Default Attributes

```
// default vertex attributes provided by BufferGeometry
attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
```

Using Uniforms in GLSL

Set color of all fragments to red, or the value of a uniform variable.

```
// Fragment shader
uniform vec3 objColor;

void main() {
    //gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
    gl_FragColor = vec4(objColor,1.0);
}
```

Offset the position of all vertices by a uniform variable.

```
// Vertex shader
uniform float xValue;
uniform float yValue;
uniform float zValue;

void main() {
    vec3 location = vec3(xValue,yValue,zValue);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position + location, 1.0 );
}
```

Modify geometry

```
// Vertex shader
uniform float zValue;

void main() {
    float zm = abs(10.0 - abs(zValue));
    float z = clamp(position.z,-zm,zm);
    vec3 newPosition = vec3(position.xy,z);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
}
```

Using Attributes in GLSL

Color the fragments based on the value of the normal vector

```
// vertex shader
uniform float zValue;
out vec3 nNormal;

void main() {
    nNormal = normalize(normal);
    float zm = abs(10.0 - abs(zValue));
    float z = clamp(position.z,-zm,zm);
    vec3 newPosition = vec3(position.xy,z);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
}

// fragment shader
in vec3 nNormal;
void main() {
    gl_FragColor = vec4( nNormal, 1.0 );
}
```

Or the position

```
// vertex shader
uniform float zValue;
out vec3 nPosition;

void main() {
    float zm = abs(10.0 - abs(zValue));
    float z = clamp(position.z,-zm,zm);
    vec3 newPosition = vec3(position.xy,z);
    nPosition = abs(newPosition/4.0);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( newPosition, 1.0 );
}

// fragment shader

in vec3 nPosition;

void main() {
    gl_FragColor = vec4( nPosition, 1.0 );
}
```

One hemisphere one color, the other hemisphere another color

Since the sphere is centered at the origin, we can use any one of {x,y,z} to determine which hemisphere the fragment is in. Make the calculation in the vertex shader and pass a single value that determines the binary color choice out to the fragment shader.

```
// vertex shader
out float hemisphere;       // out variables can only be float, vec2, vec3, vec4, mat2, mat3, ma

void main() {
    hemisphere = step(0.0,position.z);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}

// fragment shader
uniform vec3 objColor;
in float hemisphere;

void main() {
    vec3 invColor = 1.0 - objColor;
    if(hemisphere > 0.5) {
        gl_FragColor = vec4(objColor,1.0);
    } else {
        gl_FragColor = vec4(invColor,1.0);
    }
}
```

Or a smooth transition from one color to the other

```
// vertex shader
out float hemisphere;

void main() {
    hemisphere = smoothstep(-2.0,2.0,position.z);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}

// fragment shader
uniform vec3 objColor;
in float hemisphere;

void main() {
    vec3 invColor = 1.0 - objColor;
    gl_FragColor = vec4(mix(objColor,invColor,hemisphere),1.0);
}
```

Striped Sphere (Beach Ball)

```
// vertex shader
out float theta;    // This will be interpolated between vertices and passed to the fragment sha

void main() {
    // calculate the angle (i.e. latitude or longitude) of the vertex from it's position
    // this assumes it is on the surface of a sphere centered at the origin
    theta = atan(position.y,position.x);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}

// fragment shader
#define M_2PI 6.283185307179586

uniform vec3 objColor;
uniform float xValue;
in float theta;

void main() {
    vec3 invColor = 1.0 - objColor;
    float dtheta = radians(abs(360.0/xValue));
    // use the live editor here (https://thebookofshaders.com/05/ or http://editor.thebookofshad
    // to visualize the following functions
    float stripe = step(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0);
    // float stripe = smoothstep(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0,0.2);
    vec3 color = mix(objColor,invColor,stripe);
    gl_FragColor = vec4(color,1.0);
}
```

Add time to the mix (stripes that move)

