Enlightenment: Metropolis-Hastings Ray Prediction Model in 3D Space

New Mexico
Supercomputing Challenge
Final Report
April 1st, 2010

Team #69
Los Alamos High School

Team members:
1. Kathy Lin
2. Jake Poston
3. Ryan Marcus
4. Doc Shlachter

Teachers:
1. Leroy Goodwin

Project mentors:
1. Leroy Goodwin

Table of Contents

All images depicting rendering were generated with Team 69's implementation of the algorithm, and used no third party rendering software.

Executive Summary

For years, the rendering equation has posed an unbreakable enigma to the scientific world. The sheer computational complexity of light – from diffraction to reflection to diffusion – forms one of the greatest problems known to physicists and computer scientists. For example, consider the light illuminating this sheet of paper: an uncountable number of protons are streaming down from a light source and striking this sheet of paper. After they hit this sheet of paper, some may reflect back into your eyes, and some may reflect out into space. Some may be absorbed into the paper, others may diffuse against the surface, and others still may pass through the sheet of paper and strike the desk beneath you. A complicated problem, to be sure – but one with a rather intuitive and simple solution.

Through a combination of upper-level statistical theory and brute force, the laws of probability can provide the solution to this complex problem. Our unique implementation of the Metropolis-Hastings algorithm gives us a method to predict light rays based on the path of other light rays. Through this method, we can compute a few rays and then use this method to discover all the other rays.

Our process begins by creating a scene containing a camera, a light source, and some objects. We then test various paths from the camera out towards objects to determine if the camera is looking at something that is illuminated. Once we find a few rays (paths from the light source to the camera), we create a sampling distribution based on properties of those rays. From that distribution, we randomly generate new rays that are then tested for accuracy. If any given new ray is found to be accurate, it is added to the sampling distribution.

This method allows us to create very realistic images at a very efficient rate. The creation of these high-quality images can be utilized for a number of purposes, ranging from analyzing X-rays to modeling light itself.

The Rendering Equation

Essentially, the rendering equation is the formal mathematical statement of how much light is emitted from a given point given the incident angle of the light, a given viewing angle, and various properties of the material (such as luster, reflection, index of refraction, etc). When one talks about "solving the rendering equation," one does not speak of finding an algebraic solution to this formula. When considering solutions to the rendering equation, one looks at various methods that could produce the answer the rendering equation would yield. Actual mathematical manipulation of the equation itself would prove fruitless because many variables are never known, even in completely simulated situations.

Even though the equation is incredibly general, it still does not properly account for several aspects of light.

1. Fluorescence: When light bounces off an object (reflection or refraction) and has a different wave length then when it first hit the object.
2. Interference: When light waves constructively or destructively interfere with each other, such as in a double-slit experiment (described here: http://en.wikipedia.org/wiki/Double-slit_experiment)
3. Phosphorescence: When light is absorbed and not immediately emitted, such as glow-in-the-dark shirts or shoes.
4. Surface Scattering: Because the rendering equation (and almost every subsequent rendering algorithm) assumes that, with enough depth, every surface is entirely smooth, some objects may look unnaturally solid.

Because these constraints are built into the rendering equation and any given computer rendering algorithm is an attempt to provide the solution to this equation, no strictly-traditional rendering algorithm will take these into account either. The method documented in this paper, however, will (optionally) compensate for fluorescence and surface scattering.

Monte-Carlo Methods

Description and explanation

Often, a perfect mathematical solution is not available for a given problem. For example, the integration of many functions (like the normal distribution) can only be estimated. One tool in the mathematician's arsenal for resolving these troublesome situations is the Monte-Carlo method. Implementations of Monte-Carlo methods involve taking a large number of random samples from some form of distribution relating to a given problem, and then applying those samples in such a way as to estimate the actual solution.

Imagine that a mathematician is given a pair of two-dimensional closed shapes that exist across a single known domain and range. The only thing that the mathematician can test is whether a given point lies within a given shape. The mathematician wishes to determine which one has a greater area. For now, consider that the domain of both shapes is [x, y] and the range is [a, b]. If this mathematician wished to employ a Monte-Carlo method, s/he would take a simple random sample of points within [x, y] and [a, b] for both shapes. After determining a finite number of points, the mathematician could conclude that whichever shape had the lower number of points within it had the smaller area.

Two uniform properties of Monte-Carlo methods make them especially applicable to solving the rendering equation. First, as the number of samples (trials, runs, tests, etc.) increase, the answer produced by the method becomes closer and closer to the truth. Stated formally: The accuracy of a Monte-Carlo method is directly proportional to the number of samples used.

Secondly, Monte-Carlo methods allow for predictions about a population to be made using samples from that population. The difference between a Monte-Carlo method and any other statistical tool is that the Monte-Carlo method is flexible enough to be applied to a very wide range of situations. While one could simply take the mean of a sample of light rays, the result would be entirely useless. However, using a Monte-Carlo method combined with a sampling distribution proves fruitful.

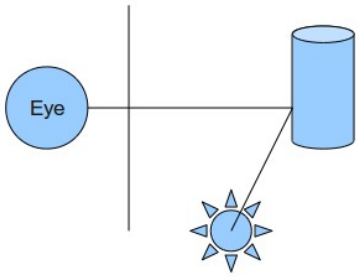
Supercomputing

The complexity (both in terms of sampling and in terms of computation based on those samples) makes Monte-Carlo methods a great candidate for supercomputing. Because these methods require a massive amount of processing power to obtain enough sample data, and because sample data (for the most part) can be taken in parallel, supercomputers seem to be an ideal platform.

Ray Tracing Basics

Description

Ray tracing has been a frequently used solution to the rendering equation. Essentially, a ray tracing model will trace rays out from an eye source, into a scene, to a object in the scene, and then to the light. A ray is traced through every point on the viewing plane (represented in by the vertical line) in this way. Based on various angles of these vectors (and rays, represented by the two-segment line in the image to the right), decisions about shading and location are made. After the tracing process is complete, each point on the viewing plane (which is not actually a line, but a 2D plane) is mapped to a pixel in an image and the color of that pixel is determined.

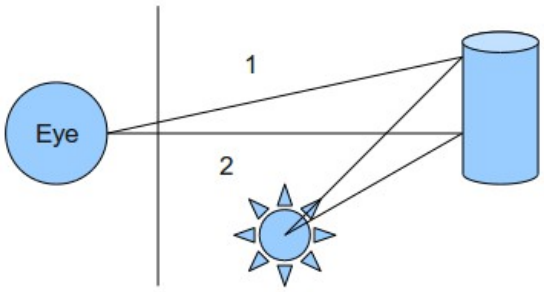


Problems with ray tracing

While ray tracing provides a decent solution to the rendering equation, it comes burdened with several disadvantages. The first and most significant disadvantage is performance. Because there are potentially an infinite number of points on the viewing plane, achieving a perfect render with ray tracing would take an infinite amount of time. Second, ray tracing treats all pixels equally. This means that a pixel where there are clearly no objects (if a ray is traced out from the eye through this point, the ray, upon entering the scene, does not even come close to any objects) is treated in the same manner as a colored pixel. This is incredibly inefficient because an optimized algorithm should concentrate only on important areas of an image.

An overview of shading

One of the difficulties faced by programmers attempting to implement a ray tracing-like model is properly shading each point. Luckily, an observable property of light makes this quite an easy task.



Consider the two traced rays in the image to the right. If one considers what this situation would look like in the real world, one would realize that the bottom of the cylinder would be more brightly illuminated then the top of the cylinder. In this case, the point represented by ray #2 should be brighter then the point represented by ray #1. In order to determine this, consider the normal of the cylinder (a vector coming out directly away from the center column of the cylinder). The angle formed between the normal and the incident ray represents the proportion of shading. The larger the angle, the less illumination. One convenience utilized by many programmers is referred to as “cosine shading” because one could easily use the cosine of the angle between the normal and the incident ray as a coefficient of shading.

3D Calculations

Defining Three-Dimensional Objects

Our three-dimensional world is created using basic shapes, such as spheres, rectangles, cylinders, and cones. We used the basic equations of these shapes combined with parametric equations to define these shapes in our program. For each shape, we need to be able to find three things:

1. Determine if an incoming ray intersects the object and where the collision point is.
2. The reflected or refracted ray from an incoming ray.
3. The cosine of the angle between the incoming ray and the normal vector from the collision point. This value is used to find the amount of lighting that points on the object should receive.

Spheres

A sphere is defined by its center point and radius, and it has the equation

$$(x-c_1)^2+(y-c_2)^2+(z-c_3)^2=s^2$$

with *(c1, c2, c3)* being the center point and *s* being the radius. Our incoming ray is defined by a starting point *(a, b, c)* and slope *(p, q, r)*. The ray can be represented by parametric equations

$$x=a+pt \quad , \quad y=b+qt \quad , \quad z=c+rt \quad .$$

In order to determine if and where the ray hits the sphere, we need to solve these two equations together. We first plug in the values for x, y, and z from the second equation into the first one.

$$(a+pt-c_1)^2+(b+qt-c_2)^2+(c+rt-c_3)^2=s^2$$

We then expand the expression and write it as a quadratic in terms of *t*.

$$(p^2+q^2+r^2)t^2+(2ap-2c_1p+2bq-2c_2q+2cr-2c_3r)t+(c_1-a)^2+(c_2-b)^2+(c_3-c)^2-s^2=0$$

To simplify notation, let

$$l=p^2+q^2+r^2 \quad , \quad m=2ap-2c_1p+2bq-2c_2q+2cr-2c_3r \quad ,$$
$$n=(c_1-a)^2+(c_2-b)^2+(c_3-c)^2-s^2$$

The previous quadratic equation becomes

$$lt^2+mt+n=0$$

To determine the real roots of this equation, we first find the discriminant.

$$d=m^2-4ln$$

If this value if negative, then *t* has no real solutions, and the ray does not hit the sphere. Otherwise, we use the quadratic formula and find the solutions. Plugging these values of *t* into the parametric equations gives us the points at which the line that contains the ray intersects the sphere. Negative values of *t* correspond to points behind the starting point of the ray, so they can be eliminated. Then we find the smallest positive value of *t*, which corresponds to the first point that the ray intersects the sphere, which is the collision point of the ray with the sphere.

To determine the resulting ray from the incident ray, we use the vector component of the incident ray in a formula to produce the vector component of the resulting ray. The starting point of the resulting ray is the collision point. Given incident vector v_I and normal vector *n* of some surface, the resulting vector is

$$\vec{v}_2=\vec{v}_1-2(\vec{v}_1\cdot\vec{n})*\vec{n}$$

The normal to the sphere at the collision point is given by the vector that goes from the center of the sphere to the collision point. We simply plug in the values into the formula and find the resulting vector.

The last part we have to find for sphere is the cosine of the angle between incident vector and the normal. We find this by using the two ways of defining the dot product. Let's call the

incident *v,* the normal vector *n,* and the angle between them *a*. (Note that v_x , v_y , and v_z refer to the x, y, and z components of *v*.)

$$\vec{v}\cdot\vec{n} = \|v\|\|n\|\cos(a)$$

$$\vec{v}\cdot\vec{n} = v_x n_x + v_y n_y + v_z n_z$$

Equating these two formulas and solving for cos(*a*) gives us

$$\cos(a) = \frac{v_x n_x + v_y n_y + v_z n_z}{\|v\|\|n\|}$$

Rotation for cylinders and cones

Cylinders and cones are more complicated because the standard equations for them are complicated. However, the standard equations for cylinders and cones that are aligned with an axis are much simpler. To take advantage of this, we define each shape with its parameters, use a rotation matrix to rotate the shape and align it with the z-axis, find the results of the methods, and rotate everything back to its original alignment.

To rotate an object, we first represent the current axis of the object as a vector. (For a cylinder, the axis is the line between centers of the two circles on the ends. For a cone, the axis is the line that connects the point of the cone with the center of the bottom circle). We then find the unit vector that has the same slope as the axis we want to rotate to, which is the z-axis in this case. Using the dot product, we can find the angle between these two vectors. Call this angle *a*.

Then we use the cross product to find the vector that is perpendicular to the plane containing the first two vectors. This is the axle of rotation about which our object will be rotated. We normalize this vector and call it *u*. Using the axis-angle formula, the matrix of rotation is:

$$R = \begin{bmatrix} u_x^2 + (1-u_x^2)c & u_x u_y (1-c) - u_z s & u_x u_z (1-c) + u_y s \\ u_x u_y (1-c) + u_z s & u_y^2 + (1-u_y^2)c & u_y u_z (1-c) - u_x s \\ u_x u_z (1-c) - u_y s & u_y u_z (1-c) + u_x s & u_z^2 + (1-u_z^2)c \end{bmatrix}$$

where $c = \cos a$ and $s = \sin a$.

To apply this matrix to object, multiply the vector parameters, like slopes of rays, by the matrix. To adjust points, such as centers of objects, multiply the point by the matrix. But remember that the matrix only rotates objects to align with the z-axis. The axes of the objects do not necessarily coincide with the z-axis. To remedy this, we calculate how far away the axes are from the z-axis along the x and y directions. Then we shift every point by that amount. The result is a transformed object with the same shape, but a different orientation. The axis of the object lies along the z-axis. Afterwards, we shift the points back and multiply the points and vectors by the inverse matrix of the rotation matrix.

Cylinders

Now that we have rotated our objects, cylinders and cones are very similar to spheres and rectangles. The general equation of a cylinder whose axis coincides with the z-axis is given by

$$x^2 + y^2 = r^2 \qquad\qquad z_0 < z < z_1$$

where r is the radius and z_0 and z_1 are the top and bottom bounds of the cylinder. Just like before, we define the incoming ray by its starting point *(a,b,c)* and its slope *(p,q,r)*. This ray can be written parametrically as

$$x = a + pt \quad , \quad y = b + qt \quad , \quad z = c + rt$$

A ray, if extended, can potentially hit multiple points on the cylinder. Like with the sphere, we find all the points where the ray could hit and take the point that is closest to the starting point of the ray.

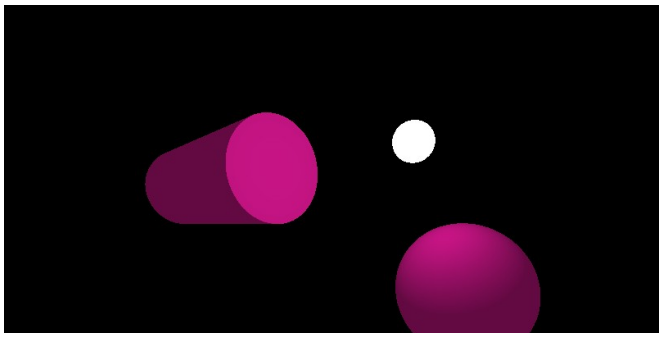


Illustration 1: Cylinders rendered with Metropolis-Hastings

First, we check if this ray will ever strike the two faces of the cylinder. We find where the ray would hit the planes of the faces. Recall that the cylinder is upright and in-line with the z-axis, so the planes of the faces are

$$z = z_0 \quad \text{and} \quad z = z_1$$

If the ray hits these planes, it will hit the planes when

$$c + rt = z_0 \qquad \text{and} \qquad c + rt = z_1$$

If the ray does hit one of these planes, we solve for *t.* We must then determine if the ray strikes the plane inside the face, which is a circle. This is only true if

$$(a + pt)^2 + (b + qt)^2 < r^2$$

If this expression is true, we save this value of t. Then we must determine if the ray strikes the lateral side of the cylinder. We plug the parametric values into the general equation

$$(a + pt)^2 + (b + qt)^2 = r^2$$

We then simplify and write the resulting equation as a quadratic equation with *t*.

$$(p^2 + q^2)t^2 + (2\text{ap} + 2\text{bq})t + (a^2 + b^2 - r^2) = 0$$

The real roots of this equation are values of *t* (if any) for which the ray hits the lateral side. To reduce the number of calculations that the computer must do, we first find the discriminant, simplify the expression and assign the variable *m* to this value.

$$m = 4(b^2 p^2 - a^2 q^2) + 4(r^2 p^2 + r^2 q^2)$$

If the discriminant is negative, the equation has no real roots. Otherwise, we use the quadratic equation to find the roots.

$$t = \frac{-2\text{ap} - 2\text{bq} \pm \sqrt{m}}{2(p^2 + q^2)}$$

Out of all the values of *t* we have so far, we find the first place that the ray hits the cylinder, which corresponds to the smallest value of *t*. However, negative values of *t* correspond to points that are behind the starting point, so we must find the smallest positive value. We plug this value of *t* back into our parametric equation to find the point that the ray hits the cylinder.

We find the resulting ray with the same formula as before. To find the normal vector of

the cylinder at the collision point, we rotate and shift the cylinder to align with the z-axis and determine whether the collision point is on the two faces of cylinder or on the lateral surface. If the point is on the top face, the normal is the vector in the positive z direction. If the point is on the bottom, the normal vector is in the negative z direction. If the point is on the lateral surface, the normal is the vector parallel to the xy plane that intersects the axis and the point. For example, if the the point is (*x, y, z*), the normal is the vector (x, y, 0). We then plug this value and the incident vector into our resulting vector formula.

Finding the cosine of the angle between the incoming ray and the normal is simple. We have already found the normal, and we just use the definitions of the dot product like before.

Cones

After rotating and shifting the cone to align with the z-axis, the general equation is

$$\frac{x^2+y^2}{m^2}=(z_0-z)^2 \qquad where \qquad m=\frac{radius}{height}$$

Just like with the other shapes, we check first if the ray hits the bottom circle. Then we write the ray as a parametric equation, plug the values into the general equation, write the expression as a quadratic equation, and find the real roots. We take the smallest positive value of *t* and plug this value back into the parametric equations to find the collision point.

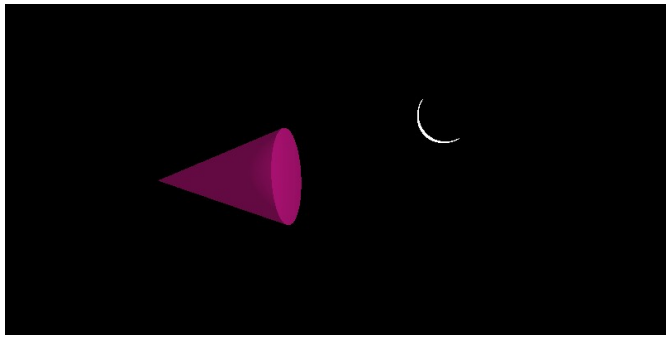
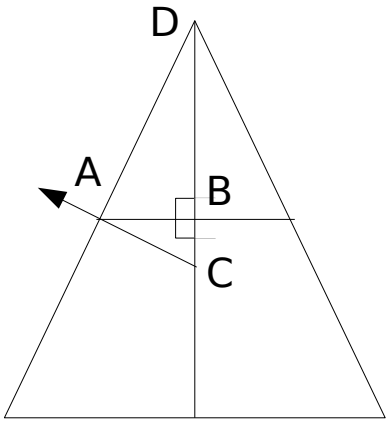


Illustration 2: A cone rendered by Metropolis-Hasting

Finding the resulting ray is a little more difficult that before because finding the normal vector is more complicated. If the point is on the bottom circle (or the top circle, if the cone is inverted), the normal is either the negative z-direction or the positive z-direction. For points on the lateral surface, we've provided a figure on the next page shows a cross section of the cone. Point A (x, y, z) is the collision point, point B (0,

0, z) is the point on the axis of the cone that is on the same horizontal plane as A. The vector between C and A is the vector we are trying to find. Let's call this vector *n*.



We note that

$$\Delta ABC \sim \Delta DAB$$

They both have a right angles, and

$$\angle CAB + \angle ACB = 90^o = \angle ACB + \angle CDA$$

So, ∠CAB = ∠CDA and the triangles are similar. We can use the dot product definitions to find the angle between the axis and the vector between D and A. This is ∠CDA, which is equivalent to ∠CAB. Let's call this angle *a*. By the definition of sine,

$$\sin(a) = \frac{\overline{CB}}{\overline{AB}}$$

We solve for $\overline{CB}$. Since we know the coordinates of B, the distance between B and C, and the fact that C is also on the z-axis, we can find the coordinates of C. From the coordinates of A, B, and C, we can find the vector from C to B and the vector from B to A. The vector sum of these vectors is the vector between C and A, which is what we want. We plug this vector and the incident vector in our resulting vector formula to find the resulting ray.

Since we now know the normal vector on the cone from the collision point, we can use

our dot product definitions to find the cosine of the angle between the incident vector and the normal vector.

The Metropolis-Hastings Light Transport Algorithm

Introduction and disclaimer

The Metropolis-Hastings algorithm is a specific Monte-Carlo method that approximates the distribution of functions that can not be directly sampled. In other words, the Metropolis-Hastings algorithm lets one use sample data to approximate the distribution of the entire population.

In terms of rendering, the Metropolis-Hastings algorithm requires a slight deviation from the formal statement of the algorithm. Therefore, the methods documented here should not necessarily be considered an implementation of the Metropolis-Hastings algorithm, but should instead be considered as a Monte-Carlo approach to rendering rooted in the logic of the Metropolis-Hasting algorithm.

Explanation of the algorithm

The algorithm begins by tracing a finite number of rays from the camera/eye through evenly spaced points on the viewing plane into the scene. These rays are then stored in memory as three points: the point where the ray intersects the viewing plane (this point will be referred to as *P*), the point where the ray strikes an object in the scene (*S*), and the point where the ray hits the light source (*L*). It is important to note that an image can be generated from this point, and a sample image is shown in Illustration 3. Notice the black lines and dots present from a lack of sampling in certain regions, as well as the "hard" shadows present behind both spheres.

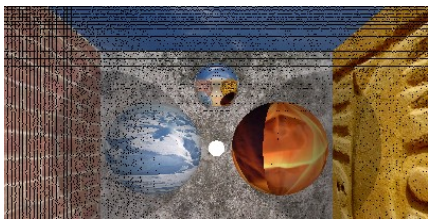


Illustration 3: An image generated from only ray tracing

Let the overall population of light rays represent every single possible light ray traveling from the light source to the eye/camera. The list of rays generated from the first stage of our algorithm serves as a random sample of this population. Because the population size is virtually infinite (as there can be an infinite number of modeled rays from the light source to the eye), the

distribution of the list of rays taken will be approximately normal, as per the central limit theorem. With this known, the algorithm creates several distributions based upon rays that pass through the viewing plane in close proximity to each other. Each of these groups of rays form a "ray cluster," a group of rays that have similar properties because they are within close proximity within the image. Realistically, there will probably be between 1,000 and 100,000 ray clusters for a given image.

