CS/CNS 171 - Assignment 7a - Fall 2015

Authors: Parker Won & Nailen Matschke

Due: December 7, 2015

Contents

1 Assignment Overview

The purpose of this assignment is to prepare you for ray tracing an entire scene, so there's a lot to read. However, the actual code you should end up writing will be at most 100 lines, and likely far less. What's important is that you thoroughly understand the contents of this document.

So far, all of our work in this course has been with polygonal models, which can be tidied up by shading methods such as the Phong model, but are fundamentally an approximation of some exact surface. Furthermore, we have not discussed dealing with shadows, reflections, or refractions through materials such as glass while using a tool such as OpenGL or our own shader program. One common technique to tackle all of these issues at once, at the sacrifice of real-time speed, is what's known as *ray tracing*.

As the name implies, this involves tracing the path that a ray of light follows to the camera as it interacts with different objects in the scene, starting from the camera itself. To determine where the ray interacts with a surface, we need a way to express it mathematically so that we can solve explicitly or at least approximately for their point(s) of intersection. While it's possible to do this for the interpolated smooth surface of tessellated models, it turns out to be easier to use implicitly defined surfaces. Specifically, we will be asking you to ray trace scenes composed of certain type of shapes called *superquadrics*, along with various light sources.

To help you with this, we have provided a modeling program composed of a command line and a viewer. The former can be used to manually build scenes out of superquadric primitives or load files specifying a series of modeling commands, while the latter allows you to view the current scene, and explore it through an arcball interface and other commands. The end goal of this assignment is to traverse the data structure provided to store objects and render the scene using ray tracing, but this week we will focus on becoming familiar with the necessary math.

2 Assignment Details

If some of this doesn't make sense, **don't panic**. Parker has also provided detailed instructions on the structure of objects present in the scene as well as how to use the command line in `understanding_the_tools.txt`, both of which are vital to understanding how the program we've provided works.

In this first half of the ray tracing assignment, you are asked to write two functions. We've provided stubs for you in the `Assignment.cpp` file; all you have to do is fill them in. They both involve drawing entities using OpenGL, but they're called by our program from within the `display()` function after all camera and arcball transformations have been applied, so you only need to worry about the actual draw calls like `glVertex3f()`, `glutSolidSphere()`, et cetera.

2.1 Part 1

The first is the `drawIOTest()` function. Its purpose is to ensure that you understand the math behind superquadrics, and can test a point in world space using the inside-outside function of a transformed superquadric. What we want for you to do is to draw a three-dimensional grid of evenly spaced small spheres or GL_POINT objects in world space, colored to indicate whether they're inside a superquadric or not. In order to do so, you must traverse the tree of objects present in the scene, a process explained in Parker's instructions on how to use the command line.

For example, an easy way to do this is to check every point of the form $(0.5i, 0.5j, 0.5k)$ where $i, j, k \in [-10, 10]$ against the inside-outside functions of each superquadric in the scene, and use `glutSolidSphere()` to draw a red sphere of radius 0.1 at that location if it's inside a superquadric, or a blue sphere if it's not. You can set the material properties (i.e. color) using `glMaterialfv()` and `glMaterialf()`. You may also want to add some "jitter" to the spheres, by which we mean some random offset in each dimension on the order of the sphere radius or less. This makes it so that they don't all lie in perfect lines, which can make it hard to distinguish between them.

Don't forget to apply the inverse superquadric transformations to each point you test, as the inside-outside function assumes the superquadric is centered at the origin. Additionally, objects in the scene may contain references to themselves, which creates a fractal pattern. As a result, you should be sure to cap the recursion depth of your tree traversal. Finally, the second part of the assignment will ask you to save a copt of the current scene before you raytrace it, so as you traverse the scene's tree, it might be a good idea to save

each primitive and its transformations to a separate `std::vector`, and then loop through it to do run your inside-outside test on each superquadric.

2.2 Part 2

The second is the `drawIntersectTest()` function. Its purpose is to ensure that you can intersect a ray with a superquadric, assuming it hits one. We want you to take the ray equation $\vec{a}t + \vec{b}$ that from the camera's location in the direction it's looking. In other words, $\vec{b}$ is the camera position, and $\vec{a}$ is the direction the camera is looking. Before any camera transformations we know that $\vec{a} = (0, 0, -1)$, so after the camera transformations $\vec{a}$ will simply be the original vector under the same rotation transformation as the camera. The `drawIntersectTest()` function receives a pointer to the scene's `Camera` data structure, which has a `getVector()` method to return its rotation vector as an `Eigen::Vector3f`, and a `getAngle()` method to return its rotation angle in radians as a `float`.

