**Parallelized Ray Tracing with Classes in CUDA**

By Jerry Feng and John Li

**1. Summary**

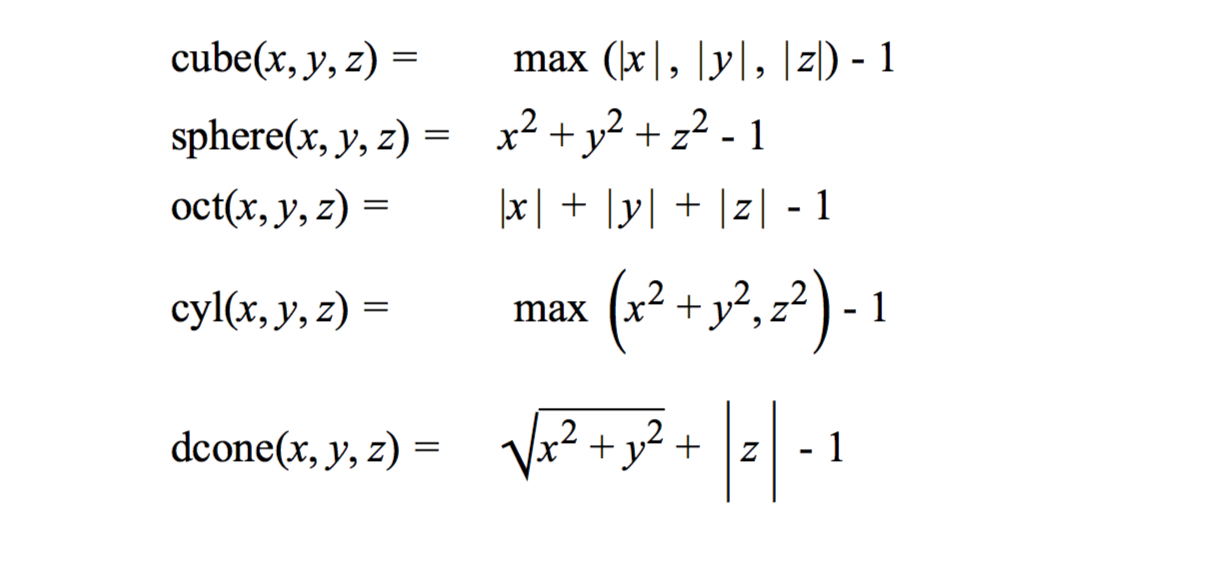
Our project was to create a parallelized ray tracer in CUDA through the GPU. This ray tracer takes in a scene and outputs a rendered image. Because ray tracing requires large amounts of calculations per pixel per object per light source per implemented feature, the algorithm for ray tracing is incredibly time consuming. For this reason, the multithreaded parallelism offered by the GPU speeds up this process.

**2. Introduction and Background**

**A. Background theory**

Ray-tracing is a popular approach for generating hyper-realistic images. The realism that the technique offers comes a direct simulation of how light interacts within a scene. In essence, a camera shoots out a light ray from a point source into a film screen. By calculating the intersection points of the scene, the color of a pixel on the film can be found. However, the process is extremely time consuming, needing a ray of light to be simulated for every pixel in the output image Additionally, when more features are added, the number of calculations required can go up exponentially. For instance, with reflections, the ray-tracing program will potentially need to treat every intersection point as another light source, up until the 100th or so generation. Moreover, as image resolution increases, the number of pixels (and thus originating rays) required also increases the number of calculations required exponentially.

