Ray Tracing Quaternion Julia Sets on the GPU*

Keenan Crane
University of Illinois at Urbana-Champaign

November 7, 2005

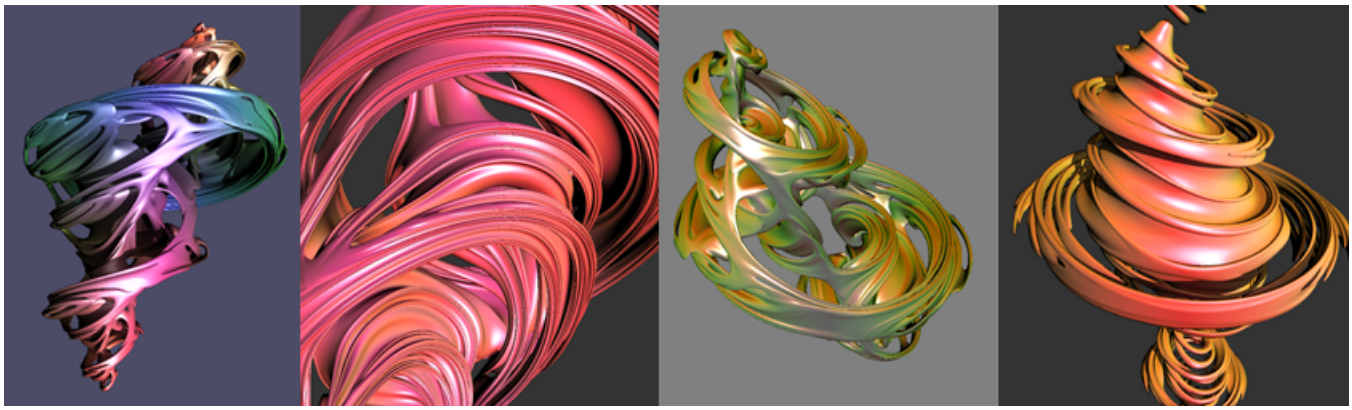

Figure 1: Quaternion Julia sets ray traced in less than a second on a pair of GeForce 7800 GTX graphics cards running in SLI.

1 Introduction

This project takes advantage of the floating point power of recent GPUs to quickly visualize quaternion Julia sets. First, a few "frequently asked questions" about the project:

Q: *What in the name of Gaston Maurice Julia are quaternion Julia sets?!*

A: A Julia set is a kind of fractal, or object that looks somewhat like itself at many different scales. The quaternion Julia set is a four-dimensional version of this fractal (more detail below). Playing with a small number of parameters for a set results in a huge number of complex, beautiful shapes such as the one shown above.

Q: *Why would you want to render these things on the GPU?*

A: In general there are two problems with quaternion Julia sets:

1. They take forever to visualize
2. They're not useful for anything

We can solve problem number one using ray tracing and the GPU: the GPU is really good at doing lots of things in parallel as long as none of those things depend on each other. Ray tracing is a great application of the GPU because each ray can work independently. Problem number two, however, is an open research problem.

*Originally appeared as an article on *DevMaster.net*.

Q: *But can't you use the quaternion Julia set to render taffy?*

A: Yes! In fact, Figure 2 shows a comparison of taffy on the Santa Cruz boardwalk with a rendering of a Julia set. Alas, Pixar is not expected to cast taffy in a lead role anytime in the near future.

