GLSL v1.20

Scott MacHaffie
Schrödinger, Inc.

http://www.schrodinger.com

Table of Contents

Introduction

Shaders are a convenient technique for getting higher quality and better performance out of modern graphics cards.

GLSL is the name of the shading language which is a core part of OpenGL. GLSL has a unique version number corresponding to an OpenGL version number.

For OpenGL 2.1, the GLSL version number is 1.20. This is the version I will be covering in this talk.

For OpenGL 3.0, the GLSL version number is 1.30. This version is neither forwards- nor backwards-compatible with 1.20.

GLSL is essentially a variant of C with extra features (additional data types, built-in functions, and built-in variables), minus some standard C features (e.g. pointers).

A shader program requires both a vertex shader and a fragment shader. These two components are linked together into a shader program.

The vertex shader is called once for each vertex, per primitive (e.g. three times for a triangle, four for a quad). The fragment shader is called once for each pixel. One of the big advantages of using shaders over traditional OpenGL code is that the lighting calculations can be done on a per-pixel basis, resulting in much higher quality.

Example 01: Trivial shader

The most trivial shader consists of transforming and passing the vertex coordinates in the vertex shader and copying the input color to the output color in the fragment shader.

```
// This is a pass-through vertex shader
void main()
{
```

```
      gl_Position = ftransform();
      gl_FrontColor = gl_Color;
      gl_BackColor = gl_Color;
}

// This is the pass-through fragment shader
void main()
{
      gl_FragColor = gl_Color;
}
```

Example 01: Basic vertex shader and basic fragment shader

This fragment shader is actually useful for debugging. If the vertex shader is doing something complicated (typically, rewriting the vertices on the fly), then it can be helpful to use a pass-through fragment shader to check that the vertex shader is doing the correct thing. For example, I used a pass-through fragment shader several times when debugging the sphere vertex shader to make sure it was generating the correct bounding box. One easy way to do this in a fragment shader is to rename the real `main()` to `main2()` and insert the pass-through fragment shader as given above.

Syntax

Types of variables

There are four types of variables in GLSL: uniform, varying, parameters, and local. Uniform variables are passed into the shader from the calling program (e.g. from Maestro). Varying variables are passed from the vertex shader to the fragment shader. Local variables are used within a function. Parameters are passed into separate functions.

Uniform variables can appear in either the vertex shader or the fragment shader. I believe they have to be declared in the shader they are used in. For example, if a vertex shader uses "current_color", then it would need to have a declaration similar to:

```
uniform vec3 current_color;
```

that would allow use of the `current_color` variable within the vertex shader. Note that uniform variables are shared between the fragment shader and vertex shader, and that they are read-only within the shaders. Thus, the fragment shader could access the same `current_color` variable as the vertex shader.

Varying variables are set inside the vertex shader and passed as read-only variables into the fragment shader. Varying variables are interpolated across the vertices. For example, if you have created a triangle and you pass in the colors for the three vertices as solid red, solid green, and solid blue, then the colors will get interpolated based on the distance from each vertex. Pixels on the line from the red to the green vertex would smoothly transition from red to green. The pixels farther away from that line would have a blue component.

Example 02: Materials vertex shader

Here is a real example. This is vertex shader designed to handle the standard lighting model on a per-pixel basis.

```
// This is a vertex shader using the current material settings

varying vec3 normal;

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

    gl_Position = ftransform();
    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
}
```

Example 02: Materials vertex shader

We are passing a normal through to the fragment shader. This shader is called with triangles, so this shader will be called three times. The `normal` variable will be interpolated for each pixel. We are setting `gl_Position` by calling `ftransform()`, which returns the projection modelview matrix

multiplied by the input position. We are also setting both the front and back colors from the predefined variable `gl_Color` (which was set via the equivalent of `glColor3d()` or by using a vertex array), although technically we probably only need to set the front color. OpenGL will convert the correct one of these (depending on whether the fragment is front-facing or back-facing) into the fragment shader predefined variable `gl_Color`.

Local variables are the same as in C or Python. There are int and float types, as well as vectors: `vec2`, `vec3`, and `vec4`. To access the elements of a vector, you can use either the syntax: `v.xyzw` or `v.rgba`. You can also access a partial set:

```
vec4 v = vec4(1., 2., 3., 4.);
vec2 v2 = v.xy;
```

There is also a matrix type. Unlike C, GLSL has no implicit type casting. See the following code:

```
int i;
float f;

// This code is wrong.  It may compile on some cards / drivers and
// fail on others
f = i;

// This is the correct way
f = float(i);
```

Function parameters can be any of the data types, but all function parameters take an additional keyword: `in`, `out`, or `inout`. `in` or `inout` means the variable can be read from. `out` or `inout` means the variable can be written to.

Example 02: Materials fragment shader

Putting all of that together, here is the fragment shader for handling materials and lighting on a per-pixel basis. It performs the basic lighting calculations to generate an output color based on two

lights (OpenGL lights 0 and 1), together with the ambient, diffuse, specular, emission, and shininess properties of the current (front) material. The lighting calculations are the same ones that OpenGL does when using the standard non-shader code, except that the shader does this for each pixel, and the standard pipeline only does this for each vertex.

```
/* This is the corresponding fragment shader for materials */

// Note that this comes from the vertex shader and has been interpolated to
// the correct value for this pixel.
varying vec3 normal;

void getLight(in int light, in vec3 N, inout vec4 ambient, inout vec4 diffuse,
        inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[light].position.xyz);
    vec3 H = normalize(gl_LightSource[light].halfVector.xyz);
    diffuse += max(0.0, dot(N, L)) * gl_FrontLightProduct[light].diffuse;
    specular += pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += gl_FrontLightProduct[light].ambient;
}

void main()
{
    vec3 N = normalize(normal);
    vec4 ambient = vec4(0, 0, 0, 1);
    vec4 diffuse = vec4(0, 0, 0, 1);
    vec4 specular = vec4(0, 0, 0, 1);

    getLight(0, N, ambient, diffuse, specular);
    getLight(1, N, ambient, diffuse, specular);

    ambient.w = 1.;
    diffuse.w = 1.;
    specular.w = 1.;

    gl_FragColor = gl_Color * (diffuse + ambient + gl_FrontMaterial.emission)
        + specular;
    /* Need to copy over alpha */
    gl_FragColor.a = gl_Color.a;
}
```

Example 02: Materials fragment shader

This shader makes use of a number of built-in variables. `gl_LightSource` is an array of

lights containing the position, color components, and a half-vector which is used for calculating specular highlights. The `gl_FrontLightProduct` array contains the product of the `gl_LightSource` array with the front material properties for specular, ambient, and diffuse components. The output of the fragment shader is `gl_FragColor` (another predefined variable).