In order to intersect this vector with a superquadric, you must test if it hits each superquadric in the scene, and choose the intersection (if there is one) closest to the camera. Then, your function should use `glBegin(GL_LINES)`, `glEnd()`, and `glVertex3f()` to draw a vector of length 1 starting from the intersection point and pointing in the direction of the surface normal there. Instances of the `Primitive` class provide a `getNormal()` function, which takes an `Eigen::Vector3f` representing a point on the superquadric's surface. The function assumes that the superquadric is centered at zero, but **does** account for the superquadric's own scaling. As a result, if you have, say, a superquadric sphere of radius 2, you should pass the `getNormal()` function a vector such as (0, 2, 0) rather than the unscaled (0, 1, 0), which isn't on its surface.

As with `drawIOTest()` function, don't forget to apply each superquadric's inverse transformations to your ray, or about the potential for an infinitely recursive tree traversal. Additionally, the arcball rotation does not affect the location of the camera or light structs, just how they appear in the viewer, so you can still use it to rotate the scene and check that the normal vector you draw looks correct.

2.3 Grading & Submission

`drawIOTest()` and `drawIntersectTest()` are worth 25 points each, out of a total of 50 for HW 7a, which makes up half of the 100 points available across HW 7a and HW 7b. All of your code should go in `Assignment.hpp` and `Assignment.cpp`, so that's all you need to turn in (via email, as usual). It should be very apparent whether your code works or not from the viewer, so please **include a screenshot** demonstrating each of your two functions. You can also come to office hours and show us.

Doing this part of the assignment correctly is necessary for the full ray tracer.

3 Renderer Controls

The keyboard controls for the renderer are as follows:

- **q** or **Esc**: switches keyboard control back to the command line
- **b**: toggles whether the scene's lights rotate with the arcball (by default, they do)
- **m**: toggles between per-pixel (i.e. Phong model, default) and per-vertex (Gouraud) shading. If the renderer is running slowly, you should try changing the shading mode to per-vertex.
- **w**: increases the scale of the scene in the viewer
- **s**: decreases the scale of the scene in the viewer

- **r**: refreshes the entities in the scene
- **t**: toggles wireframe mode (defaults to off). When wireframe mode is off, we load all the scene data into the GPU and then draw, which is fast, but for wireframes we have to draw each edge one by one from the CPU side of things, which is much slower. If your scene has a lot of high-resolution polygons in it, wireframe mode will be very slow. **The result of `drawIOTest()` is only shown in wireframe mode.**
- **n**: toggles the drawing of normals in wireframe mode (defaults to off). If this is turned on, a vector indicating the direction of the normal at each vertex will be drawn in wireframe mode.
- **o**: toggles whether your `drawIOTest()` function is called during non-wireframe mode rendering (defaults to on, which does nothing at first).
- **i**: toggles whether your `drawIntersectTest()` function is called during wireframe mode rendering (defaults to on, which does nothing at first).

4 Superquadrics

4.1 Mathematical Details

4.1.1 Definition

In order to define the shapes we'll be ray tracing, we use an *inside-outside* function, defined to be some $f : \mathbb{R}^3 \to \mathbb{R}$ such that

$$f(x, y, z) \begin{cases} < 0 & \text{inside the object} \\ = 0 & \text{on the object's surface} \\ > 0 & \text{outside the object} \end{cases}$$

Many common solids can be defined this way, such as the unit cube, unit sphere, octahedron, unit cylinder, and double cone:

$$\begin{aligned} \text{cube}(x, y, z) &= \max(|x|, |y|, |z|) - 1 \\ \text{sphere}(x, y, z) &= x^2 + y^2 + z^2 - 1 \\ \text{octahedron}(x, y, z) &= |x| + |y| + |z| - 1 \\ \text{cylinder}(x, y, z) &= \max(x^2 + y^2, z^2) - 1 \\ \text{cone}(x, y, z) &= \sqrt{x^2 + y^2} + |z| - 1 \end{aligned}$$

One can see that by setting the left-hand side of side of these equations to 0, we get an implicit definition for each of these shapes.