Figure 2: Left: *saltwater taffy.* Right: *quaternion Julia set. Photo courtesy of Yisong Yue.*

2 Julia Sets in 2D

Before explaining Julia sets in the quaternions, let's take a look at the traditional two-dimensional Julia sets. These eponymous fractals were the invention of French mathematician Gaston Julia. The fractal exists in the complex plane, a coordinate system where the x component of a point's location corresponds to a real number, and the y component corresponds to an imaginary number (i.e., a single number x such that x^2 is less than zero). Each point in the complex plane is a complex number of the form $z = a + bi$. A Julia set (technically a filled-in Julia set) is the set of all points z_0 in the complex plane for which the sequence

$$z_{n+1} = z_n^2 + c$$

has a finite limit (i.e., does not get arbitrarily large as n approaches infinity). Geometrically this recurrence pushes points around on the complex plane, since each iteration will produces a new complex number / point on the plane. Points included in a Julia set will hover around the origin, while points not in the set will shoot off to infinity. Different constants c specify a particular Julia set, and can be thought of as placing differently shaped boundaries on the plane which prevent points inside the boundary from escaping.

In practice, to determine whether a given point diverges we start to compute points in the sequence above and see how quickly their magnitude (geometrically: their distance from the origin) gets large. In fact, if the magnitude of any point in the sequence exceeds a value of 2 (geometrically: the point leaves a circular region of radius 2 centered at the origin), the sequence diverges and z_0 is not in the set. By applying the convergence test to all the points corresponding to pixels in an image, we get an image like the one shown in Figure 3.

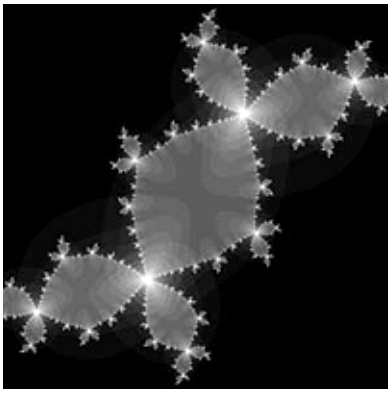

Figure 3: The "Rabbit Ears" Julia set ($c = -0.12 + 0.75i$). Grey values correspond to the number of iterations used to guess if a point is in the set.

3 Julia Sets in 4D

The quaternion Julia sets are almost exactly like the original Julia sets, except that points in a set are from a four-dimensional complex space called the quaternions—this variation on the fractal was first explored by Norton [Nor89]. The quaternions are similar to the complex numbers, except that a quaternion has three imaginary components: $z = a + bi + cj + dk$. Plugging one of these points into the same recurrence will give you similar behavior: some of the points will "escape" (their magnitude will approach infinity), and some will move around inside a region close to the origin. However, visualizing this fractal becomes tougher in 4D for a couple reasons.

First, we can't visualize the set of 4D points directly as we can with the 2D fractal. Instead, we have to come up with a way to "project" the set into a lower dimension. One way to look at a high dimensional object in a lower dimension is to take slices of the object by finding its intersection with several lower-dimensional planes. For example, if we intersect a three-dimensional sphere with a two-dimensional plane, we get a two-dimensional circle whose radius depends on the location of the plane relative to the sphere. By looking at a number of these cross sections we could infer the true shape of the sphere in 3D. Putting an egg through an egg slicer gives a similar result: each blade defines a slicing plane through the egg, and we can infer the shape of the yolk from the shape of yellow regions in each slice. So if we take 3D slices of a 4D Julia set we can get some idea of what it really looks like (Figure 5).

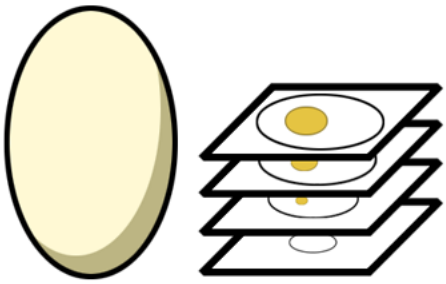

Figure 4: 2D slices of a 3D egg.

