided_lighting.vert

```
/*
It is assumed that the OpenGL 2.0+ application provides:
        * projection and model view matrices;
        * several light sources using light indices GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7;
        * materials for front and back faces; and
        * a normal vector for each vertex.
*/

varying vec3 normal;
varying vec3 vertex;

void main()
{
    // calculate and normalize the normal vector
    normal = normalize(gl_NormalMatrix * gl_Normal);

    // transform the vertex position to eye space
    vertex = vec3(gl_ModelViewMatrix * gl_Vertex);

    gl_Position = ftransform();
}
```

Two sided lighting, part I

Fragment shader: two_sided_lighting.frag

```
varying vec3 normal;
varying vec3 vertex;

const vec4 AMBIENT_BLACK = vec4(0.0, 0.0, 0.0, 1.0);
const vec4 DEFAULT_BLACK = vec4(0.0, 0.0, 0.0, 0.0);

// i denotes the index of a light source.
bool IsLightEnabled(in int i)
{
    bool enabled = true;

    // If all the colors of the Light are set to BLACK then we know we do not need to bother
    // doing a lighting calculation on it.
    if ((gl_LightSource[i].ambient  == AMBIENT_BLACK) &&
        (gl_LightSource[i].diffuse  == DEFAULT_BLACK) &&
        (gl_LightSource[i].specular == DEFAULT_BLACK))
        enabled = false;

    return (enabled);
}

// distance is measured from current vertex to the ith light source.
float CalculateAttenuation(in int i, in float distance)
{
    return(1.0 / (gl_LightSource[i].constantAttenuation +
                  gl_LightSource[i].linearAttenuation * distance +
                  gl_LightSource[i].quadraticAttenuation * distance * distance));
}

// N denotes the unit varying normal vector.
void DirectionalLight(in int i, in vec3 N, in float shininess,
                      inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[i].position.xyz);

```

Two sided lighting, part II

Fragment shader: two_sided_lighting.frag

```
    float N_dot_L = dot(N, L);

    if (N_dot_L > 0.0)
    {
        vec3 H = gl_LightSource[i].halfVector.xyz;

        float pf = pow(max(dot(N,H), 0.0), shininess);

        diffuse  += gl_LightSource[i].diffuse  * N_dot_L;
        specular += gl_LightSource[i].specular * pf;
    }

    ambient  += gl_LightSource[i].ambient;
}

// N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
void PointLight(in int i, in vec3 N, in vec3 V, in float shininess,
                inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    vec3 D = gl_LightSource[i].position.xyz - V;
    vec3 L = normalize(D);

    float distance = length(D);
    float attenuation = CalculateAttenuation(i, distance);

    float N_dot_L = dot(N,L);

    if (N_dot_L > 0.0)
    {
        vec3 E = normalize(-V);
        vec3 R = reflect(-L, N);

        float pf = pow(max(dot(R,E), 0.0), shininess);

        diffuse  += gl_LightSource[i].diffuse  * attenuation * N_dot_L;
```

Two sided lighting, part III

Fragment shader: two_sided_lighting.frag

```glsl
        specular += gl_LightSource[i].specular * attenuation * pf;
    }

    ambient  += gl_LightSource[i].ambient * attenuation;
}

// N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
void Spotlight(in int i, in vec3 N, in vec3 V, in float shininess,
               inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    vec3 D = gl_LightSource[i].position.xyz - V;
    vec3 L = normalize(D);

    float distance = length(D);
    float attenuation = CalculateAttenuation(i, distance);

    float N_dot_L = dot(N,L);

    if (N_dot_L > 0.0)
    {
        float spot_effect = dot(normalize(gl_LightSource[i].spotDirection), -L);

        if (spot_effect > gl_LightSource[i].spotCosCutoff)
        {
            attenuation *=  pow(spot_effect, gl_LightSource[i].spotExponent);

            vec3 E = normalize(-V);
            vec3 R = reflect(-L, N);

            float pf = pow(max(dot(R,E), 0.0), shininess);

            diffuse  += gl_LightSource[i].diffuse  * attenuation * N_dot_L;
            specular += gl_LightSource[i].specular * attenuation * pf;
        }
    }
```