Geometry transformations

Vertex shaders can be used to modify the geometry. This is useful because you can pass in a minimal set of vertices and pack them with additional data, then use that data to calculate complex shapes.

Example 03: Sphere vertex shader

For example, in the sphere shader, we pass in a quad with all four of the vertices set to the center of the sphere. This gives us the center coordinates to pass on to the fragment shader. We use a pair of variables, `up` and `right` to transform each vertex to the corners of a screen-aligned bounding box containing the sphere.

```
// Vertex shader
uniform vec3 right_vector;
uniform vec3 up_vector;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
varying vec3 point;

void main(void)
{
    // Get billboard attributes
    float right = gl_MultiTexCoord0.x;
    float up = gl_MultiTexCoord0.y;
    float radius = gl_MultiTexCoord0.z;
    radius2 = radius * radius; // compute squared radius

    // We need to project the vertex out to the edge of the square, which
```

```
    // is the following distance:
    // float corner_distance = sqrt(2.0 * radius2);
    // but since we need to normalize the corner vector computed below
    // which has length sqrt(2.0), we can simply use radius as corner distance
    // to compute vertex position of screen-oriented quad.

    // Compute corner vector
    vec3 corner_direction =  up * up_vector + right * right_vector;

    // Calculate vertex of screen-oriented quad (billboard)
    vec4 vertex = vec4(gl_Vertex.xyz + radius * corner_direction,
            gl_Vertex.w);

    // Calculate vertex position in modelview space
    vec4 eye_space_pos = gl_ModelViewMatrix * vertex;

    // Compute sphere position in modelview space
    vec4 tmppos = gl_ModelViewMatrix * gl_Vertex;
    sphere_center = vec3(tmppos) / tmppos.w;

    // Compute ray direction and origin point
    point = vec3(eye_space_pos) / eye_space_pos.w;
    view_direction = normalize(point);

    // Pass fog coordinate
    gl_FogFragCoord = abs(sphere_center.z);

    // Pass the transformed vertex for clipping plane calculations
    gl_ClipVertex = eye_space_pos;

    // Pass color
    color = gl_Color;

    // Pass transformed vertex
    gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

Example 03: Sphere vertex shader

In this example, the up and right directions are passed as part of the texture for the vertex. These variables are passed in as all combinations of 1 and -1 (e.g. 1, 1; -1, 1). Then, `vertex` is set to the center plus the up and right vectors. We also pass in some other useful variables (e.g. `radius2`) to the fragment shader to avoid repetitive computations at the fragment level (which will be called for each pixel). We also set `gl_ClipVertex` to let OpenGL know that we've changed the vertex. It will

use this modified vertex as the basis for the interpolation for the data sent into the fragment shader.

Example 03: Sphere fragment shader

The vertex shader produces a quad bounding the sphere, but we still have to draw an actual sphere. The fragment shader takes the data that is passed in and calculates a sphere-ray intersection to determine which points are actually part of the sphere.

```
// Fragment shader

// 0 is no fog, 1 is linear, 2 is exp, 3 is exp2
uniform float fogging[4];

uniform float perspective;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
varying vec3 point;

const float INV_LOG2 = 1.442695;

void getLight(in int light, in vec3 N, inout vec4 ambient, inout vec4 diffuse,
        inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[light].position.xyz);
    vec3 H = normalize(gl_LightSource[light].halfVector.xyz);
    diffuse += max(0.0, dot(N, L)) * gl_FrontLightProduct[light].diffuse;
    specular += pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += gl_FrontLightProduct[light].ambient;
}

void main(void)
{
    vec3 ray_origin = mix(point, vec3(0., 0., 0.), perspective);
    vec3 ray_direction = mix(vec3(0., 0., 1.), normalize(view_direction),
            perspective);
    vec3 sphere_direction = mix(ray_origin - sphere_center, sphere_center,
            perspective);

    // Calculate sphere-ray intersection
    float b = mix(dot(sphere_direction, ray_direction),
                  dot(ray_direction, sphere_direction), perspective);
```

```
    float position = b * b + radius2 - dot(sphere_direction, sphere_direction);

    // Check if the ray missed the sphere
    if (position < 0.0)
        discard;

    // Calculate nearest point of intersection
    float nearest = mix(sqrt(position) - b, b - sqrt(position), perspective);

    // Calculate intersection point on the sphere surface.  The ray
    // origin is at the quad (center point), so we need to project
    // back towards the user to get the front face.
    vec3 ipoint = nearest * ray_direction + ray_origin;

    // Calculate normal at the intersection point
    vec3 N = normalize(ipoint - sphere_center);

    // Calculate depth in fragment space
    vec2 clipZW = ipoint.z * gl_ProjectionMatrix[2].zw +
        gl_ProjectionMatrix[3].zw;
    gl_FragDepth = 0.5 + 0.5 * clipZW.x / clipZW.y;

    // Calculate lighting
    vec4 diffuse = vec4(0, 0, 0, 1);
    vec4 ambient = vec4(0, 0, 0, 1);
    vec4 specular = vec4(0, 0, 0, 1);

    getLight(0, N, ambient, diffuse, specular);
    getLight(1, N, ambient, diffuse, specular);

    ambient.w = 1.;
    diffuse.w = 1.;
    specular.w = 1.;

    // Calculate fog
    // No fogging
    float fog = fogging[0];