Each ray cluster forms nine graphs/distributions – one for each coordinate in each point of each ray. For example, the *P* point's X, Y, and Z components are each plotted in their own distribution. The algorithm calculates the mean and standard deviation (which, in this case, is actually the standard error) of each distribution and uses the results as parameters to create a normal sampling distribution. The algorithm then takes random samples from each of these distributions to propose a new ray. This proposed ray is comprised of 9 random points taken from 9 different sampling distributions. Thus, a proposed ray can be defined as:

$$P\{X, Y, Z\}\ S\{A,B,C\}\ L\{Q, R, S\}$$

Once this proposed ray has been generated, it is tested against the scene itself (i.e., the algorithm makes sure that the proposed ray is actually valid within the modeled world). If the proposed ray is added, the proposed ray is added to the current ray cluster and the mean and standard deviation of the normal distribution being used is recalculated and another ray is proposed. This process repeats until an adequate number of rays have been calculated, or a finite time has been reached. Illustration 4 is what Illustration 3 looks like after the Metropolis-Hasting's algorithm finds 1280000 rays. The result speaks for itself.

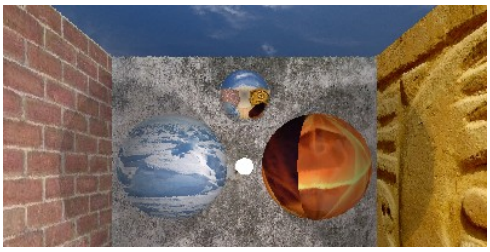
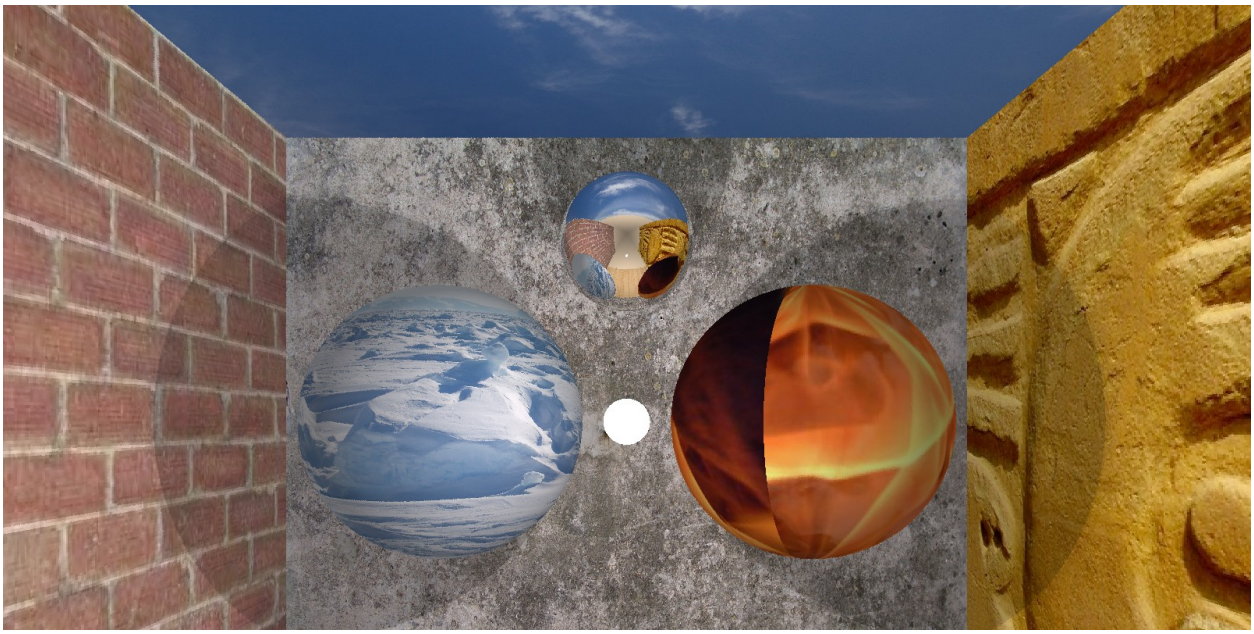
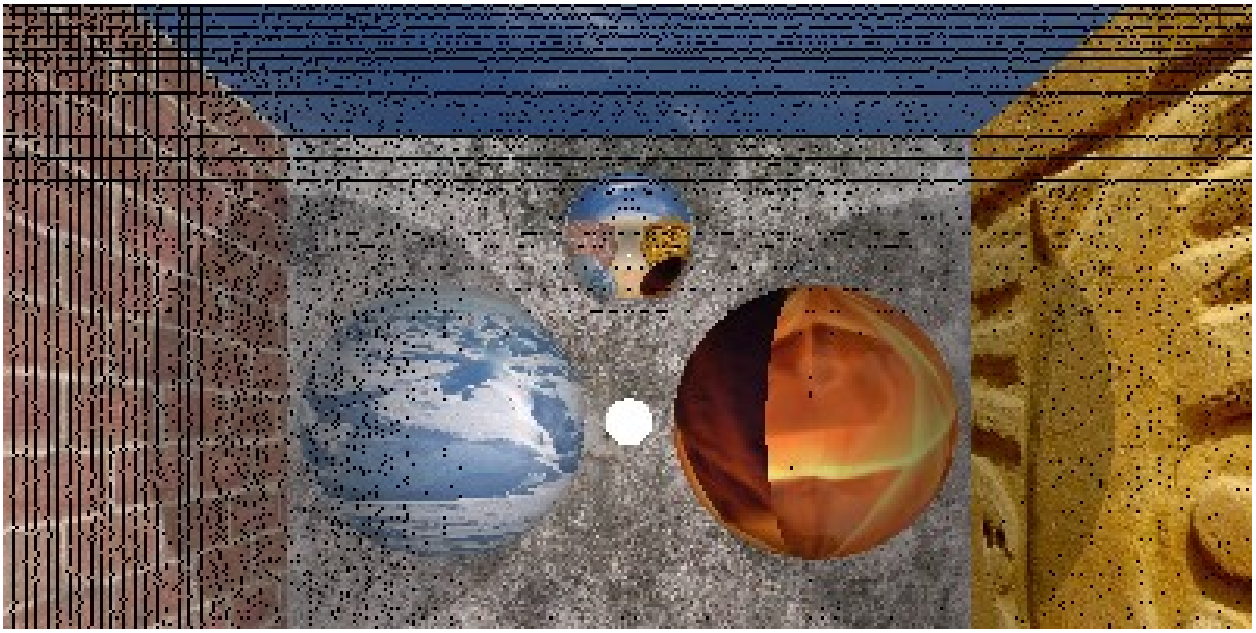


Illustration 4: Low quality Metropolis-Hasting render

Image comparison

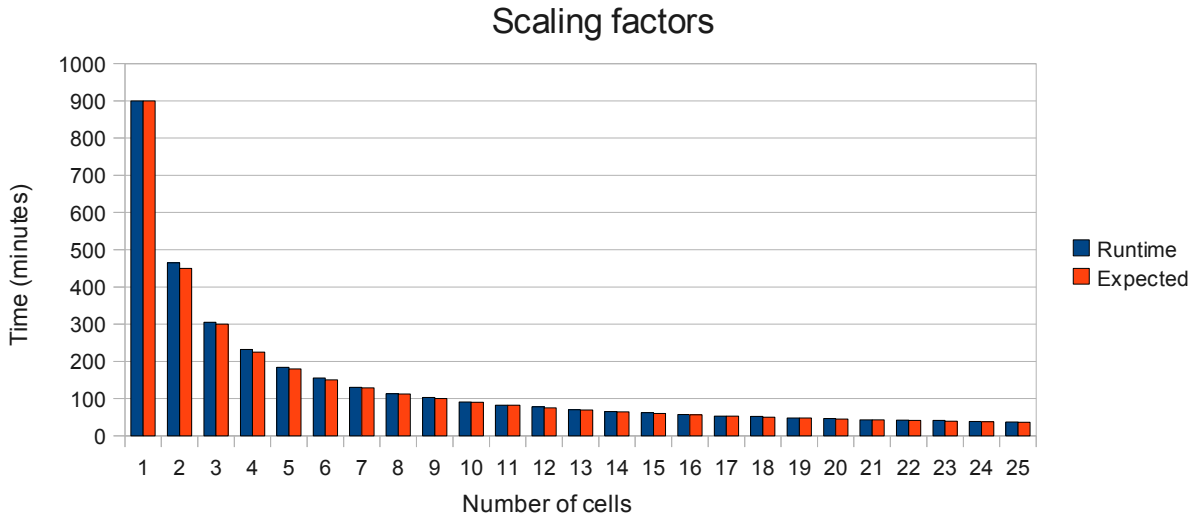
The first image is a high quality image produced by the Metropolis-Hastings algorithm. The second image is created from standard ray tracing. Specifically, note the softer shadows and lack of graininess in the Metropolis-Hastings image.





Parallel advantages of the algorithm

Rendering algorithms are great candidates for parallel computing. In fact, most modern day graphics cards (such as those designed by nVidia and ATI) contain hundreds of vector processors for quickly rendering images. Sadly, the methods and algorithms used are aimed at creating realistic *looking* graphics, not an accurate model of light. However, the same capabilities for parallelization exist within the algorithm's more accurate and realistic model of light.



The graph above depicts the runtime for a high-quality render using the Metropolis-Hastings algorithm. The blue bar represents the time it took for a given number of cells (in this case, each cell is a separate computer running 3 threads) to complete the render. The red bar represents how long it would take to render the image in a perfect world – i.e., one with perfect scaling. For example, with one cell the render took 900 minutes. In a perfect world, eight computers would be able to preform the same task eight times faster, which would be 112.5 minutes. The algorithm took 113 minutes with 8 cells, demonstrating that the algorithm is incredibly distributable, and thus an excellent candidate for a supercomputer.

There are three properties of the algorithm and its implementation that make it particularly distributable:

1. Each ray cluster can be calculated independently of other ray clusters. This has

two implications. First, it means that each cell requires no communication with the cells around it. Each cell only needs to be able to talk to one central cell (perhaps called a server). Second, each ray cluster can be calculated at the same time, which means as the number of cells increase, the algorithm continues to scale because the number of clusters will always exceed the number of cells (one can control the number of ray clusters for any given image, giving the algorithm almost infinite scalability).

2. The actual rendering – the creation of the pixels themselves – can be done independently from each ray cluster. This means that each cell does not have to send back all the data contained in its ray cluster, only the RGB values (red, green, and blue) of each pixel the cell was assigned to calculate. This provides for a very low amount of network traffic, thus decreasing the cost of adding an additional cell.
3. The algorithm has a low traffic-to-processing ratio. This means that very little data needs to be sent across the network to instruct each cell of what it needs to do. Each cell needs only two components in order to operate: the area of the viewing plane that contains the ray cluster a given cell would be responsible for, and a copy of the scene. The area is only two doubles, one representing a starting point and another representing a stopping point. The digital world is small as well, and could even be preloaded. One may think the scene would take up a lot of space, but keep in mind that a sphere is really just a point and a radius. Everything within the algorithm is represented in terms of dimensions.

Video produced by the algorithm

Little known fact: The movie Shrek took several years to render. In designing the algorithm, it was pointed out that if high quality images could be produced, then high quality video could also be produced by stitching several images together. The results proved illuminating (pun intended). Because this medium (paper) does not easily allow for the

publishing of a video file, a sample video produced by the algorithm can be viewed here: http://marcusfamily.info/~ryan/Export.mov

The ability to render video makes the algorithm an even better candidate for a supercomputer. Because each frame can rendered independently of each other, the algorithm can efficiently encompass even more cells while gaining a phenomenal performance boost.

Conclusions

While the algorithm described here (and developed, in full, by team 69) does not come close to providing an all encompassing model for light, it does succeed in providing a supercomputer-ready, high quality, and incredibly accurate model of light.

Through a combination of statistics, theory, and the power of computers, this implementation of the Metropolis-Hastings Light Transport algorithm was not only incredibly successful, but also a pleasure to develop.

Light is a tricky thing. It surrounds every human everyday, and is probably one of the most taken-for-granted elements of human life. Hopefully, this algorithm will provide some insight into the surprisingly dark mystery of light. That is everyone's goal anyway: enlightenment.

A wise grandmother once said: “computers seem to be a compilation of pretty pictures on top of loud boxes.” The algorithm creates some of the prettiest pictures on some of the loudest boxes, so at a minimum, it is grandmother-approved.