However, instead of using a different function for each type of surface we wish to ray trace, it is easier to instead use superquadrics, as they can easily be defined to mimic those described above, amongst others.

Superquadrics themselves are a three-dimensional generalization of Lamé curves (also known as Piet Hein's "superellipses"). While it is possible to extend the math to create toroids, we will be dealing solely with ellipsoids, defined by longitudinal shape parameter e and latitudinal parameter n, with the inside-outside function given as

$$\text{sq\_io}(x, y, z, \quad e, n) = \left(\left(x^2\right)^{\frac{1}{e}} + \left(y^2\right)^{\frac{1}{e}} \right)^{\frac{e}{n}} + \left(z^2\right)^{\frac{1}{n}} - 1$$

One can verify that with $e = n = 1$ this equation becomes the same as that of the unit sphere above, while with $e = n = 2$ we have an octahedron. **It is important to square x, y, and z before exponentiating them, as otherwise the math will break for negative numbers**.

4.1.2 Normals

This definition is highly useful not only because it defines a surface, but it also gives us a way to find the surface normal at any point (x, y, z). The normal is simply the gradient of the sq_io function, or in other words

$$\nabla \text{sq\_io}(x, y, z, e, n) = \begin{bmatrix} \frac{\partial \text{sq\_io}}{\partial x} \\ \frac{\partial \text{sq\_io}}{\partial y} \\ \frac{\partial \text{sq\_io}}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{2x(x^2)^{\frac{1}{e}-1}\left((x^2)^{\frac{1}{e}}+(y^2)^{\frac{1}{e}}\right)^{\frac{e}{n}-1}}{n} \\ \frac{2y(y^2)^{\frac{1}{e}-1}\left((x^2)^{\frac{1}{e}}+(y^2)^{\frac{1}{e}}\right)^{\frac{e}{n}-1}}{n} \\ \frac{2z(z^2)^{\frac{1}{n}-1}}{n} \end{bmatrix}.$$

This vector may not be normalized. Thus, given a point on the surface of a superquadric, we can find the surface normal there as well.

4.1.3 Parameterization

It turns out that we can also parameterize these three-dimensional surfaces (for fixed values of e and n) in terms of two variables u and v, by taking advantage of the longitudinal and latitudinal nature of e and n mentioned earlier. If we had a unit sphere superquadric, we could express points on its surface in terms of polar coordinates as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(v)\cos(u) \\ \cos(v)\sin(u) \\ \sin(v) \end{bmatrix},$$

with $u \in (-\pi, \pi]$ (longitude) and $v \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ (latitude). It turns out that this formula generalizes to superquadrics through specialized analogs of the sine and cosine functions, defined as

$$s(\theta, p) = \begin{cases} \text{sign}(\sin(\theta)) \cdot |\sin(\theta)|^p & \sin(\theta) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

and

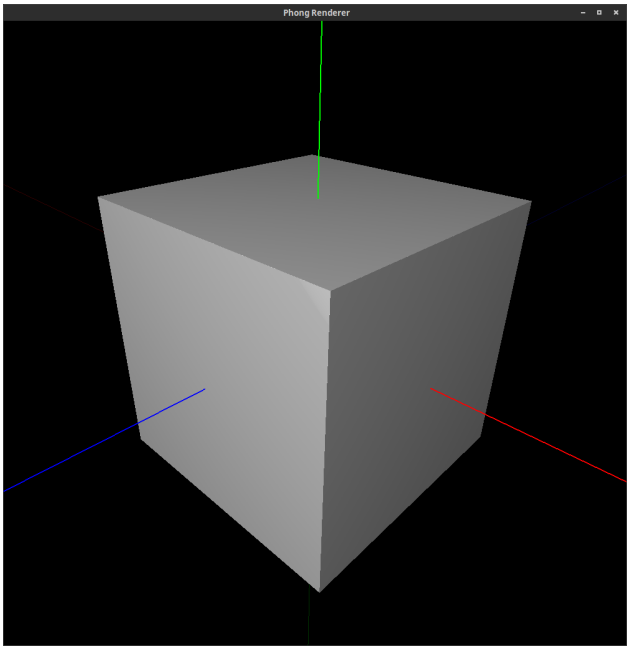
$$c(\theta, p) = \begin{cases} \text{sign}(\cos(\theta)) \cdot |\cos(\theta)|^p & \cos(\theta) \neq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Using these functions, we can parameterize a point on the superquadric in terms of its longitude and latitude u and v as