It is important to note that the operations themselves needed are not terribly complex in and of themselves – it is more the sheer number of calculations that causes the process to be so time consuming. The basis of ray tracing lies in utilizing an inside-outside function for a closed surface. Exactly as it sounds, the inside-outside function returns a negative number if a given point lies within the inside of the surface, 0 if on the surface, and a positive number otherwise. Simple inside-outside functions include:



Borrowed from CS171 Problem Set 6, by Kevin Li

However, as nice as these functions are, they do not give us a very wide variety of objects to draw. A common workaround is to use what is called a superquadric surface. The superquadric shape is an extension of Piet Hein's Lame curves and it contains two shape parameters, known as the east-west and north-south shape parameters (the eccentricity parameters cover these). With these parameters, the superquadric ellipsoid can mimic the surfaces of several three-dimensional surfaces, such as spheres, cylinders, and cubes. The superquadric surface is defined as

It is easy to turn this into an inside-outside function by simply subtracting 1 from both sides. We then notice that points that lie directly on the surface will satisfy

For the ray tracer, the function that can be used to render the image of a superquadric will be the following:

Additionally

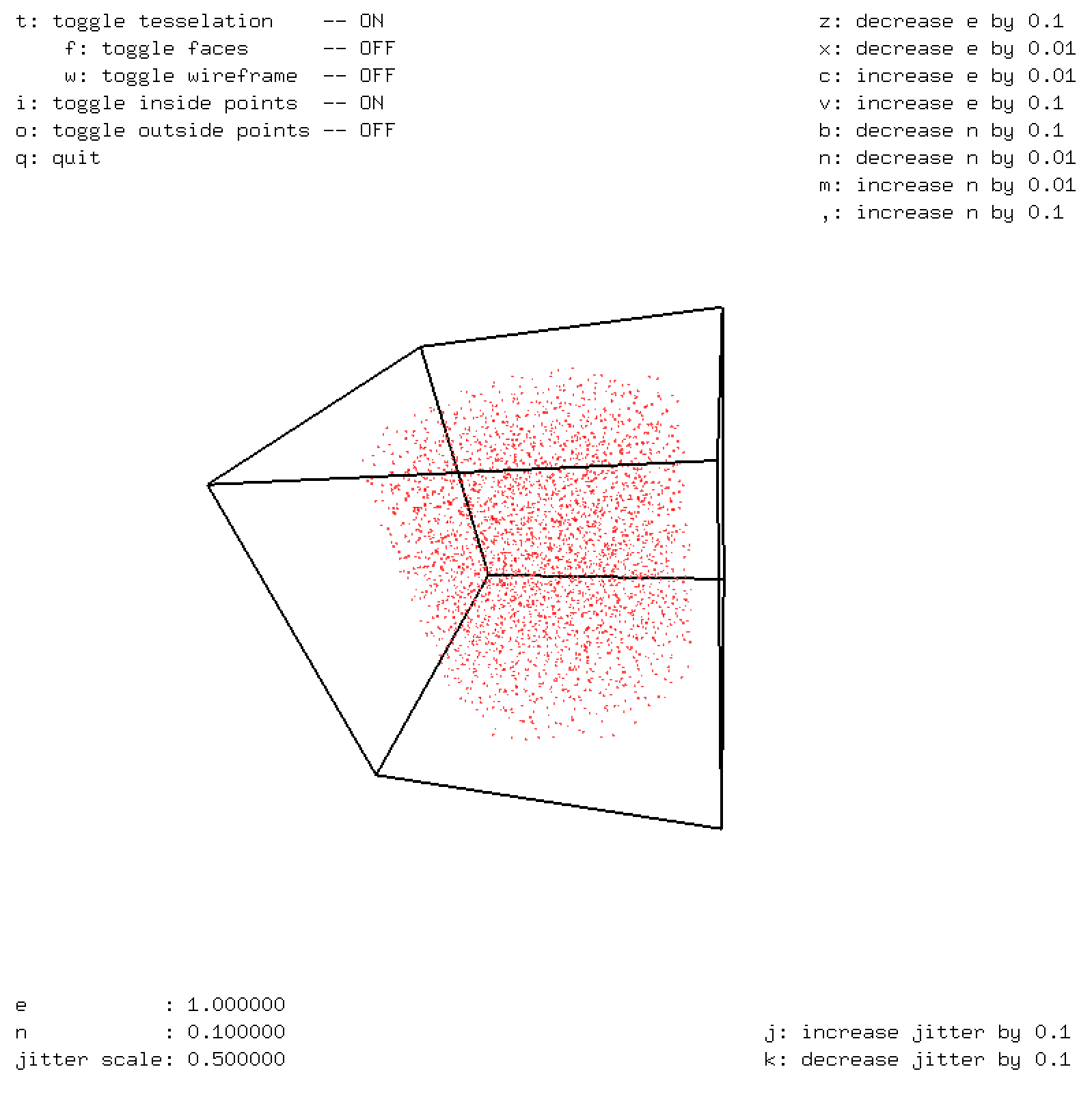
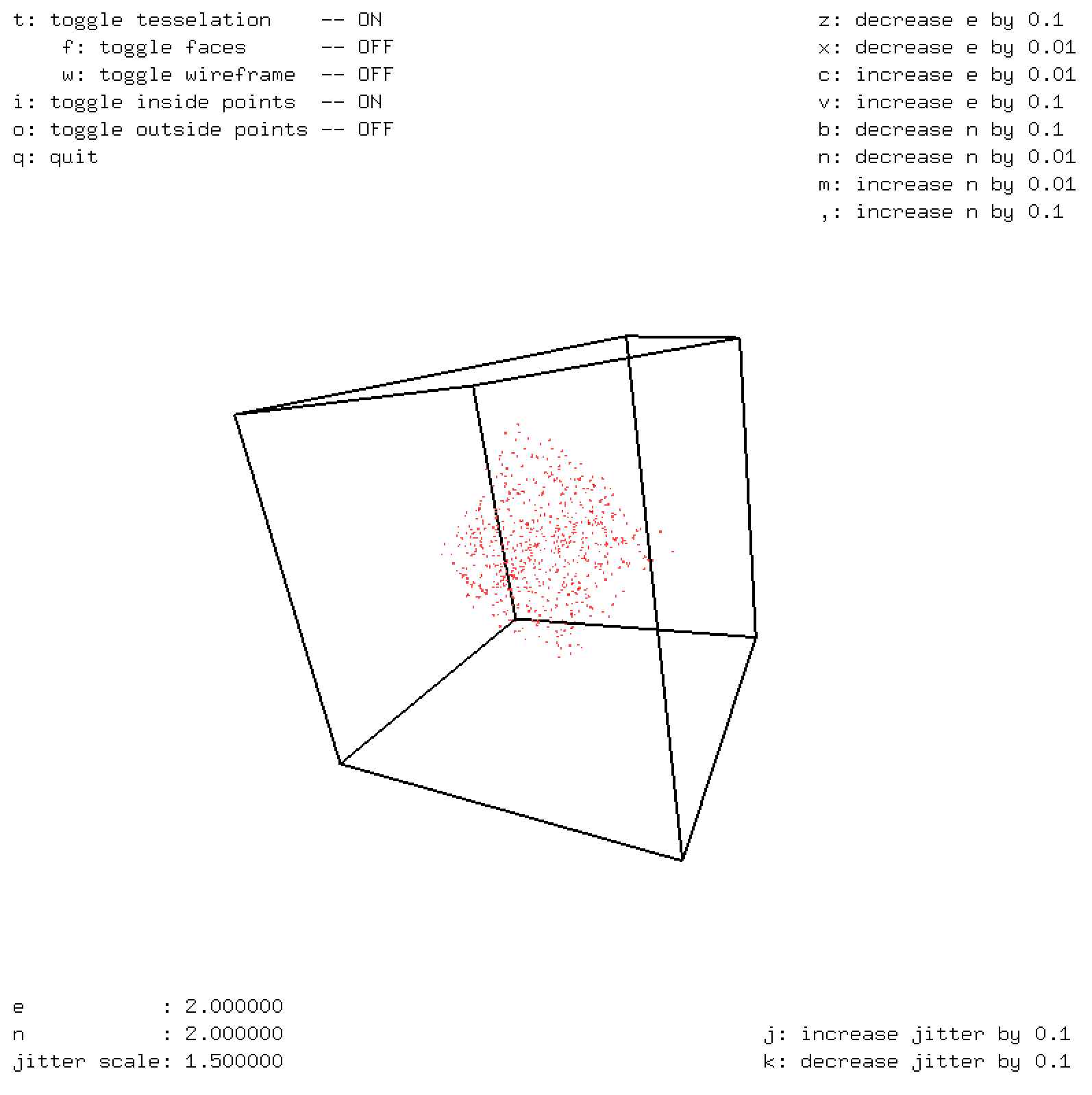


Fig 1. Generation of random points to demonstrate the inside-outsideness defined by a superquadric surface with the following exponents. (A) e = 2 n = 2 (an octahedron), and (B) e = 1, n = 0.1 (a soft cylinder)

Furthermore, by applying transformation matrices, superquadrics can create a limitless number of scenes.