    // Linear fog
    fog += mix(0.0, clamp((gl_Fog.end - gl_FogFragCoord) * gl_Fog.scale,
                0.0, 1.0), fogging[1]);

    // Exp fog
    fog += mix(0.0, clamp(exp(-gl_Fog.density * gl_FogFragCoord), 0.0,
                1.0), fogging[2]);

    // Exp^2 fog: exp(x) = 2^(x/log(2))
    fog += mix(0.0, clamp(exp2(-gl_Fog.density * gl_Fog.density *
                    gl_FogFragCoord * gl_FogFragCoord *
```

```
                    INV_LOG2), 0.0, 1.0), fogging[3]);

    // Calculate final color
    gl_FragColor = mix(gl_Fog.color, color * (diffuse + ambient +
                gl_FrontMaterial.emission) + specular, fog);
}
```

Example 03: Sphere fragment shader

The `getLight()` function is the same as in the materials shader. For spheres, we add in fogging. The fog code we have replicates the fogging modes in the fixed pipeline.

if statements and the mix() function

In graphics cards, `if` statements are typically very slow. On the other hand, the `mix()` function is very fast. To handle perspective versus orthogonal drawing, we use the `perspective` variable in combination with `mix()`.

The `mix()` statement works as an if statement when you restrict your conditional values to 0 and 1. Basically: `result = mix(x, y, condition)` sets `result` to `y` if `condition` is 1 and `x` if `condition` is 0.

Passing data

You can pass data which doesn't vary across vertices (e.g. a highlight color) by using `uniform` variables. However, when you need to pass distinct data on a per-vertex basis, that won't work. Instead, you need to make full use of the variables which do vary on a per-vertex basis:

- Coordinate data (4 floats): you can use the fourth component of the coordinates to pass data, but you need to reset the fourth coordinate to 1 before operating on it.
- Color data (4 floats): if you have a fixed alpha, then you can use the color's alpha value to pass in different information.
- Texture data (4 floats): this is arbitrary data you can use for what you want.

- Normals (3 floats): if you don't need the normal data, you can pass in your own values.

Variable lights

In Maestro, by default we enable only lights 0 and 1. Through the GUI, the user can turn off either one of those lights. Through commands, the user can turn on any combination of the eight OpenGL lights. We optimize the default case (as shown in the previous examples). For any other combination of lights, we fall back to a generalized lighting mechanism.

Example 04: Variable lights fragment shader

To handle the lights, we pass in an array indicating whether or not each light is on (1 for on, 0 for off). In GLSL, `if` statements are typically slow. Therefore, we want to minimize `if` statements. Specifically, we want to avoid doing eight if statements in a row. To do that, we make heavy use of the `mix()` statement. We do a straight multiply against `enabled`, and we use `mix()` to generate valid vectors.

```
/* This is the corresponding fragment shader for materials */

uniform float light_enabled[8];
varying vec3 normal;

const vec3 UNIT_VEC3 = vec3(1.0, 0.0, 0.0);

void getLight(in int light, in float enabled, in vec3 N, inout vec4 ambient,
        inout vec4 diffuse, inout vec4 specular)
{
    vec3 L = normalize(mix(UNIT_VEC3, gl_LightSource[light].position.xyz,
                enabled));
    vec3 H = normalize(mix(UNIT_VEC3, gl_LightSource[light].halfVector.xyz,
                enabled));
    diffuse += enabled * max(0.0, dot(N, L)) *
        gl_FrontLightProduct[light].diffuse;
    specular += enabled * pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += enabled * gl_FrontLightProduct[light].ambient;
}
```

```
void main()
{
    vec3 N = normalize(normal);
    vec4 ambient = vec4(0, 0, 0, 1);
    vec4 diffuse = vec4(0, 0, 0, 1);
    vec4 specular = vec4(0, 0, 0, 1);

    getLight(0, light_enabled[0], N, ambient, diffuse, specular);
    getLight(1, light_enabled[1], N, ambient, diffuse, specular);
    getLight(2, light_enabled[2], N, ambient, diffuse, specular);
    getLight(3, light_enabled[3], N, ambient, diffuse, specular);
    getLight(4, light_enabled[4], N, ambient, diffuse, specular);
    getLight(5, light_enabled[5], N, ambient, diffuse, specular);
    getLight(6, light_enabled[6], N, ambient, diffuse, specular);
    getLight(7, light_enabled[7], N, ambient, diffuse, specular);

    ambient.w = 1.;
    diffuse.w = 1.;
    specular.w = 1.;

