Generative Methods, Group 5, week 4

CODE:

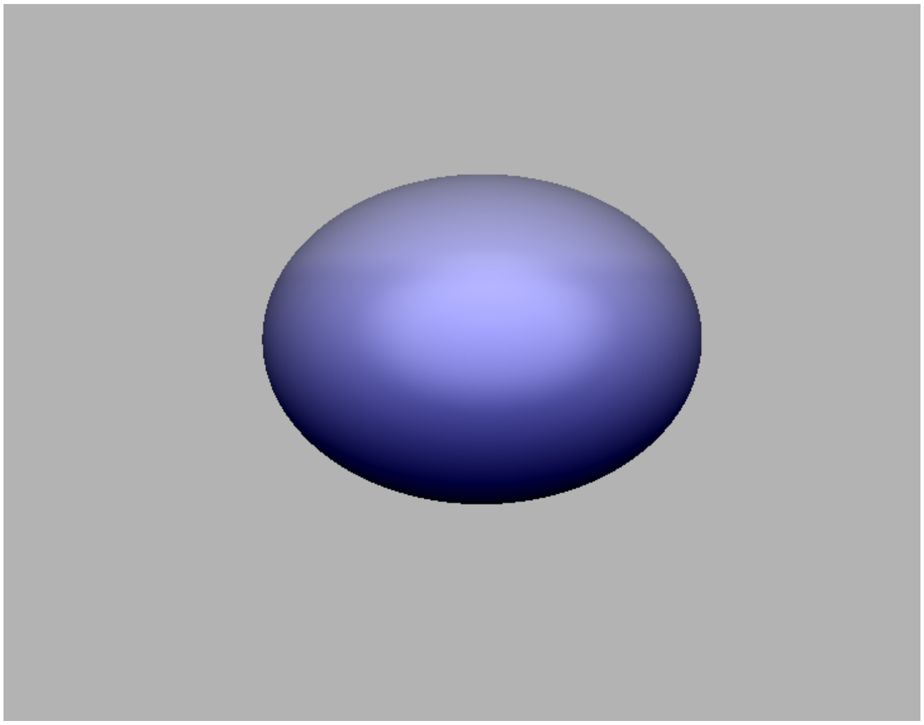
```
#version 410

// ----------------------------------------------------------------------------------------
-------
// An almost fully functional ray tracer for implicit surfaces. Presently, it renders
only a
// single sphere and it takes constant length steps. Your tasks are to
// 1. Compute the gradient correctly   //done
// 2. make it more efficient by taking adaptive steps.   //done
// 3. Change the sphere to an ellipsoid. //done
// 4. Make a composition by blending at least three implicits.
// 5. Draw a periodic structure (e.g. gyroid or grid of spheres)
// 6. NON-MANDATORY: Add shadows, ambient occlusion, reflections or other rendering
features.
// ----------------------------------------------------------------------------------------
-------


uniform sampler2DRect gtex;
uniform float T;        // Time
uniform vec4 eye_pos;
uniform vec3 y_dir;     // What direction considered up

in vec3 ray_dir;        // Direction of ray
in vec3 ray_origin;     // Origin of ray: where the ray intersects the bounding box
in vec3 norm;           // Normal at ray origin
out vec4 fragColor;     // Output color

const float thresh = 1e-4;                      // Required surface proximity.
const float inv_thresh_times_2 = 1/2.0*thresh;    // Used for central differences

// For distance fields, Lipschitz const should be 1:
const float lipschitz_const = 20*3.5;

// Compute distance. Right now this function computes the distance to a single
// sphere

/*
```