However, in the case of ray-tracing, simply knowing whether a point is in or out of a surface is insufficient. A ray may very well tell an origin and direction, but simply simulating the propagation of a ray forward in time to find intersections is undecidable. There needs to be a good initial-guess for the time required to approach an object from a point-source ray. For that, we may use what is known as a bounding sphere. The bounding sphere encompasses all possible points that a superquadric may cover. In the case of a vanilla superquadric, all points lie within a 1-unit radius from the origin. With this, it is easy to surmise that a good initial guess can be found by finding an intersection between the given ray object and the surface defined by

which can be solved through simple application of the quadratic formula.

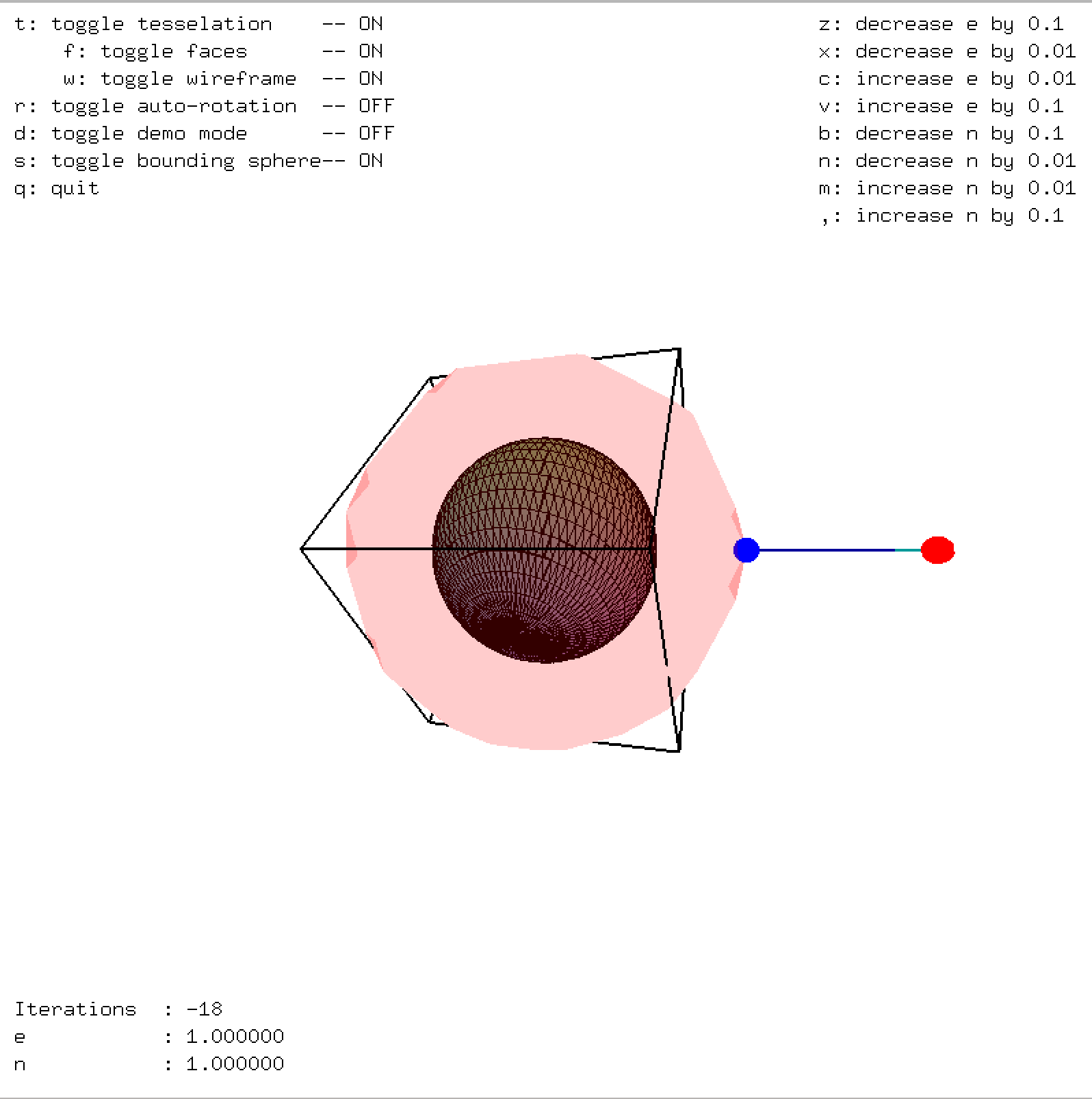


Fig 2. Finding an estimate by using a bounding sphere

Now that a good approximate for where a ray intersects a superquadric has been found (and if there is none, the roots will be in the complex plane), there needs to be a method to inch the point closer to the surface. For this, the method we will use is Newton’s method, which simply updates the timestep with the following rule:

where g is the value from *isq(ray, e, n)* and g’ is the gradient at the point dotted with the direction. The stop condition is defined by when g is close to zero (touching the surface) or when g’ changes sign (went through the bounding sphere but did not intersect).