    gl_FragColor = gl_Color * (diffuse + ambient + gl_FrontMaterial.emission)
        + specular;
    /* Need to copy over alpha */
    gl_FragColor.a = gl_Color.a;
}
```

Example 04: Variable lights fragment shader

Debugging

Different drivers accept slightly different syntaxes for GLSL. For example, the nvidia drivers tend to be fairly permissive in what they accept. The code which loads the shaders in Maestro is set up to display any error messages we get from the driver at any stage (vertex shader compilation, fragment shader compilation, and linking). It can be useful to try shaders on different drivers: nvidia, Mesa v7.7 or later, and ATI. There are also some programs which can compile shaders and generate error messages for invalid syntax. I've used GLSL Validate from 3D labs (Windows or Linux under wine).

One error that doesn't get reported well is either having too many main() functions in a shader or not enough. The vertex shader and the fragment shader each need to have exactly one main() function. nvidia in particular generates an unhelpful message (something like "couldn't link").

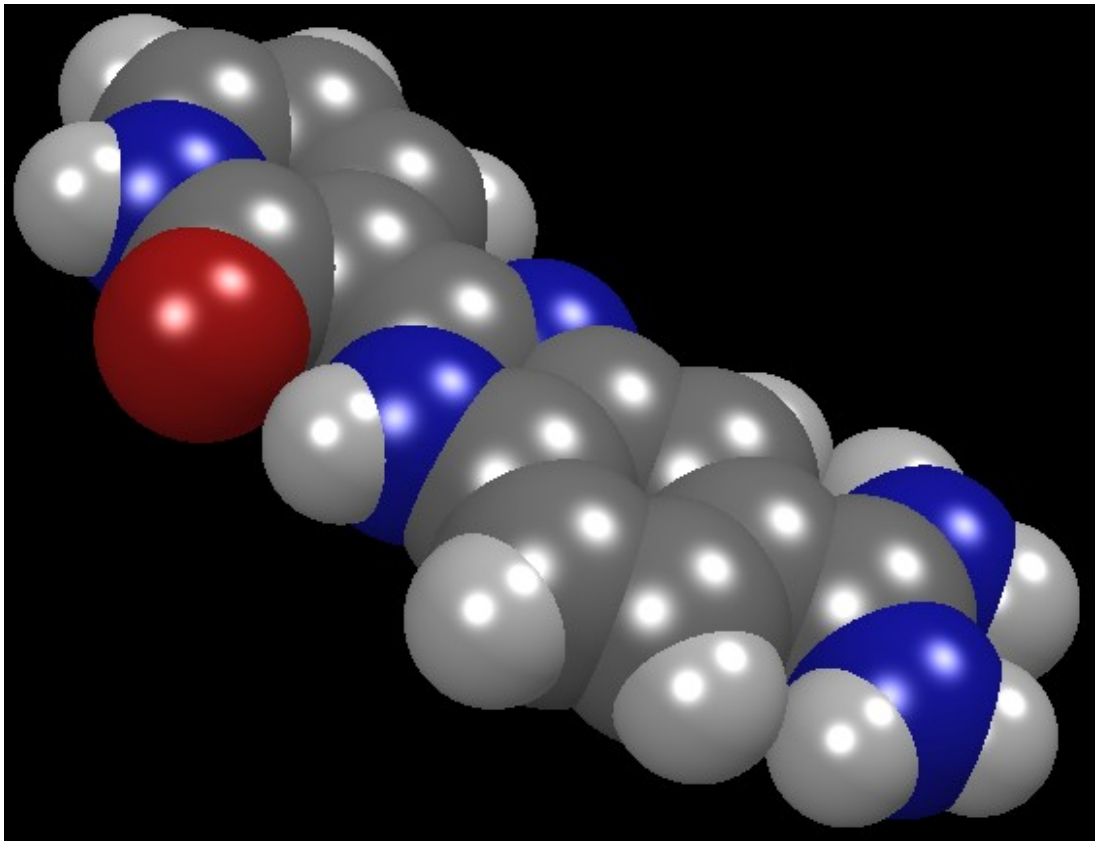
There are some OpenGL debuggers available, but I haven't used any of them so, I can't comment on their quality. The other way of debugging is to set specific colors to catch specific things (e.g. red for error #1, blue for error #2, yellow for error #3, white for no error). I have used this technique to identify when a calculation was not producing the expected result.

If you are specifically trying to debug a vertex shader which rewrites the vertices, you can temporarily switch the fragment shader to simply pass through the input color. That will show you specifically where the vertices are, which might be useful.

Effects

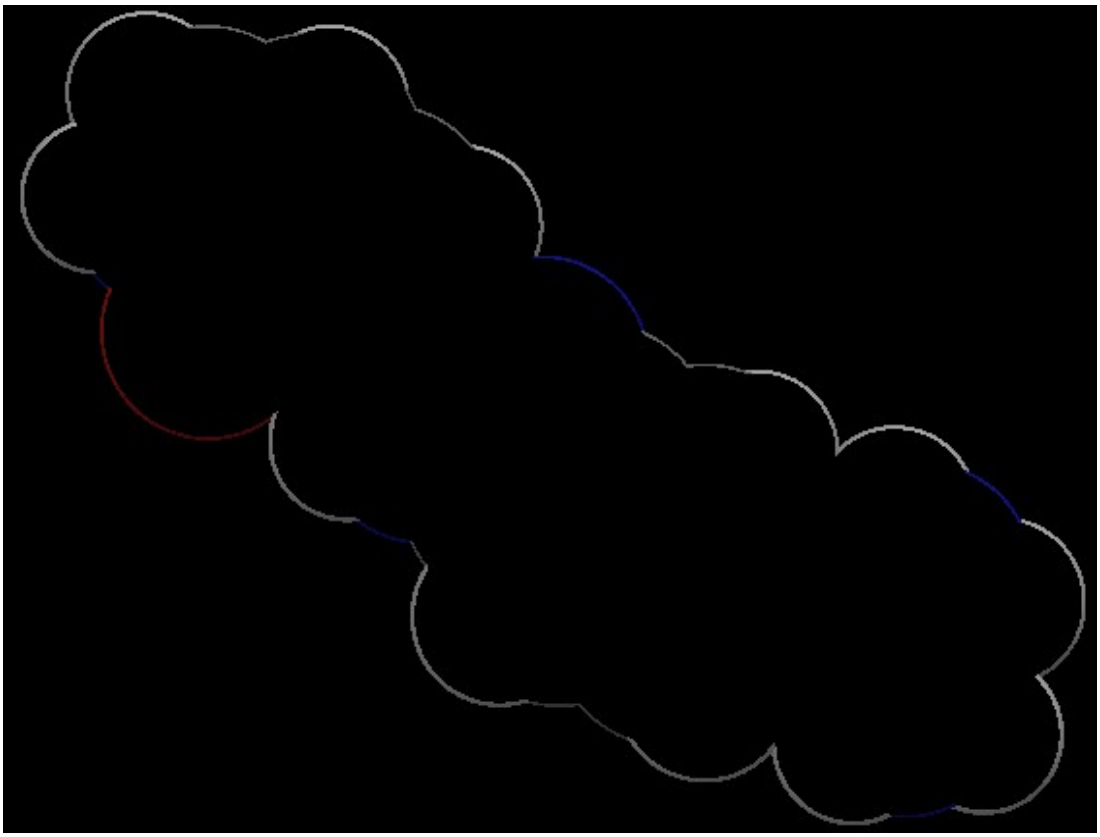
There are a number of effects which are possible using shaders.

For reference, here is an image using the standard shader:



Example 01: Outlines

This shader produces an outline around the outside of the structure.



The idea with this shader is to set the interior of each sphere to black without adjusting the Z value, so that the spheres erase each other. Given the standard atom sphere fragment shader, the basic changes are to check if we're on the border (position < 0.1), and only then do we adjust the Z value and do the lighting calculations. Otherwise, we set the color to black.