Code

```
**MySQL.java

package globalStuff;

import java.sql.Connection;
import java.sql.DriverManager;

/**
 * This class just provides some global methods to work with the
MySQL database
 * @author ryan
 *
 */
public class MySQL {
     private final static String databaseHost = "localhost";
     private final static String databaseName = "metro";
     private final static String databaseUsername = "scc";
     private final static String databasePassword = "metro";

     public static Connection getSQLConnection() {
          // load the JDBC driver
          try {

     Class.forName("com.mysql.jdbc.Driver").newInstance();
               return
DriverManager.getConnection(MySQL.constructJDBCConnectionString(
));
          } catch (Exception e) {
               e.printStackTrace();
          }

          return null;
     }

     /**
      * Returns the string to use with JDBC to connect to the
database. Uses information from this class' fields
      * @return the string to use with JDBC
      */
```

```
     private static String constructJDBCConnectionString() {
          StringBuilder myBuilder = new StringBuilder();

          myBuilder.append("jdbc:mysql://");
          myBuilder.append(databaseHost);
          myBuilder.append("/");
          myBuilder.append(databaseName);
          myBuilder.append("?user=");
          myBuilder.append(databaseUsername);
          myBuilder.append("&password=");
          myBuilder.append(databasePassword);

          return myBuilder.toString();
     }
}

** Timekeeper.java

package globalStuff;

import java.util.ArrayList;

/**
 * This class is responsible for "keeping time" for the entire
project -- it uses a collection of static methods and fields
 * to allow classes to report how much time they spend doing
things. Very useful for optimizing.
 * @author ryan
 *
 */
public class Timekeeper {
     public static final boolean timingEnabled = true;

     // hell, why not? we already have one constant like that
here.
     public static final boolean useCUDA = false;

     public static final int timeCodeForSphereCheckCollision =
0;
     public static final int timeCodeForSphereGetLP = 1;

     public static final int timeCodeForRectangleCheckCollision
```

```
= 2;
     public static final int timeCodeForRectangleGetLP = 3;

     public static final int timeCodeForRayRender = 4;
     public static final int timeCodeForGetClose = 5;

     public static final int timeCodeForCosineShading = 6;
     public static final int timeCodeForMapping = 7;


     private static class Timing {
          public long totalTime;
          public int totalTimes;
     }

     private static ArrayList<Timing> theTimes;

     /**
      * Gets the timekeeper ready to go
      */
     public static void init() {
          theTimes = new ArrayList<Timing>();
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
          theTimes.add(new Timing());
     }

     /**
      * Reports a time to the time keeper
      * @param timeCode the time code
      * @param timeTaken the amount of time used
      */
     public static void addTime(int timeCode, long timeTaken) {
          theTimes.get(timeCode).totalTime += timeTaken;
          theTimes.get(timeCode).totalTimes++;
     }
```

```
    /**
     * Generates a human-readable time report
     * @return the report
     */
    public static String getTimeReport() {
        StringBuilder myB = new StringBuilder();


        myB.append("TASK\t\t\tTIME\tAVG\t\t\tNUM\n");

        Timing t = theTimes.get(0);
        myB.append("Sphere check collision\t" + t.totalTime +
"\t" + Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n");

        t = theTimes.get(1);
        myB.append("Sphere get LP\t\t" + t.totalTime + "\t" +
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n\n");

        t = theTimes.get(2);
        myB.append("Rect check collision\t" + t.totalTime +
"\t" + Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n");

        t = theTimes.get(3);
        myB.append("Rect get LP\t\t" + t.totalTime + "\t" +
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n\n");

        t = theTimes.get(4);
        myB.append("Ray render\t\t" + t.totalTime + "\t" +
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n");

        t = theTimes.get(5);
        myB.append("Ray close\t\t" + t.totalTime + "\t" +
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n\n");

        t = theTimes.get(6);
        myB.append("Cosine shading\t\t" + t.totalTime + "\t" +
```

```
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n");

          t = theTimes.get(7);
          myB.append("Ray mapping\t\t" + t.totalTime + "\t" +
Double.valueOf(t.totalTime / (t.totalTimes == 0 ? 1.0 :
Double.valueOf(t.totalTimes))) + "\t" + t.totalTimes + "\n");


          return myB.toString();
     }

}

** Distribution.java
package monteCarlo.distributions;

/**
 * The standard definition of a distribution to be used with
Monte Carlo methods.
 * @author ryan
 *
 */
public interface Distribution {

     /**
      * Set the scaling factor of the distribution. Defaults to
0.
      *
      * @param d the scaling factor
      */
     public void setScalingFactor(double d);

     /**
      * Sets the scaling factor of the distribution to something
that makes the maximum value 1.
      */
     public void setUnitScale();

     /**
      * Returns the scaling factor of the distribution
      *
```

```
      * @return the scaling factor
      */
     public Double getScalingFactor();

     /**
      * Moves the position of the distribution forward
      */
     public void moveForward();

     /**
      * Moves the position of the distribution backwards
      */
     public void moveBackward();

     /**
      * Resets the distribution to its main position
      */
     public void reset();

     /**
      * Gets the value of the distribution at the current
position.
      *
      * @return the value
      */
     public double getValue();
}

** EulerZetaDistribution.java
package monteCarlo.distributions;

/**
 * A symetric zeta function (using absolute value on negative
numbers, so z(1) = z(-1)).
 *
 * Zeta(x) = 1 + (1 / 2^x) + (1 / 3^x) + (1 / 4^x)...
 *
 * We'll use 50 iterations.
 *
 * @author ryan
 *
 */
```

```
public class EulerZetaDistribution implements Distribution {

     private double sf;
     private double value;

     public Double getScalingFactor() {
          return sf;
     }

     @Override
     public double getValue() {
          int i = 2;
          double toReturn = 1.0;

          while (i != 50) {
               toReturn += (1.0 / Math.pow(Double.valueOf(i),
Double.valueOf(Math.abs(value))));
               //System.out.println("X" + toReturn);
               i++;
          }

          return sf * toReturn;
     }

     @Override
     public void moveBackward() {
          value -= 0.05;
     }

     @Override
     public void moveForward() {
          value += 0.05;
     }

     @Override
     public void reset() {
          value = 0;
     }

     @Override
     public void setScalingFactor(double d) {
          sf = d;
```

```
     }

     /**
      * Tests and prints a couple of values.
      * @param args none
      */
     public static void main(String args[]) {
          EulerZetaDistribution myDist = new
EulerZetaDistribution();

          myDist.setUnitScale();
          myDist.reset();


          int i = 0;

          while (i != 25) {
               myDist.moveBackward();
               i++;
          }

          i = 0;

          while (i != 50) {
               System.out.println(myDist.getValue());
               myDist.moveForward();
               i++;
          }

     }

     @Override
     public void setUnitScale() {
          this.setScalingFactor(1.0/49.0);

     }

}

** NormalDistribution.java

package monteCarlo.distributions;
```

```
/**
 * Defines the standard normal distribution.
 *
 * f(x) = e^(-x^2 / 2)
 *
 * Peaks at 1. If you want the area under the normal
distribution to be 1, use (1/sqrt(2pi)) as your scaling factor.
 *
 * Moves by 10ths.
 *
 * Symetric
 *
 * Please note that you'll need to set your scaling factor to at
least 1, or else 0 will be used.
 *
 * @author ryan
 *
 */
public class NormalDistribution implements Distribution {

     private double sf;
     private double position;

     @Override
     public Double getScalingFactor() {
          return sf;
     }

     @Override
     public double getValue() {
          return sf * Math.pow(Math.E, 0 - Math.pow(position, 2)
/ 2);
     }

     @Override
     public void moveBackward() {
          // to provide good coverage, we'll move by tenths.
          position = position - 0.1;
     }

     @Override
```

```
	public void moveForward() {
		// to provide good coverage, we'll move by tenths.
		position = position + 0.1;

	}

	@Override
	public void reset() {
		position = 0;
	}

	@Override
	public void setScalingFactor(double d) {
		sf = d;

	}

	/**
	 * Tests and prints a couple of values.
	 * @param args none
	 */
	public static void main(String args[]) {
		NormalDistribution myDist = new NormalDistribution();

		myDist.setUnitScale();
		myDist.reset();


		int i = 0;

		while (i != 25) {
			myDist.moveBackward();
			i++;
		}

		i = 0;

		while (i != 50) {
			System.out.println(myDist.getValue());
			myDist.moveForward();
			i++;
		}
```

```
	}

	@Override
	public void setUnitScale() {
		this.setScalingFactor(1.0);

	}

}

** SimpleRightLeftDistirbution.java

package monteCarlo.distributions;

/**
 * A distribution that is incredibly simple. Starts at 0, one
forward is 1, one backwards is 1.
 *
 * @author ryan
 *
 */
public class SimpleRightLeftDistribution implements Distribution
{

	private double sf;
	private int value;

	@Override
	public Double getScalingFactor() {
		return sf;
	}

	@Override
	public double getValue() {
		return Math.abs(value) * sf;
	}

	@Override
	public void moveBackward() {
		value--;
	}
```

```
     @Override
     public void moveForward() {
          value++;
     }

     @Override
     public void reset() {
          value = 0;
     }

     @Override
     public void setScalingFactor(double d) {
          sf = d;
     }

     /**
      * Tests and prints a couple of values.
      * @param args none
      */
     public static void main(String args[]) {
          SimpleRightLeftDistribution myDist = new
SimpleRightLeftDistribution();

          myDist.setUnitScale();
          myDist.reset();


          int i = 0;

          while (i != 25) {
               myDist.moveBackward();
               i++;
          }

          i = 0;

          while (i != 50) {
               // abs to make sym
               System.out.println(Math.abs(myDist.getValue()));
               myDist.moveForward();
               i++;
```

```
          }

     }

     @Override
     public void setUnitScale() {
          this.setScalingFactor(0.1);

     }

}

** MetropolisFunnel.java

package monteCarlo.metropolis;

import java.awt.Color;

import monteCarlo.distributions.Distribution;

import threeDWorld.World;
import threeDWorld.raysAndVectors.Ray3D;
import tracing.RayProcessor;

/**
 * This class works kind of like an upside-down funnel. When it
gets a ray, it applies a Metropolis/Monte Carlo algorithm to it
(assuming the law of large
 * numbers) and returns a multitude of rays to the set ray
processor. One ray comes in, many metropolis-generated rays go
out.
 * @author ryan
 *
 */
public class MetropolisFunnel implements RayProcessor {

     /**
      * Controls the number of threads used by the funnel.
      */
     public int numThreads = 1;

     /**
```

```
      * The number of iterations to calculate per thread. This
is how "deep" we'll try to go into our distribution before
stopping.
      */
     public int iterationsPerThread = 5;

     private Distribution theDistro;
     private RayProcessor toGive;

     /**
      * Creates a new metropolis funnel
      * @param theDist the distribution to use
      * @param toForward where to pass the calculated rays
      * @param theWorld the world the funnel is working in
      */
     public MetropolisFunnel(Distribution theDist, RayProcessor
toForward, World theWorld) {
          theDistro = theDist;
          toGive = toForward;
     }

     @Override
     public void gotRay(Ray3D theRay, Color realColor) {
          // TODO Auto-generated method stub

     }

}


** JakeRenderer.java
package rendering;


import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
```

```
import threeDWorld.raysAndVectors.Ray3D;
import threeDWorld.raysAndVectors.ThreeDPoint;


public class JakeRenderer implements Renderer {
     private double PlaneX, PlaneY, PlaneZ; // In Units
     private double X1, X2, Y1, Y2, Z1, Z2; // The Largest and
smallest X,Y,Z's
     private double xTrans, yTrans;
     private int ScalingFactor = 0;
     private int PixalsX, PixalsY;
     private ArrayList<VisualRay> visRay = new
ArrayList<VisualRay>();
     public double BlendRatio = .5;
     protected Color tempC;




     @Override
     public BufferedImage run() {
          BufferedImage lolImage = new BufferedImage(PixalsX,
PixalsY, java.awt.image.BufferedImage.TYPE_INT_RGB);

          Graphics2D myG = lolImage.createGraphics();

          int count = 0;

          for (VisualRay v1 : visRay) {
               count++;
               double progress = Double.valueOf(count) /
Double.valueOf(visRay.size());

               if (count % 1000 == 0)
{ System.out.println(progress); }

               if (v1.flag) continue;
               ArrayList<VisualRay> thingsHitting = new
ArrayList<VisualRay>();
```

```
                for (VisualRay v2 : visRay) {
                     if (v2.flag) continue;
                     if (v2.X == v1.X && v2.Y == v1.Y) {
                          v2.flag = true;
                          thingsHitting.add(v2);
                     }
                }

                int avgR = 0, avgG = 0, avgB = 0;

                for (VisualRay v : thingsHitting) {
                     avgR += v.color.getRed();
                     avgG += v.color.getGreen();
                     avgB += v.color.getBlue();
                }

                avgR /= thingsHitting.size();
                avgG /= thingsHitting.size();
                avgB /= thingsHitting.size();

                myG.setColor(new Color(avgR, avgG, avgB));
                myG.fillRect(v1.X, v1.Y, 1,1);
           }




           return lolImage;

           /*
            * To output the BufferedImage as a file-
            *
            * try {
            BufferedImage bi = getMyImage(); // retrieve image
            File outputfile = new File("saved.png");
           ImageIO.write(bi, "png", outputfile);
           * }catch (IOException e) {
      *  ...
           * }

           */
      }
```

```
     @Override
     public void setRays(ArrayList<Ray3D> theRays) {
          for(int i = 0; i < theRays.size(); i++){
               if(theRays.get(i).getFlag() == 1){
                    VisualRay ray = new VisualRay();
                    ThreeDPoint Start =
theRays.get(i).getVector(0).startingPosition;
                    ray.setPoint((int)(((Start.x -
xTrans)/PlaneX) * PixalsX), Math.abs((int)(((Start.y -
yTrans)/PlaneY)* PixalsY)));


     ray.setColor(theRays.get(i).calculateFinalColor());
                    visRay.add(ray);
               }


               if(theRays.get(i).getFlag() == 2){
                         VisualRay ray = new VisualRay();
                         ThreeDPoint Start =
theRays.get(i).getVector(theRays.get(i).getVectorCount() -
1).startingPosition;
                         ray.setPoint((int)(((Start.x -
xTrans)/PlaneX) * PixalsX), Math.abs((int)(((Start.y -
yTrans)/PlaneY)* PixalsY)));
                         System.out.println(((int)(((Start.x -
xTrans)/PlaneX) * PixalsX) + "," + Math.abs((int)(((Start.y -
yTrans)/PlaneY)* PixalsY))));

     ray.setColor(theRays.get(i).calculateFinalColor());
                         visRay.add(ray);
                    }
}



     }

     @Override
     public void setPlaneBounds(ArrayList<ThreeDPoint>
theBounds) {
```

```
                X1 = theBounds.get(0).x;
                X2 = theBounds.get(3).x;
                Y1 = theBounds.get(0).y;
                Y2 = theBounds.get(2).y;
                //Z1 = theBounds.get(0).z;
                //Z2 = theBounds.get(0).z;


           xTrans = theBounds.get(1).x;
           yTrans = theBounds.get(1).y;
           PlaneX = X2-X1;
           PlaneY = Y2-Y1;
           //PlaneZ = Z2 - Z1;
           System.out.println(PlaneX + ", " + PlaneY + ", " +
PlaneZ);
           // At the moment Ignoring the Z Axis
           if(ScalingFactor == 0){
                PixalsX = (int) Math.floor(PlaneX * 400);
                PixalsY = (int) Math.floor(PlaneY * 400);
           }
           else{
                PixalsX = (int) Math.floor(PlaneX *
ScalingFactor);
                PixalsY = (int) Math.floor(PlaneY *
ScalingFactor);
           }
           System.out.println(PixalsX + " X " + PixalsY);

     }
     // This is basically the same method, but in this method,
you can change the resolution
     // since Default scaling factor is 400 Pixels
     public void setPlaneBounds(ArrayList<ThreeDPoint>
theBounds, int ScalingFactor) {
           if(ScalingFactor > 0)
                this.ScalingFactor = ScalingFactor;
           else
                System.out.println("Fail -> Scaling Factor Must be
Greater than 1.");

           setPlaneBounds(theBounds);
```

```
     }

}

** OrthoDraw.java
package rendering;

import java.awt.Image;
import java.util.ArrayList;
import threeDWorld.raysAndVectors.Ray3D;
import threeDWorld.raysAndVectors.ThreeDPoint;

import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.util.ArrayList;




import threeDWorld.raysAndVectors.Ray3D;

public class OrthoDraw implements Renderer {
     private double PlaneX, PlaneY, PlaneZ; // In Units
     private double X1, X2, Y1, Y2, Z1, Z2; // The Largest and
smallest X,Y,Z's
     private double xTrans, yTrans, zTrans;
     private int ScalingFactor = 0;
     private int PixalsX, PixalsY;
     protected int ecks, why;
     private ArrayList<VisualRay> visRay = new
ArrayList<VisualRay>();

     @Override
     public BufferedImage run() {
          BufferedImage lolImage = new BufferedImage(PixalsX,
PixalsY, java.awt.image.BufferedImage.TYPE_INT_RGB);

          Graphics2D myG = lolImage.createGraphics();

          for(int i = 0; i < visRay.size(); i++){
               myG.setColor(visRay.get(i).color);
```

```
                myG.fillRect(visRay.get(i).X, visRay.get(i).Y,
1,1);
                System.out.println(visRay.get(i).X + ", " +
visRay.get(i).Y);
          }

          return lolImage;

          /*
           * To output the BufferedImage as a file-
           *
           * try {
           BufferedImage bi = getMyImage(); // retrieve image
           File outputfile = new File("saved.png");
          ImageIO.write(bi, "png", outputfile);
          * }catch (IOException e) {
     *  ...
          * }

          */
     }

     @Override
     public void setRays(ArrayList<Ray3D> theRays) {
          // x'  = (x-z)cos(30)
          // y'  =  y+(x+z)sin(30)

          for(int i = 0; i < theRays.size(); i++){
               if(theRays.get(i).getFlag() == 1){
                    VisualRay ray = new VisualRay();
                    ThreeDPoint Start =
theRays.get(i).getVector(0).startingPosition;
                    //ray.setPoint((int)(((Start.x -
xTrans)/PlaneX) * PixalsX), Math.abs((int)(((Start.y -
yTrans)/PlaneY)* PixalsY)));
                    //ecks = (int) ((((Start.x - xTrans)/PlaneX)
- ((Start.z - zTrans)/PlaneZ)) * Math.cos(30));
                    //why =  (int) (((Start.y - yTrans)/PlaneY) +
(((Start.x - xTrans)/PlaneX) + ((Start.z - zTrans)/PlaneZ)) *
Math.cos(30));
                    //ecks =  (int) (((((Start.x - xTrans) -
(Start.z - zTrans)) * Math.cos(30))/PlaneX) * PixalsX);
```

```
                    //why =  (int) Math.abs(((((Start.y -
yTrans)) + ((((Start.x - xTrans)) + ((Start.z - zTrans))) *
Math.sin(30)))/PlaneY) * PixalsY));
                    ecks =  (int) ((((((Start.x) - (Start.z)) *
Math.cos(30)) - xTrans)/PlaneX) * PixalsX);
                    why =  (int) Math.abs((((((Start.y)) +
((((Start.x)) + ((Start.z))) * Math.sin(30)))- yTrans)/PlaneY) *
PixalsY));

                    //System.out.println(ecks + ", " + why);


     ray.setColor(theRays.get(i).calculateFinalColor());
                    ray.setPoint(ecks, why);
                    visRay.add(ray);
               }


               if(theRays.get(i).getFlag() == 2){
                    VisualRay ray = new VisualRay();
                    ThreeDPoint Start =
theRays.get(i).getVector(theRays.get(i).getVectorCount() -
1).startingPosition;
                    //ray.setPoint((int)(((Start.x -
xTrans)/PlaneX) * PixalsX), Math.abs((int)(((Start.y -
yTrans)/PlaneY)* PixalsY)));
                    //ecks = (int) ((((Start.x - xTrans)/PlaneX)
- ((Start.z - zTrans)/PlaneZ)) * Math.cos(30));
                    //why =  (int) (((Start.y - yTrans)/PlaneY) +
(((Start.x - xTrans)/PlaneX) + ((Start.z - zTrans)/PlaneZ)) *
Math.cos(30));
                    //ecks =  (int) ((((((Start.x - xTrans)) -
((Start.z - zTrans))) * Math.cos(30)))/PlaneX) * PixalsX);
                    //why =  (int) Math.abs(((((Start.y -
yTrans)) + ((((Start.x - xTrans)) + ((Start.z - zTrans))) *
Math.sin(30)))/PlaneY) * PixalsY));
                    ecks =  (int) ((((((Start.x) - (Start.z)) *
Math.cos(30)) - xTrans)/PlaneX) * PixalsX);
                    why =  (int) Math.abs((((((Start.y)) +
((((Start.x)) + ((Start.z))) * Math.sin(30)))- yTrans)/PlaneY) *
PixalsY));
```

```
                    ray.setPoint(ecks, why);

     ray.setColor(theRays.get(i).calculateFinalColor());
                    visRay.add(ray);
                    }


          }
     }

     @Override
     public void setPlaneBounds(ArrayList<ThreeDPoint>
theBounds) {

          X1 = theBounds.get(0).x;
          X2 = theBounds.get(3).x;
          Y1 = theBounds.get(0).y;
          Y2 = theBounds.get(2).y;
          Z1 = theBounds.get(0).z;
          Z2 = theBounds.get(2).z;


          xTrans = theBounds.get(1).x;
          yTrans = theBounds.get(1).y;
          zTrans = theBounds.get(1).z;
          PlaneX = X2-X1;
          PlaneY = Y2-Y1;
          PlaneZ = Z2-Z1;
          System.out.println(PlaneX + ", " + PlaneY + ", " +
PlaneZ);

          if(ScalingFactor == 0){
               PixalsX = (int) Math.floor(PlaneX * 400);
               PixalsY = (int) Math.floor(PlaneY * 400);
          }
          else{
               PixalsX = (int) Math.floor(PlaneX *
ScalingFactor);
               PixalsY = (int) Math.floor(PlaneY *
ScalingFactor);
          }
```

```
          System.out.println(PixalsX + " X " + PixalsY);

}
// This is basically the same method, but in this method, you
can change the resolution
// since Default scaling factor is 400 Pixels
     public void setPlaneBounds(ArrayList<ThreeDPoint>
theBounds, int ScalingFactor) {
          if(ScalingFactor > 0)
               ScalingFactor = this.ScalingFactor;
          else
               System.out.println("Fail -> Scaling Factor Must be
Greater than 1.");

          setPlaneBounds(theBounds);
     }

}

** Renderer.java
package rendering;

import java.awt.Image;
import java.awt.image.BufferedImage;
import java.util.ArrayList;

import threeDWorld.raysAndVectors.Ray3D;
import threeDWorld.raysAndVectors.ThreeDPoint;

/**
 * The default interface for a render. Renders will take in a
list of rays and spit out an image.
 * @author jake
 *
 */
public interface Renderer {
     /**
      * The renderer will use these rays to produce its images
      * @param theRays the rays to use
      */
     public void setRays(ArrayList<Ray3D> theRays);
```

```
     /**
      * Gives you the bounds of the viewing plane
      *
      * Assume that:
      *
      *  2--------3
      *  |        |
      *  |        |
      *  |        |
      *  |        |
      *  1--------4
      *
      * @param theBounds the bounds of the plane
      */
     public void setPlaneBounds(ArrayList<ThreeDPoint>
theBounds);

     /**
      * Render the image and return it.
      * Precondition: setRays and setPlaneBounds have already
been called
      * @return the image
      */
     public BufferedImage run();
}

** VisualRay.java
package rendering;
import java.awt.Color;


public class VisualRay {
     public int X, X2, Y, Y2; // X1 & Y1 are start points, X2 &
Y2 are end points
     public Color color;
     public boolean flag;

     //public void setStartPoint(int x, int y){
     //   X1 = x;
     //   Y1 = y;
     //}
```

```
     public void setPoint(int x, int y){
          X = x;
          Y = y;
     }
     public void setColor(Color c){
          color = c;
     }

}


** Benchmark.java

package runners;

import globalStuff.Timekeeper;
import threeDWorld.World;
import tracing.local.RayTraceRender;

public class Benchmark {

     public double run() {
          //World myWorld, double ambLight, int thread, int w,
int h, int aa, int saa
          RayTraceRender myRender = new
RayTraceRender(World.getFireAndIce(), 0.5, 12, 400, 200, 2, 5,
null);

          Timekeeper.init();

          long currentTime = System.currentTimeMillis();

          (new Thread(myRender)).run();

          while (!myRender.isFinished()) {
               System.out.println(myRender.getProgress());
               try {
                    Thread.sleep(50);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }
```

```
          return Double.valueOf(myRender.getTotalRays() /
((System.currentTimeMillis() - currentTime) / 1000.0));
     }

     /**
      * @param args
      */
     public static void main(String[] args) {
          Benchmark bm = new Benchmark();
          System.out.println("Rays per second: " + bm.run());

     }

}

** MySQLPixelRender.java

package runners;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import javax.imageio.ImageIO;

import threeDWorld.raysAndVectors.ThreeDPoint;
import tracing.network.PixelPassback;

/**
 * This class renders all the pixels stored in the pixel
database.
 * @author ryan
 *
 */
public class MySQLPixelRender implements PixelPassback {
     private int width;
     private int height;

     private int pixelsReturned;
```

```
     private int pixelsNeeded;

     private int threadsDone;

     private long startingTime;

     private ExecutorService myExe;

     private BufferedImage myImage;

     private int threadCount;

     /**
      * Gets a new pixel renderer ready to go
      * @param w the width of the image
      * @param h the height of the image
      * @param t the number of the threads to use in the render
      */
     public MySQLPixelRender(int w, int h, int t) {
          width = w;
          height = h;
          pixelsNeeded = w * h;
          threadCount = t;
     }

     /**
      * Renders the image found in the table on the database
      */
     public void renderImage() {
          startingTime = System.currentTimeMillis();
          myImage = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
          myExe = Executors.newFixedThreadPool(threadCount);


          // +1 to give some overlap and insure we don't miss
any.
          int pixelShares = (pixelsNeeded / threadCount) + 1;
          int widthIncrement = width / threadCount;

          int currentWidth = 0;
```

```
          int i = 0;
          while (i < threadCount) {
               myExe.execute(new MySQLPixelRenderThread(width,
height, currentWidth, pixelShares, this));

               currentWidth += widthIncrement;

               i++;
          }
     }



     @Override
     public void done() {
          threadsDone++;
     }

     @Override
     public void gotPixel(ThreeDPoint pixel) {
          try {
               myImage.setRGB((int) pixel.x, (int) pixel.y, (int)
pixel.z);
               pixelsReturned++;
          } catch (Exception e) {
               // do nothing. strange Java bug.
          }
     }

     /**
      * Gets the progress of the current render, if any.
      * @return a number between 0 (just starting) and 1 (done)
      */
     public double getProgress() {
          return Double.valueOf(pixelsReturned) /
Double.valueOf(pixelsNeeded);
     }

     /**
      * Returns the number of pixels being calculated per second
      * @return pixels per second
      */
```

```
     public double getPixelsPerSecond() {
          long time = System.currentTimeMillis() - startingTime;
          time /= 1000;

          if (time == 0 || pixelsReturned == 0) { return 0.0; }

          return pixelsReturned / time;
     }

     /**
      * Returns true if the render is complete
      * @return
      */
     public boolean isFinished() {
          if (threadCount == threadsDone) {
               myExe.shutdown();
               return true;
          }

          return false;
     }

     /**
      * Gets the image rendered
      * @return the image
      */
     public BufferedImage getImage() { return myImage; }

     public static void main(String[] args) {
          MySQLPixelRender myRender = new MySQLPixelRender(800,
400, 7);
          myRender.renderImage();

          while (!myRender.isFinished()) {
               System.out.println("Progress: " +
myRender.getProgress() + " at a rate of " +
myRender.getPixelsPerSecond() + " pixels per second.");
               try {
                    Thread.sleep(5000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
```

```
          }

          System.out.println("Done");

          BufferedImage myImage = myRender.getImage();

          File outputfile = new File("saved.png");
         try {
               boolean b = ImageIO.write(myImage, "png",
outputfile);
               System.out.println("Complete: " + (b ? "saved" :
"not saved"));
          } catch (IOException e) {
               e.printStackTrace();
          }
     }


}

** MySQLPixelRenderThread.java

package runners;

import globalStuff.MySQL;

import java.awt.Color;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import threeDWorld.raysAndVectors.ThreeDPoint;
import tracing.network.PixelPassback;

/**
 * This class is responsible for sampling part of the data in
the MySQL database and returning the pixels containted (after
averaging)
 * @author ryan
 *
 */
public class MySQLPixelRenderThread implements Runnable {
```

```
     private PixelPassback toPB;

     private int height;

     private int startingWidth;
     private int totalPixels;

     public MySQLPixelRenderThread(int w, int h, int sw, int tp,
PixelPassback toReturn) {
          System.out.println("Starting at: " + sw + " to get " +
tp);
          height = h;
          startingWidth = sw;
          totalPixels = tp;

          toPB = toReturn;
     }

     @Override
     public void run() {
          int i = startingWidth;
          int ii = 0;
          int donePixels = 0;

          Connection myConn = MySQL.getSQLConnection();


          try {
               PreparedStatement myStatement =
myConn.prepareStatement("select * from pixels where CoordX=(?)
and CoordY=(?)");
               while (donePixels != totalPixels) {
                    myStatement.setInt(1, i);
                    myStatement.setInt(2, ii);

                    ResultSet mySet = myStatement.executeQuery();
                    int red = 0;
                    int green = 0;
                    int blue = 0;
                    int numPixels = 0;
```

```
                    while (mySet.next()) {
                         Color myColor = new
Color(mySet.getInt("Color"));

                         red += myColor.getRed();
                         green += myColor.getGreen();
                         blue += myColor.getBlue();

                         numPixels++;
                    }
                    if (numPixels != 0) {
                         red /= numPixels;
                         green /= numPixels;
                         blue /= numPixels;

                         Color currentColor = new Color(red,
green, blue);
                         toPB.gotPixel(new ThreeDPoint(i, ii,
currentColor.getRGB()));
                    } else {
                         // there were no pixels there, but we
still have to return something.
                         System.out.println("nothing at " + i +
"," + ii + " when starting at " + startingWidth);
                         toPB.gotPixel(new ThreeDPoint(-1, -1,
-1));
                    }

                    ii++;
                    if (ii > height) { ii = 0; i++; }

                    donePixels++;
               }


          } catch (Exception e) {
               e.printStackTrace();
          }


          toPB.done();
     }
```

```
}

** RenderServer.java

package runners;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import threeDWorld.World;
import tracing.network.RayTraceServer;

public class RenderServer {

     private final int port = 6002;
     private final double ambLight = 0.4;
     private final int width = 2000;
     private final int height = 1000;
     private final int aa = 16;
     private final int saa = 50;

     public void run() {
          // int port, World myWorld, double ambLight, int w,
int h, int aa)
          RayTraceServer myServ = new RayTraceServer(port,
World.getFireAndIce(), ambLight, width, height, aa, saa);

          myServ.addSpeedMapEntry("172.17.114.215", 25);
          myServ.addSpeedMapEntry("172.17.114.210", 43);
          myServ.addSpeedMapEntry("172.17.114.209", 44);
          myServ.addSpeedMapEntry("172.17.114.214", 25);

          myServ.addSpeedMapEntry("172.17.114.156", 160);
          myServ.addSpeedMapEntry("172.17.114.233", 160);
          myServ.addSpeedMapEntry("172.17.114.190", 160);
          myServ.addSpeedMapEntry("172.17.114.234", 160);

          myServ.addSpeedMapEntry("172.17.114.218", 160);
```

```
          myServ.addSpeedMapEntry("172.17.114.247", 160);
          myServ.addSpeedMapEntry("172.17.114.236", 160);
          myServ.addSpeedMapEntry("172.17.114.223", 160);


          System.out.println("Waiting for connections...");

          // listen for 30 seconds..
          int got = myServ.listenForIncomingConnection(30000);

          System.out.println("Got " + got + " connections.");

          myServ.start();

          while (myServ.getProgress() != 1.0) {
               System.out.println(myServ.getProgress());
               try {
                    Thread.sleep(5000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }

          BufferedImage theImage = myServ.getImage();
          File outputfile = new File("saved.png");
         try {
               boolean b = ImageIO.write(theImage, "png",
outputfile);
               System.out.println("Complete: " + (b ? "saved" :
"not saved"));
          } catch (IOException e) {
               e.printStackTrace();
          }
     }

     public static void main(String[] args) {
          RenderServer myServer = new RenderServer();
          myServer.run();

     }

}
```

```
** TradSimpleDynamicWorld.java

package runners;

import globalStuff.Timekeeper;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import threeDWorld.World;
import tracing.local.RayTraceRender;
import tracing.processors.ImageCreator;

/**
 * This class renders a bunch of images that make up a simple
tween
 * @author ryan
 *
 */
public class TradSimpleDynamicWorld {
     private final int threadCount = 7;
     private final int width = 800;
     private final int height = 400;
     private final int antialias = 4;
     private final int shadowAA = 10;

     public static void main(String[] args) {
          TradSimpleDynamicWorld myWorld = new
TradSimpleDynamicWorld();

          int startFrame = Integer.valueOf(args[0]);
          int endFrame = Integer.valueOf(args[1]);

          myWorld.run(startFrame, endFrame);
     }

     public void run(int start, int end) {
          long totalTime = System.currentTimeMillis();
```

```
          System.out.println("Building world...");


          Timekeeper.init();


          // Stop! Ray tracing time!

          System.out.println("Built. Starting trace...");

          // create the image processor
          ImageCreator myCreator = new ImageCreator(width,
height);

          System.out.println("Starting ray trace...");
          int i = start;
          while (i <= end) {
               System.out.println("Frame " + i);
               World myWorld = World.getFireAndIceTween(i);
               RayTraceRender myTrace = new
RayTraceRender(myWorld, 0.5, threadCount, width, height,
antialias, shadowAA, myCreator);

               (new Thread(myTrace)).run();

               while (!myTrace.isFinished()) {
                    System.out.println("Frame " + i + " progress:
" + myTrace.getProgress());
                    try {
                         Thread.sleep(1000);
                    } catch (InterruptedException e) {
                         e.printStackTrace();
                    }
               }

               myTrace.shutdown();

               BufferedImage theImage = myCreator.getImage();
               File outputfile = new File("saved" + i + ".png");
               try {
                    ImageIO.write(theImage, "png", outputfile);
               } catch (IOException e) {
```

```
                    e.printStackTrace();
               }

               i++;
          }



          System.out.println("Total time:" +
(System.currentTimeMillis() - totalTime));
          System.out.println();
          System.out.println(Timekeeper.getTimeReport());
     }
}

** TradSimpleStaticWorld.java

package runners;

import globalStuff.Timekeeper;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

import threeDWorld.World;
import tracing.local.RayTraceRender;
import tracing.processors.ImageCreator;

public class TradSimpleStaticWorld {

     private final int threadCount = 7;
     private final int width = 1600;
     private final int height = 800;
     private final int antialias = 16;
     private final int shadowAA = 32;


     private final boolean rayTrace = true;
```

```
     public void run() {
          long totalTime = System.currentTimeMillis();
          System.out.println("Building world...");

          World myWorld = World.getFireAndIce();
          Timekeeper.init();


          // Stop! Ray tracing time!

          System.out.println("Built. Starting trace...");

          // create the image processor
          ImageCreator myCreator = new ImageCreator(width,
height);

          if (rayTrace) {
               System.out.println("Starting ray trace...");
               RayTraceRender myTrace = new
RayTraceRender(myWorld, 0.5, threadCount, width, height,
antialias, shadowAA, myCreator);


               (new Thread(myTrace)).run();

               while (!myTrace.isFinished()) {
                    // block
                    double progress = myTrace.getProgress();
                    double currTime = System.currentTimeMillis()
- totalTime;

                    double rate = progress / currTime;
                    double ETA = (1.0 - progress) / rate;
                    ETA /= 1000.0;
                    ETA /= 60.0;

                    System.out.println("Ray: " +
myTrace.getProgress() + " ETA: " + ETA + " minutes");
                    try {
                         Thread.sleep(5000);
```

```
                    } catch (InterruptedException e) {
                         e.printStackTrace();
                    }
               }


               myTrace.shutdown();
               System.out.println("Ray trace done.");
          }


          BufferedImage theImage = myCreator.getImage();
          File outputfile = new File("saved.png");
         try {
               boolean b = ImageIO.write(theImage, "png",
outputfile);
               System.out.println("Complete: " + (b ? "saved" :
"not saved"));
          } catch (IOException e) {
               e.printStackTrace();
          }

          System.out.println("Total time:" +
(System.currentTimeMillis() - totalTime));
          System.out.println();
          System.out.println(Timekeeper.getTimeReport());

     }

     public static void main(String[] args) {
          TradSimpleStaticWorld myTrace = new
TradSimpleStaticWorld();
          myTrace.run();
     }
}

** LightProperties.java
package threeDWorld.raysAndVectors;

import java.awt.Color;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
```