In the case of the Julia set, we specify a 3D slice by picking three basis quaternions. The idea of a basis in a 2D plane is simple: pick two basis vectors which point in different directions in the plane. We can then get to any point in the plane by starting at the origin and moving some distance in the direction of the first basis vector and some other distance in the direction second basis vector. (Think about where we would be able to go if we had only picked one basis vector). If we instead pick two (and only two) 3D vectors, then we still have a basis for a 2D plane, but this is a 2D plane sitting in a 3D space. We can now get to any point in a slice of 3D space by moving different distances along our two 3D basis vectors. If we need to know where a point (x, y) from our 2D plane sits in the 3D space, we simply start at the origin of the 3D space, move a distance x along the first basis vector, and a distance y along the second basis vector. Similarly, if at any time we need to know where a 3D point (x, y, z) is in the 4D quaternion space, we start at the origin and move a distance x along our first basis quaternion, a distance y along our second basis quaternion, and a distance z along our third basis quaternion.

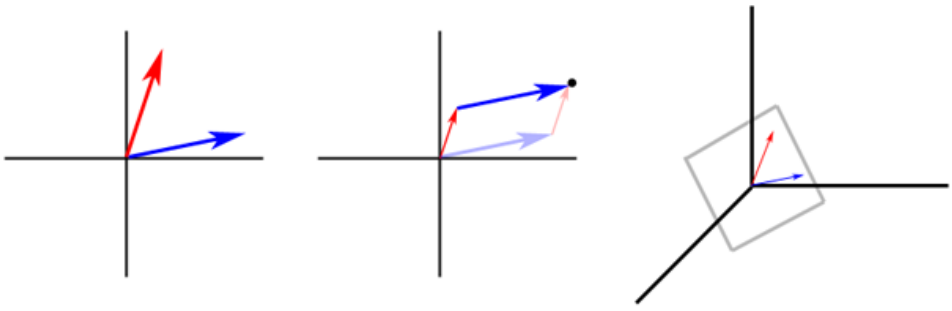

Figure 5: Left: *two 2D vectors which define a basis for the 2D plane. Middle: any point in the 2D plane can be thought of as some combination of these two vectors.* Right: *two 3D vectors which define a basis for a 2D subset of points in 3D space.*

The second problem with the 4D version of the Julia set is that our scheme for generating an image of the set is much less efficient in 4D. While we could test the convergence of every point on a 4D grid, the cost would be the same as rendering n stacks of n 2D images, or n^2 times more expensive, and would consume n^2 times as much memory. Even if we used a grid of only three dimensions and tested the convergence of points in the corresponding slice there would be wasted effort: in the end we only want to look at the surface of the 3D projection, and we don't care about interior points.

Since we are only interested in looking at the surface of a 3D object, ray tracing seems like it's worth investigating: ray tracing is often useful for detecting the thing that's closest to the viewer. However, there is no known way to analytically find the nearest point along a ray which intersects a Julia set (this is the usual method for ray tracing simple objects like spheres or cones). Instead, we could take small steps along the ray, testing the convergence at each point, and stop when we reach the first point which does not diverge. This might work, but we would have to evaluate a huge number of points for every pixel rendered. Additionally, there is no obvious way to shade the surface once we find an intersection, because the fractal surface does not yield a surface normal.

4 Unbounding Volumes

Fortunately, there is a distance estimator which will tell us, given any point z in the quaternion space, the distance to the closest point in the Julia set. This distance estimator can be used to accelerate the ray tracing process using unbounding volumes, a method presented by Hart et al [HSK89]. Note that the formula given for the distance estimator in this paper is a slight misprint—it should be:

$$d(z) = \frac{|z_n|}{2|z'_n|} \log |z_n|$$

where z' is just another sequence of points:

$$z'_{n+1} = 2 z_n z'_n$$

which can be computed alongside z_n. The first point in this sequence should always be $z'_0 = 1 + 0i + 0j + 0k$. Just as the magnitude or norm of a complex number is its distance from the origin of the complex plane, the magnitude of a quaternion $z = a + bi + cj + dk$ is simply its distance from the origin of quaternion space $\sqrt{a^2 + b^2 + c^2 + d^2}$.