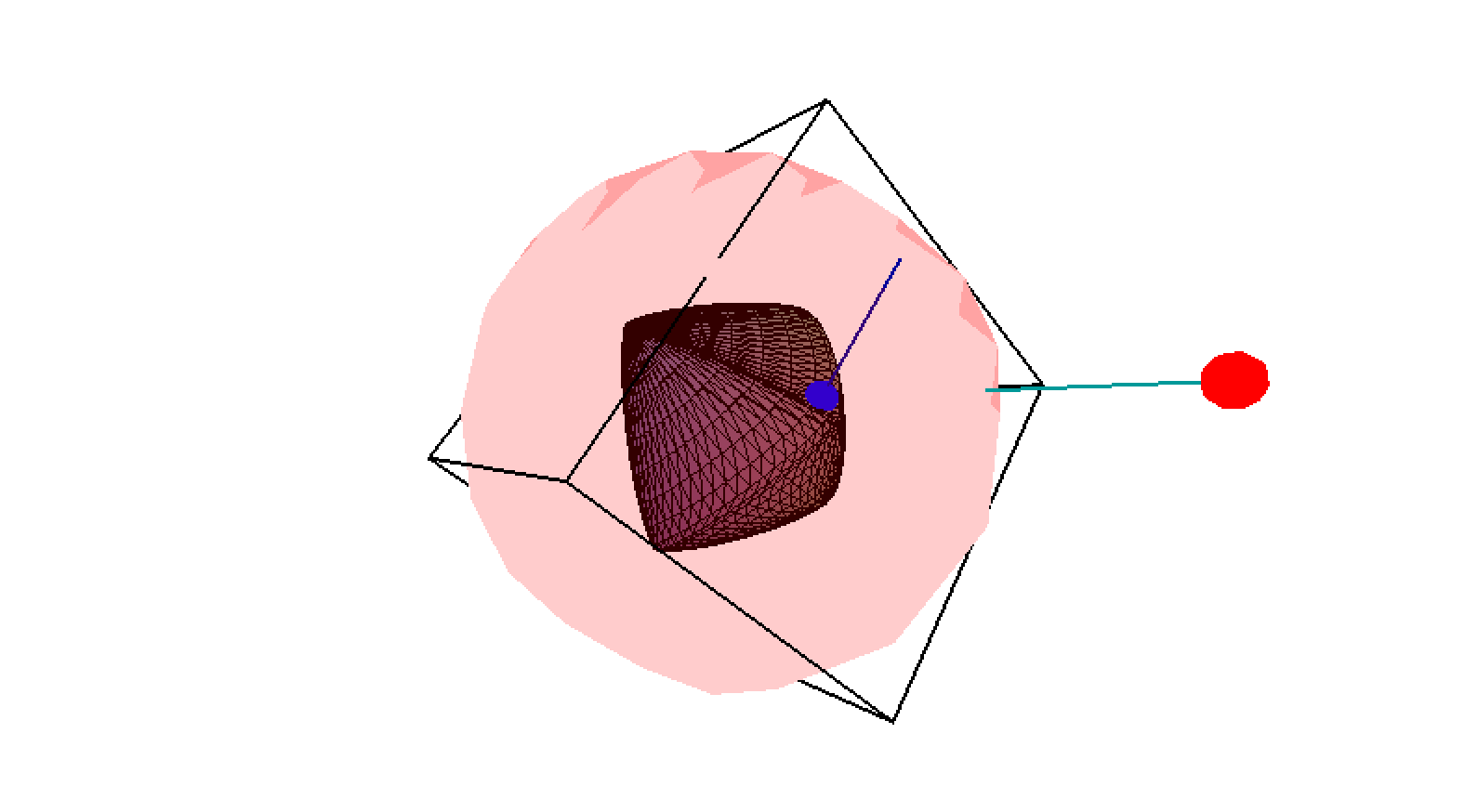
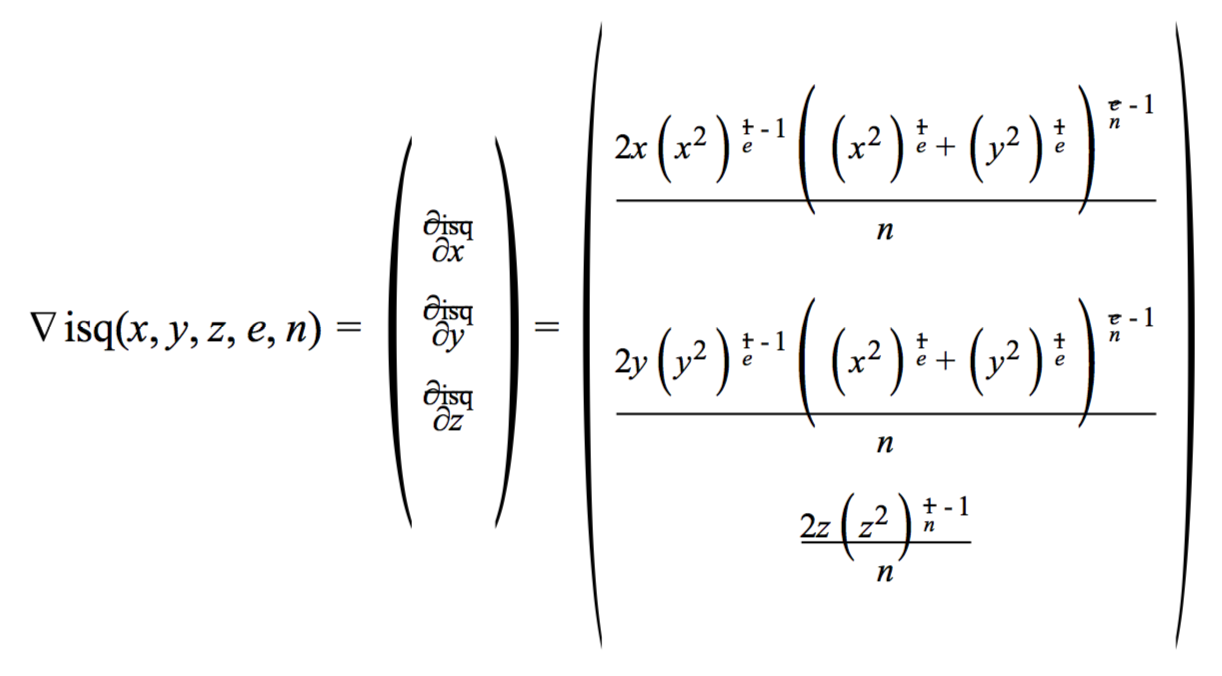


Fig 3. Using Newton’s Method to find an intersection point. The normal is also displayed.