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

/**
 * This class stores all of the information we may need to
determine the color of a final pixel. I.E., this class stores
color,
 * transparency, reflectivity, etc. These are abstracted into a
seperate class so we can easily add more properties later.
 *
 * @author ryan
 *
 */
public class LightProperties {

     public static final int numBytes = ((Integer.SIZE / 8) * 6)
+ 1;

     public LightProperties() {
          objectColor = new Color(0,0,0);
     }

     /**
      * The object that the light properties came from. Should
be set if possible.
      */
     public Object from;

     /**
      * If the object is a light.
      *
      * If no, then we can have transparency and reflection
      * If yes, then we only have lightIntensity
      */
     public boolean isLight;

     /**
      * The color of the object
      */
     public Color objectColor;
```

```
     /**
      * Scale of 0 (completely opaque) to 254 (completely
transparent)
      */
     public int transparency;

     /**
      * The index of the refraction of the object
      */
     public double indexOfRefraction;

     /**
      * True if mirror, false if otherwise
      */
     public boolean reflection;

     /**
      * Set to true to not show shadows on this surface
      */
     public boolean noShadow;

     /**
      * Returns a copy of the object
      * @return a copy
      */
     public LightProperties clone() {
          LightProperties toReturn = new LightProperties();
          toReturn.from = this.from;
          toReturn.indexOfRefraction = this.indexOfRefraction;
          toReturn.isLight = this.isLight;
          toReturn.objectColor = new Color(objectColor.getRed(),
objectColor.getGreen(), objectColor.getBlue());
          toReturn.reflection = this.reflection;
          toReturn.transparency = this.transparency;
          toReturn.noShadow = this.noShadow;

          return toReturn;
     }

     public byte[] getBinaryData() {
          ByteArrayOutputStream myBAOS = new
ByteArrayOutputStream();
```

```
          DataOutputStream dos = new DataOutputStream(myBAOS);
          try {
               dos.writeInt(transparency);
               dos.writeBoolean(reflection);
               dos.writeInt(objectColor.getRGB());
               dos.writeBoolean(isLight);
          } catch (IOException e) {
               e.printStackTrace();
          }

          return myBAOS.toByteArray();
     }

     public void setBinaryData(byte[] s) {
          DataInputStream dis = new DataInputStream(new
ByteArrayInputStream(s));

          try {
               this.transparency = dis.readInt();
               this.reflection = dis.readBoolean();
               this.objectColor = new Color(dis.readInt());
               this.isLight = dis.readBoolean();
          } catch (IOException e) {
               e.printStackTrace();
          }
     }

     /**
      * Tests the toString and parseString methods
      * @param args none
      */
     public static void main(String[] args) {
          LightProperties ourProps = new LightProperties();

          ourProps.reflection = true;
          ourProps.objectColor = Color.orange;

          System.out.println(ourProps.objectColor.getRGB());

          byte[] theData = ourProps.getBinaryData();

          ourProps = new LightProperties();
```

```
          ourProps.setBinaryData(theData);

          System.out.println(ourProps.objectColor.getRGB());
     }
}

** Ray3D.java

package threeDWorld.raysAndVectors;

import java.awt.Color;
import java.util.ArrayList;


/**
 * Rays are a series of "vectors". -- remember, these vectors
are really line segments -- they have a start and end point.
 *
 * If the ray is done being traced, the last vector's magnitude
will be 0,0,0.
 * @author ryan
 *
 */
public class Ray3D {
     protected ArrayList<Vector3D> theVectors;
     protected int flag;
     protected double angle;
     protected int realX;
     protected int realY;

     /**
      * This flag indicates that this ray's flag has not been
set
      */
     public static final int RAY_FLAG_UNFLAGGED = 0;

     /**
      * Use this flag if the ray is fully traced and started
from the eye (ended at a light source)
      */
     public static final int RAY_FLAG_COMPLETE_FROM_EYE = 1;
```

```
     /**
      * Use this flag if the ray is fully traced and started
from the light source (ended at the eye)
      */
     public static final int RAY_FLAG_COMPLETE_FROM_LIGHT = 2;

     /**
      * Use this flag if you are done tracing the ray for some
reason besides its termination. For example, exceeding the max
number of bounces
      */
     public static final int RAY_FLAG_COMPLETE_KILLED = 3;

     /**
      * Use this flag if the ray is currently being traced from
the eye
      */
     public static final int RAY_FLAG_TRACING_FROM_EYE = 4;

     /**
      * Use this flag if the ray is currently being traced from
the light
      */
     public static final int RAY_FLAG_TRACING_FROM_LIGHT = 5;

     public Ray3D() {
          theVectors = new ArrayList<Vector3D>();
     }

     /**
      * Add a vector to the ray. This will reduce the vector.
      * @param toAdd the vector to add
      */
     public void addVector(Vector3D toAdd) {
          // first, reduce the vector
          theVectors.add(toAdd);
     }

     /**
      * Removes the given vector at the passed index from the
ray
```

```
     * @param index the index of the vector to remove
     */
    public void removeVector(int index) {
        theVectors.remove(index);
    }

    /**
     * Gets the number of vectors currently stored in the ray
     * @return the number of vectors
     */
    public int getVectorCount() {
        return theVectors.size();
    }

    /**
     * Gets the vector at index i. The first vector stored is
0.
     * @param i the index
     * @return the vector
     */
    public Vector3D getVector(int i) {
        return theVectors.get(i);
    }

    /**
     * Set the coordinates where this ray should be rendered on
the actual final image
     * @param x the x value
     * @param y the y value
     */
    public void setRealCoords(int x, int y) {
        realX = x;
        realY = y;
    }

    /**
     * Returns the X coordinate where this ray should be
rendered on the actual final image
     * @return the X coordinate
     */
    public int getRealX() { return realX; }
```

```
     /**
      * Returns the Y coordinate where this ray should be
rendered on the actual final image
      * @return the Y coordinate
      */
     public int getRealY() { return realY; }

     /**
      * Sets the rays flag
      * @param flg the flag to set
      */
     public void setFlag(int flg) {
          flag = flg;
     }

     /**
      * Returns the current flag of the ray
      * @return
      */
     public int getFlag() {
          return flag;
     }

     /**
      * Optionally, set the angle between the normal and the
light. Needed for ray trace rendering.
      * @param theAngle the angle
      */
     public void setAngle(double theAngle) {
          if (theAngle < 0.0 || theAngle == Double.NaN ||
String.valueOf(theAngle).equals("NaN")) {
               this.angle = 0.0;
          } else {
               this.angle = theAngle;
          }
     }

     /**
      * Returns the angle between the normal and the light, if
set.
      * @return the angle
      */
```

```
     public double getAngle() {
          return this.angle;
     }
     public String toString() {
          StringBuilder sb = new StringBuilder();

          sb.append(flag);
          sb.append("~");

          for (Vector3D vtd : theVectors) {
               sb.append(vtd);
               sb.append("~");
          }

          return sb.substring(0, sb.length() - 1);
     }


     /**
      * Precondition: The ray must be completed. Otherwise,
we'll return null.
      *
      * Returns the final color that should be rendered on the
camera plane.
      * @return the color
      */
     public Color calculateFinalColor() {

          if (this.getFlag() == Ray3D.RAY_FLAG_COMPLETE_KILLED)
return Color.black;

          // a super fancy way to count down from the eye and up
from the light
          int i = (this.getFlag() ==
Ray3D.RAY_FLAG_COMPLETE_FROM_EYE ? this.getVectorCount() - 1 :
0);
          int inc = (this.getFlag() ==
Ray3D.RAY_FLAG_COMPLETE_FROM_EYE ? -1 : 1);
          int finish = (i == 0 ? this.getVectorCount() - 1 : 0);
// we want to stop before we hit the camera
```

```
          Color myColor =
this.getVector(i).theProps.objectColor;
          i += inc;

          int newR = myColor.getRed();
          int newB = myColor.getBlue();
          int newG = myColor.getGreen();

          while (i != finish) {
               if (this.getVector(i).theProps == null ||
this.getVector(i).theProps.reflection) {
                    // no light properties? we don't care.
                    i += inc;
                    continue;
               }


               newR -= (255 -
(this.getVector(i).theProps.objectColor.getRed()));
               newB -= (255 -
(this.getVector(i).theProps.objectColor.getBlue()));
               newG -= (255 -
(this.getVector(i).theProps.objectColor.getGreen()));

               //newR =
(this.getVector(i).theProps.objectColor.getRed());
               //newB =
(this.getVector(i).theProps.objectColor.getBlue());
               //newG =
(this.getVector(i).theProps.objectColor.getGreen());

               myColor = new Color((newR > 0 ? newR : 0), (newG >
0 ? newG : 0), (newB > 0 ? newB : 0));
               i += inc;
          }

          return myColor;

     }
```

```
     public Ray3D clone() {
          Ray3D newRay = new Ray3D();

          newRay.angle = this.angle;
          newRay.flag = this.flag;
          for (Vector3D vtd : this.theVectors) {
               newRay.addVector(vtd);
          }

          return newRay;
     }

}

** ThreeDPoint.java
package threeDWorld.raysAndVectors;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;


/**
 * Stores a simple point in 3D space.
 * @author ryan
 *
 */
public class ThreeDPoint {

     public double x,y,z;
     public ThreeDPoint() { }

     public ThreeDPoint(double valueX, double valueY, double
valueZ) {
          x = valueX;
          y = valueY;
          z = valueZ;
     }

     /**
```

```
      * Gets the distance between this point and the passed
point
      * @param p the point to test
      * @return the distance
      */
     public double getDistance(ThreeDPoint p) {
          double xDiff,yDiff,zDiff;

          xDiff = Math.pow(x - p.x, 2);
          yDiff = Math.pow(y - p.y, 2);
          zDiff = Math.pow(z - p.z, 2);

          return Math.sqrt(xDiff + yDiff + zDiff);
     }

     /**
      * Returns a distance that can be used in comparision only.
It is the distance formula without the square root.
      * @param p the point ot measure the distance from
      * @return the comparable distance
      */
     public double getComparitveDistance(ThreeDPoint p) {
          double xDiff,yDiff,zDiff;

          xDiff = Math.pow(x - p.x, 2);
          yDiff = Math.pow(y - p.y, 2);
          zDiff = Math.pow(z - p.z, 2);

          return (xDiff + yDiff + zDiff);
     }

     /**
      * Returns the dot product of this point and the point
passed
      * @param p the point to use
      * @return the dot product
      */
     public double getDotProduct(ThreeDPoint p) {
          // a dot product is equal to the product of each term
added together.
          // for example, [1 2 3] dot [4 5 6] = 1*4 + 2*5 + 3*6
= 32
```

```
          return (this.x * p.x) + (this.y * p.y) + (this.z *
p.z);
     }

     /**
      * Returns the cross product of this point and the point
passed
      * @param p the point to use
      * @return the cross product
      */
     public ThreeDPoint getCrossProduct(ThreeDPoint p) {
          // Wikipedia:
http://en.wikipedia.org/wiki/Cross_product
          // defines the cross product as: (a2b3 - a3b2, a3b1 -
a1b3, a1b2 - a2b1)

          return new ThreeDPoint((this.y * p.z) - (this.z *
p.y), (this.z * p.x) - (this.x * p.z), (this.x * p.y) - (this.y
* p.x));
     }

     /**
      * Multiplies each component of the current point by i
      * @param i the number to multiply by
      * @return the resulting vector
      */
     public ThreeDPoint scalerMultiply(double i) {
          return new ThreeDPoint(x * i, y * i, z * i);
     }

     /**
      * Subtracts i from each of the current points.
      * @param i the number to subtract
      * @return the resulting vector
      */
     public ThreeDPoint scalerSubtract(double i) {
          return new ThreeDPoint(x - i, y - i, z - i);
     }

     /**
      * Adds i to each of the current points
      * @param i the number to add
```

```
     * @return the resulting vector
     */
    public ThreeDPoint scalerAddition(double i) {
        return new ThreeDPoint(x + i, y + i, z + i);
    }

    /**
     * Takes in a vector and returns the result of subtraction
     * @param i the vector
     * @return the result
     */
    public ThreeDPoint vectorSubtract(ThreeDPoint i) {
        return new ThreeDPoint(x - i.x, y - i.y, z - i.z);
    }

    /**
     * Takes in a vector and returns the result of addition
     * @param i the vector
     * @return the result
     */
    public ThreeDPoint vectorAdd(ThreeDPoint i) {
        return new ThreeDPoint(x + i.x, y + i.y, z + i.z);
    }

    /**
     * Makes the vector into a unit vector -- i.e., of length
1, but keeping the same proportions.
     */
    public void makeUnitVector() {
        double divisor = Math.sqrt(Math.pow(x, 2) +
Math.pow(y, 2) + Math.pow(z, 2));
        x /= divisor;
        y /= divisor;
        z /= divisor;
    }


    public String toString() {
        return "{" + x + "," + y + "," + z + "}";
    }

    public byte[] getBinaryData() {
```

```
          ByteArrayOutputStream myBAO = new
ByteArrayOutputStream();
          DataOutputStream myDOS = new DataOutputStream(myBAO);
          try {
               myDOS.writeDouble(x);
               myDOS.writeDouble(y);
               myDOS.writeDouble(z);
          } catch (IOException e) {
               e.printStackTrace();
          }

          return myBAO.toByteArray();
     }

     public void setBinaryData(byte[] theData) {
          ByteArrayInputStream myBAI = new
ByteArrayInputStream(theData);
          DataInputStream myDIS = new DataInputStream(myBAI);

          try {
               x = myDIS.readDouble();
               y = myDIS.readDouble();
               z = myDIS.readDouble();
          } catch (IOException e) {
               e.printStackTrace();
          }
     }

     public static void main(String[] args) {
          ThreeDPoint myPoint = new ThreeDPoint(1,2,3);

          System.out.println(myPoint);

          byte[] theBytes = myPoint.getBinaryData();

          myPoint = new ThreeDPoint();
          myPoint.setBinaryData(theBytes);

          System.out.println(myPoint);

     }
```

```
}

** ThreeDPointAvg.java

package threeDWorld.raysAndVectors;
import java.util.ArrayList;


/**
 * This class allows you to average a bunch of ThreeDPoints. You
can put a number of points, calculate the average, add more
points,
 * and repeat.
 *
 * @author ryan
 *
 */
public class ThreeDPointAvg {
     private ArrayList<ThreeDPoint> theList;

     public ThreeDPointAvg() {
          theList = new ArrayList<ThreeDPoint>();
     }

     public void addPoint(ThreeDPoint toAdd) {
          theList.add(toAdd);
     }

     public void reset() {
          theList = new ArrayList<ThreeDPoint>();
     }

     public ThreeDPoint getAverage() {
          int c = 0;
          ThreeDPoint toReturn = new ThreeDPoint();


          for (ThreeDPoint p : theList) {
               toReturn.x += p.x;
               toReturn.y += p.y;
               toReturn.z += p.z;
               c++;
```

```
          }

          return toReturn.scalerMultiply(1.0 / c);
     }
}

** Vector3D.java

package threeDWorld.raysAndVectors;

import java.awt.Color;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;


/**
 * This isn't really a vector -- it is more of a line. I.E., the
components do not indicate a magnitude, only a direction.
 *
 * @author ryan
 *
 */
public class Vector3D {

     public Vector3D() {
          startingPosition = new ThreeDPoint();
          direction = new ThreeDPoint();
          theProps = new LightProperties();
     }


     /**
      * The position of the vector: x, y, and z
      */
     public ThreeDPoint startingPosition;

     /**
      * The direction of the vector: i, j, and k
      */
```

```
     public ThreeDPoint direction;

     /**
      * Use this to store the properites of the starting point.
We'll figure out how it looks when we do
      * the ray tracing.
      */
     public LightProperties theProps;

     /**
      * Set this to true if the vector is starting at the eye
      */
     public boolean isAtEye = false;

     /**
      * Set this to true is the vector is starting at the light
source
      */
     public boolean isAtLightSource = false;

     public Vector3D clone() {
          Vector3D toReturn = new Vector3D();

          toReturn.startingPosition.x = startingPosition.x;
          toReturn.startingPosition.y = startingPosition.y;
          toReturn.startingPosition.z = startingPosition.z;

          toReturn.direction.x = direction.x;
          toReturn.direction.y = direction.y;
          toReturn.direction.z = direction.z;

          toReturn.isAtEye = isAtEye;
          toReturn.isAtLightSource = isAtLightSource;

          toReturn.theProps = theProps;
          return toReturn;
     }

     public byte[] getBinaryData() {
          ByteArrayOutputStream myBAO = new
ByteArrayOutputStream();
          DataOutputStream myDOS = new DataOutputStream(myBAO);
```

```
          try {
               myDOS.writeBoolean(isAtEye);
               myDOS.writeBoolean(isAtLightSource);

     myDOS.write(this.startingPosition.getBinaryData());
               myDOS.write(this.direction.getBinaryData());
               myDOS.write(this.theProps.getBinaryData());
          } catch (IOException e) {
               e.printStackTrace();
          }

          return myBAO.toByteArray();
     }

     public void setBinaryData(byte[] b) {
          // now read in the data
          ByteArrayInputStream myStream = new
ByteArrayInputStream(b);
          DataInputStream myDIS = new DataInputStream(myStream);

          try {
               this.isAtEye = myDIS.readBoolean();
               this.isAtLightSource = myDIS.readBoolean();

               this.startingPosition.x = myDIS.readDouble();
               this.startingPosition.y = myDIS.readDouble();
               this.startingPosition.z = myDIS.readDouble();

               this.direction.x = myDIS.readDouble();
               this.direction.y = myDIS.readDouble();
               this.direction.z = myDIS.readDouble();

               theProps.transparency = myDIS.readInt();
               theProps.reflection = myDIS.readBoolean();
               theProps.objectColor = new Color(myDIS.readInt());
               theProps.isLight = myDIS.readBoolean();
          } catch (IOException e) {
               e.printStackTrace();
          }
     }

     /**
```

```
      * This method tests the binary input/output functions of
the ray
      * @param args
      */
     public static void main(String[] args) {
          Vector3D myVec = new Vector3D();
          myVec.direction = new ThreeDPoint(3,2,1);
          myVec.startingPosition = new ThreeDPoint(1,2,3);
          myVec.isAtLightSource = true;
          myVec.theProps.objectColor = Color.pink;

          byte[] data = myVec.getBinaryData();

          myVec = new Vector3D();
          myVec.setBinaryData(data);


          System.out.println(myVec.startingPosition);
          System.out.println(myVec.direction);
          System.out.println(myVec.theProps.objectColor.getRGB()
+ " should be " + Color.pink.getRGB());
          System.out.println((myVec.isAtLightSource ? "true" :
"false"));

     }


}

** Camera3D.java

package threeDWorld.threeDObjectsDefinitions;

import java.util.ArrayList;

import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;

/**
 * Our "cameras" contain two parts: an eye, and a screen. The
eye sits behind the screen and looks at everything on the
screen. The screen
```

```
 * is a 2D plane, which will be represented by 4 points in 3D
space -- one for each corner. The eye is simply a point in
space.
 *
 * @author ryan
 *
 */
public interface Camera3D {


     /**
      * return the area of the plane (w*h). This is useful for
deciding how many vectors we are going to ask you for.
      * @return the area of the plane, in units squared.
      */
     public double getPlaneArea();