```
// Fragment shader

// 0 is no fog, 1 is linear, 2 is exp, 3 is exp2
uniform float fogging[4];

uniform float perspective;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
```

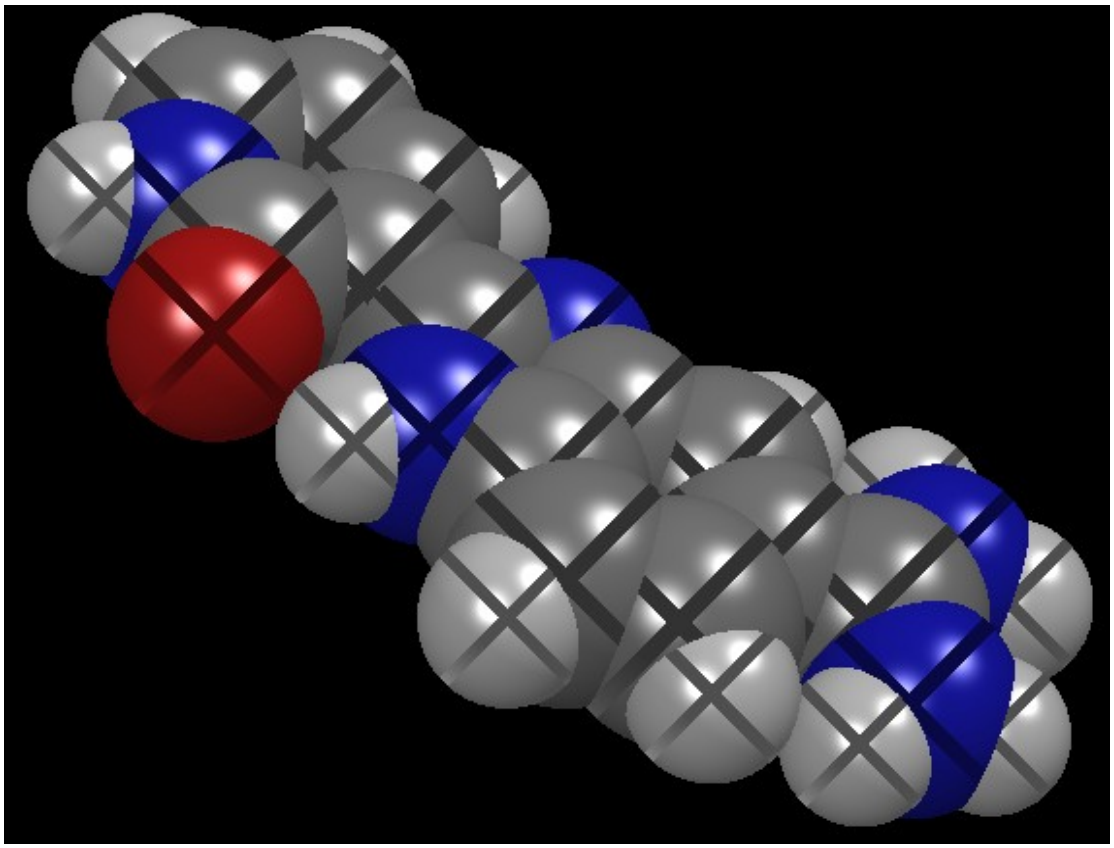
```
varying vec3 point;

const float INV_LOG2 = 1.442695;

void getLight(in int light, in vec3 N, inout vec4 ambient, inout vec4 diffuse,
        inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[light].position.xyz);
    vec3 H = normalize(gl_LightSource[light].halfVector.xyz);
    diffuse += max(0.0, dot(N, L)) * gl_FrontLightProduct[light].diffuse;
    specular += pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += gl_FrontLightProduct[light].ambient;
}

void main(void)
{
    vec3 ray_origin = mix(point, vec3(0., 0., 0.), perspective);
    vec3 ray_direction = mix(vec3(0., 0., 1.), normalize(view_direction),
            perspective);
    vec3 sphere_direction = mix(ray_origin - sphere_center, sphere_center,
            perspective);

    // Calculate sphere-ray intersection
    float b = mix(dot(sphere_direction, ray_direction),
                  dot(ray_direction, sphere_direction), perspective);
    float position = b * b + radius2 - dot(sphere_direction, sphere_direction);

    // Check if the ray missed the sphere
    if (position < 0.0)
        discard;

    if (position < 0.1) {
        // Calculate nearest point of intersection
        float nearest = mix(sqrt(position) - b, b - sqrt(position), perspective);

        // Calculate intersection point on the sphere surface.  The ray
        // origin is at the quad (center point), so we need to project
        // back towards the user to get the front face.
        vec3 ipoint = nearest * ray_direction + ray_origin;

        // Calculate normal at the intersection point
        vec3 N = normalize(ipoint - sphere_center);

        // Calculate depth in fragment space
        vec2 clipZW = ipoint.z * gl_ProjectionMatrix[2].zw +
            gl_ProjectionMatrix[3].zw;
        gl_FragDepth = 0.5 + 0.5 * clipZW.x / clipZW.y;
```

```
        // Calculate lighting
        vec4 diffuse = vec4(0, 0, 0, 1);
        vec4 ambient = vec4(0, 0, 0, 1);
        vec4 specular = vec4(0, 0, 0, 1);

        getLight(0, N, ambient, diffuse, specular);
        getLight(1, N, ambient, diffuse, specular);

        ambient.w = 1.;
        diffuse.w = 1.;
        specular.w = 1.;

        // Calculate fog
        // No fogging
        float fog = fogging[0];