The idea of unbounding volumes is simple: imagine we're tracing a ray through quaternion space, looking for the first point in the set along this ray. At any time we can query the distance estimator, and it will tell us the distance from our ray's origin to the closest point in the Julia set (in any direction—not necessarily in the direction of the ray). We can then safely take a step of this same distance along the ray direction, because we are guaranteed not to skip over any points in the set (for if we passed a point, that point would have been the closest one, and the distance estimate would have been smaller).

The image sequence in Figure 6 illustrates several steps of this process. If we pay attention to the first ray tracing step in the animation, we see the benefit of this method: a ray originating far from the Julia set will move much more quickly than if we had taken many small, uniform steps. This is because the initial few unbounding steps account for a large part of the distance between the ray origin and the surface.

5 Zeno's Paradox

Because we are always taking steps which only cover part of the remaining distance, we can never actually reach the surface (except in the unusual case that the closest point in the set is directly along our ray). This phenomenon is known as *Zeno's paradox of motion*. Zeno was an ancient Greek philosopher whose paradox of motion was something like the following: in order to get from one point to another you must first travel half the total distance. However, to travel the remaining distance, you must first cover half the distance between the midpoint and the destination. Since there will always be another half to cover (he claims), one can never reach the final destination.

In reality we will eventually reach a stopping point because of the limited precision of floating point values. However, getting to that point would probably take a huge number of steps, and we would like to be able to stop before then for the

Figure 6: Several steps of ray tracing using unbounding volumes.

sake of computational efficiency. In practice we specify some small value epsilon and say that if the minimum distance to the set (given by our distance estimator) is less than or equal to epsilon, then we have reached the surface.

In this case we are actually rendering an isosurface of the distance estimator rather than the Julia set itself. An isosurface of a function is simply all the points where that function has the same value. An example of this would be rings on a tree stump: each ring can be thought of as an isosurface of a function mapping cells of the tree to their age in years, since all cells in a ring were grown in the same year (Figure 7).

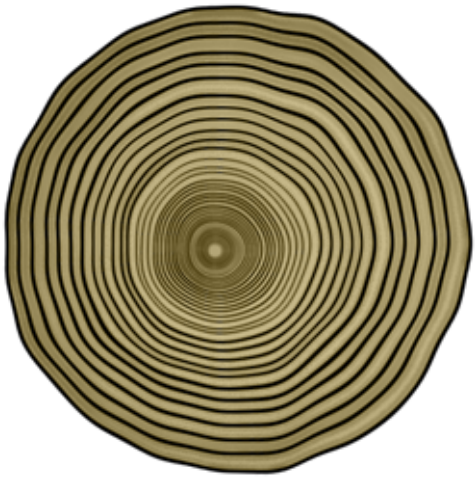

Figure 7: A tree trunk displays age-isosurfaces of its cells.

The isosurface we're rendering will not be as detailed as the Julia set itself – the distance function looks more like a smoothed version of the set. Detail can be increased by making epsilon smaller, but this also increases rendering time due to the greater number of steps required to reach the isosurface. There is one redeeming property of rendering an isosurface, however: since the isosurface is a continuous function we can generate normals for shading the surface. Surface normals are impossible to define on the Julia set itself because there is detail at every level. More details on generating normals from the isosurface can be found in Hart et al [HSK89].