Two sided lighting, part IV

Fragment shader: two_sided_lighting.frag

```glsl

    ambient  += gl_LightSource[i].ambient * attenuation;
}

// N denotes the unit varying normal vector, while V corresponds to the varying vertex position.
void CalculateLighting(in int number_of_light_sources, in vec3 N, in vec3 V, in float shininess,
                       inout vec4 ambient, inout vec4 diffuse, inout vec4 specular)
{
    // Just loop through each light, and if its enabled add
    // its contributions to the color of the pixel.
    for (int i = 0; i < number_of_light_sources; i++)
    {
        if (IsLightEnabled(i))
        {
            if (gl_LightSource[i].position.w == 0.0)
                DirectionalLight(i, N, shininess, ambient, diffuse, specular);
            else if (gl_LightSource[i].spotCutoff == 180.0)
                PointLight(i, N, V, shininess, ambient, diffuse, specular);
            else
                Spotlight(i, N, V, shininess, ambient, diffuse, specular);
        }
    }
}

void main()
{
    // Normalize the normal. A varying variable CANNOT be modified by a fragment shader.
    // So a new variable needs to be created.
    vec3 n = normalize(normal);

    vec4 ambient, diffuse, specular, color;

    // Initialize the contributions for the front-face-pass over the lights.
    ambient  = vec4(0.0);
    diffuse  = vec4(0.0);
```

Two sided lighting, part V

Fragment shader: two_sided_lighting.frag

```
    specular = vec4(0.0);

    // In this case the built-in uniform gl_MaxLights is used to denote the number of lights.
    // A better option may be passing in the number of lights as a uniform or replacing the
    // current value with a smaller value.
    CalculateLighting(gl_MaxLights, n, vertex, gl_FrontMaterial.shininess,
                      ambient, diffuse, specular);

    color  = gl_FrontLightModelProduct.sceneColor  +
             (ambient  * gl_FrontMaterial.ambient) +
             (diffuse  * gl_FrontMaterial.diffuse) +
             (specular * gl_FrontMaterial.specular);

    // Re-initialize the contributions for the back-face-pass over the lights.
    ambient  = vec4(0.0);
    diffuse  = vec4(0.0);
    specular = vec4(0.0);

    // Now caculate the back contribution. All that needs to be done is to flip the normal.
    CalculateLighting(gl_MaxLights, -n, vertex, gl_BackMaterial.shininess,
                      ambient, diffuse, specular);

    color += gl_BackLightModelProduct.sceneColor  +
             (ambient  * gl_BackMaterial.ambient) +
             (diffuse  * gl_BackMaterial.diffuse) +
             (specular * gl_BackMaterial.specular);

    color = clamp(color, 0.0, 1.0);

    gl_FragColor = color;
}
```

Two sided lighting

Results

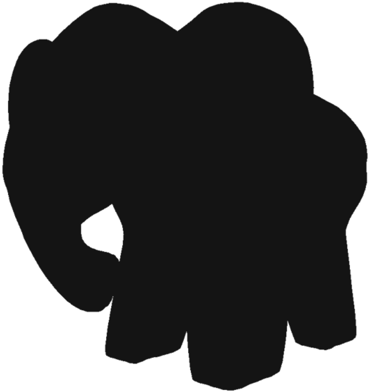


Fig. 1: Simple color shader.

Two sided lighting

Results

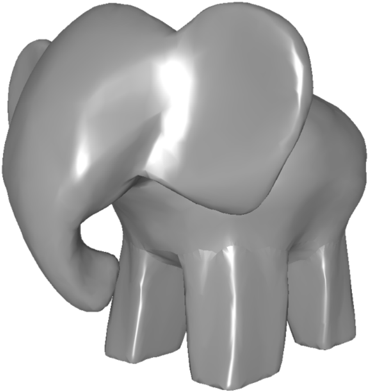


Fig. 2: A directional light.

Two sided lighting

Results

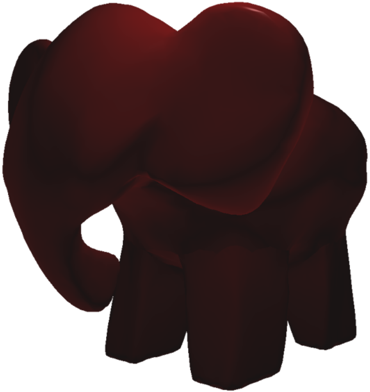


Fig. 3: A point light.