     /**
      *
      * Test to see if the vector (a) passes through the plane
and (b) hits the eye.
      * You may want to give your eye a certain threshold --
very few vectors pass through point
      * (4.0000000, 4.0000000, 4.0000000), so you may want to
accept vectors with a range (+/- 0.5, for example).
      *
      * This method needs to be hyper-fast -- it will be called
for EVERY SINGLE VECTOR (not just ray!) when performing
      * photon tracing. (That's a lot!) It will also be called
as soon as we start the metropolis algorithm (after all
      * bidirectional tracing is done.)
      *
      * Return the point where the ray hit the PLANE, not the
eye.
      *
      * @param theVector the vector to test
      * @return the point the vector hit, or null if it did not
hit.
      */
     public ThreeDPoint testVector(Vector3D theVector);

     /**
```

```
     * This method is mainly used for ray tracing (i.e. eye to
light source).
     *
     * It needs to return a given number of EQUALLY SPACED
vectors that go from the eye, through the plane. For example, if
the area of your
     * plane is 4 units^2, and this method is called with count
= 4, then you'd return four vectors, each centered at the middle
of each
     * of your square units. The returned vectors starting
point should be the point where the vector hits the plane, and
the direction of each
     * vector should be directed through the plane.
     *
     * This method is only called once, and thus does not have
to be that speedy.
     *
     * @param count the number of vectors to return
     * @return
     */
    public ArrayList<Vector3D> generateVectors(int count);

    /**
     * Return a vector with magnitude 0,0,0 that starts at the
eye. Make sure to set the "isAtEye" property to true.
     * @return the null vector
     */
    public Vector3D getNullVector();

    /**
     * Returns the width of the plane
     * @return the width
     */
    public double getPlaneWidth();

    /**
     * Returns the height of the plane
     * @return the height
     */
    public double getPlaneHeight();

    /**
```

```
      * Generate a vector that passes through the given X,Y
point on the plane of the camera. The x value will always be
between 0 and the
      * width of the plane (as defined by getPlaneWidth()) and
the y value will always be between 0 and the height of the
plane.
      * @param x the x value
      * @param y the y value
      * @return
      */
     public Vector3D getVectorThroughPoint(double x, double y);

     /**
      * Returns an x,y coordinate (don't return a Z point) that
maps the passed point (which you may assume is on the camera's
viewing plane) on to an x,y plane
      * like an image (i.e., not a standard cord. grid)
      * @param pointHit the point we hit
      * @return the mapped point
      */
     public ThreeDPoint getXYPointAt(ThreeDPoint pointHit);

     /**
      * Return the top left point of the viewing plane
      * @return
      */
     public ThreeDPoint getTopLeftPoint();



}

** Lightsource.java

package threeDWorld.threeDObjectsDefinitions;

import java.util.ArrayList;

import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;

/**
```

```
 * Reprsents a lightsource.
 *
 * @author ryan
 *
 */
public interface Lightsource {

     /**
      *
      * Test to see if the vector hits the lightsource.
      * You may want to give your light a certain threshold --
very few vectors pass through point
      * (4.0000000, 4.0000000, 4.0000000), so you may want to
accept vectors with a range (+/- 0.5, for example).
      *
      * This method needs to be hyper-fast -- it will be called
for EVERY SINGLE VECTOR (not just ray!) when performing
      * ray tracing. (That's a lot!) It will also be called as
soon as we start the metropolis algorithm (after all
      * bidirectional tracing is done.)
      *
      * @param theVector the vector to test
      * @return the point the vector hit, or null if it did not
hit.
      */
     public ThreeDPoint testVector(Vector3D theVector);

     /**
      * This method is mainly used for photon tracing (i.e.
light to eye).
      *
      * It needs to return a given number of EQUALLY SPACED
vectors that go out from the light. The returned vectors
starting point should be
      * the lightsource.
      *
      * This method is only called once, and thus does not have
to be that speedy.
      *
      * @param count the number of vectors to return
      * @return
      */
```

```
     public ArrayList<Vector3D> generateVectors(int count);

     /**
      * Return a vector with magnitude 0,0,0 that starts at the
light. Make sure to set the "isAtLight" property to true, and to
set the proper
      * light properties.
      * @return the null vector
      */
     public Vector3D getNullVector();

     /**
      * Return a list of ThreeDPoints evenly distributed across
the surface of the light.
      * @param num the number of ThreeDPoints to return
      * @return
      */
     public ArrayList<ThreeDPoint> getPossibleHittingPoints(int
num);


}

** Object3D.java
package threeDWorld.threeDObjectsDefinitions;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;

public interface Object3D {
     /**
      * Check to see if the passed vector is affected by this
object. If it is, return true. If not, return false.
      *
      * @param theVector the vector to check
      * @return the point the vector hit, or null if it did not
hit.
      */
     public ThreeDPoint checkForCollision(Vector3D theVector);

     /**
```

```
      * Precondition: Assume that checkForCollision returned
true on theVector when this method is called.
      *
      * Given a vector that has hit your object (theVector),
return the resulting vector. It is important to note that you
need to set
      * the LightProperties of the new vector -- do not make
your new vector's LightProperties dependent on the passed vector
-- that math is
      * done later.
      *
      * @param theVector the vector that has hit your object
      * @param hittingPoint the point on your object that the
vector is hitting, as passed by you in checkForCollision.
      * @return the resulting vector
      */
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint);

     /**
      * Returns the light properties of your object at the
passed point
      * @param passed the ThreeDPoint to get the LPs
      * @return the LPs
      */
     public LightProperties getLP(ThreeDPoint passed);

     /**
      * Return the cosine of the angle between the hit point
from the incident ray and the normal
      * @param hit the point to get the angle at
      * @param incident the ray coming into the object that
we're trying to calculate for
      * @return the angle
      */
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
ThreeDPoint incident);
}