        // Linear fog
        fog += mix(0.0, clamp((gl_Fog.end - gl_FogFragCoord) * gl_Fog.scale,
                    0.0, 1.0), fogging[1]);

        // Exp fog
        fog += mix(0.0, clamp(exp(-gl_Fog.density * gl_FogFragCoord), 0.0,
                    1.0), fogging[2]);

        // Exp^2 fog: exp(x) = 2^(x/log(2))
        fog += mix(0.0, clamp(exp2(-gl_Fog.density * gl_Fog.density *
                        gl_FogFragCoord * gl_FogFragCoord *
                        INV_LOG2), 0.0, 1.0), fogging[3]);

        // Calculate final color
        gl_FragColor = mix(gl_Fog.color, color * (diffuse + ambient +
                    gl_FrontMaterial.emission) + specular, fog);
    }
    else {
        gl_FragColor = vec4(0., 0., 0., 1.);
    }
}
```

Example 01: Outline fragment shader

Example 02: X

This example puts an X on all of the spheres.



This shader makes use of the up and right data which is passed into the vertex shader. Since these values are not exported from the vertex shader in the standard shaders, this shader requires both a new vertex shader and a new fragment shader.

```
// Vertex shader
uniform vec3 right_vector;
uniform vec3 up_vector;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
varying vec3 point;

varying float u1;
varying float r1;
```

```
void main(void)
{
    // Get billboard attributes
    float right = gl_MultiTexCoord0.x;
    float up = gl_MultiTexCoord0.y;
    float radius = gl_MultiTexCoord0.z;
    radius2 = radius * radius; // compute squared radius

    u1 = up;
    r1 = right;

    // We need to project the vertex out to the edge of the square, which
    // is the following distance:
    // float corner_distance = sqrt(2.0 * radius2);
    // but since we need to normalize the corner vector computed below
    // which has length sqrt(2.0), we can simply use radius as corner distance
    // to compute vertex position of screen-oriented quad.

    // Compute corner vector
    vec3 corner_direction =  up * up_vector + right * right_vector;

    // Calculate vertex of screen-oriented quad (billboard)
    vec4 vertex = vec4(gl_Vertex.xyz + radius * corner_direction,
            gl_Vertex.w);

    // Calculate vertex position in modelview space
    vec4 eye_space_pos = gl_ModelViewMatrix * vertex;

    // Compute sphere position in modelview space
    vec4 tmppos = gl_ModelViewMatrix * gl_Vertex;
    sphere_center = vec3(tmppos) / tmppos.w;

    // Compute ray direction and origin point
    point = vec3(eye_space_pos) / eye_space_pos.w;
    view_direction = normalize(point);

    // Pass fog coordinate
    gl_FogFragCoord = abs(sphere_center.z);

    // Pass the transformed vertex for clipping plane calculations
    gl_ClipVertex = eye_space_pos;

    // Pass color
    color = gl_Color;

    // Pass transformed vertex
    gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

Example 02: X vertex shader

Notice that we are simply copying the `up` and `right` variables to `u1` and `r1`.

In the fragment shader, we produce a cheap unlighted effect for the X by inverting the normal.

```
// Fragment shader

// 0 is no fog, 1 is linear, 2 is exp, 3 is exp2
uniform float fogging[4];

uniform float perspective;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
varying vec3 point;

varying float u1;
varying float r1;

const float INV_LOG2 = 1.442695;

void getLight(in int light, in vec3 N, inout vec4 ambient, inout vec4 diffuse,
        inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[light].position.xyz);
    vec3 H = normalize(gl_LightSource[light].halfVector.xyz);
    diffuse += max(0.0, dot(N, L)) * gl_FrontLightProduct[light].diffuse;
    specular += pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += gl_FrontLightProduct[light].ambient;
}

void main(void)
{
    vec3 ray_origin = mix(point, vec3(0., 0., 0.), perspective);
    vec3 ray_direction = mix(vec3(0., 0., 1.), normalize(view_direction),
            perspective);
    vec3 sphere_direction = mix(ray_origin - sphere_center, sphere_center,
            perspective);

    // Calculate sphere-ray intersection
    float b = mix(dot(sphere_direction, ray_direction),
                  dot(ray_direction, sphere_direction), perspective);
    float position = b * b + radius2 - dot(sphere_direction, sphere_direction);
```