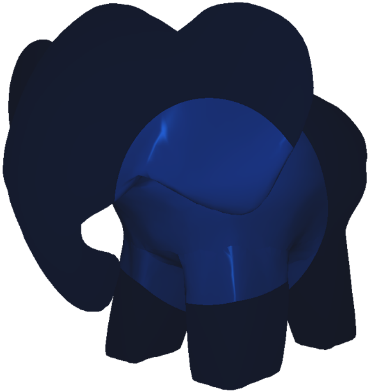


Fig. 4: A spotlight.

Two sided lighting

Results

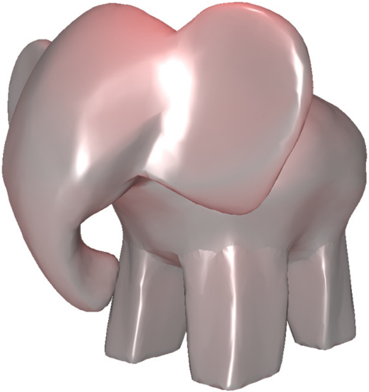


Fig. 5: The contributions of previous directional and point lights.

Two sided lighting

Results

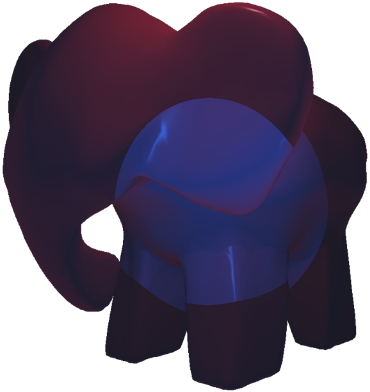


Fig. 6: The contributions of previous point and spotlights.

Two sided lighting

Results

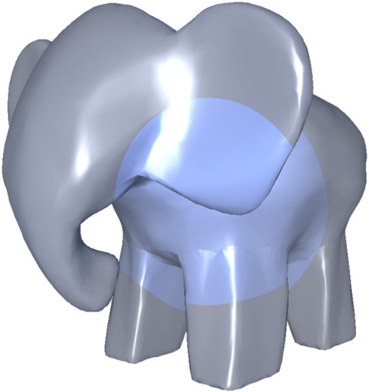


Fig. 7: The contributions of previous directional and spotlights.

Two sided lighting

Results

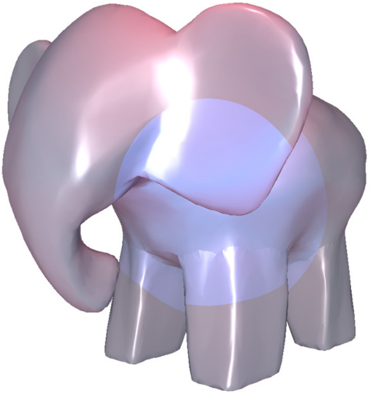


Fig. 8: The contributions of previous directional, point, and spotlights.

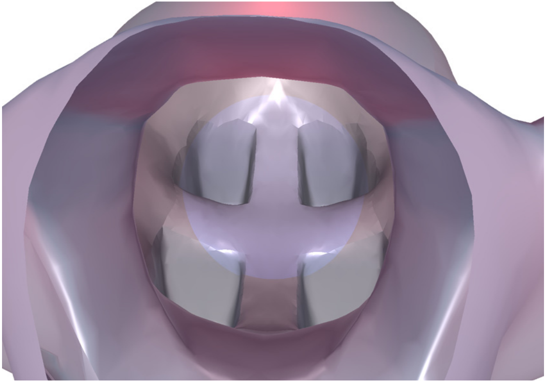


Fig. 9: The interior (i.e. the set of back faces) of the previous model is also lit up.