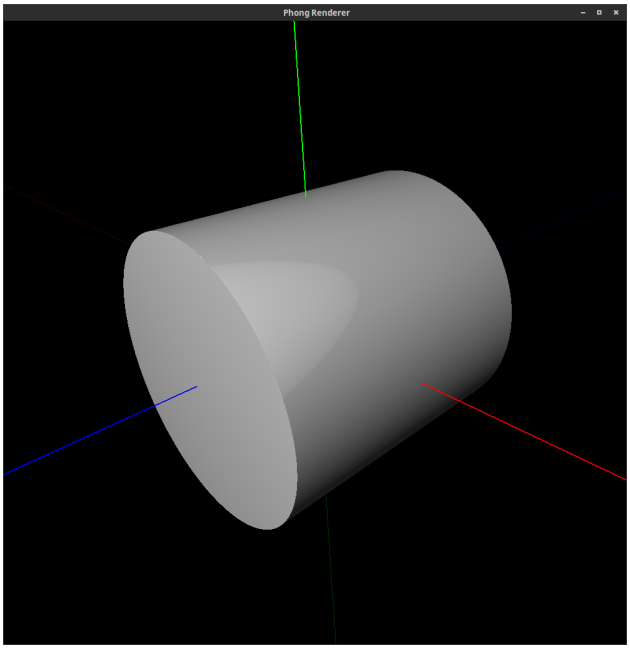
$$\text{sq\_p}(u, v) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} c(v, n)c(u, e) \\ c(v, n)s(u, e) \\ s(v, n) \end{bmatrix}.$$

4.1.4 Some Examples

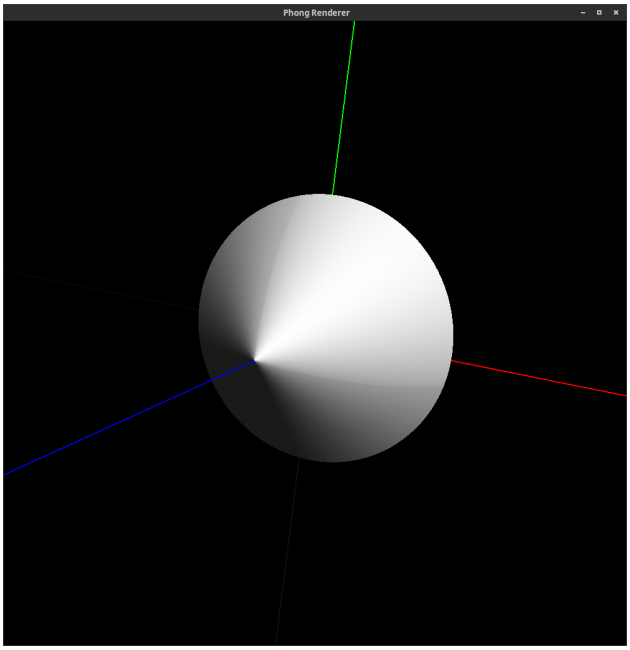
The following images were produced using our command line and renderer, which are built on the above math. By choosing fractional exponents such as $e = n = 0.0001$, we can make a cube:



while $e = 1$ and $n = 0.0001$ produces a cylinder:



or $e = 1$ and $n = 2$ yields a double cone:



This should illustrate the power superquadrics give us to express different solids.

