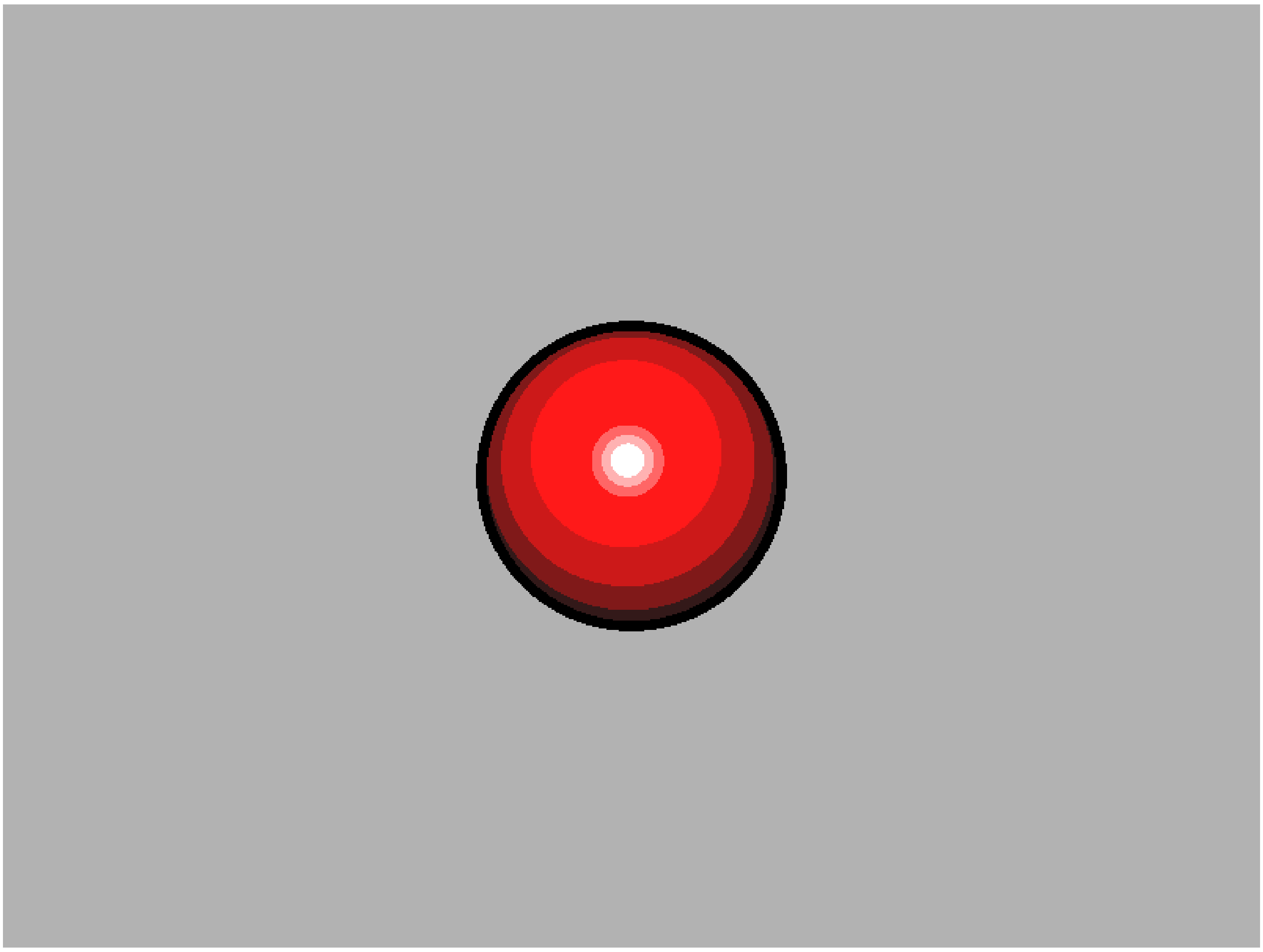
xdPixel

RAY MARCHING

RAY MARCHING 101 – PART 3

JANUARY 7, 2016 | ADMIN | LEAVE A COMMENT



URL: http://glslsandbox.com/e#29884.0

```
// Ray Marching Tutorial (With Toon Shading)
// By: Brandon Fogerty
// bfogerty at gmail dot com
// xdpixel.com

// Ray Marching is a technique that is very similar to Ray Tracing.
```

```
// In both techniques, you cast a ray and try to see if the ray intersects
// with any geometry.  Both techniques require that geometry in the scene
// be defined using mathematical formulas.  However the techniques differ
// in how the geometry is defined mathematically.  As for ray tracing,
// we have to define geometry using a formula that calculates the exact
// point of intersection.  This will give us the best visual result however
// some types of geometry are very hard to define in this manner.
// Ray Marching using distance fields to decribe geometry.  This means all
// we need to know to define a kind of geometry is how to mearsure the distance
// from any arbitrary 3d position to a point on the geometry.  We iterate or "march"
// along a ray until one of two things happen.  Either we get a resulting distance
// that is really small which means we are pretty close to intersecting with some kind
// of geometry or we get a really huge distance which most likely means we aren't
// going to intersect with anything.

// Ray Marching is all about approximating our intersection point.  We can take a pretty
// good guess as to where our intersection point should be by taking steps along a ray
// and asking "Are we there yet?".  The benefit to using ray marching over ray tracing is
// that it is generally much easier to define geometry using distance fields rather than
// creating a formula to analytically find the intersection point.  Also, ray marching makes
// certain effects like ambient occlusion almost free.  It is a little more work to compute
// the normal for geometry.  I will cover more advanced effects using ray marching in a late
// For now,  we will simply ray march a scene that consists of a single sphere at the origi
// We will not bother performing any fancy shading to keep things simple for now.

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 resolution;
uniform float time;

//-------------------------------------------------------------------------------------------
// The sphere function takes in a point along the ray
// we are marching and a radius.  The sphere function
// will then return the distance from the input point p
// to the closest point on the sphere.  The sphere is assumed
// to be centered on the origin which is (0,0,0).
float sphere( vec3 p, float radius )
{
    return length( p ) - radius;
}

//-------------------------------------------------------------------------------------------
// The map function is the function that defines our scene.
// Here we can define the relationship between various objects
// in our scene.  To keep things simple for now, we only have a single
// sphere in our scene.
float map( vec3 p )
{
    return sphere( p, 3.0 );
}

//-------------------------------------------------------------------------------------------
// This function will return the normal of any point in the scene.
// This function is pretty expensive so if you need the normal, you should
// call this function once and store the result.  Essentially the way it works
// is by offsetting the input surface point "p" along each axis and then determining the
// change in distance at each new point along each axis.
vec3 getNormal( vec3 p )
{