Toon shader

Vertex shader: toon.vert

```glsl
varying float intensity;
varying vec3  normal, light_direction;

void main()
{
        normal = normalize(gl_NormalMatrix * gl_Normal);

        light_direction = normalize(vec3(gl_LightSource[0].position));

        intensity = dot(light_direction, normal);

        gl_Position = ftransform();
}
```

Toon shader

Fragment shader: toon.frag

```
uniform vec4 default_outline_color;

varying float intensity;
varying vec3  normal, light_direction;

void main()
{
        vec4 color = gl_FrontMaterial.ambient;
        vec3 N = normalize(normal), L = normalize(light_direction);
        float N_dot_L = max(0.0, dot(N, L));

        if (intensity > 0.95)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.95;
        else if (intensity > 0.85)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.85;
        else if (intensity > 0.75)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.75;
        else if (intensity > 0.65)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.65;
        else if (intensity > 0.55)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.55;
        else if (intensity > 0.45)
                color += gl_FrontMaterial.diffuse * N_dot_L *  0.45;
        else
        {
            color = default_outline_color;
        }

        gl_FragColor = color;
}
```

Toon shader

Results

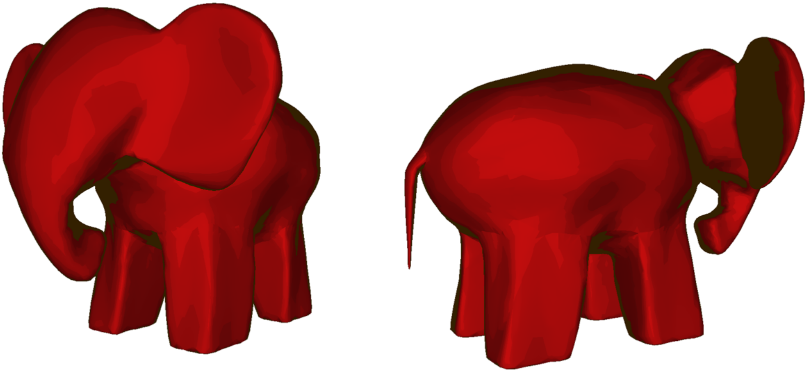


Fig. 10: Toon shader (the default outline color is $(0.2, 0.125, 0.0, 1.0)$; the applied material is MatFBRuby).

Toon shader

Results

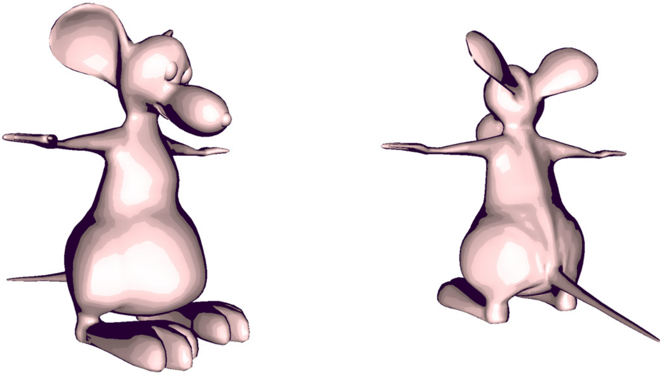
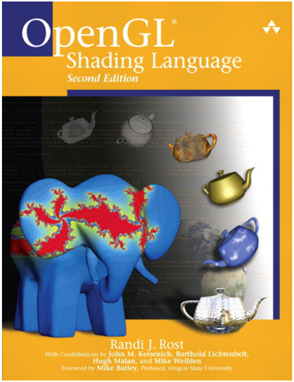
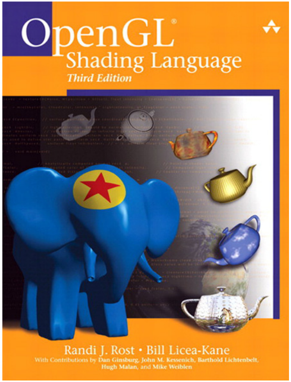


Fig. 11: Toon shader (the default outline color is $(0.15, 0.0, 0.2, 1.0)$; the applied material is MatFBPearl).

Guide to writing shaders


OpenGL®
Shading Language
Second Edition
Randi J. Rost
With Contributions by John M. Kessenich, Barthold Lichtenbelt,
Hugh Malan, and Mike Weiblen
Foreword by Mike Bailey, Professor, Oregon State University


OpenGL®
Shading Language
Third Edition
Randi J. Rost • Bill Licea-Kane
With Contributions by Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt,
Hugh Malan, and Mike Weiblen