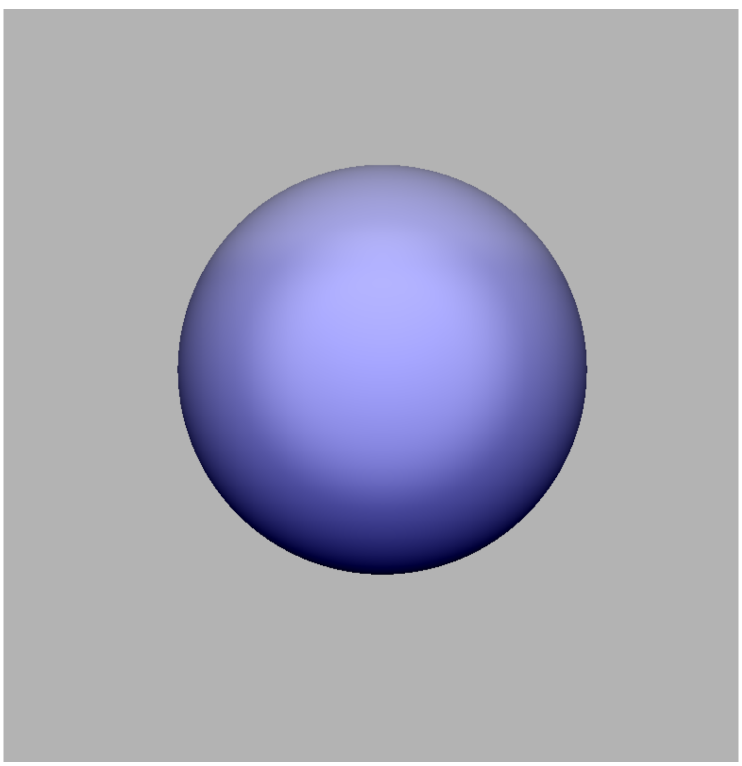
```
float dist(vec3 p) {    //SPHERE
   vec3 c = vec3(0.0, -0.1, 0.0);
   float r = 0.5;
   return length(p-c) - r;
}
*/
/*
float dist(vec3 p) {
   vec3 c = vec3(0.0, -0.1, 0.0);
   vec3 r = vec3(0.5, 0.3, 0.4); // radius for x, y, and z axes
   vec3 q = p - c;               // translate to origin
   return length(q/r) - 1.0;   // scale and subtract 1
}*/

float sphere(vec3 p, float val) {
   vec3 c = val*vec3(0.0, -0.1, 0.0);
   float r = 0.5;
   return length(p-c) - r;
}

float box(vec3 p) {
   vec2 s = vec2(0.5, 0.5);
   vec2 d = abs(p.xy) - s;
   return min(max(d.x,d.y),0.0) + length(max(d,0.0));
}

float smin(float a, float b){
   float m = min(a,b);
   float k = 0.2;
   float M = max(k - length(a-b), 0);
   return m - ((M*M)/4*k);
}

float dist(vec3 p) {
   float sphere_coord  = sphere(p, 5.0);
   float box_coord     = box(p);
   float sphere_coord2 = sphere(p, -3.0);
   return smin(sphere_coord, sphere_coord2);
}

/*
float dist(vec3 p){
```