    vec3 e = vec3( 0.001, 0.00, 0.00 );

    float deltaX = map( p + e.xyy ) - map( p - e.xyy );
    float deltaY = map( p + e.yxy ) - map( p - e.yxy );
    float deltaZ = map( p + e.yyx ) - map( p - e.yyx );

```

```
    return normalize( vec3( deltaX, deltaY, deltaZ ) );
}

//-------------------------------------------------------------------------------------------
// The trace function is our integration function.
// Given a starting point and a direction, the trace
// function will return the distance from a point on the ray
// to the closest point on an object in the scene.  In order for
// the trace function to work properly, we need functions that
// describe how to calculate the distance from a point to a point
// on a geometric object.  In this example, we have a sphere function
// which tells us the distance from a point to a point on the sphere.
float trace( vec3 origin, vec3 direction, out vec3 p )
{
    float totalDistanceTraveled = 0.0;

    // When ray marching,  you need to determine how many times you
    // want to step along your ray.  The more steps you take, the better
    // image quality you will have however it will also take longer to render.
    // 32 steps is a pretty decent number.  You can play with step count in
    // other ray marchign examples to get an intuitive feel for how this
    // will affect your final image render.
    for( int i=0; i <32; ++i)
    {
        // Here we march along our ray and store the new point
        // on the ray in the "p" variable.
        p = origin + direction * totalDistanceTraveled;

        // "distanceFromPointOnRayToClosestObjectInScene" is the
        // distance traveled from our current position along
        // our ray to the closest point on any object
        // in our scene.  Remember that we use "totalDistanceTraveled"
        // to calculate the new point along our ray.  We could just
        // increment the "totalDistanceTraveled" by some fixed amount.
        // However we can improve the performance of our shader by
        // incrementing the "totalDistanceTraveled" by the distance
        // returned by our map function.  This works because our map function
        // simply returns the distance from some arbitrary point "p" to the closest
        // point on any geometric object in our scene.  We know we are probably about
        // to intersect with an object in the scene if the resulting distance is very small.
        float distanceFromPointOnRayToClosestObjectInScene = map( p );
        totalDistanceTraveled += distanceFromPointOnRayToClosestObjectInScene;

        // If our last step was very small, that means we are probably very close to
        // intersecting an object in our scene.  Therefore we can improve our performance
        // by just pretending that we hit the object and exiting early.
        if( distanceFromPointOnRayToClosestObjectInScene < 0.0001 )
        {
            break;
        }

        // If on the other hand our totalDistanceTraveled is a really huge distance,
        // we are probably marching along a ray pointing to empty space.  Again,
        // to improve performance,  we should just exit early.  We really only want
        // the trace function to tell us how far we have to march along our ray
        // to intersect with some geometry.  In this case we won't intersect with any
        // geometry so we will set our totalDistanceTraveled to 0.00.
        if( totalDistanceTraveled > 10000.0 )
        {
            totalDistanceTraveled = 0.0000;
            break;
        }
    }

    return totalDistanceTraveled;
}