With an intersection point found, it is possible to derive the normal vector at the intersection point using the following formula:



Borrowed from CS171 Problem Set 6, by Kevin Li

With the normal and the intersection point, the light reported at the pixel can be calculated using the Phong Lighting Model.

**B. Data Structures**

The first data structure we used was the “Point” data structure, where each “Point” represented a particular pixel on the screen. This “Point” structure considers the 3-dimensional screen, so as such, each point contains an x, y, and z coordinate. This is necessary for the ray tracing computations, such as rotation, scaling, and translation.

After the “Point” data structure, we also have a “Ray” object that represents a ray of light with that encounters an object. Once the light encounters an object, the distance is recorded and the color of the surface is also stored into the ray. Thus, the “Ray” structure contains a distance (expressed in time) member, and three integer members that represent the intensity of the colors Red, Blue, and Green.

Another related structure that was used is the “pointLight” structure that simply holds the position, color, and attenuation constant (essentially how much energy dissipates with time for that light).

The major structure that was implemented in this program was the “Superquadric” structure. This structure contains the position matrix, orientation matrix, and a scaling matrix, as well as two eccentricity values. The matrices are there to specify size, location, and orientation of the shape on the screen, while the eccentricity values indicate what shape the superquadric is. Additionally, the structure contains three color sets: ambient color (the natural color of the object), diffusive color (affects how light is diffused onto the object’s surface), and specular color (which dictates how light reflects off the object). Finally, there is a shininess attribute that dictates how *much* light is reflected.

The function “isq” represents an “inside-outside” function for the ray tracer. If the output of the function is greater than zero, then it is outside of the object. If the output is zero, then it is on the surface of the object. Lastly, if the output is less than zero, then it is inside the object. The x, y, and z represent the pixel coordinates that are found through the “Point” data structure. The inputs “e” and “n” represent the eccentricity values of the surface. The “e” represents an “east-west” eccentricity parameter, while the “n” represents a “north-south” eccentricity parameter. The class contains methods such as isq(Point \*) and contains(Point \*) that checks the position of a point with respect to the superquadric. The functionality of our “isq” function was tested graphically through the use of the io-test.cpp portion of our code.

In order for the ray-tracing to actually work, two other objects are necessary: a Camera object and a Screen object. The Camera object represents the viewpoint when viewing the image, and the

screen image is necessary to determine the 2D projection of the superquadric object.

**3. Approaches**

TODO:

Languages: C++, CUDA

Machines: UNIX machines, GPUs

When approaching this ray-tracing project,

Due to a previous graphics laboratory class, code for a ray tracer program already exists, but we are unable to use that code due to the fact that CUDA does not support several of the data structures that the CPU supports. An example of this is STL, or the Standard Template Library. Any data structures that are in this library are not CUDA supported, so a challenge was to work around this and to use other data structures that are CUDA supported for parallelization to occur. We overcame this challenge by using the “thrust” library, which provides STL-like data structures that are supported by CUDA. Since we are using vectors in the CPU version of the ray tracer, we need a device\_vector from thrust to implement something similar in CUDA.

Another challenge that we had to overcome was the fact that CUDA does not support virtual classes and pointer references as well. When memcpying from the host to the device, the pointer reference may be lost. Therefore, we had to work around the original code by refactoring it to not include virtual functions and any pointer references in any of the classes. A consequence is that the optimization of the ray tracing decreases.

One other challenge that also occurred was the additional complications of classes. Passing a C++ class object to CUDA is not entirely simple; there needs to be several modifications to code in order to be able to access things appropriately. However, we wanted to preserve the code organization and structure for the ray tracer, so we implemented a tactic known as Device Code Linking1, which links the class methods for use in both a host machine and a GPU. For this reason, a lot of the class methods include a \_\_device\_\_ and \_\_host\_\_ descriptor to indicate if the class method is supported by the GPU or the CPU.

**References**