4.2 Transformations of Implicit Functions

4.2.1 Derivation

So far, we've only considered superquadrics centered at the origin, but in order to make scenes, we want to be able to position our objects arbitrarily. Specifically, given some inside-outside function $f(\vec{x})$, we want to know how to find the resulting function $F(\vec{X})$ which serves an inside-outside function for the same surface under some transformation O that takes $\vec{x}$ to $\vec{X}$. As an example, let's consider the transformation given by

$$\vec{X} = O(\vec{x}) = T(R(S(\vec{x}))),$$

where T, R, and S are some translation, rotation, and scaling operations respectively. Then, we have the inverse transformation O^{-1} as

$$\vec{x} = O^{-1}(\vec{X}) = S^{-1}(R^{-1}(S^{-1}(\vec{X})))$$

We can then plug this into the relation between our original and transformed inside-outside functions to get that

$$F(\vec{X}) = f(\vec{x}) = f(O^{-1}(\vec{X})).$$

Thus, in order to calculate the inside-outside function of a transformed superquadric, you must first apply the inverse transformation to the position vector in question, and feed this into the original untransformed function. The same must be done to calculate surface normals, as the above holds for $\nabla f(\vec{x})$ and $\nabla F(\vec{X})$.

4.2.2 A Note on Inverse Transforms

While you may rely on Eigen to compute inverse transformations, it's a good idea to understand what inverse transformation matrices look like relative to the originals. Given a translation matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

then the inverse scaling can be accomplished with

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -a \\ 0 & 1 & 0 & -b \\ 0 & 0 & 1 & -c \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For scaling, if the forward transformation is

$$S = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

then the inverse can be done with

$$S^{-1} = \begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & \frac{1}{b} & 0 & 0 \\ 0 & 0 & \frac{1}{c} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For rotations, the transpose of a rotation matrix is actually the same as its inverse, which is left to the reader to verify.

5 Ray Tracing

5.1 Intersections

Now that we know how to express the surface of a superquadric and compute its normal vectors, how do we follow light from the camera back to the surface of objects? For simple contours such as planes we can solve for the intersection between a vector and the surface explicitly, but for superquadrics we must use an iterative solver to produce an estimation. One such technique that produces accurate results but is still fairly straightforward is that of *Newton's method*.

5.1.1 Newton's Method

Many of you will recall from calculus that a Taylor series $T(g(t), t_0)$ is a method of approximating the function $g(t)$ around the point $t = t_0$, given by

$$g(t) = g(t_0) + g'(t_0)(t - t_0) + \dots$$

If we ignore the higher order terms as they are comparatively small in the neighborhood of t_0, then we can say that if $g(t) = 0$, we have

$$0 = g(t_0) + g'(t_0)(t - t_0) \implies t = t_0 - \frac{g(t_0)}{g'(t_0)}$$

This tells us that if we're at some point t_0 and want to get to another point t where t is a root of $g(t)$, then we can approximate t using t_0, $g(t_0)$, and $g'(t_0)$.

This can be made into an iterative process by replacing t with t_{new} and t_0 with t_{old}, so that if we start from some initial value of t_{old}, we can compute our next approximation of t as

$$t_{\text{new}} = t_{\text{old}} - \frac{g(t_{\text{old}})}{g'(t_{\text{old}})},$$

and then plug this again in as t_{old} to get another approximation of t, and so on and so forth until we hit some "stopping condition." We label this value as t_{final}, and take it to be that $g(t_{\text{final}}) \approx 0$. As you may have guessed, we will be using the sq_io function and its derivative for g and g', since finding the intersection of a vector and a superquadric's surface is the same as finding where along the vector we will have $\text{sq\_io}(x, y, z) = 0$.

It is of course possible that $g'(t_0) = 0$, which would cause divide-by-zero errors at runtime - you should check for this.

5.1.2 Initial Guess

So what do we use as our initial value of t_{old}? A simple way to bound superquadrics is to use a sphere larger than the largest unit superquadric, which is a cube. The radius of the bounding sphere must therefore be $\sqrt{1^2 + 1^2 + 1^2} = \sqrt{3}$, which in turn gives us an inside-outside function for it:

$$\text{bsphere}(\vec{x}) = \vec{x} \cdot \vec{x} - 3.$$

Now, we know we can express a line parallel to the vector $\vec{a}$ that passes through $\vec{b}$ as

$$\text{ray}(t) = \vec{a}t + \vec{b}$$

Where t varies across $\mathbb{R}$. If we let $\vec{b}$ be the location of our camera, then we can solve for the intersection of a ray exiting the camera and the bounding sphere of a superquadric by solving

$$\vec{a}t + \vec{b} = \vec{x} \implies (\vec{a}t + \vec{b}) \cdot (\vec{a}t + \vec{b}) - 3 = (\vec{a} \cdot \vec{a})t^2 + 2(\vec{a} \cdot \vec{b})t + (\vec{b} \cdot \vec{b}) - 3 = 0.$$

This is just a quadratic equation in t with coefficients $a = \vec{a} \cdot \vec{a}$, $b = 2(\vec{a} \cdot \vec{b})$, and $c = \vec{b} \cdot \vec{b}$, so we can solve for t:

$$t^{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Now, there are two things to consider. The first is that our discriminant may be negative, leading to imaginary values of t. This corresponds with cases where the ray misses the bounding sphere entirely, so you should check for this and handle it accordingly.