```

```
//-------------------------------------------------------------------------------------
// This function essentially simulates a texture with sharp gradients going from completely
// black to pure white.  To see a visual example of this function, check out my ramp shader.
// http://glslsandbox.com/e#23880.0
float calculateRampCoefficient( float t, int stripeCount )
{
    float fStripeCount = float(stripeCount);
    float modifiedT = mod( floor( t * fStripeCount ), fStripeCount );
    float rampCoefficient = mix( 0.1, 1.0, modifiedT / (fStripeCount-1.0) );

    return rampCoefficient;
}

//-------------------------------------------------------------------------------------
// Standard Blinn lighting model.
// This model computes the diffuse and specular components of the final surface color.
vec3 calculateLighting(vec3 pointOnSurface, vec3 surfaceNormal, vec3 lightPosition, vec3 cam
{
    vec3 fromPointToLight = normalize(lightPosition - pointOnSurface);
    float diffuseStrength = clamp( dot( surfaceNormal, fromPointToLight ), 0.0, 1.0 );

    diffuseStrength = calculateRampCoefficient( diffuseStrength, 4 );
    vec3 diffuseColor = diffuseStrength * vec3( 1.0, 0.0, 0.0 );
    vec3 reflectedLightVector = normalize( reflect( -fromPointToLight, surfaceNormal ) );

    vec3 fromPointToCamera = normalize( cameraPosition - pointOnSurface );
    float specularStrength = pow( clamp( dot(reflectedLightVector, fromPointToCamera), 0.0,
    specularStrength = calculateRampCoefficient(specularStrength, 4);
    // Ensure that there is no specular lighting when there is no diffuse lighting.
    specularStrength = min( diffuseStrength, specularStrength );
    vec3 specularColor = specularStrength * vec3( 1.0 );

    vec3 finalColor = diffuseColor + specularColor;

    // Draw a thick silhouette around our object
    if( dot( fromPointToCamera, surfaceNormal ) < 0.2 )
    {
        finalColor = vec3( 0.0 );
    }

    return finalColor;
}

//-------------------------------------------------------------------------------------
// This is where everything starts!
void main( void )
{
    // gl_FragCoord.xy is the coordinate of the current pixel being rendered.
    // It is in screen space.  For example if you resolution is 800x600, gl_FragCoord.xy
    // could be (300,400).  By dividing the fragcoord by the resolution, we get normalized
    // coordinates between 0.0 and 1.0.  I would like to work in a -1.0 to 1.0 space
    // so I multiply the result by 2.0 and subtract 1.0 from it.
    // if (gl_FragCoord.xy / resolution.xy) equals 0.0, then 0.0 * 2.0 - 1.0 = -1.0
    // if (gl_FragCoord.xy / resolution.xy) equals 1.0, then 1.0 * 2.0 - 1.0 =  1.0
    vec2 uv = ( gl_FragCoord.xy / resolution.xy ) * 2.0 - 1.0;

    // I am assuming you have more pixels horizontally than vertically so I am multiplying
    // the x coordinate by the aspect ratio.  This means that the magnitude of x coordinate
    // be larger than 1.0.  This allows our image to not look squashed.
    uv.x *= resolution.x / resolution.y;

    // We would like to cast a ray through each pixel on the screen.
    // In order to use a ray, we need an origin and a direction.
    // The cameraPosition is where we want our camera to be positioned.  Since our sphere wi
    // positioned at (0,0,0), I will push our camera back by -10 units so we can see the sph
    vec3 cameraPosition = vec3( 0.0, 0.0, -10.0 );

```

```
    // We will need to shoot a ray from our camera's position through each pixel.  To do thi
    // we will exploit the uv variable we calculated earlier, which describes the pixel we c
    // currently rendering, and make that our direction vector.
    vec3 cameraDirection = normalize( vec3( uv.x, uv.y, 1.0) );

    // Now that we have our ray defined,  we need to trace it to see how far the closest poi
    // in our world is to this ray.  We will simply shade our scene.
    vec3 pointOnSurface;
    float distanceToClosestPointInScene = trace( cameraPosition, cameraDirection, pointOnSur

    // We will now shade the sphere if our ray intersected with it.
    vec3 finalColor = vec3(0.7);
    if( distanceToClosestPointInScene > 0.0 )
    {
        // Move our light around on both the x and y axis.
        float lx = mix( -1.5, 1.5, sin(time) * 0.5 + 0.5);
        float ly = 3.0 + mix( -1.5, 1.5, sin(time * 1.3) * 0.5 + 0.5);
        vec3 lightPosition = vec3( lx, ly, -10.0 );
        vec3 surfaceNormal = getNormal( pointOnSurface );
        finalColor = calculateLighting( pointOnSurface, surfaceNormal, lightPosition, camera
    }

    // And voila!  We are done!  We should now have a sphere!  =D
    // gl_FragColor is the final color we want to render for whatever pixel we are currently
    gl_FragColor = vec4( finalColor, 1.0 );
}
```

Share It!