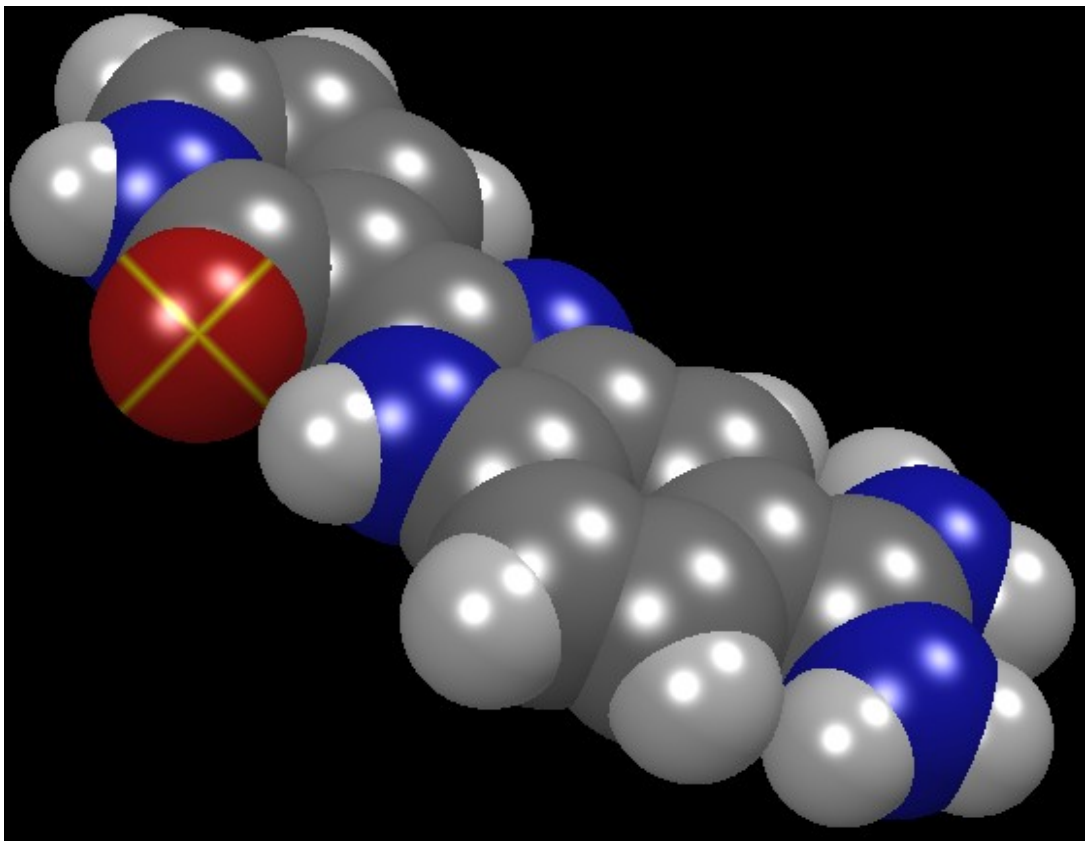
```
    // Check if the ray missed the sphere
    if (position < 0.0)
        discard;

    // Calculate nearest point of intersection
    float nearest = mix(sqrt(position) - b, b - sqrt(position), perspective);

    // Calculate intersection point on the sphere surface.  The ray
    // origin is at the quad (center point), so we need to project
    // back towards the user to get the front face.
    vec3 ipoint = nearest * ray_direction + ray_origin;

    // Calculate normal at the intersection point
    vec3 N = normalize(ipoint - sphere_center);

    if (abs(abs(u1) - abs(r1)) < 0.1)
        N = -N;

    // Calculate depth in fragment space
    vec2 clipZW = ipoint.z * gl_ProjectionMatrix[2].zw +
        gl_ProjectionMatrix[3].zw;
    gl_FragDepth = 0.5 + 0.5 * clipZW.x / clipZW.y;

    // Calculate lighting
    vec4 diffuse = vec4(0, 0, 0, 1);
    vec4 ambient = vec4(0, 0, 0, 1);
    vec4 specular = vec4(0, 0, 0, 1);

    getLight(0, N, ambient, diffuse, specular);
    getLight(1, N, ambient, diffuse, specular);

    ambient.w = 1.;
    diffuse.w = 1.;
    specular.w = 1.;

    // Calculate fog
    // No fogging
    float fog = fogging[0];

    // Linear fog
    fog += mix(0.0, clamp((gl_Fog.end - gl_FogFragCoord) * gl_Fog.scale,
                0.0, 1.0), fogging[1]);

    // Exp fog
    fog += mix(0.0, clamp(exp(-gl_Fog.density * gl_FogFragCoord), 0.0,
                1.0), fogging[2]);

    // Exp^2 fog: exp(x) = 2^(x/log(2))
    fog += mix(0.0, clamp(exp2(-gl_Fog.density * gl_Fog.density *
```

```
                    gl_FogFragCoord * gl_FogFragCoord *
                    INV_LOG2), 0.0, 1.0), fogging[3]);

    // Calculate final color
    gl_FragColor = mix(gl_Fog.color, color * (diffuse + ambient +
                gl_FrontMaterial.emission) + specular, fog);
}
```

Example 02: X fragment shader

Example 03: Yellow X on red atoms

The next example is the same as the previous example, except that the X only appears on red atoms (e.g. oxygen with the element coloring scheme).



We just expand the "X" test to include a color check. Instead of:

```
    vec4 color2 = color;
    if (abs(abs(u1) - abs(r1)) < 0.1) {
        float factor = abs(abs(u1) - abs(r1)) * 10.;
        // Set the color to yellow
        color2 = mix(vec4(1., 1., 0., 1.), color, factor);
```

```
    }
```

Clip from example 02

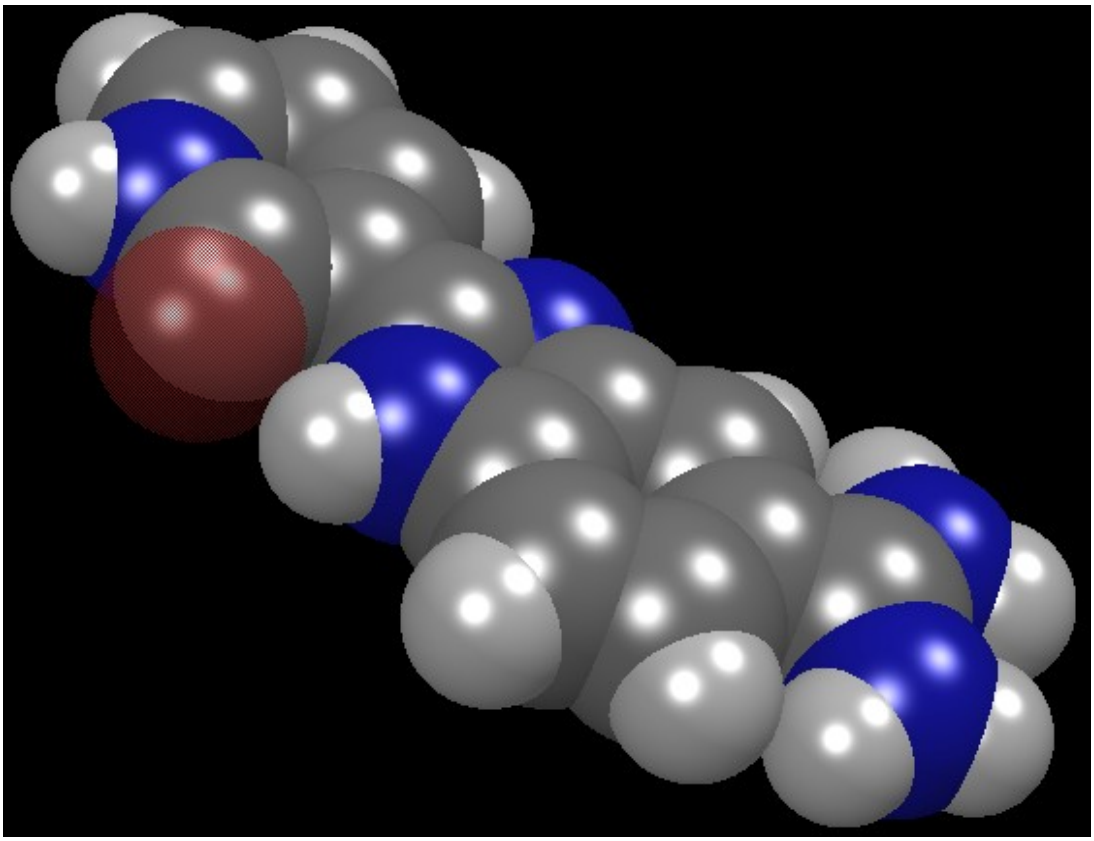
We do:

```
    vec4 color2 = color;
    // If the sphere is mostly red, then draw an X on it
    if (abs(abs(u1) - abs(r1)) < 0.1 && color.r > 0.7 && color.g < 0.3 &&
            color.b < 0.3) {
        float factor = abs(abs(u1) - abs(r1)) * 10.;
        // Set the color to yellow
        color2 = mix(vec4(1., 1., 0., 1.), color, factor);
    }
```