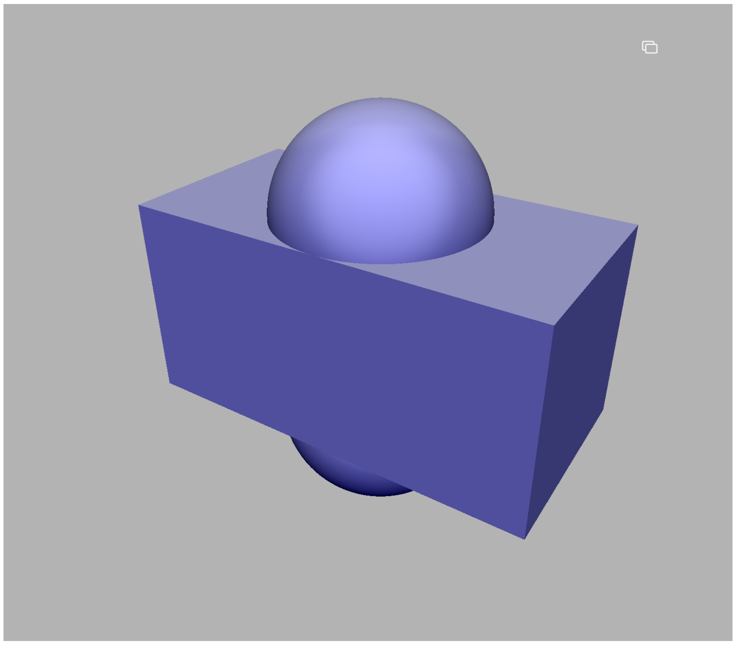
```
    float scale = 15; // Adjust this value to change the scale of the gyroid
    p *= scale;
    return sin(p.x)*cos(p.y) + sin(p.y)*cos(p.z) + sin(p.z)*cos(p.x);
}
*/

// Compute the gradient. For many implcits, we can easily compute the
// analytic derivatives, but this is a general solution that does not
// require us to find the derivative for every new implicit.
vec3 dist_grad(vec3 p) {  //part 1
    float delta = thresh;
    float two_delta = inv_thresh_times_2;
    float dx = (dist(vec3(p.x + delta, p.y, p.z)) - dist(vec3(p.x - delta, p.y, p.z)))
* two_delta;
    float dy = (dist(vec3(p.x, p.y + delta, p.z)) - dist(vec3(p.x, p.y - delta, p.z)))
* two_delta;
    float dz = (dist(vec3(p.x, p.y, p.z + delta)) - dist(vec3(p.x, p.y, p.z - delta)))
* two_delta;
    return vec3(dx,dy,dz);
}



vec3 shade(vec3 p, vec3 n, float t) {
    // This function computes the shading as a sum of three terms:
    // ambient, diffuse, and specular. There are diffuse and specular
    // contributions for two lights - a skylight above and a light source
    // in the eye. The amount of specular reflection is computed via a
    // smooth step function. Note that specular is just a highlight.
    // It would not be hard to add secondary rays and use these to compute
    // specular reflections and cast shadows, but it would reduce performance.

    // Color is initialized to the ambient color
    vec3 color = vec3(0.05,0.05, 0.15);
    // Add diffuse and specular contributions for light source in the eye
    float d = dot(n,normalize(eye_pos.rgb));
    color +=  vec3(0.5,0.5,0.75) * d;
    color +=  vec3(0.1,0.1,0.1) * smoothstep(0.75,0.95,d);
    d = dot(n, y_dir);
    // Add diffuse and specular contribution for skylight
    color +=  vec3(0.25,0.25,0.2) * d;
    color +=  vec3(0.05,0.05,0.04) * smoothstep(0.4,0.6,d);
    return color;//*clamp(1.0-1.0*t, 0.0, 1.0);
```