6 Ray Tracing on the GPU

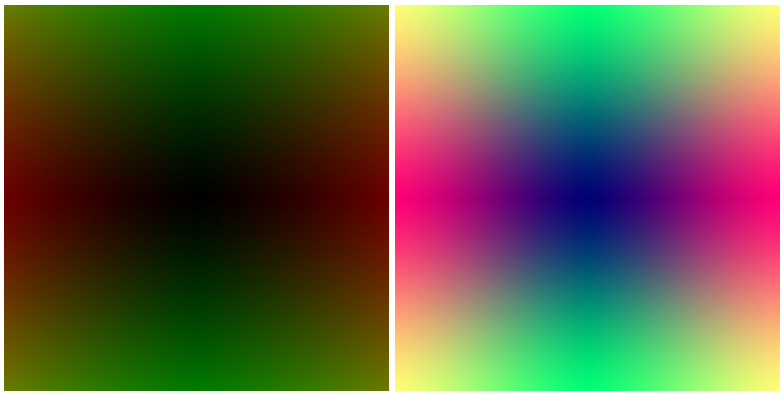

Figure 8: Left: *Input ray origins.* Right: *Input ray directions.*

The GPU was never meant to do strange stuff like rendering fractal sets or tracing rays in four-dimensional space, but in the last few years the GPU has become much more powerful than the CPU, and many people are becoming interested in its use for applications outside of typical video game graphics. This topic is known as GPGPU or General-Purpose computation on Graphics Processing Units, and encompasses anything from protein folding to options pricing.

Most GPGPU applications ignore the majority of graphics card features (vertex processing, fog, lighting, etc.) and instead run a complex fragment program on a single rectangle which covers the entire screen. Each fragment in this rectangle can be thought of as a separate process which works independently of all other processes, and only differs from other processes as a result of its input data.

To ray trace a Julia set, we run a fragment program which steps a single ray through quaternion space using unbounding volumes as described above. In a high-level shader language like Cg, the fragment code looks similar to CPU ray tracing

code, and can be found in Appendix A. The only information that needs to be sent to the program is the origin (starting point) and direction of the rays which make up the image. Since the rays vary linearly across the screen, we need only to specify this information at vertices and the correct ray values for each fragment will be interpolated across the rectangle. We can use, e.g., the three color components (red, green, blue), to send three spatial coordinates (x, y, z), resulting in input rectangles like the ones showin in Figure 8.

Although the GPU is good at quickly doing many floating point operations relative to the CPU, it still suffers from coarse control flow across processes (fragments). Because the GPU is built to render simple graphics, it expects that nearby pixels will usually follow the same branches in a program. All pixels within a relatively large block must wait for each other to finish before moving on to the next batch, but this wait time is typically small (or zero) for the simple shaders used in video games. Unfortunately, many GPGPU applications do not behave this way. In the ray tracer, for instance, two pixels in a block may differ greatly in their distance from the viewer, meaning closer rays effectively take the same amount of time to render as distant ones. Figure 9 illustrates the general difficulties of branching on the GPU: if the black tree on the left represents all possible branches a process can take, and the colored lines indicate the paths actually taken by three different processes from the same block, then the three lines on the right represent the time needed to actually run each process, highlighting the region where useful work is done. Despite all this, a GPU quaternion Julia ray tracer still far outperforms a CPU implementation.

Figure 9: Left: *Potential control flow decisions in a fragment program, and paths taken by different processes in a block.* Right: *Effective running times of processes from the same block.*

7 More Questions

Q: *The pictures are neat, but this quaternion-fractal-GPU stuff is truly esoteric and convoluted. Is there some code I can just play with?*

A: Yes! Well-documented source code and executables for a couple GPU implementations of the ray tracer are available for download from the author's home page.

Q: *2D, 4D, ...what about 3D? 5D? n-D?*

A: A three-dimensional Julia set cannot be defined in the same way as the 2D and 4D versions because multiplication (which is needed to generate a sequence of points) in a three-dimensional complex space is ill-defined. However, the Cayley-Dickson construction gives us a method of generating any $2n$-dimensional complex space, and these hypercomplex spaces have a sum, product, and norm defined, which are all we need to evaluate the distance estimator. The first few hypercomplex spaces are the quaternions (4D), the octonions (8D), and the sedenions (16D). Unfortunately, as we continue to increase the dimension of a Julia set, a conservative distance estimator becomes less and less useful: the closest point in the entire $2n$-dimensional set probably won't be a good estimate for the closest point in the slice we are rendering.

References

[HSK89] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. *SIGGRAPH Comput. Graph.*, 23(3):289–296, July 1989.