** SphereLight.java

package threeDWorld.threeDObjectsImpl.lights;
```

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Lightsource;
import threeDWorld.threeDObjectsImpl.objects.SolidSphere;

/**
 * A light represented by a sphere, emits light in all
directions.
 * @author ryan
 *
 */
public class SphereLight implements Lightsource{
     private ThreeDPoint center;
     private double radius;
     private LightProperties ourProps;
     private SolidSphere ourSphere;
     private ArrayList<ThreeDPoint> pointsInside;

     /**
      * Creates a new SphereLight with the passed properties
      * @param r the radius
      * @param c the center point
      * @param i the intensity of the light
      * @param theColor the color of the light
      */
     public SphereLight(double r, ThreeDPoint c, Color theColor)
{
          center = c;
          radius = r;
          ourProps = new LightProperties();
          ourProps.isLight = true;
          ourProps.objectColor = theColor;
          ourProps.from = this;
          ourSphere = new SolidSphere(r, c, null);
          pointsInside = new ArrayList<ThreeDPoint>();
          generatePointsInside();
```

```
     }

     public ArrayList<Vector3D> generateVectors(int count) {
          ArrayList<Vector3D> toReturn = new
ArrayList<Vector3D>();
          Random r = new Random(42);

          while (toReturn.size() != count) {
               Vector3D myVec = new Vector3D();
               ThreeDPoint toStart =
pointsInside.get(r.nextInt(pointsInside.size()));
               myVec.startingPosition = new
ThreeDPoint(toStart.x, toStart.y, toStart.z);
               myVec.direction = new ThreeDPoint((r.nextDouble()
* 2) - 1, (r.nextDouble() * 2) - 1, (r.nextDouble() * 2) - 1);
               myVec.isAtLightSource = true;
               myVec.theProps = ourProps;
               toReturn.add(myVec);
          }

          return toReturn;
     }


     public Vector3D getNullVector() {
          Vector3D newVec = new Vector3D();
          newVec.startingPosition = center;
          newVec.isAtLightSource = true;
          newVec.theProps = ourProps;


          return newVec;
     }

     public double getRadius() {
          return radius;
     }

     public ThreeDPoint testVector(Vector3D theVector) {
          return ourSphere.checkForCollision(theVector);
     }
```

```
     private void generatePointsInside() {
          Random r = new Random(42);


     pointsInside.add(this.getNullVector().startingPosition);

          while (pointsInside.size() != 500) {
               Vector3D myVec = new Vector3D();
               myVec.startingPosition = new ThreeDPoint(center.x,
center.y, center.z);

               myVec.direction.x = (r.nextDouble() * 2.0) - 1.0;
               myVec.direction.y = (r.nextDouble() * 2.0) - 1.0;
               myVec.direction.z = (r.nextDouble() * 2.0) - 1.0;

               myVec.direction.makeUnitVector();

               myVec.startingPosition =
myVec.startingPosition.vectorAdd(myVec.direction.scalerMultiply(
getRadius()));

               pointsInside.add(myVec.startingPosition);

          }

     }

     @Override
     public ArrayList<ThreeDPoint> getPossibleHittingPoints(int
num) {
          // draw a sample of size NUM from pointsInside

          ArrayList<ThreeDPoint> toReturn = new
ArrayList<ThreeDPoint>();
          while (toReturn.size() != num) {
               toReturn.add(pointsInside.get(toReturn.size()));
          }
          return toReturn;
     }


}
```

```
** SquareLight.java

package threeDWorld.threeDObjectsImpl.lights;

import java.awt.Color;
import java.util.ArrayList;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.ThreeDPointAvg;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Lightsource;
import threeDWorld.threeDObjectsImpl.objects.SolidRectangle;

public class SquareLight implements Lightsource {

     private SolidRectangle myRect;
     private LightProperties ourProps;
     private ThreeDPoint vec1,vec2,vec3,vec4;

     /**
      * Creates a new light
      * @param v1 see the SolidRectangle documentation
      * @param v2 see the SolidRectangle documentation
      * @param v3 see the SolidRectangle documentation
      * @param v4 see the SolidRectangle documentation
      * @param theColor the color of the light
      * @param i the intensity of the light
      */
     public SquareLight(ThreeDPoint v1, ThreeDPoint v2,
ThreeDPoint v3, ThreeDPoint v4, Color theColor) {
          ourProps = new LightProperties();
          ourProps.isLight = true;
          ourProps.objectColor = theColor;
          ourProps.from = this;

          vec1 = v1;
          vec2 = v2;
          vec3 = v3;
          vec4 = v4;
```

```
          myRect = new SolidRectangle(v1, v2, v3, v4, null);

     }

     public ArrayList<Vector3D> generateVectors(int count) {
          // TODO Auto-generated method stub
          return null;
     }

     @Override
     public Vector3D getNullVector() {
          Vector3D newVec = new Vector3D();

          ThreeDPointAvg myAvg = new ThreeDPointAvg();
          myAvg.addPoint(vec1);
          myAvg.addPoint(vec2);
          myAvg.addPoint(vec3);
          myAvg.addPoint(vec4);

          newVec.startingPosition = myAvg.getAverage();
          newVec.isAtLightSource = true;
          newVec.theProps = ourProps;


          return newVec;
     }

     @Override
     public ThreeDPoint testVector(Vector3D theVector) {
          return myRect.checkForCollision(theVector);
     }

     @Override
     public ArrayList<ThreeDPoint> getPossibleHittingPoints(int
num) {
          // TODO Auto-generated method stub
          return null;
     }

}

** CylinderTester.java
```

```
package threeDWorld.threeDObjectsImpl.objects;

import java.awt.Color;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;

public class CylinderTester {
     public static void main(String[] args){
          ThreeDPoint point1 = new ThreeDPoint(2.0, 2.0, 0.0);
          ThreeDPoint point2 = new ThreeDPoint(2.0, 2.0, 2.0);
          ThreeDPoint point;

          LightProperties myProps = new LightProperties();

          myProps.isLight = false;
          myProps.objectColor = new Color(199,21,133);
          myProps.reflection = false;
          myProps.transparency = 0;
          myProps.indexOfRefraction = 1.0 + (1.0/3.0);

          SolidCylinder myCylinder = new SolidCylinder(point1,
point2, 1.0, myProps);
          myCylinder.transformAxis(point1);
          myCylinder.transformAxis(point2);
          System.out.println("point1 is " + point1);
          System.out.println("point2 is " + point2);
          ThreeDPoint starting = new ThreeDPoint (2,2,3);
          ThreeDPoint slope = new ThreeDPoint(0,0,-1);
          Vector3D myVec = new Vector3D();
          myVec.startingPosition = starting;
          myVec.direction = slope;
          myVec.theProps = myProps;
          if (myCylinder.checkForCollision(myVec) == null)
               System.out.println("no collision");
          else{
               point = myCylinder.checkForCollision(myVec);
               System.out.println(point.x + ", " + point.y + ", "
+ point.z);
//             myCylinder.transformAxis(point1);
```

```
//             myCylinder.transformAxis(point2);
//             myCylinder.transformAxis(point);
//             System.out.println("point1 is " + point1);
//             System.out.println("point2 is " + point2);
//             System.out.println("point is " + point);
          }
     }
}

** ImageCylinder.java

package threeDWorld.threeDObjectsImpl.objects;

import java.awt.image.BufferedImage;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import globalStuff.Timekeeper;

/**
 * This is a subclass of the SolidCylinder class that projects
the passed image onto the cylinder.
 * @author kathy
 *
 */

public class ImageCylinder extends SolidCylinder{

     private BufferedImage img;
     private LightProperties theLP;
     private int rot;

     public ImageCylinder(ThreeDPoint cp1, ThreeDPoint cp2,
double radius, BufferedImage toProject, LightProperties lps, int
rotation) {
          super(cp1, cp2, radius, lps);
          img = toProject;
          theLP = super.getLP(null);
          rot = rotation;
     }

     public LightProperties getLP(ThreeDPoint passed) {
```

```
          super.transformAxis(passed);
          long currentTime = 0;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();

          LightProperties toReturn = theLP.clone();
          if(passed.z == super.c1.z || passed.z == super.c2.z)
               return toReturn;




          return null;
     }

}

** ImageRectangle.java
package threeDWorld.threeDObjectsImpl.objects;

import globalStuff.Timekeeper;

import java.awt.Color;
import java.awt.image.BufferedImage;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.ThreeDPointAvg;

public class ImageRectangle extends SolidRectangle {

     private BufferedImage img;
     private LightProperties lp;
     private boolean dropX, dropY, dropZ;

     public ImageRectangle(ThreeDPoint vp1, ThreeDPoint vp2,
ThreeDPoint vp3, ThreeDPoint vp4, BufferedImage toProject,
LightProperties theLP) {
          super(vp1, vp2, vp3, vp4, theLP);
          img = toProject;
          lp = super.getLP(null);
```

```
            /*
             * Ryan attempting math again... oh boy!
             *
             * first, we need to convert the plane into 2D
             *
             * Drop the least significant component, i.e., drop
the one with the least change amongst all four points.
             *
             *We're going to do it by finding the MAD for each
point
             *
             */

            ThreeDPointAvg avg = new ThreeDPointAvg();
            avg.addPoint(v1);
            avg.addPoint(v2);
            avg.addPoint(v3);
            avg.addPoint(v4);

            ThreeDPoint theAvg = avg.getAverage();

            ThreeDPoint difference = new ThreeDPoint();

            difference.x = Math.abs(v1.x - theAvg.x) +
Math.abs(v2.x - theAvg.x) + Math.abs(v3.x - theAvg.x) +
Math.abs(v4.x - theAvg.x);
            difference.y = Math.abs(v1.y - theAvg.y) +
Math.abs(v2.y - theAvg.y) + Math.abs(v3.y - theAvg.y) +
Math.abs(v4.y - theAvg.y);
            difference.z = Math.abs(v1.z - theAvg.z) +
Math.abs(v2.z - theAvg.z) + Math.abs(v3.z - theAvg.z) +
Math.abs(v4.z - theAvg.z);

            //System.out.println(difference.x + "," + difference.y
+ "," + difference.z);

            dropZ = false;
            dropX = false;
            dropY = false;

            if (difference.z < difference.x && difference.z <
difference.y) {
```

```
                dropZ = true;
          }

          if (difference.x < difference.z && difference.x <
difference.y) {
                dropX = true;
          }

          if (difference.y < difference.x && difference.y <
difference.z) {
                dropY = true;
          }

          // now see if any of them were equal...
          if (difference.y == difference.x && difference.y <
difference.z) {
                dropY = true;
          }

          if (difference.y == difference.z && difference.y <
difference.x) {
                dropY = true;
          }

          if (difference.x == difference.z && difference.x <
difference.y) {
                dropX = true;
          }

          //if (dropX) { System.out.println("Dropping x"); }
     }

     @Override
     public LightProperties getLP(ThreeDPoint hittingPoint) {
          long currentTime = 0;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();


          ThreeDPoint newV2 = new ThreeDPoint();
          ThreeDPoint newHit = new ThreeDPoint();
```

```
          if (dropX) {
               newV2.x = v2.z;
               newV2.y = v2.y;
               newHit.x = hittingPoint.z;
               newHit.y = hittingPoint.y;
          } else if (dropY) {
               newV2.x = v2.x;
               newV2.y = v2.z;
               newHit.x = hittingPoint.x;
               newHit.y = hittingPoint.z;
          } else if (dropZ){
               newV2.x = v2.x;
               newV2.y = v2.y;
               newHit.x = hittingPoint.x;
               newHit.y = hittingPoint.y;
          } else {
               // worse comes to worse, dropZ.
               newV2.x = v2.x;
               newV2.y = v2.y;
               newHit.x = hittingPoint.x;
               newHit.y = hittingPoint.y;
          }

          /*
           * Ok, now:
           *
           * Let:
           *
           * imageX = newHit.x - newV2.x
           * imageY = -( newHit.y - newV2.y )
           *
           * Divide imageX by the width of the rectangle
           * Divide imageY by the height of the rectangle
           *
           * Multiply imageX by the width of the image
           * Multiply imageY by the height of the image
           *
           * imageX and imageY are now the points in the image
that should be at this point
           */

          double imageX = newHit.x - newV2.x;
```

```
          double imageY = (newV2.y - newHit.y);

          imageX /= this.getWidth();
          imageY /= this.getHeight();

          imageX *= img.getWidth();
          imageY *= img.getHeight();

          imageX = Math.abs(imageX);
          imageY = Math.abs(imageY);

          //System.out.println(imageX + "," + imageY);

          LightProperties toReturn = lp.clone();
          toReturn.from = this;

          try {
               toReturn.objectColor = new Color(img.getRGB(((int)
imageX), ((int) imageY)));
          } catch (Exception e) {
               // we can ignore it. It is some strange thing
going on with how java maps pixels...
          }

          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForRectangleGetLP,
System.currentTimeMillis() - currentTime);
          return toReturn;

     }

}

** ImageSphere.java

package threeDWorld.threeDObjectsImpl.objects;

import globalStuff.Timekeeper;

import java.awt.Color;
import java.awt.image.BufferedImage;
```

```
import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;

/**
 * This is a subclass of the SolidSphere class that projects the
passed image onto the sphere.
 * @author ryan
 *
 */
public class ImageSphere extends SolidSphere {

     private BufferedImage img;
     private LightProperties theLP;
     private int rot;

     /**
      * The constructor
      * @param r the radius of the sphere
      * @param c the center of the sphere
      * @param toProject the image to project onto the sphere
      * @param rotation how rotated the sphere is from its
normal axis, in degrees (0 - 360)
      */
     public ImageSphere(double r, ThreeDPoint c, BufferedImage
toProject, LightProperties lp, int rotation) {
          super(r, c, lp);
          img = toProject;
          theLP = super.getLP(null);
          rot = rotation;

     }

     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          long currentTime = 0;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();

          LightProperties toReturn = theLP.clone();

          /*
           *
```

```
           * First, find:
           *
           * Vn, which is the vector from the center of the
sphere to the "north pole"
           * Ve, which is the vector from the center of the
sphere to any point on the equator
           * Vp, which is the vector from the center of the
sphere to the point we are coloring
           *
           * Normalize all vectors
           *
           * Then, calculate the longitude and latitude:
           *
           * The latitude is the angle between Vp and Vn:
arccos( - (Vn dot Vp)) / PI
           *
           * Finding the longitude is a little more
complicated...
           *
           * Define some theta to be: ( arccos( (Vp dot Ve) /
sin(latitude * PI) ) ) / 2PI
           * Then:
           *
           * if (Vn X Ve) dot Vp) > 0 then longitude = theta
           * Otherwise, longitude = 1 - theta
           *
           * Then, get pixel:
           * (latitude * width, longitude * height)
           *
           */

          // first, calculate our required vectors
          ThreeDPoint Vn = new ThreeDPoint(0.0, 1.0, 0.0);
          ThreeDPoint Vp =
this.getCenter().vectorSubtract(passed);

          // this the vector we are going to use for rotation.
          // picture a unit circle centered at the origin.
          // it is defined by: x^2 + y^2 = r^2 or x^2 + y^2 = 1

          // we're going to use our rotation factor as a
proportion between 0 and 360 for where we should point our
```

```
vector
          // 0 and 360 being the same place.

          // so, in order to do this, we'll calculate a sign and
proportion out of 180.
          int sign = (rot > 180 ? -1 : 1);
          double value = (rot > 180 ? 180 - (rot - 180) : rot);
          value /= 180.0;

          // value is now the X coord of the point we want to
get to.
          // to calculate the Y coord:
          // y = (+/-)sqrt(1 + x^2)

          double yVal = sign * Math.sqrt(1 - (value * value));

          // now our value represents an X and our yVal
represents a Y point on our circle. We want to calculate a
vector from the
          // center of the circle (0,0) to these points.
Luckily, we don't really need to subtract zero to know what it
is. Because we want to rotate
          // about the y-axis, we map the X value to X and the Y
value to Z
          ThreeDPoint Ve = new ThreeDPoint(value, 0.0, yVal);

          //System.out.println(rot + "|" + value + "|" + yVal);

          Vn.makeUnitVector();
          Ve.makeUnitVector();
          Vp.makeUnitVector();

          // latitude
          double longitude = Math.acos(0.0 -
Vn.getDotProduct(Vp)) / Math.PI;

          // longitude
          double latitude = ( Math.acos(Vp.getDotProduct(Ve) /
Math.sin(longitude * Math.PI))) / (2.0 * Math.PI);

          if (Vn.getCrossProduct(Ve).getDotProduct(Vp) <= 0) {
               latitude = 1.0 - latitude;
```

```
          }

          // get the pixels
          double x = latitude * img.getWidth(null);
          double y = longitude * img.getHeight(null);

          try {
               toReturn.objectColor = new Color(img.getRGB(((int)
x), ((int) y)));
          } catch (Exception e) {
               // we can ignore it. It is some strange thing
going on with how java maps pixels...
          }

          //if (toReturn.objectColor.getRed() == 0 &&
toReturn.objectColor.getBlue() == 0 &&
toReturn.objectColor.getGreen() == 0)
{ System.out.println("Black"); }

          toReturn.from = this;

          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForSphereGetLP,
System.currentTimeMillis() - currentTime);
          return toReturn;


     }

     public void setRotation(int rotation) {
          rot = rotation;
     }

     public int getRotataion() { return rot; }

}

** ObjectUtils.java

package threeDWorld.threeDObjectsImpl.objects;

import threeDWorld.raysAndVectors.ThreeDPoint;
```

```
public class ObjectUtils {

     public static double[][] getRotationMatrix(ThreeDPoint a){
          double angle = Math.acos(a.z/(Math.sqrt((a.x*a.x) +
(a.y*a.y) + (a.z*a.z))));
          //System.out.println(angle);
          double cos = Math.cos(angle);
          double sin = Math.sin(angle);
          ThreeDPoint vec = new ThreeDPoint(0,0,1);
          ThreeDPoint u;
          double[][] rMatrix = new double[3][3];

          if(angle < (Math.PI/2))
               u = vec.getCrossProduct(a);
          else
               u = vec.getCrossProduct(a);
          if(u.x == 0 && u.y ==0 && u.z == 0){
               rMatrix[0][0] = 1;
               rMatrix[1][1] = 1;
               rMatrix[2][2] = 1;
               rMatrix[0][1] = 0;
               rMatrix[1][2] = 0;
               rMatrix[2][0] = 0;
               rMatrix[0][2] = 0;
               rMatrix[1][0] = 0;
               rMatrix[2][1] = 0;

               //System.out.println("original u = " + u.x + ", "
+ u.y + ", " + u.z);
          }
          else {
               u.makeUnitVector();
               //System.out.println("unit u = " + u.x + ", " +
u.y + ", " + u.z);
               rMatrix[0][0] = (u.x* u.x) + cos*(1- (u.x* u.x));
               rMatrix[1][1] = (u.y* u.y) + cos*(1- (u.y* u.y));
               rMatrix[2][2] = (u.z* u.z) + cos*(1- (u.z* u.z));
               rMatrix[0][1] = (u.x * u.y)*(1-cos) + (u.z*sin);
               rMatrix[1][2] = (u.y * u.z)*(1-cos) + (u.x*sin);
               rMatrix[2][0] = (u.x * u.z)*(1-cos) + (u.y*sin);
               rMatrix[0][2] = (u.x * u.z)*(1-cos) - (u.y*sin);
```

```
                rMatrix[1][0] = (u.x * u.y)*(1-cos) - (u.z*sin);
                rMatrix[2][1] = (u.y * u.z)*(1-cos) - (u.x*sin);
            }
            return rMatrix;
        }
        /**only for 2x2 matrices
         *   0 1
         * 0 a b
         * 1 c d
         * @param matrix
         * @return
         */
        public static double get2Determinant(double[][] matrix){
            double a = matrix[0][0];
            double b = matrix[1][0];
            double c = matrix[0][1];
            double d = matrix[1][1];
            return (a*d - b*c);
        }
        /**only for 3x3 matrices
         *   0 1 2
         * 0 a b c
         * 1 d e f
         * 2 g h i
         * @param matrix
         * @return
         */
        public static double get3Determinant(double[][] matrix){
            double deter;
            double a = matrix[0][0];
            double b = matrix[1][0];
            double c = matrix[2][0];
            double d = matrix[0][1];
            double e = matrix[1][1];
            double f = matrix[2][1];
            double g = matrix[0][2];
            double h = matrix[1][2];
            double i = matrix[2][2];
            deter = (a*e*i) - (a*f*h) + (b*f*g) - (b*d*i) +
(c*d*h) - (c*e*g);
            return deter;
        }
```

```
     /* only for 3x3 matrices
      *   0 1 2
      * 0 a b c
      * 1 d e f
      * 2 g h i
      */
     public static double[][] getInverseMatrix(double[][]
matrix){
          double[][] inverse = new double[3][3];
          double coef = 1/ (get3Determinant(matrix));
          double a = matrix[0][0];
          double b = matrix[1][0];
          double c = matrix[2][0];
          double d = matrix[0][1];
          double e = matrix[1][1];
          double f = matrix[2][1];
          double g = matrix[0][2];
          double h = matrix[1][2];
          double i = matrix[2][2];
          inverse[0][0] = coef * (e*i - f*h);
          inverse[1][0] = coef * (c*h - b*i);
          inverse[2][0] = coef * (b*f - c*e);
          inverse[0][1] = coef * (f*g - d*i);
          inverse[1][1] = coef * (a*i - c*g);
          inverse[2][1] = coef * (c*d - a*f);
          inverse[0][2] = coef * (d*h - e*g);
          inverse[1][2] = coef * (b*g - a*h);
          inverse[2][2] = coef * (a*e - b*d);
          return inverse;
     }

     /* apply matrix to a point
      * if a point is multiplied by both the rotation matrix and
its inverse
      * the result should be the original point
      */
     public static void multiplyMatrices(double[][] matrix,
ThreeDPoint p){
          double a = (p.x * matrix[0][0]) + (p.y * matrix[0][1])
+ (p.z * matrix[0][2]);
          double b = (p.x * matrix[1][0]) + (p.y * matrix[1][1])
```

```
+ (p.z * matrix[1][2]);
          double c = (p.x * matrix[2][0]) + (p.y * matrix[2][1])
+ (p.z * matrix[2][2]);
          p.x = a;
          p.y = b;
          p.z = c;
     }
     //shift the points you want to line up with the axis
     public static double[] shiftToAxis(ThreeDPoint p){
          double[] shift = new double[2];
          shift[0] = p.x;
          shift[1] = p.y;
          p.x = p.x - shift[0];
          p.y = p.y - shift[1];
          return shift;
     }
     //shift the vector and anything else in the shape that
doesn't line up with the axis
     public static void shiftWithShape(ThreeDPoint p, double[]
shift){
          p.x = p.x-shift[0];
          p.y = p.y-shift[1];
     }

     public static void shiftBack(ThreeDPoint p, double[]
values){
          p.x += values[0];
          p.y += values[1];
     }

     //public static ThreeDPoint, double[] transform()
     /* tests if the inverse methods work
      *
      */
     public static void main (String[] args) {
          ThreeDPoint p1 = new ThreeDPoint(-1.0, 2.0, 3.0);
          ThreeDPoint p2 = new ThreeDPoint(-4.0, 2.0, 6.0);
          ThreeDPoint dir = new ThreeDPoint(3.0, 0.0, -3.0);
          ThreeDPoint point = new ThreeDPoint(-
1.0547902469430162,1.0084588425239303,2.9452097530569836);
          System.out.println("The distance is " +
point.getDistance(p1));
```

```
          ThreeDPoint newSlope = p1.vectorSubtract(point);
          point.makeUnitVector();
          dir.makeUnitVector();
          System.out.println("The new dir is " +
newSlope.getDotProduct(dir));
          double[][] matrix = getRotationMatrix(dir);
          //System.out.println("matrix is" + matrix[0][0]);
          double[][] inverse = getInverseMatrix(matrix);

          multiplyMatrices(matrix, p1);
          multiplyMatrices(matrix, p2);
          ThreeDPoint dir2 = new ThreeDPoint(p1.x-p2.x, p1.y-
p2.y, p1.z-p2.z);
          multiplyMatrices(matrix, dir);
//        System.out.println("the new direction is " + dir2.x +
"," + dir2.y + "," + dir2.z);
//        System.out.println("the new other direction is " +
dir.x + "," + dir.y + "," + dir.z);
//        System.out.println("the new p1 is " + p1.x + "," +
p1.y + "," + p1.z);
//        System.out.println("the new p2 is " + p2.x + "," +
p2.y + "," + p2.z);
          double[] values = shiftToAxis(p1);
          double[] values2 = shiftToAxis(p2);
          System.out.println("the new p1 is " + p1);
          System.out.println("the new p2 is " + p2);
//        System.out.println(p1.x + "," + p1.y + "," + p1.z);
//        System.out.println(p2.x + "," + p2.y + "," + p2.z);

          //shiftBack(p1, values);
          //shiftBack(p2, values2);
          //multiplyMatrices(inverse, p1);
          //multiplyMatrices(inverse, p2);
//        System.out.println(p1.x + "," + p1.y + "," + p1.z);
//        System.out.println(p2.x + "," + p2.y + "," + p2.z);

          multiplyMatrices(matrix, point);
          shiftWithShape(point, values);
          System.out.println("the new point is " + point);
          //{0.4802435092189756,-0.22010920691453506,-
2.82842712474619}
          //{0.0774851103125851,-0.9915411574760699,-
```

```
2.82842712474619}
     }

}

** SolidCone.java

package threeDWorld.threeDObjectsImpl.objects;
import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;


public class SolidCone implements Object3D {

     private ThreeDPoint top, cen;

     private double rad;

     private LightProperties lp;

     private double[][] rMatrix;

     private double[][] rInverse;

     private double[] shift;

     /**
      *
      * @param top1 = top of cone
      * @param cen1 = center of bottom circle
      * @param radius = radius of bottom circle
      * @param theProps
      */

     public SolidCone(ThreeDPoint top1, ThreeDPoint cen1, double
radius, LightProperties theProps){
          top = top1;
          cen = cen1;
          rad = radius;
          ThreeDPoint slope = new ThreeDPoint(top.x-cen.x,
```

```
top.y-cen.y, top.z-cen.z);
          System.out.println("the slope is " + slope.x + ", " +
slope.y + ", " + slope.z);
          rMatrix = ObjectUtils.getRotationMatrix(slope);
          rInverse = ObjectUtils.getInverseMatrix(rMatrix);
          //shift everything

          ObjectUtils.multiplyMatrices(rMatrix, top);
          top.x = 0;
          top.y = 0;
          ObjectUtils.multiplyMatrices(rMatrix, cen);
          shift = ObjectUtils.shiftToAxis(cen);
          lp = theProps;
     }
     /* slope is the slope between cp1 and cp2
      * we are transforming everything so that this slope is in
line with the z-axis
      * see ObjectUtils for methods
      */
     private void transformAxis(ThreeDPoint thePoint){
          //System.out.println("the rMatrix is " + rMatrix[0]
[0]);
          ObjectUtils.multiplyMatrices(rMatrix, thePoint);
          ObjectUtils.shiftWithShape(thePoint, shift);
     }

     private void transformBack(double[] shift, ThreeDPoint
thePoint){
          ObjectUtils.shiftBack(thePoint, shift);
          ObjectUtils.multiplyMatrices(rInverse, thePoint);
     }

     @Override
     public ThreeDPoint checkForCollision(Vector3D theVector) {
          Vector3D ourVector = theVector.clone();
          //shift the vector
          transformAxis(ourVector.startingPosition);
          ObjectUtils.multiplyMatrices(rMatrix,
ourVector.direction);

          double a = ourVector.startingPosition.x;
          double b = ourVector.startingPosition.y;
```

```
          double c = ourVector.startingPosition.z;
          double p = ourVector.direction.x;
          double q = ourVector.direction.y;
          double r = ourVector.direction.z;

          //this checks for a collision on the circle
          double t = (cen.z-c)/r;
          double finalt = t;
          if( t < 0 || Math.pow( (a + (p*t)), 2) + Math.pow( (b
+ (q*t) ), 2) >= (rad * rad)){
               finalt = -1.0;
          }

          //this checks for a collision on th curvy part
          double m = rad/(top.z-cen.z);
          double w = (p*p) + (q*q) - (m*m*r*r);
          double u = (2*a*p) + (2*b*q) + (2*m*m*top.z*r) -
(2*m*m*c*r);
          double v = (a*a) + (b*b) - (m*m*top.z*top.z) +
(2*m*m*c*top.z)- (m*m*c*c);
          double alpha = Math.pow(u, 2) - (4*w*v);
          double acounter = -1;
          if(alpha < 0)
               acounter = -1;
          else{
               double hit1 = ( (0-u) + Math.sqrt(alpha) )/(2*w);
               double hit2 = ( (0-u) - Math.sqrt(alpha) )/(2*w);
               if(hit1 < 0 && hit2 < 0){
                    acounter = -1;
               }
               else if((hit1 >= 0 && hit2 < 0) || (hit1 >= 0 &&
hit1 < hit2)){
                    acounter = hit1;
               }
               else if((hit1 < 0 && hit2 >= 0) || (hit2 >= 0 &&
hit2 < hit1) ){
                    acounter = hit2;
               }
          }
          double testingPoint = c + (r*acounter);;
          if( (testingPoint < Math.min(top.z, cen.z)) ||
(testingPoint > Math.max(top.z, cen.z)) ){
```

```
                acounter = -1;
           }

           double finalcounter = -1;
           double point1;
           double point2;
           double point3;

           if (finalt < 0 && acounter < 0){
                return null;
           }
           else if ((finalt >= 0 && acounter < 0) || (finalt >= 0
&& finalt < acounter)){
                finalcounter = finalt;
                point3 = cen.z;
                point1 = a + (p*finalcounter);
                point2 = b + (q*finalcounter);
                ThreeDPoint collisionPoint = new
ThreeDPoint(point1, point2, point3);
                transformBack(shift, collisionPoint);
                return collisionPoint;
           }
           else if ((acounter >= 0 && finalt < 0) || (acounter >=
0 && finalt > acounter)){
                finalcounter = acounter;

                point3 = c + (r*finalcounter);
                if( (point3 < Math.min(top.z, cen.z)) || (point3 >
Math.max(top.z, cen.z)) ){
                     return null;
                }
                point1 = a + (p*finalcounter);
                point2 = b + (q*finalcounter);
                ThreeDPoint collisionPoint = new
ThreeDPoint(point1, point2, point3);
                transformBack(shift, collisionPoint);
                return collisionPoint;
           }

           return null;
      }
```

```
     private ThreeDPoint getCenter(){
          return new ThreeDPoint(0,0,top.z + ((top.z-cen.z)/2));
     }

     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
ThreeDPoint incident) {
          ThreeDPoint ourHit = new ThreeDPoint(hit.x, hit.y,
hit.z);
          ThreeDPoint ourIncident = new ThreeDPoint(incident.x,
incident.y, incident.z);
          transformAxis(ourHit);
          ObjectUtils.multiplyMatrices(rMatrix, ourIncident);
          ThreeDPoint normal;
          ourIncident.makeUnitVector();

          if(Math.abs(ourHit.z - cen.z) < 0.000000000001){
               normal = new ThreeDPoint(0, 0, cen.z -
getCenter().z);
               normal.makeUnitVector();
               return (normal.getDotProduct(ourIncident));
          }

          //see pictures to understand...
          else{
               double tanTheta = (rad/(Math.abs(top.z-cen.z)));
               ThreeDPoint insidePoint = new
ThreeDPoint(0,0,ourHit.z);
               ThreeDPoint parallel =
ourHit.vectorSubtract(insidePoint);
               ThreeDPoint perpen = new ThreeDPoint(0,0,
(parallel.getDistance(ourHit) * tanTheta));
               normal = perpen.vectorAdd(parallel);
               normal.makeUnitVector();
               return (normal.getDotProduct(ourIncident));
          }
     }

     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          // TODO Auto-generated method stub
          return lp;
```

```
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          // TODO Auto-generated method stub
          return null;
     }

}

** SolidCube.java

package threeDWorld.threeDObjectsImpl.objects;

import java.awt.Color;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;

/**
 * This class forms a set of 6 planes in cube form. The center
of the object is the center of the cube, and the "radius" is the
distance from the center
 * of the cube to each vertex.
 * @author ryan
 *
 */
public class SolidCube implements Object3D {

     private ArrayList<SolidRectangle> theFaces;

     /**
      * This constructor creates a normal 6 sided cube, with
each vertex's distance from the center being the radius.
      * @param center the center of the cube
      * @param radius the distance from the center to each
```

```
vertex
      * @param c1 the color of face 1
      * @param c2 the color of face 2
      * @param c3 the color of face 3
      * @param c4 the color of face 4
      * @param c5 the color of face 5
      * @param c6 the color of face 6
      */
     public SolidCube(ThreeDPoint center, double radius, Color
c1, Color c2, Color c3, Color c4, Color c5, Color c6) {

          // first, let's map our all of our vertexs

          // upper right back
          ThreeDPoint firstVertex = new
ThreeDPoint(1.0,1.0,1.0);
          firstVertex.makeUnitVector();
          firstVertex = firstVertex.scalerMultiply(radius);
          firstVertex = firstVertex.vectorAdd(center);

          // upper right front
          ThreeDPoint secondVertex = new ThreeDPoint(1.0,1.0,-
1.0);
          secondVertex.makeUnitVector();
          secondVertex = firstVertex.scalerMultiply(radius);
          secondVertex = firstVertex.vectorAdd(center);

          // upper left back
          ThreeDPoint thirdVertex = new ThreeDPoint(-1.0, 1.0,
1.0);
          thirdVertex.makeUnitVector();
          thirdVertex = firstVertex.scalerMultiply(radius);
          thirdVertex = firstVertex.vectorAdd(center);

          // upper left front
          ThreeDPoint fourthVertex = new ThreeDPoint(-1.0, 1.0,
-1.0);
          fourthVertex.makeUnitVector();
          fourthVertex = firstVertex.scalerMultiply(radius);
          fourthVertex = firstVertex.vectorAdd(center);

          // lower right back
```

```
          ThreeDPoint fifthVertex = new ThreeDPoint(1.0, -1.0,
1.0);
          fifthVertex.makeUnitVector();
          fifthVertex = firstVertex.scalerMultiply(radius);
          fifthVertex = firstVertex.vectorAdd(center);

          // lower right front
          ThreeDPoint sixthVertex = new ThreeDPoint(1.0, -1.0,
-1.0);
          sixthVertex.makeUnitVector();
          sixthVertex = firstVertex.scalerMultiply(radius);
          sixthVertex = firstVertex.vectorAdd(center);

          // lower left back
          ThreeDPoint seventhVertex = new ThreeDPoint(-1.0,
-1.0, 1.0);
          seventhVertex.makeUnitVector();
          seventhVertex = firstVertex.scalerMultiply(radius);
          seventhVertex = firstVertex.vectorAdd(center);

          // lower left front
          ThreeDPoint eightVertex = new ThreeDPoint(-1.0, -1.0,
-1.0);
          eightVertex.makeUnitVector();
          eightVertex = firstVertex.scalerMultiply(radius);
          eightVertex = firstVertex.vectorAdd(center);

          // now create the following 6 rectangles:
          // 1 2 3 4 roof
          // 5 6 7 8 floor
          // 1 3 5 7 back wall
          // 2 4 6 8 front wall
          // 3 4 7 8 left wall
          // 1 2 5 6 right wall

          theFaces = new ArrayList<SolidRectangle>();

          // order: lower left, upper left, upper right, lower
right

          // face 1
          LightProperties theProps = new LightProperties();
```

```
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c1;
          theProps.reflection = false;

          SolidRectangle myRect = new
SolidRectangle(fourthVertex, thirdVertex, firstVertex,
secondVertex, theProps);
          theFaces.add(myRect);

          // face 2
          theProps = new LightProperties();
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c2;
          theProps.reflection = false;

          myRect = new SolidRectangle(eightVertex,
seventhVertex, fifthVertex, sixthVertex, theProps);
          theFaces.add(myRect);

          // face 3
          theProps = new LightProperties();
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c3;
          theProps.reflection = false;

          myRect = new SolidRectangle(seventhVertex,
thirdVertex, firstVertex, fifthVertex, theProps);
          theFaces.add(myRect);

          // face 4
          theProps = new LightProperties();
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c4;
          theProps.reflection = false;
```

```
          myRect = new SolidRectangle(eightVertex, fourthVertex,
secondVertex, sixthVertex, theProps);
          theFaces.add(myRect);

          // face 5
          theProps = new LightProperties();
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c5;
          theProps.reflection = false;

          myRect = new SolidRectangle(seventhVertex,
thirdVertex, fourthVertex, eightVertex, theProps);
          theFaces.add(myRect);

          // face 6
          theProps = new LightProperties();
          theProps.from = this;
          theProps.isLight = false;
          theProps.noShadow = false;
          theProps.objectColor = c6;
          theProps.reflection = false;

          myRect = new SolidRectangle(sixthVertex, secondVertex,
firstVertex, fifthVertex, theProps);
          theFaces.add(myRect);
     }


     @Override
     public ThreeDPoint checkForCollision(Vector3D theVector) {
          HashMap<ThreeDPoint, Object3D> theMap = new
HashMap<ThreeDPoint, Object3D>();

          for (Object3D otd : theFaces) {
               ThreeDPoint hit =
otd.checkForCollision(theVector);
               if (hit != null) {