Needs the updated code that adds updateable to the Sphere class. Time passed in is in seconds

```
// vertex shader
out float theta;

void main() {
    // calculate the angle (i.e. latitude or longitude) of the vertex from it's position
    // this assumes it is on the surface of a sphere centered at the origin
    theta = atan(position.y,position.x);
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}

// fragment shader
#define M_2PI 6.283185307179586

uniform vec3 objColor;
uniform float xValue;
uniform float time;
in float theta;

void main() {
    float theta_time = theta + time;
    vec3 invColor = 1.0 - objColor;
    float dtheta = radians(abs(360.0/xValue));
    // use the live editor here (https://thebookofshaders.com/05/ or http://editor.thebookofshac
    // to visualize the following functions
    //float stripe = step(mod(theta_time/M_2PI,dtheta)/dtheta-0.5,0.0);
    float stripe = smoothstep(mod(theta_time/M_2PI,dtheta)/dtheta-0.5,0.0,0.2);
    vec3 color = mix(objColor,invColor,stripe);
    gl_FragColor = vec4(color,1.0);
}
```

Oscillating sphere

Use time in the vertex shader to oscillate the position of each vertex. This one approximates the lowest volumetric normal mode of oscillation of a sphere.

```
// vertex shader
#define M_2PI 6.283185307179586

uniform float time;
out float theta;

void main() {
    theta = atan(position.y,position.x);
    // make our own scale transformation matrix
    float scaleFactor = 1.0 - 0.07*sin(M_2PI * time);
    mat4 S = mat4(scaleFactor);   // same values along the diagonal
    S[3][3] = 1.0;
    gl_Position = projectionMatrix * modelViewMatrix * S * vec4( position, 1.0 );
}

// fragment shader
#define M_2PI 6.283185307179586

uniform vec3 objColor;
uniform float xValue;
in float theta;

void main() {
    vec3 invColor = 1.0 - objColor;
    float dtheta = radians(abs(360.0/xValue));
    float stripe = step(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0);
    //float stripe = smoothstep(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0,0.2);
    vec3 color = mix(objColor,invColor,stripe);
    gl_FragColor = vec4(color,1.0);
}
```

Marshmallow mode!

```
// vertex shader
#define M_2PI 6.283185307179586
#define MAGNITUDE 0.07
uniform float time;
out float theta;

void main() {
    theta = atan(position.y,position.x);
    float phi = atan(position.y, position.z);
    // make our own scale transformation matrix
    float scaleFactor = MAGNITUDE*sin(M_2PI * time);
    mat4 S = mat4(1.0);
    S[0][0] = 1.0 - scaleFactor*cos(2.0*theta);
    S[1][1] = 1.0 - scaleFactor*sin(2.0*theta);
    S[2][2] = 1.0 - scaleFactor*sin(2.0*phi);
    S[3][3] = 1.0;
    gl_Position = projectionMatrix * modelViewMatrix * S * vec4( position, 1.0 );
}

// fragment shader
#define M_2PI 6.283185307179586

uniform vec3 objColor;
uniform float xValue;
in float theta;

void main() {
    vec3 invColor = 1.0 - objColor;
    float dtheta = radians(abs(360.0/xValue));
    float stripe = step(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0);
    //float stripe = smoothstep(mod(theta/M_2PI,dtheta)/dtheta-0.5,0.0,0.2);
    vec3 color = mix(objColor,invColor,stripe);
    gl_FragColor = vec4(color,1.0);
}
```

Lighting

Ambient Light

Needs

- ia: ambient light intensity/color
- ka: ambient reflection coefficient
- ca: object color
  Does not depend on the location of the light source or of the camera. It is a constant value for all fragments.

```
uniform vec3 objColor;

// normally would be uniforms
// Ambient
vec3 ia = vec3(0.9,0.9,0.6) * 0.2;
vec3 ka = vec3(1.0,1.0,1.0);

void main() {
    vec3 ca = objColor;
    vec3 ambient = ia * ka * ca;
    vec3 color = ambient;
    gl_FragColor = vec4( color, 1.0 );
}
```

Diffuse Light

Needs:

- id: diffuse light intensity/color
- kd: diffuse reflection coefficient
- cd: object color for diffuse component
- theta: angle between the normal and the light direction
  Which to get this needs:
  - normalized normal vector
  - normalized direction of or to the light source
    We'll assume a directional light source for now and leave the point source for an exercise.

```
// Vertex shader
out vec3 wsNormal;

void main() {
    vec4 wsn = modelMatrix * vec4(normal,0.0);
    wsNormal = wsn.xyz;
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}


// Fragment shader
uniform vec3 objColor;

in vec3 wsNormal;
uniform float xValue;
uniform float yValue;
uniform float zValue;


// normally would be uniforms
// Ambient
vec3 ia = vec3(0.9,0.9,0.6) * 0.2;
vec3 ka = vec3(1.0,1.0,1.0);
// Diffuse
vec3 id = vec3(1.0,1.0,1.0) * 0.6;
vec3 kd = vec3(1.0,1.0,1.0);
// use the same color for both ambient and diffuse

void main() {
    vec3 ca = objColor;
    vec3 cd = ca;
    vec3 wsLight = vec3(xValue,yValue,zValue);  // this is in world coordinates
    vec3 wsNormalizedNormal = normalize(wsNormal);
    vec3 wsNormalizedLight = normalize(wsLight);
    vec3 ambient = ia * ka * ca;
    vec3 diffuse = id * kd * ca * max(dot(wsNormalizedNormal,wsNormalizedLight),0.0);
    vec3 color = ambient + diffuse;
    gl_FragColor = vec4( color, 1.0 );
}
```

Specular Light

Needs to calculate the reflection of the light direction about the normal. This is the direction of the light that would be reflected off the surface if it were a perfect mirror. The angle between this direction and the direction to the camera is the angle of reflection. The specular reflection is the intensity of the light times the specular reflection coefficient times the cosine of the angle of reflection raised to the power of the shininess coefficient. The shininess coefficient is a measure of how shiny the object is. The higher the value, the smaller the specular reflection will be. The specular reflection is added to the ambient and diffuse reflections to get the final color.

```
// Vertex shader
out vec3 wsNormal;
out vec3 wsPosition;

void main() {
    vec4 wsn = modelMatrix * vec4(normal,0.0);
    vec4 wsp = modelMatrix * vec4(position,1.0);
    wsNormal = wsn.xyz;
    wsPosition = wsp.xyz;
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}

// Fragment shader
uniform vec3 objColor;

in vec3 wsNormal;
in vec3 wsPosition;  // world space interpolated position of fragment (per fragment specular)
uniform float xValue;
uniform float yValue;
uniform float zValue;
// uniform vec3 cameraPosition; // this is declared for us, and is in world coordinates

// Ambient
vec3 ia = vec3(0.9,0.9,0.6) * 0.2;
vec3 ka = vec3(1.0,1.0,1.0);
// Diffuse
vec3 id = vec3(1.0,1.0,1.0) * 0.6;
vec3 kd = vec3(1.0,1.0,1.0);
// Specular
vec3 is = vec3(1.0,1.0,1.0) * 0.4;
vec3 ks = vec3(1.0,1.0,1.0);
float shinyness = 50.0;

void main() {
    vec3 ca = objColor;
    vec3 cd = ca;
    vec3 wsLight = vec3(xValue,yValue,zValue);  // this is in world coordinates
    vec3 wsNormalizedNormal = normalize(wsNormal);
    vec3 wsNormalizedLight = normalize(wsLight);
    vec3 ambient = ia * ka * ca;
    vec3 diffuse = id * kd * ca * max(dot(wsNormalizedNormal,wsNormalizedLight),0.0);

    vec3 wsEye = normalize(cameraPosition - wsPosition);
```

```
    vec3 wsReflect = 2.0 * dot(wsNormalizedLight,wsNormalizedNormal) * wsNormalizedNormal - wsN
    vec3 specular = is * ks * ca * pow(max(dot(wsReflect,wsEye),0.0),shinyness);

    vec3 color = ambient + diffuse + specular;
    gl_FragColor = vec4( color, 1.0 );
}
```

See "Phong reflection model" at Wikipedia.