[Nor89] Alan Norton. Julia sets in the quaternions. *Computers & Graphics*, 13(2):267–278, 1989.

A Code Listing

```
//////////////////////////////////////
//
//  QJuliaFragment.cg
//  4/17/2004
//
//
//  Intersects a ray with the qj set w/ parameter mu and returns
//  the color of the phong shaded surface (estimate)
//  (Surface also colored by normal direction)
//
//     Keenan Crane (kcrane@uiuc.edu)
//
//
//

// Some constants used in the ray tracing process.  (These constants
// were determined through trial and error and are not by any means
// optimal.)

#define BOUNDING_RADIUS_2  3.0      // radius of a bounding sphere for the set used to accelerate intersection

#define ESCAPE_THRESHOLD   1e1      // any series whose points' magnitude exceed this threshold are considered
                                    // divergent

#define DEL                1e-4     // delta is used in the finite difference approximation of the gradient
                                    // (to determine normals)


// --------- quaternion representation ----------------------------------------
//
// Each quaternion can be specified by four scalars q = A + Bi + Cj + Dk, so are
// stored as a float4.  I've tried a struct containing a separate scalar and
// 3-vector to avoid a lot of swizzling, but the float4 representation ends up
// using fewer instructions.  A matrix representation is also possible.
//


// -------- quatMult() -------------------------------------------------------
//
// Returns the product of quaternions q1 and q2.
// Note that quaternion multiplication is NOT commutative (i.e., q1 ** q2 != q2 ** q1 ).
//

float4 quatMult( float4 q1, float4 q2 )
{
   float4 r;

   r.x   = q1.x*q2.x - dot( q1.yzw, q2.yzw );
   r.yzw = q1.x*q2.yzw + q2.x*q1.yzw + cross( q1.yzw, q2.yzw );

   return r;
}

// ------- quatSq() ---------------------------------------------------------
//
// Returns the square of quaternion q.  This function is a special (optimized)
// case of quatMult().
```

```
//

float4 quatSq( float4 q )
{
   float4 r;

   r.x   = q.x*q.x - dot( q.yzw, q.yzw );
   r.yzw = 2*q.x*q.yzw;

   return r;
}

// ------- iterateIntersect() ----------------------------------------------------
//
// Iterates the quaternion q for the purposes of intersection.  This function also
// produces an estimate of the derivative at q, which is required for the distance
// estimate.  The quaternion c is the parameter specifying the Julia set, and the
// integer maxIterations is the maximum number of iterations used to determine
// whether a point is in the set or not.
//
// To estimate membership in the set, we recursively evaluate
//
// q = q*q + c
//
// until q has a magnitude greater than the threshold value (i.e., it probably
// diverges) or we've reached the maximum number of allowable iterations (i.e.,
// it probably converges).  More iterations reveal greater detail in the set.
//
// To estimate the derivative at q, we recursively evaluate
//
// q' = 2*q*q'
//
// concurrently with the evaluation of q.
//

void iterateIntersect( inout float4 q, inout float4 qp, float4 c, int maxIterations )
{
   for( int i=0; i<maxIterations; i++ )
   {
      qp = 2.0 * quatMult(q, qp);
      q = quatSq(q) + c;

      if( dot( q, q ) > ESCAPE_THRESHOLD )
      {
         break;
      }
   }
}

// ----------- normEstimate() --------------------------------------------------------
//
// Create a shading normal for the current point.  We use an approximate normal of
// the isosurface of the potential function, though there are other ways to
// generate a normal (e.g., from an isosurface of the potential function).
//

float3 normEstimate(float3 p, float4 c, int maxIterations )
{
   float3 N;
   float4 qP = float4( p, 0 );
   float gradX, gradY, gradZ;

   float4 gx1 = qP - float4( DEL, 0, 0, 0 );
   float4 gx2 = qP + float4( DEL, 0, 0, 0 );
   float4 gy1 = qP - float4( 0, DEL, 0, 0 );
   float4 gy2 = qP + float4( 0, DEL, 0, 0 );
   float4 gz1 = qP - float4( 0, 0, DEL, 0 );
   float4 gz2 = qP + float4( 0, 0, DEL, 0 );

   for( int i=0; i<maxIterations; i++ )
   {
      gx1 = quatSq( gx1 ) + c;
```

```
      gx2 = quatSq( gx2 ) + c;