The second is that we have two solutions for t, t^+ and t^-. The physical interpretation of this fact is intuitive, as a ray may intersect a sphere in at most two locations. If both values of t are positive, then starting point $\vec{b}$ is outside the sphere and the smaller value t^- is on the side of the sphere closer to the camera, and thus closer to the ray's actual intersection with the superquadric. In this case, your initial guess should be t^-.

If one value is positive and the other negative, then $\vec{b}$ is inside the sphere, and the ray may or may not hit the superquadric while traveling away from the eye. In this case, both t^- and t^+ could be tested as initial guesses. If either produces a positive solution for t through the Newton's method solver, then the trick is to know whether the ray intersects the superquadric on the way in or the way out. If one solution is positive and one is negative, then we know $\vec{b}$ is inside the superquadric, and the outside surface can't be seen. If both are positive, however, then we know the ray goes in one side and comes out the other. If both are negative, then the superquadric is entirely behind the camera, and can't be seen.

As with the Newton's method solutions in the previous case, we know that if both t^+ and t^- are negative, then the bounding sphere is behind the camera and thus invisible, since the ray has to travel backwards to hit it.

5.1.3 Trick for Better Numerical Solutions

If for any reason you want to use t^+ as the initial guess for your solver, it happens to be that the limitations of floating point math can result in poor accuracy for certain values of t. To counteract this, one trick is to write the original quadratic equation so that $b \geq 0$ (if it's not originally, just multiply both sides by -1), and then multiply the numerator and denominator of t^+ by $-b - \sqrt{b^2 - 4ac}$ to produce $t^{+\text{alt}}$:

$$t^{+\text{alt}} = t^+ \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = \frac{2c}{-b - \sqrt{b^2 - 4ac}},$$

a result which you can verify yourself.

5.1.4 Stopping Conditions

Between floating point math and the inherent approximation made by our Newton's method solver, we can't simply iterate until we find a value of t such that $\text{sq\_io}(\vec{a}t + \vec{b}) = 0$. Instead, we can stop when the value of a superquadric's inside-outside function is *close enough* to zero. A good approximation of this is when we are within one twentieth or so of a pixel's width away from the actual surface, but the gist is that $|\text{sq\_io}(\vec{a}t + \vec{b})|$ must be sufficiently close to 0.

However, we can also take advantage of the fact that we know some information about $g'(t) = \vec{a} \cdot \nabla \text{sq\_io}(\vec{a}t + \vec{b})$. As long as we are approaching the superquadric along a ray, the inside-outside function should be positive and decreasing, and thus $g'(t)$ should be negative. If ever it becomes positive, then traveling further along the ray should take us farther away from the superquadric. Thus, if the sign of $g'(t)$ ever changes from $-$ to $+$, we know we've missed the superquadric.

Additionally, on the surface of a superquadric we will have $g'(t) = 0$, so if this is ever the case we should also check if $g(t) = 0$. If this is satisfied as well, then we know we're on the superquadric's surface already, and can just use our current value for t_{old} without computing t_{new}. On the other hand, if $g'(t) = 0$ and $g(t)$ is nowhere near 0, the math is effectively the same as the sign change case mentioned above.

Above all else, remember that $\vec{b}$ is the position of the camera, $\vec{a}$ is the direction the camera is looking,

$$\text{ray}(t) = \vec{a}t + \vec{b}$$

describes a ray leaving the camera,

$$g(t) = \text{sq\_io}(\text{ray}(t))$$

is the function to use in Newton's method as it tests whether the ray is inside a superquadric, and

$$g'(t) = \vec{a} \cdot \nabla \text{sq\_io}(\text{ray}(t))$$

is its derivative.