                    theMap.put(hit, otd);
               }
```

```
          }

          if (theMap.size() == 0) return null;

          if (theMap.size() == 1) {
               return theMap.keySet().iterator().next();
          }

          Iterator<ThreeDPoint> ourPoints =
theMap.keySet().iterator();

          ThreeDPoint current = ourPoints.next();
          double smallestValue =
current.getComparitveDistance(theVector.startingPosition);

          while (ourPoints.hasNext()) {
               ThreeDPoint toTest = ourPoints.next();
               double testing =
toTest.getComparitveDistance(theVector.startingPosition);

               if (testing < smallestValue) {
                    smallestValue = testing;
                    current = toTest;
               }
          }

          return current;
     }

     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
ThreeDPoint incident) {
          // first, figure out which surface was hit...
          for (Object3D otd : theFaces) {
               SolidRectangle myRect = (SolidRectangle) otd;
               if (myRect.isPointOnRectangle(hit)) {
                    return myRect.getCosineOfAngleFromNormal(hit,
incident);
               }
          }

          return 0;
```

```
     }

     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          // first, figure out which surface was hit...
          for (Object3D otd : theFaces) {
               SolidRectangle myRect = (SolidRectangle) otd;
               if (myRect.isPointOnRectangle(passed)) {
                    return myRect.getLP(passed);           }
          }

          return null;
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          // first, figure out which surface was hit...
          for (Object3D otd : theFaces) {
               SolidRectangle myRect = (SolidRectangle) otd;
               if (myRect.isPointOnRectangle(hittingPoint)) {
                    return myRect.getNewVector(theVector,
hittingPoint);
               }
          }

          return null;
     }

}

** SolidCylinder.java

package threeDWorld.threeDObjectsImpl.objects;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;

public class SolidCylinder implements Object3D {
```

```
    protected ThreeDPoint c1, c2;

    private double rad;

    private LightProperties lp;

    private double[][] rMatrix;

    private double[][] rInverse;

    private double[] shift;

    /**
     *
     * @param cp1 - the center of a circle
     * @param cp2 - the center of the other circle
     * @param radius - the radius
     * @param lps - light properties
     */

    public SolidCylinder(ThreeDPoint cp1, ThreeDPoint cp2,
double radius, LightProperties lps){
        c1 = cp1;
        c2 = cp2;
        rad = radius;
        ThreeDPoint slope = new ThreeDPoint(c1.x-c2.x, c1.y-
c2.y, c1.z-c2.z);
        rMatrix = ObjectUtils.getRotationMatrix(slope);
        rInverse = ObjectUtils.getInverseMatrix(rMatrix);
        //shift everything

        ObjectUtils.multiplyMatrices(rMatrix, c1);
        c1.x = 0;
        c1.y = 0;
        ObjectUtils.multiplyMatrices(rMatrix, c2);
        shift = ObjectUtils.shiftToAxis(c2);

        lp = lps;
    }
    /* slope is the slope between cp1 and cp2
     * we are transforming everything so that this slope is in
line with the z-axis
```

```
      * see ObjectUtils for methods
      */
     public void transformAxis(ThreeDPoint thePoint){
          ObjectUtils.multiplyMatrices(rMatrix, thePoint);
          ObjectUtils.shiftWithShape(thePoint, shift);
     }

     public void transformBack(double[] shift, ThreeDPoint
thePoint){
          ObjectUtils.shiftBack(thePoint, shift);
          ObjectUtils.multiplyMatrices(rInverse, thePoint);
     }
     @Override
     public ThreeDPoint checkForCollision(Vector3D theVector) {
          Vector3D ourVector = theVector.clone();
          //shift the vector
          transformAxis(ourVector.startingPosition);
          ObjectUtils.multiplyMatrices(rMatrix,
ourVector.direction);

          double a = ourVector.startingPosition.x;
          double b = ourVector.startingPosition.y;
          double c = ourVector.startingPosition.z;
          double p = ourVector.direction.x;
          double q = ourVector.direction.y;
          double r = ourVector.direction.z;

          //this is to check for collisions on the two circles
          double finalt = -1;
          double t = (c1.z - c)/r;
          double tcounter = t;
          if( t < 0 || Math.pow( (a + (p*t)), 2) + Math.pow( (b
+ (q*t) ), 2) >= (rad * rad)){
               tcounter = -1.0;
          }
          double t2 = (c2.z - c)/r;
          double t2counter = t2;
          if( t2 < 0 || Math.pow( (a + (p*t2)), 2) +
Math.pow( (b + (q*t2) ), 2) >= (rad * rad)){
               t2counter = -1.0;
          }
          if(tcounter < 0 && t2counter < 0)
```

```
                finalt = -1;
           else if((tcounter < 0 && t2counter >= 0) || (t2counter
>= 0 && t2counter < tcounter))
                finalt = t2counter;
           else if((tcounter >= 0 && t2counter < 0) || (tcounter
>=0 && tcounter < t2counter))
                finalt = tcounter;

           //this is to check for collisions on the curvy part
           double m = (8*a*b*p*q) - (4*b*b*p*p) + (4*rad*rad*p*p)
- (4*a*a*q*q) + (4*rad*rad*q*q);
           double mcounter = -1.0;
           if(m < 0){
                mcounter = -1;
           }

           else if(m >= 0){
                double hit1 = ( (0 -2*a*p) - (2*b*q) +
Math.sqrt(m) )/ (2*((p*p) + (q*q)));
                double hit2 = ( (0 -2*a*p) - (2*b*q) -
Math.sqrt(m) )/ (2*((p*p) + (q*q)));
                if(hit1 < 0 && hit2 < 0){
                     mcounter = -1;
                }
                else if((hit1 >= 0 && hit2 < 0) || (hit1 >= 0 &&
hit1 < hit2)){
                     mcounter = hit1;
                }
                else if((hit1 < 0 && hit2 >= 0) || (hit2 >= 0 &&
hit2 < hit1) ){
                     mcounter = hit2;
                }
           }
           double testingPoint = c + (r*mcounter);;
           if( (testingPoint < Math.min(c1.z, c2.z)) ||
(testingPoint > Math.max(c1.z, c2.z)) ){
                mcounter = -1;
           }
           double finalcounter = -1;
           double point1;
           double point2;
           double point3;
```

```
          if (finalt < 0 && mcounter < 0){
               return null;
          }
          else if ((finalt >= 0 && mcounter < 0) || (finalt >= 0
&& finalt < mcounter)){
               finalcounter = finalt;
               if(finalcounter == tcounter){
                    point3 = c1.z;
                    point1 = a + (p*finalcounter);
                    point2 = b + (q*finalcounter);
                    ThreeDPoint collisionPoint = new
ThreeDPoint(point1, point2, point3);
                    transformBack(shift, collisionPoint);
                    return collisionPoint;
               }
               if(finalcounter == t2counter){
                    point3 = c2.z;
                    point1 = a + (p*finalcounter);
                    point2 = b + (q*finalcounter);
                    ThreeDPoint collisionPoint = new
ThreeDPoint(point1, point2, point3);
                    transformBack(shift, collisionPoint);
                    return collisionPoint;
               }
          }
          else if ((mcounter >= 0 && finalt < 0) || (mcounter >=
0 && finalt > mcounter)){
               finalcounter = mcounter;

               point3 = c + (r*finalcounter);
               if( (point3 < Math.min(c1.z, c2.z)) || (point3 >
Math.max(c1.z, c2.z)) ){
                    return null;
               }
               point1 = a + (p*finalcounter);
               point2 = b + (q*finalcounter);
               ThreeDPoint collisionPoint = new
ThreeDPoint(point1, point2, point3);
               transformBack(shift, collisionPoint);
               return collisionPoint;
          }
```

```
          return null;
     }

     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          return lp;
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          //shift the point and vector to the right orientation
          Vector3D ourVector = theVector.clone();
          ThreeDPoint ourHittingPoint = new
ThreeDPoint(hittingPoint.x, hittingPoint.y, hittingPoint.z);

          transformAxis(ourVector.startingPosition);
          ObjectUtils.multiplyMatrices(rMatrix,
ourVector.direction);
          transformAxis(ourHittingPoint);

          ThreeDPoint normal;
          if(ourHittingPoint.z == c1.z){
               normal = new ThreeDPoint(c2.x-c1.x, c2.y-c1.y,
c2.z-c1.z);
          }
          if(ourHittingPoint.z == c2.z){
               normal = new ThreeDPoint(c1.x-c2.x, c1.y-c2.y,
c1.z-c2.z);
          }
          else{
               normal = new ThreeDPoint(0-ourHittingPoint.x, 0-
ourHittingPoint.y, 0);
          }
          normal.makeUnitVector();

          ThreeDPoint incident = ourVector.direction;
          incident.makeUnitVector();


          // reflect or refract?
          if (lp.reflection) {
```

```
                /*
                 * Now apply this equation:
                 *
                 * Let the slope of the incident vector = n1
                 * Let the slope of the resulting vector = n2
                 * Let the slope of the normal = s
                 *
                 * n2 = n1 - (2(n1 dot s) * s)
                 */


                double d = 2 * (incident.getDotProduct(normal));
                normal = normal.scalerMultiply(d);
                //normal.makeUnitVector();
                Vector3D toReturn = new Vector3D();
                ThreeDPoint newDirection =
incident.vectorSubtract(normal);
                //shift the direction to the right orientation
                ObjectUtils.multiplyMatrices(rInverse,
newDirection);
                toReturn.direction = newDirection;
                //shift the point back to the right place
                transformBack(shift, ourHittingPoint);
                toReturn.startingPosition = ourHittingPoint;
                toReturn.theProps = this.getLP(hittingPoint);

                return toReturn;
           }

           if (lp.transparency > 0) {
                // if we are still here, refract.
                /*
                 * Let:
                 *
                 * if the vector is pointing into the sphere:
                 *   n = 1 / index of refraction
                 * else:
                 *   n = index of refraction / 1
                 *
                 * d1 = - (normal dot theVector)
                 * d2 = sqrt(1 - n^2 * (1 - c1^2))
                 *
```

```
                * The refracted ray:
                *
                * refrac = (n * theVector) + (n * d1 - d2) *
normal
                */

               ThreeDPoint centerOfVector = new
ThreeDPoint(theVector.startingPosition.x + ((hittingPoint.x-
theVector.startingPosition.x)/2), theVector.startingPosition.y +
((hittingPoint.y - theVector.startingPosition.y)/2),
theVector.startingPosition.z + ((hittingPoint.z -
theVector.startingPosition.z)/2));
               double distanceFromHitToCenter =
hittingPoint.getDistance(getCenter());
               double distanceFromCenterToCenter =
centerOfVector.getDistance(getCenter());

               //if (distanceFromStartToCenter >
newPointDistance) {
               //   System.out.println("in");
               //} else {
               //   System.out.println("out");
               //}

               double n = (distanceFromHitToCenter <
distanceFromCenterToCenter ? 1.0 / lp.indexOfRefraction :
lp.indexOfRefraction);
               double d1 = 0.0 - normal.getDotProduct(incident);
               double d2 = Math.sqrt(1.0 - (Math.pow(n, 2) * (1.0
- Math.pow(d1, 2))));

               //System.out.println(n + "," + c1 + ","  + c2);

               Vector3D toReturn = new Vector3D();
               toReturn.startingPosition = hittingPoint;
               toReturn.theProps = this.getLP(hittingPoint);

               toReturn.direction = incident.scalerMultiply(n);

     toReturn.direction.vectorAdd(normal.scalerMultiply((n * d1)
- d2));
               ObjectUtils.multiplyMatrices(rInverse,
```

```
toReturn.direction);
                //System.out.println(toReturn.direction.x + "," +
toReturn.direction.y + "," + toReturn.direction.z);

                // move the hitting point down the path a little
bit to avoid dumb roundoff errors..
                toReturn.startingPosition =
toReturn.startingPosition.vectorAdd(toReturn.direction.scalerMul
tiply(rad / 100.0));
                transformBack(shift, toReturn.startingPosition);
                //transformBack(shift, hittingPoint);
                //transformBack(shift,
theVector.startingPosition);
                //ObjectUtils.multiplyMatrices(rInverse,
theVector.direction);
                return toReturn;

           }
           return null;
     }

     private ThreeDPoint getCenter(){
           return new ThreeDPoint(0,0,c1.z + ((c2.z-c1.z)/2));
     }

     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
ThreeDPoint incident) {
           ThreeDPoint ourHit = new ThreeDPoint(hit.x, hit.y,
hit.z);
           ThreeDPoint ourIncident = new ThreeDPoint(incident.x,
incident.y, incident.z);
           transformAxis(ourHit);
           ObjectUtils.multiplyMatrices(rMatrix, ourIncident);
           ThreeDPoint normal;
           ourIncident.makeUnitVector();

           if(Math.abs(ourHit.z - c1.z) < 0.000000000001){
                normal = new ThreeDPoint(0, 0, c1.z -
getCenter().z);
                normal.makeUnitVector();
                return (normal.getDotProduct(ourIncident));
```

```
          }
          else if(Math.abs(ourHit.z - c2.z) < 0.000000000001){
               normal = new ThreeDPoint(0, 0, c2.z -
getCenter().z);
               normal.makeUnitVector();
               return (normal.getDotProduct(ourIncident));
          }
          else
               normal = new ThreeDPoint(ourHit.x, ourHit.y, 0);
          normal.makeUnitVector();
          return (normal.getDotProduct(ourIncident));
     }




     public static void main(String[] args){

     }
}

** SolidRectangle.java

package threeDWorld.threeDObjectsImpl.objects;

import globalStuff.Timekeeper;
import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;

public class SolidRectangle implements Object3D {

     protected ThreeDPoint v1,v2,v3,v4;

     private ThreeDPoint normal, vec1, vec2;

     private double a,b,c,d;

     private LightProperties lp;

     /**
```

```
      * Defines a new rectangle. vp1 to vp4 must be given in a
clockwise fasion, as in:
      *  2-----3
      *  |     |
      *  |     |
      *  |     |
      *  |     |
      *  1-----4
      *
      * @param vp1 the lower left point
      * @param vp2 the upper left point
      * @param vp3 the upper right point
      * @param vp4 the lower right point
      * @param LP the light properties of the rectangle
      */
     public SolidRectangle(ThreeDPoint vp1, ThreeDPoint vp2,
ThreeDPoint vp3, ThreeDPoint vp4, LightProperties LP) {
          // v1 - v4 must be defined in a clockwise fashion
          v1 = vp1;
          v2 = vp2;
          v3 = vp3;
          v4 = vp4;

          fillInData();

          lp = LP;
          if (lp != null && lp.from != null) {
               lp.from = this;
          }
     }


     private void fillInData() {
          /*
           * Given:
           * V1 - V4's locations are x1 - x4, y1 - y4, z1 - z4
           * x0, y0, and z0 are the starting points of the ray
           * Rd = (xd, yd, zd) is the slope of the vector
           * Vec1 = V2 - V1
           * Vec2 = V4 - V1
           *
           * Let:
```

```
           * A = y1(z2-z3) + y2(z3-z1) + y3(z1-z2)
           * B = z1(x2-x3) + z2(x3-x1) + z3(x1-x2)
           * C = x1(y2-y3) + x2(y3-y1) + x3(y1-y2)
           * D = -x1(y2*z3 - y3*z2) - x2(y3*z1 - y1*z3) -
x3(y1*z2 - y2*z1)
           *
           */

          vec1 = v2.vectorSubtract(v1);
          vec2 = v4.vectorSubtract(v1);

          normal = vec1.getCrossProduct(vec2);
          normal.makeUnitVector();

          a = (v1.y * (v2.z - v3.z)) + (v2.y * (v3.z-v1.z)) +
(v3.y * (v1.z - v2.z));
          b = (v1.z * (v2.x - v3.x)) + (v2.z * (v3.x-v1.x)) +
(v3.z * (v1.x - v2.x));
          c = (v1.x * (v2.y - v3.y)) + (v2.x * (v3.y-v1.y)) +
(v3.x * (v1.y - v2.y));

          d = (-1.0 * v1.x * ( (v2.y * v3.z) - (v3.y * v2.z))) -
(v2.x * ( (v3.y * v1.z) - (v1.y * v3.z))) - (v3.x * ((v1.y *
v2.z) - (v2.y * v1.z)));

     }

     @Override
     public ThreeDPoint checkForCollision(Vector3D theVector) {

          long currentTime = 0;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();

          /*
           * Given:
           * V1 - V4's locations are x1 - x4, y1 - y4, z1 - z4
           * x0, y0, and z0 are the starting points of the ray
           * Rd = (xd, yd, zd) is the slope of the vector
           * Vec1 = V2 - V1
           * Vec2 = V4 - V1
           *
```

```
            * Let:
            *
            * W = (A*xd + B*yd + C*zd)
            *
            * If w = 0, no hit
            *
            * T = -(A*x0 + B*y0 + C*z0 + D) / W
            *
            * Make sure T is not negative. If it is, we're facing
the wrong way.
            *
            * The hitting point MAY be: (x0 + xd*t, y0 + yd*t, z0
+ zd*t)
            */

          theVector.direction.makeUnitVector();

          double W = ((a * theVector.direction.x) + (b *
theVector.direction.y) + (c * theVector.direction.z));

          if (W == 0) return null;

          double T = -1.0 * ((a * theVector.startingPosition.x)
+ (b * theVector.startingPosition.y) + (c *
theVector.startingPosition.z) + d);
          T /= W;

          if (T < 0) {
               // t is negative.
               if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForRectangleCheckCollision
, System.currentTimeMillis() - currentTime);
               return null;
          }

          ThreeDPoint testHit =
theVector.startingPosition.vectorAdd(theVector.direction.scalerM
ultiply(T));

          if (String.valueOf(testHit.x).equals("NaN")) {
               System.out.println("NaN!");
          }
```

```
          /*
           * Make sure we actually hit the point. Google answers
at http://answers.google.com/answers/threadview?id=18979 says:
           *
           * Let Vec3 = V4 - V3 be the vector opposite from
Vec1.
           * Let Vec4 be the vector from V1 to the point of
intersection
           * Let Vec5 be the vector from V3 to the point of
intersection.
           *
           * Normalize vectors Vec1, Vec3, Vec4 and Vec5.
           *
           * Find the dot products Vec1 dot Vec4 and Vec3 dot
Vec5. If both are
           * non-negative, then the point is in the rectangle.
          ThreeDPoint vec3, vec4, vec5;

          vec3 = v4.vectorSubtract(v3);
          vec4 = testHit.vectorSubtract(v1);
          vec5 = testHit.vectorSubtract(v3);

          vec1.makeUnitVector();
          vec3.makeUnitVector();
          vec4.makeUnitVector();
          vec5.makeUnitVector();

          double dot1, dot2;

          dot1 = vec1.getDotProduct(vec4);
          dot2 = vec3.getDotProduct(vec5);

          if (dot1 < 0 || dot2 < 0) {
               return null;
          }
           *
           */

          /*
           * But, Ryan likes this way better. See if the point
works in the plane equation.
```

```
            *
            * ax + by + cz + d = 0
            *
            * a,b,c = the normal x,y,z
            * x,y,z = our points
            *
            * d = -(aj + bk + cl)
            *
            * Now isn't that spiffy?
            */

          // because nothing ever really equals zero because of
roundoff...
          if (testHit.getDotProduct(normal) + (0.0 -
normal.getDotProduct(v1)) > 0.0000000001) {
               if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForRectangleCheckCollision
, System.currentTimeMillis() - currentTime);
               return null;
          }

          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForRectangleCheckCollision
, System.currentTimeMillis() - currentTime);
          return testHit;

     }

     public boolean isPointOnRectangle(ThreeDPoint testHit) {
          // using the plane equation...
          // because nothing ever really equals zero because of
roundoff...
          return !(testHit.getDotProduct(normal) + (0.0 -
normal.getDotProduct(v1)) > 0.0000000001);
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          // first, we need to calculate the normal to the point
we are hitting.
```

```
          ThreeDPoint incident = theVector.direction;
          incident.makeUnitVector();


          // reflect or refract?
          if (lp.reflection) {
               /*
                * Now apply this equation:
                *
                * Let the slope of the incident vector = n1
                * Let the slope of the resulting vector = n2
                * Let the slope of the normal = s
                *
                * n2 = n1 - (2(n1 dot s) * s)
                */


               double d = 2 * (incident.getDotProduct(normal));
               ThreeDPoint modNormal = normal.scalerMultiply(d);
               Vector3D toReturn = new Vector3D();
               toReturn.direction =
incident.vectorSubtract(modNormal);
               toReturn.startingPosition = hittingPoint;
               toReturn.theProps = this.getLP(hittingPoint);

               // move the hitting point down the path a little
bit to avoid dumb roundoff errors..
               toReturn.startingPosition =
toReturn.startingPosition.vectorAdd(toReturn.direction.scalerMul
tiply(0.0000001));

               return toReturn;
          }

          if (lp.transparency > 0) {
               // if we are still here, refract.
               /*
                * Let:
                *
                * if the vector is pointing into the sphere:
                *   n = 1 / index of refraction
                * else:
```

```
                *   n = index of refraction / 1
                *
                * c = - (normal dot theVector)
                * d = sqrt(1 - n^2 * (1 - c^2))
                *
                * The refracted ray:
                *
                * refrac = (n * theVector) + (n * c - d) * normal
                */


               double n = 1.0 / lp.indexOfRefraction;
               double c =
normal.getDotProduct(theVector.direction);
               double d = Math.sqrt(1.0 - Math.pow(n, 2) * (1.0 -
Math.pow(c, 2)));

               Vector3D toReturn = new Vector3D();
               toReturn.startingPosition = hittingPoint;
               toReturn.theProps = this.getLP(hittingPoint);

               toReturn.direction =
theVector.direction.scalerMultiply(n);

     toReturn.direction.vectorAdd(normal.scalerMultiply(n * c -
d));

               return toReturn;

          }

          return null;



     }



     public double getArea() {
          double w = v1.getDistance(v4);
          double h = v1.getDistance(v2);
```

```
          return w * h;
     }

     public double getWidth() {
          return v1.getDistance(v4);
     }

     public double getHeight() {
          return v1.getDistance(v2);
     }

     /**
      * Just a little main method that tests a few points.
      * @param args none
      */
     public static void main(String args[]) {
          SolidRectangle myS = new SolidRectangle(new
ThreeDPoint(0,2,0), new ThreeDPoint(2,2,0), new
ThreeDPoint(2,0,0), new ThreeDPoint(0,0,0), null);

          Vector3D myV = new Vector3D();
          myV.startingPosition = new ThreeDPoint(1, 1, 1);
          myV.direction = new ThreeDPoint(0,0,-1);

          System.out.println(myS.checkForCollision(myV));

     }



     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          return lp;
     }


     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
ThreeDPoint incident) {
          return Math.abs(normal.getDotProduct(incident));
     }
```

```
}

** SolidSphere.java

package threeDWorld.threeDObjectsImpl.objects;

import globalStuff.Timekeeper;
import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;

public class SolidSphere implements Object3D {

     private double radius;
     private ThreeDPoint center;
     private LightProperties lp;
     private long currentTime = 0;

     public SolidSphere(double r, ThreeDPoint c, LightProperties
theProps) {
          radius = r;
          center = c;
          lp = theProps;
          if (lp != null) {
               lp.from = this;
          }


     }

     public ThreeDPoint checkForCollision(Vector3D theVector) {
          if (Timekeeper.timingEnabled) {
               currentTime = System.currentTimeMillis();
          }

          if (Timekeeper.useCUDA) {
               // do the CUDA stuff
```

```
                return this.cudaCheckForCollision(theVector);
           } else {
                // do it in plain ol' Java
                return this.softwareCheckForCollision(theVector);
           }

     }

     private ThreeDPoint cudaCheckForCollision(Vector3D
theVector) {
           return null;
     }

     private ThreeDPoint softwareCheckForCollision(Vector3D
theVector) {
           // see if they are inside of us...
           if (theVector.startingPosition.getDistance(center) <=
radius) {
                if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForSphereCheckCollision,
System.currentTimeMillis() - currentTime);
                return null;
           }

           /**
            * <Kathy Math-Magic>
            *
            * the center of the sphere = (p,q,r)
            * the initial point = (a,b,c)
            * the slope = (x,y,z)
            *
            * m = x^2 + y^2 + z^2
            * n = 2ax - 2px + 2by - 2qy + 2cz - 2rz
            * o = (p-a)^2 + (q-b)^2 + (r-c)^2
            *
            * define a variable t = (px - ax + qy - by + rz - cz)
/ m
            *
            * If t is negative, stop. No hit.
            *
            * plug t into this formula: t^2(x^2 + y^2 + z^2) + tn
+ o
```

```
           * if that is less than (radius^2), then it hits the
sphere
           *
           *
           * </Kathy Math-Magic>
           *
           * Go-Go-Gadget Comp Sci!
           */

          // squares
          double xs = Math.pow(theVector.direction.x, 2);
          double ys = Math.pow(theVector.direction.y, 2);
          double zs = Math.pow(theVector.direction.z, 2);
          double rs = Math.pow(radius, 2);

          double m = xs + ys + zs;
          double n = (2 * theVector.startingPosition.x *
theVector.direction.x) - (2 * center.x * theVector.direction.x)
+ (2 * theVector.startingPosition.y * theVector.direction.y) -
(2 * center.y * theVector.direction.y) + (2 *
theVector.startingPosition.z * theVector.direction.z) - (2 *
center.z * theVector.direction.z);
          double o = Math.pow(center.x -
theVector.startingPosition.x, 2) + Math.pow(center.y -
theVector.startingPosition.y, 2) + Math.pow(center.z -
theVector.startingPosition.z, 2);

          double t = ((center.x * theVector.direction.x) -
(theVector.startingPosition.x * theVector.direction.x) +
(center.y * theVector.direction.y) -
(theVector.startingPosition.y * theVector.direction.y) +
(center.z * theVector.direction.z) -
(theVector.startingPosition.z * theVector.direction.z)) / m;


          if (t < 0.0) {
               // no hit.
               return null;
          }

          // t^2(x^2 + y^2 + z^2)
          double toCheck = Math.pow(t, 2) * m;
```

```
          // + t(2ax - 2px + 2by - 2qy + 2cz - 2rz)
          toCheck += t * n;

          // + (p-a)^2 + (q-b)^2 + (r-c)^2
          toCheck += o;


          // if that is less than (radius^2), then it hits the
sphere
          if (toCheck < rs) {
               // hit
               /**
                * to get the point:
                *
                * new o = o - s^2 ( s = radius )
                *
                * (a + xt, b + yt, c + zt)
                *
                * Where T = the quadratic formula on m n o
                * The largest will always be the -.
                *
                * (-b - sqrt(bb - 4ac)) / 2a
                *
                * Sub our values:
                *
                * (-n - sqrt(nn - 4mo)) / 2m
                *
                */

               o = o - rs;

               double zeroT = ((0 - n) - Math.sqrt(Math.pow(n, 2)
- (4 * m * o))) / (2 * m);

               ThreeDPoint toReturn = new ThreeDPoint();
               toReturn.x = theVector.startingPosition.x +
(theVector.direction.x * zeroT);
               toReturn.y = theVector.startingPosition.y +
(theVector.direction.y * zeroT);
               toReturn.z = theVector.startingPosition.z +
(theVector.direction.z * zeroT);
```

```
                if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForSphereCheckCollision,
System.currentTimeMillis() - currentTime);
                return toReturn;
          }

          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForSphereCheckCollision,
System.currentTimeMillis() - currentTime);
          // no hit
          return null;
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          // first, we need to calculate the normal to the point
we are hitting.

          ThreeDPoint normal = new ThreeDPoint(center.x -
hittingPoint.x, center.y - hittingPoint.y, center.z -
hittingPoint.z);
          normal.makeUnitVector();

          ThreeDPoint incident = theVector.direction;
          incident.makeUnitVector();


          // reflect or refract?
          if (lp.reflection) {
               /*
                * Now apply this equation:
                *
                * Let the slope of the incident vector = n1
                * Let the slope of the resulting vector = n2
                * Let the slope of the normal = s
                *
                * n2 = n1 - (2(n1 dot s) * s)
                */
```

```
                double d = 2 * (incident.getDotProduct(normal));
                normal = normal.scalerMultiply(d);
                Vector3D toReturn = new Vector3D();
                toReturn.direction =
incident.vectorSubtract(normal);
                toReturn.startingPosition = hittingPoint;
                toReturn.theProps = this.getLP(hittingPoint);

                return toReturn;
           }

           if (lp.transparency > 0) {
                // if we are still here, refract.
                /*
                 * Let:
                 *
                 * if the vector is pointing into the sphere:
                 *   n = 1 / index of refraction
                 * else:
                 *   n = index of refraction / 1
                 *
                 * c1 = - (normal dot theVector)
                 * c2 = sqrt(1 - n^2 * (1 - c1^2))
                 *
                 * The refracted ray:
                 *
                 * refrac = (n * theVector) + (n * c1 - c2) *
normal
                 */

                double distanceFromStartToCenter =
hittingPoint.getDistance(getCenter());
                double newPointDistance =
hittingPoint.vectorAdd(incident.scalerMultiply(radius /
2.0)).getDistance(getCenter());

                //if (distanceFromStartToCenter >
newPointDistance) {
                //   System.out.println("in");
                //} else {
                //   System.out.println("out");
                //}
```

```
                double n = (newPointDistance <
distanceFromStartToCenter ? 1.0 / lp.indexOfRefraction :
lp.indexOfRefraction);
                double c1 = 0.0 - normal.getDotProduct(incident);
                double c2 = Math.sqrt(1.0 - (Math.pow(n, 2) * (1.0
- Math.pow(c1, 2))));

                //System.out.println(n + "," + c1 + ","  + c2);

                Vector3D toReturn = new Vector3D();
                toReturn.startingPosition = hittingPoint;
                toReturn.theProps = this.getLP(hittingPoint);

                toReturn.direction = incident.scalerMultiply(n);

     toReturn.direction.vectorAdd(normal.scalerMultiply((n * c1)
- c2));
                //System.out.println(toReturn.direction.x + "," +
toReturn.direction.y + "," + toReturn.direction.z);

                // move the hitting point down the path a little
bit to avoid dumb roundoff errors..
                toReturn.startingPosition =
toReturn.startingPosition.vectorAdd(toReturn.direction.scalerMul
tiply(radius / 100.0));
                return toReturn;

           }

           return null;
     }

     /**
      * Gets the center of the sphere
      * @return the center
      */
     public ThreeDPoint getCenter() {
           return center;
     }

     /**
```

```
      * This method tests the CUDA implementation of the
checkForCollisions method against the software method.
      * @param args
      */
     public static void main(String[] args) {


System.out.println(System.getProperty("java.library.path"));

          Timekeeper.init();
          SolidSphere ss = new SolidSphere(3.0, new
ThreeDPoint(0,0,0), null);

          Vector3D myVTD = new Vector3D();
          myVTD.startingPosition = new ThreeDPoint(0,0,5);
          myVTD.direction = new ThreeDPoint(0,0,-7);

          ThreeDPoint hit = ss.softwareCheckForCollision(myVTD);
          ThreeDPoint hit2 = ss.cudaCheckForCollision(myVTD);

          System.out.println("Software: " + hit);
          System.out.println("CUDA: " + hit2);
     }


     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          return lp;
     }


     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint
hittingPoint, ThreeDPoint incident) {
          ThreeDPoint myPoint =
hittingPoint.vectorSubtract(getCenter());
          myPoint.makeUnitVector();
          incident.makeUnitVector();

          return(myPoint.getDotProduct(incident));

          //Double toReturn =
```

```
Math.min(Math.acos(myPoint.getDotProduct(incident)) / (Math.PI),
1.0);
          //if (String.valueOf(toReturn).equals("NaN")) {
          //   System.out.println(hittingPoint + " + " +
incident);
          //}
          //return toReturn;
     }

     /**
      * Returns the radius of the sphere
      * @return the radius
      */
     public double getRadius() {
          return radius;
     }

}

** SolidTorus.java

package threeDWorld.threeDObjectsImpl.objects;

import threeDWorld.raysAndVectors.LightProperties;
import java.util.ArrayList;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Object3D;

public class SolidTorus implements Object3D {

     protected ThreeDPoint midx, midy, midz;

     private double dir;

     private double R;

     private double s;

     private LightProperties lp;

     private double[][] rMatrix;
```

```
     private double[][] rInverse;

     private double[] shift;

     /**
      *
      * @param xmid = x coord of middle point
      * @param ymid = y coord of middle point
      * @param zmid = z coord of middle point
      * @param dir = random vector from center to random point
on outmost edge
      * @param Rad = distance from center of torus to center of
ring
      * @param rad = radius of ring
      * @param lps = light properties
      */

     public SolidTorus(ThreeDPoint xmid, ThreeDPoint ymid,
ThreeDPoint zmid, double dir, double Rad, double rad,
LightProperties lps){
          midx = xmid;
          midy = ymid;
          midz = zmid;
          R = Rad;
          s = rad;
          lp = lps;

          //ThreeDPoint
     }

     private ArrayList<Double> solveQuartic(double a, double b,
double c, double d, double e){
          ArrayList<Double> roots = new ArrayList<Double>();
          double f = ( (-3*b*b)/(8*a*a) ) + (c/a);
          double g = ( (b*b*b)/(8*a*a*a)) - ( (b*b)/(2*a*a) ) +
(d/a);
          double h = ( (-3*Math.pow(b, 4))/ (256*Math.pow(a, 4))
) + ( (c*Math.pow(b, 2))/(16*Math.pow(a, 3)) ) - ( (b*d)/
(4*Math.pow(a, 2))) + (e/a);
          double ba = (-b/(4*a));
          if(g == 0){
```

```
                double ff = (f*f) - (4*h);
                if(ff < 0){
                     return null;
                }
                double ff1 = (-f + Math.sqrt(ff))/2;
                if(ff1 >=0){
                     roots.add(ba + Math.sqrt(ff1));
                     roots.add(ba - Math.sqrt(ff1));
                }
                double ff2 = (-f - Math.sqrt(ff))/2;
                if(ff2 >=0){
                     roots.add(ba + Math.sqrt(ff2));
                     roots.add(ba - Math.sqrt(ff2));
                }
                return roots;
           }
           double p = ((-f*f)/12) - h;
           double q = ((-Math.pow(f,3))/108) + ((f*h)/3) -
((Math.pow(g, 2))/2);
           double r = (-q/2) + Math.sqrt(((q*q)/4) +
((Math.pow(p,3))/27));
           double u = Math.pow(r, (1/3));
           double y = ((-5/6)*f) + u;
           if(u == 0){
                y += (p/(3*u));
           }
           else
                y += (Math.pow(q, (1/3)));
           double w = Math.sqrt(f + (2*y));
           double gg = (2*g/w);
           double gg1 = -((3*f) + (2*y) + gg);
           if(gg1 >= 0){
                double ggsq = Math.sqrt(gg1);
                roots.add(ba + ((w + ggsq)/2));
                roots.add(ba + ((w - ggsq)/2));
           }
           double gg2 = -((3*f) + (2*y) - gg);
           if(gg2 >= 0){
                double ggsq = Math.sqrt(gg2);
                roots.add(ba + ((-w + ggsq)/2));
                roots.add(ba + ((-w - ggsq)/2));
           }
```

```
          return roots;

     }

     @Override
     public ThreeDPoint checkForCollision(Vector3D theVector) {
          // TODO Auto-generated method stub
          return null;
     }

     @Override
     public double getCosineOfAngleFromNormal(ThreeDPoint hit,
               ThreeDPoint incident) {
          // TODO Auto-generated method stub
          return 0;
     }

     @Override
     public LightProperties getLP(ThreeDPoint passed) {
          // TODO Auto-generated method stub
          return null;
     }

     @Override
     public Vector3D getNewVector(Vector3D theVector,
ThreeDPoint hittingPoint) {
          // TODO Auto-generated method stub
          return null;
     }

     public static void main(String[] args){
          ThreeDPoint point = new ThreeDPoint(1,1,1);
          SolidTorus st = new SolidTorus(point, point, point, 1,
1, 1, null);
          ArrayList<Double> roots = st.solveQuartic(1, -6, 11,
-6, 0);
          if(roots.size() == 0)
               System.out.println("There are no real roots");
          double smallest = 1000000000;
          for(int i = 1; i < roots.size(); i++){
               System.out.println(roots.get(i));
               if((roots.get(i) < 0))
```

```
                    continue;
               if(roots.get(i) < smallest)
                    smallest = roots.get(i);
          }
          System.out.println(smallest);
     }

}
```