      gy1 = quatSq( gy1 ) + c;
      gy2 = quatSq( gy2 ) + c;
      gz1 = quatSq( gz1 ) + c;
      gz2 = quatSq( gz2 ) + c;
   }

   gradX = length(gx2) - length(gx1);
   gradY = length(gy2) - length(gy1);
   gradZ = length(gz2) - length(gz1);

   N = normalize(float3( gradX, gradY, gradZ ));

   return N;
}

// ---------- intersectQJulia() -----------------------------------------
//
// Finds the intersection of a ray with origin rO and direction rD with the
// quaternion Julia set specified by quaternion constant c.  The intersection
// is found using iterative sphere tracing, which takes a conservative step
// along the ray at each iteration by estimating the minimum distance between
// the current ray origin and the closest point in the Julia set.  The
// parameter maxIterations is passed on to iterateIntersect() which determines
// whether the current ray origin is in (or near) the set.
//

float intersectQJulia( inout float3 rO, inout float3 rD, float4 c, int maxIterations, float epsilon )
{
   float dist; // the (approximate) distance between the first point along the ray within
               // epsilon of some point in the Julia set, or the last point to be tested if
               // there was no intersection.

   while( 1 )
   {

      float4 z = float4( rO, 0 );         // iterate on the point at the current ray origin.  We
                                          // want to know if this point belongs to the set.

      float4 zp = float4( 1, 0, 0, 0 );   // start the derivative at real 1.  The derivative is
                                          // needed to get a lower bound on the distance to the set.

      // iterate this point until we can guess if the sequence diverges or converges.
      iterateIntersect( z, zp, c, maxIterations );

      // find a lower bound on the distance to the Julia set and step this far along the ray.
      float normZ = length( z );
      dist = 0.5 * normZ * log( normZ ) / length( zp );  //lower bound on distance to surface

      rO += rD * dist; // (step)

      // Intersection testing finishes if we're close enough to the surface
      // (i.e., we're inside the epsilon isosurface of the distance estimator
      // function) or have left the bounding sphere.
      if( dist < epsilon || dot(rO, rO) > BOUNDING_RADIUS_2 )
         break;
   }

   // return the distance for this ray
   return dist;
}

// ----------- Phong() -------------------------------------------------
//
// Computes the direct illumination for point pt with normal N due to
// a point light at light and a viewer at eye.
//

float3 Phong( float3 light, float3 eye, float3 pt, float3 N )
{
   float3 diffuse = float3( 1.00, 0.45, 0.25 ); // base color of shading
   const int specularExponent = 10;             // shininess of shading
```

```
   const float specularity = 0.45;              // amplitude of specular highlight

   float3 L     = normalize( light - pt );  // find the vector to the light
   float3 E     = normalize( eye   - pt );  // find the vector to the eye
   float  NdotL = dot( N, L );              // find the cosine of the angle between light and normal
   float3 R     = L - 2 * NdotL * N;        // find the reflected vector

   diffuse += abs( N )*0.3;  // add some of the normal to the
                             // color to make it more interesting

   // compute the illumination using the Phong equation
   return diffuse * max( NdotL, 0 ) + specularity*pow( max(dot(E,R),0), specularExponent );
}

// ---------- intersectSphere() ------------------------------------
//
// Finds the intersection of a ray with a sphere with statically
// defined radius BOUNDING_RADIUS centered around the origin.  This
// sphere serves as a bounding volume for the Julia set.

float3 intersectSphere( float3 rO, float3 rD )
{
   float B, C, d, t0, t1, t;

   B = 2 * dot( rO, rD );
   C = dot( rO, rO ) - BOUNDING_RADIUS_2;
   d = sqrt( B*B - 4*C );
   t0 = ( -B + d ) * 0.5;
   t1 = ( -B - d ) * 0.5;
   t = min( t0, t1 );
   rO += t * rD;

   return rO;
}

// ------------ main() -----------------------------------------------
//