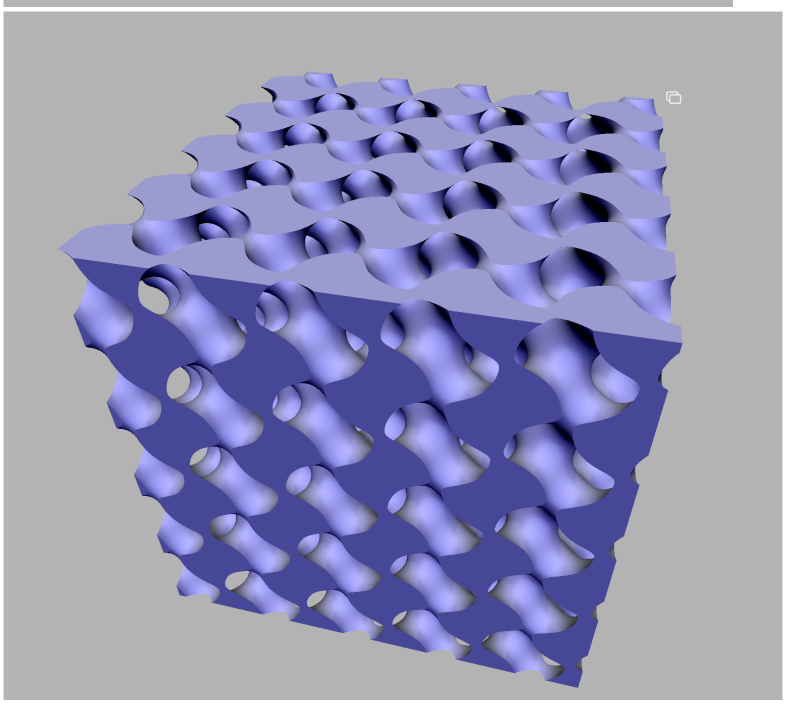
```
   // Isophote rendering below - great for debugging.
   //    float x = cos(20.0*dot(n, normalize(eye_pos.rgb)));
   //    return vec3(x*x);
}

void main()
{
   vec3 r = normalize(ray_dir); // Ray direction
   vec3 p0; // ray starting point
   vec3 p1 = texture(gtex, gl_FragCoord.xy).xyz; // ray terminus
   float d; // pseudo-distance aka value of implicit

   // We test whether the normal in the pixel points towards the eye
   // or away from the eye.
   if (dot(norm,r) <-0.01) {
       // If the normal points toward the eye, the pixel contains the
       // front facing normal, and the ray origin is simply the position
       // of the pixel passed from the vertex shader.
       p0 = ray_origin;
       d = dist(p0);
       // If we are inside the implicit, we shade using the normal
       // stored in the pixel.
       if (d<0)  {
           fragColor.rgb = shade(p0, norm, 0);
           // Uncomment line below to clip bounding box as near plane.
           // fragColor.rgb = vec3(.2,0,0);
           return;
       }
   }
   else {
       // Otherwise, the near clipping plane intersects the triangle
       // geometry, and set the ray starting point to be on the near
       // clipping plane.
       p0 = eye_pos.rgb + r/dot(r, -normalize(eye_pos.rgb));
       d = dist(p0);
       if (d<0)  {
           // Now, if we are inside the solid, we have clipped it and set
           // the color to a very very dark red.
           fragColor.rgb = vec3(.2,0,0);
           return;
       }
```

```
    }

    vec3 ray_span = p1-p0;
    r= normalize(ray_span);
    float t_max = length(ray_span);

    float t = 0.0; // distance along ray
    for (int i=0;i<10000;++i) {
        // Update ray parameter such that we take uniform small steps.
        // This is SLOW. Change it such that you use the distance value
        // and Lipschitz constant to choose a good step length.    adaptive????
        t += d/lipschitz_const;

        // Compute position along ray.
        vec3 p = p0 + t * r;

        // Next, we compute the value of the implicit
        d = dist(p);

        // If the point p is now outside the domain and we did not hit the
        // surface, break out of the loop
        if(t>t_max && d>-thresh)
            break;

        // If we are sufficiently close to the surface, we set the
        // fragment color and return
        if(abs(d) < thresh) {
            fragColor.rgb = shade(p, normalize(dist_grad(p)), t);
            return;
        }
    }

    // We missed ... and set the fragment color to background color!
    fragColor.rgb = vec3(0.7);// * dot(vec2(0.5,0.5),smoothstep(0.45,0.55,
fract(gl_FragCoord.xy / 30.0)));
}
```









1) how blending works:

Blending is the process of combining the color of a pixel output by the fragment shader with the color of the corresponding pixel that is already on the screen . The way that these colors are combined is based on the blending function and the source and destination blend factors, which are set up in the OpenGL/WebGL.

2) how adaptive step ray casting works:

Adaptive ray casting is a technique used to render a volume. Adaptive steps' purpose is to render the volume as efficiently as possible. The more steps a ray takes the more expensive that ray is going to be. On the other hand, a ray that takes very large steps run the risk of overshooting the volume.

Adaptive ray casting works by taking steps in the direction of the ray but with the size being equal to the distance to the volume. This ensures that we never overshoot the volume. This is done repeatedly until the steps get smaller than a certain threshold, whereafter the algorithm will return the shade.

The algorithm can also be adapted to situations where there is no distance field. It works for functions that are Lipschitz continuous. A function is Lipschitz continuous if there is exists a constant, k, where k times the absolute difference between the points, $k \cdot |x_1 - x_0|$, is greater than the absolute value of the slope/gradient, $|f(x_1) - f(x_0)|$ everywhere on the function. This can be used to safely take a step towards the function, because we know that the distance is less than f/k.

Lipschitz inequality: $|f(x_1) - f(x_0)| < k \cdot |x_1 - x_0|$

3) how your periodic structure was modeled:

A periodic structure is made when the implicit function used to define the structure is periodic. This defines surfaces or volumes with values that repeat at regular intervals and thus enables the generation of complex patterns and structures with a high degree of regularity.

The periodic structure we have chosen to render is the gyroid.

Its functions is so defined:

$$G(x,y,z) = \sin(x)\cos(y) + \sin(y)\cos(z) + \sin(z)\cos(z).$$