Example 03: Yellow X for red atoms fragment shader

Example 04: Transparent atoms

This effect produces a cheesy transparency (50% transparency) on red atoms.



We simply discard every other pixel for red atoms. This example uses the standard vertex

shader, so only the fragment shader is shown.

```
// Fragment shader

// 0 is no fog, 1 is linear, 2 is exp, 3 is exp2
uniform float fogging[4];

uniform float perspective;

varying vec4 color;
varying vec3 view_direction;
varying vec3 sphere_center;
varying float radius2;
varying float depth_level;
varying vec3 point;

const float INV_LOG2 = 1.442695;

void getLight(in int light, in vec3 N, inout vec4 ambient, inout vec4 diffuse,
        inout vec4 specular)
{
    vec3 L = normalize(gl_LightSource[light].position.xyz);
    vec3 H = normalize(gl_LightSource[light].halfVector.xyz);
    diffuse += max(0.0, dot(N, L)) * gl_FrontLightProduct[light].diffuse;
    specular += pow(max(0.0, dot(N, H)), gl_FrontMaterial.shininess) *
        gl_FrontLightProduct[light].specular;
    ambient += gl_FrontLightProduct[light].ambient;
}

void main(void)
{
    vec3 ray_origin = mix(point, vec3(0., 0., 0.), perspective);
    vec3 ray_direction = mix(vec3(0., 0., 1.), normalize(view_direction),
            perspective);
    vec3 sphere_direction = mix(ray_origin - sphere_center, sphere_center,
            perspective);

    // Calculate sphere-ray intersection
    float b = mix(dot(sphere_direction, ray_direction),
                  dot(ray_direction, sphere_direction), perspective);
    float position = b * b + radius2 - dot(sphere_direction, sphere_direction);

    // Check if the ray missed the sphere
    if (position < 0.0)
        discard;

    // Draw a 50% cheesy transparency effect on red (oxygen) atoms
    if (color.r > 0.7 && color.g < 0.3 && color.b < 0.3 &&
            (int(abs(gl_FragCoord.x + gl_FragCoord.y)) % 2) == 1)
```

```
    discard;

// Calculate nearest point of intersection
float nearest = mix(sqrt(position) - b, b - sqrt(position), perspective);

// Calculate intersection point on the sphere surface.  The ray
// origin is at the quad (center point), so we need to project
// back towards the user to get the front face.
vec3 ipoint = nearest * ray_direction + ray_origin;

// Calculate normal at the intersection point
vec3 N = normalize(ipoint - sphere_center);

// Calculate depth in fragment space
vec2 clipZW = ipoint.z * gl_ProjectionMatrix[2].zw +
    gl_ProjectionMatrix[3].zw;
gl_FragDepth = 0.5 + 0.5 * clipZW.x / clipZW.y;

// Calculate lighting
vec4 diffuse = vec4(0, 0, 0, 1);
vec4 ambient = vec4(0, 0, 0, 1);
vec4 specular = vec4(0, 0, 0, 1);

getLight(0, N, ambient, diffuse, specular);
getLight(1, N, ambient, diffuse, specular);

ambient.w = 1.;
diffuse.w = 1.;
specular.w = 1.;

// Calculate fog
// No fogging
float fog = fogging[0];

// Linear fog
fog += mix(0.0, clamp((gl_Fog.end - gl_FogFragCoord) * gl_Fog.scale,
            0.0, 1.0), fogging[1]);

// Exp fog
fog += mix(0.0, clamp(exp(-gl_Fog.density * gl_FogFragCoord), 0.0,
            1.0), fogging[2]);

// Exp^2 fog: exp(x) = 2^(x/log(2))
fog += mix(0.0, clamp(exp2(-gl_Fog.density * gl_Fog.density *
                gl_FogFragCoord * gl_FogFragCoord *
                INV_LOG2), 0.0, 1.0), fogging[3]);

// Calculate final color
gl_FragColor = mix(gl_Fog.color, color * (diffuse + ambient +
```

```
                gl_FrontMaterial.emission) + specular, fog);
}
```

Example 04: Transparent red atom fragment shader

Resources

This is a list of useful resources for GLSL.

OpenGL v3.2 (GLSL 1.5) quick reference card:

http://www.khronos.org/files/opengl-quick-reference-card.pdf

GLSL v1.20 quick reference card:

http://www.cs.brown.edu/courses/cs123/resources/glslQuickRef.pdf

Shader Toy (allows experimenting with different shaders in very recent web browsers):

http://www.iquilezles.org/apps/shadertoy/