//  Each fragment performs the intersection of a single ray with
//  the quaternion Julia set.  In the current implementation
//  the ray's origin and direction are passed in on texture
//  coordinates, but could also be looked up in a texture for a
//  more general set of rays.
//
//  The overall procedure for intersection performed in main() is:
//
//  -move the ray origin forward onto a bounding sphere surrounding the Julia set
//  -test the new ray for the nearest intersection with the Julia set
//  -if the ray does include a point in the set:
//      -estimate the gradient of the potential function to get a "normal"
//      -use the normal and other information to perform Phong shading
//      -cast a shadow ray from the point of intersection to the light
//      -if the shadow ray hits something, modify the Phong shaded color to represent shadow
//  -return the shaded color if there was a hit and the background color otherwise
//

float4 main( float3 rO : TEXCOORD0,               // ray origin
             float3 rD : TEXCOORD1,               // ray direction (unit length)

             uniform float4 mu,                   // quaternion constant specifying the particular set
             uniform float epsilon,               // specifies precision of intersection
             uniform float3 eye,                  // location of the viewer
             uniform float3 light,                // location of a single point light
             uniform bool renderShadows,          // flag for turning self-shadowing on/off
             uniform int maxIterations )          // maximum number of iterations used to test convergence
: COLOR
{
   const float4 backgroundColor = float4( 0.3, 0.3, 0.3, 0 );  //define the background color of the image

   float4 color;  // This color is the final output of our program.

   // Initially set the output color to the background color.  It will stay
   // this way unless we find an intersection with the Julia set.
```

```
    color = backgroundColor;


    // First, intersect the original ray with a sphere bounding the set, and
    // move the origin to the point of intersection.  This prevents an
    // unnecessarily large number of steps from being taken when looking for
    // intersection with the Julia set.

    rD = normalize( rD );  //the ray direction is interpolated and may need to be normalized
    rO = intersectSphere( rO, rD );


    // Next, try to find a point along the ray which intersects the Julia set.
    // (More details are given in the routine itself.)

    float dist = intersectQJulia( rO, rD, mu, maxIterations, epsilon );



    // We say that we found an intersection if our estimate of the distance to
    // the set is smaller than some small value epsilon.  In this case we want
    // to do some shading / coloring.

    if( dist < epsilon )
    {
       // Determine a "surface normal" which we'll use for lighting calculations.
       float3 N = normEstimate( rO, mu, maxIterations );

       // Compute the Phong illumination at the point of intersection.
       color.rgb = Phong( light, rD, rO, N );
       color.a = 1;  // (make this fragment opaque)

       // If the shadow flag is on, determine if this point is in shadow
       if( renderShadows == true )
       {
          // The shadow ray will start at the intersection point and go
          // towards the point light.  We initially move the ray origin
          // a little bit along this direction so that we don't mistakenly
          // find an intersection with the same point again.

          float3 L = normalize( light - rO );
          rO += N*epsilon*2.0;
          dist = intersectQJulia( rO, L, mu, maxIterations, epsilon );

          // Again, if our estimate of the distance to the set is small, we say
          // that there was a hit.  In this case it means that the point is in
          // shadow and should be given darker shading.
          if( dist < epsilon )
             color.rgb *= 0.4;  // (darkening the shaded value is not really correct, but looks good)
       }
    }

    // Return the final color which is still the background color if we didn't hit anything.
    return color;
}
```