** SphereCamera.java

```
package threeDWorld.threeDObjectsImpl;

import java.util.ArrayList;
import java.util.Random;

import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.ThreeDPointAvg;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Camera3D;
import threeDWorld.threeDObjectsImpl.objects.SolidRectangle;
import threeDWorld.threeDObjectsImpl.objects.SolidSphere;

/**
 * The most basic and most natural camera class. This class
contains two objects -- a sphere, to represent the camera, and a
rectangle,
 * to represent the viewing plane. In order for a ray to be
valid, it must pass through the viewing plane and hit the
sphere. The radius
 * of the sphere can be considered the size of a retna -- the
larger it is, the more effect light will have. The smaller it
is, the more
 * resolution, but the darker the image.
 *
 * @author ryan
 *
 */
public class SphereCamera implements Camera3D {

     private SolidSphere theSphere;
     private SolidRectangle thePlane;
```

```
    private boolean dropX, dropY, dropZ;

    private ThreeDPoint ver1,ver2,ver3,ver4;

    private Random myR = new Random();

    /**
     * Constructs a new camera
     *
     * v1 - v4 should be in a clockwise order as according to
the documentation in SolidRectangle
     *
     * @param v1 the first vertex of the window
     * @param v2 the second vertex of the window
     * @param v3 the third vertex of the window
     * @param v4 the fourth vertex of the window
     * @param c the center of the sphere
     * @param radius the radius of the sphere
     */
    public SphereCamera(ThreeDPoint v1, ThreeDPoint v2,
ThreeDPoint v3, ThreeDPoint v4, ThreeDPoint c, double radius) {
        thePlane = new SolidRectangle(v1, v2, v3, v4, null);
        theSphere = new SolidSphere(radius, c, null);

        ver1 = v1;
        ver2 = v2;
        ver3 = v3;
        ver4 = v4;

        /*
         * Ryan attempting math again... oh boy!
         *
         * first, we need to convert the plane into 2D
         *
         * Drop the least significant component, i.e., drop
the one with the least change amongst all four points.
         *
         *We're going to do it by finding the MAD for each
point
         *
         */
```

```
          ThreeDPointAvg avg = new ThreeDPointAvg();
          avg.addPoint(v1);
          avg.addPoint(v2);
          avg.addPoint(v3);
          avg.addPoint(v4);

          ThreeDPoint theAvg = avg.getAverage();

          ThreeDPoint difference = new ThreeDPoint();

          difference.x = Math.abs(v1.x - theAvg.x) +
Math.abs(v2.x - theAvg.x) + Math.abs(v3.x - theAvg.x) +
Math.abs(v4.x - theAvg.x);
          difference.y = Math.abs(v1.y - theAvg.y) +
Math.abs(v2.y - theAvg.y) + Math.abs(v3.y - theAvg.y) +
Math.abs(v4.y - theAvg.y);
          difference.z = Math.abs(v1.z - theAvg.z) +
Math.abs(v2.z - theAvg.z) + Math.abs(v3.z - theAvg.z) +
Math.abs(v4.z - theAvg.z);

          //System.out.println(difference.x + "," + difference.y
+ "," + difference.z);

          dropZ = false;
          dropX = false;
          dropY = false;

          if (difference.z < difference.x && difference.z <
difference.y) {
               dropZ = true;
          }

          if (difference.x < difference.z && difference.x <
difference.y) {
               dropX = true;
          }

          if (difference.y < difference.x && difference.y <
difference.z) {
               dropY = true;
          }
```

```
          // now see if any of them were equal...
          if (difference.y == difference.x && difference.y <
difference.z) {
               dropY = true;
          }

          if (difference.y == difference.z && difference.y <
difference.x) {
               dropY = true;
          }

          if (difference.x == difference.z && difference.x <
difference.y) {
               dropX = true;
          }
     }

     @Override
     public ArrayList<Vector3D> generateVectors(int count) {
          /*
           *
           * Time for some RYAN MATH MAGIC!
           *
           * Let's make the following assumption:
           *
           * Let vec1 to vec4 be the vectors that pass through
the four vertices of the square from the center of the camera
           *
           * The maximum and minimum values of vec1 through vec4
(for each component) make up the range for our vectors.
           *
           * Works best if your screen is vertical.
           */

          // calculate vec1 to vec4
          ThreeDPoint vec1 =
ver1.vectorSubtract(theSphere.getCenter());
          ThreeDPoint vec2 =
ver2.vectorSubtract(theSphere.getCenter());
          ThreeDPoint vec3 =
ver3.vectorSubtract(theSphere.getCenter());
```

```
          ThreeDPoint vec4 =
ver4.vectorSubtract(theSphere.getCenter());

          // min/max instead of ? to make code more readable.
Only runs once -- no effic. problem
          double xMin = Math.min(vec1.x, vec2.x);
          xMin = Math.min(xMin, vec3.x);
          xMin = Math.min(xMin, vec4.x);

          double yMin = Math.min(vec1.y, vec2.y);
          yMin = Math.min(yMin, vec3.y);
          yMin = Math.min(yMin, vec4.y);

          double zMin = Math.min(vec1.z, vec2.z);
          zMin = Math.min(zMin, vec3.z);
          zMin = Math.min(zMin, vec4.z);

          double xMax = Math.max(vec1.x, vec2.x);
          xMax= Math.max(xMax, vec3.x);
          xMax = Math.max(xMax, vec4.x);

          double yMax = Math.max(vec1.y, vec2.y);
          yMax= Math.max(yMax, vec3.y);
          yMax = Math.max(yMax, vec4.y);

          double zMax = Math.max(vec1.z, vec2.z);
          zMax= Math.max(zMax, vec3.z);
          zMax = Math.max(zMax, vec4.z);


          ArrayList<Vector3D> toReturn = new
ArrayList<Vector3D>();

          int i = 0;
          while (i != count) {
               Vector3D myVec = new Vector3D();
               myVec.startingPosition = theSphere.getCenter();

               double randX = (myR.nextDouble() * (xMax - xMin))
+ xMin;
               double randY = (myR.nextDouble() * (yMax - yMin))
+ yMin;
```

```
                double randZ = (myR.nextDouble() * (zMax - zMin))
+ zMin;

                myVec.direction = new ThreeDPoint(randX, randY,
randZ);

                // just a note: thePlane.checkForCollision will
normalize our vector. We don't really care.
                ThreeDPoint hittingPoint =
thePlane.checkForCollision(myVec);
                if (hittingPoint == null) {
                     continue;
                }

                // move the vector up so the starting point is on
the plane...
                myVec.startingPosition = hittingPoint;
                myVec.isAtEye = true;

                toReturn.add(myVec);
                i++;
           }

           //System.out.println(xMin + "|" + xMax);
           //System.out.println(yMin + "|" + yMax);
           //System.out.println(zMin + "|" + zMax);

           return toReturn;

     }

     @Override
     public Vector3D getNullVector() {
          Vector3D toReturn = new Vector3D();
          toReturn.isAtEye = true;
          toReturn.direction = new ThreeDPoint();
          toReturn.startingPosition = theSphere.getCenter();

          return toReturn;
     }

     @Override
```

```
     public double getPlaneArea() {
          return thePlane.getArea();
     }

     @Override
     public ThreeDPoint testVector(Vector3D theVector) {
          ThreeDPoint test =
thePlane.checkForCollision(theVector);

          if (test == null) { return null; }

          if (theSphere.checkForCollision(theVector) != null)
{ return test; }
          return null;
     }

     public static void main(String[] args) {
          SphereCamera myCamera = new SphereCamera(new
ThreeDPoint(0,4,0), new ThreeDPoint(4,4,0), new
ThreeDPoint(4,0,0), new ThreeDPoint(0,0,0), new
ThreeDPoint(1,1,1), 0.5);

          ArrayList<Vector3D> theList =
myCamera.generateVectors(100);
          for (Vector3D v : theList) {
               System.out.println(v.direction);
          }

          System.out.println(theList.size());

     }

     @Override
     public double getPlaneHeight() {
          return thePlane.getHeight();
     }

     @Override
     public double getPlaneWidth() {
          return thePlane.getWidth();
     }
```

```
    @Override
    public Vector3D getVectorThroughPoint(double x, double y) {
        /*
         * Ok, so Ryan is going to try to do some math...
         *
         * The plane equation:
         *
         * ax + by + cz + d = 0
         *
         * Where: a,b,c are the normal unit vector of the
plane
         *        x,y,z are any given points on the plane
         *        d = (-a * x0) - (b * y0) - (c * z0) where
x0,y0,z0 are some point on the plane
         *
         * Now we just need to figure out two of the three
coordinates... that's kind of hard. The following is wrong:
         *
         * Therefore, our new X coord is: ver1.x + paramX
(paramX = the passed in X)
         *            our new Y coord is: ver1.y + paramY
(paramY = the passed in Y)
         *
         * Thus, solving the plane equation for z yields:
         *
         * new Z coord = ( 0 - (ax + by + d)) / c
         *
         */

        // TODO assuming all Z points are the same
        ThreeDPoint hittingPoint = new ThreeDPoint(ver1.x + x,
ver1.y + y, ver1.z);

        Vector3D toReturn = new Vector3D();
        toReturn.startingPosition = theSphere.getCenter();

        // thing we are going to minus the thing we are at
        toReturn.direction =
hittingPoint.vectorSubtract(toReturn.startingPosition);

        // now move it so the starting point is on the plane..
        toReturn.startingPosition =
```

```
thePlane.checkForCollision(toReturn);


          return toReturn;

     }

     @Override
     public ThreeDPoint getTopLeftPoint() {
          return ver2;
     }

     @Override
     public ThreeDPoint getXYPointAt(ThreeDPoint pointHit) {
          ThreeDPoint newV2 = new ThreeDPoint();
          ThreeDPoint newHit = new ThreeDPoint();

          if (dropX) {
               newV2.x = ver2.z;
               newV2.y = ver2.y;
               newHit.x = pointHit.z;
               newHit.y = pointHit.y;
          } else if (dropY) {
               newV2.x = ver2.x;
               newV2.y = ver2.z;
               newHit.x = pointHit.x;
               newHit.y = pointHit.z;
          } else if (dropZ){
               newV2.x = ver2.x;
               newV2.y = ver2.y;
               newHit.x = pointHit.x;
               newHit.y = pointHit.y;
          } else {
               // worse comes to worse, dropZ.
               newV2.x = ver2.x;
               newV2.y = ver2.y;
               newHit.x = pointHit.x;
               newHit.y = pointHit.y;
          }

          /*
           * Ok, now:
```

```
            *
            * Let:
            *
            * imageX = newHit.x - newV2.x
            * imageY = -( newHit.y - newV2.y )
            *
            */

           double imageX = newHit.x - newV2.x;
           double imageY = (newV2.y - newHit.y);

           return new ThreeDPoint(imageX, imageY, 0);
     }

}

** World.java

package threeDWorld;

import java.awt.Color;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;

import javax.imageio.ImageIO;

import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.threeDObjectsDefinitions.Camera3D;
import threeDWorld.threeDObjectsDefinitions.Lightsource;
import threeDWorld.threeDObjectsDefinitions.Object3D;
import threeDWorld.threeDObjectsImpl.SphereCamera;
import threeDWorld.threeDObjectsImpl.lights.SphereLight;
import threeDWorld.threeDObjectsImpl.objects.ImageRectangle;
import threeDWorld.threeDObjectsImpl.objects.ImageSphere;
import threeDWorld.threeDObjectsImpl.objects.SolidCone;
import threeDWorld.threeDObjectsImpl.objects.SolidCube;
import threeDWorld.threeDObjectsImpl.objects.SolidCylinder;
import threeDWorld.threeDObjectsImpl.objects.SolidSphere;

/**
```

```
 * We store every object in one array list, every light source
in another, and the one camera.
 * @author ryan
 *
 */
public class World {

     private final static String pathToTexture =
System.getProperty("user.home") +
"/Documents/metro/Images/ice.jpg";
     private final static String pathToTexture2 =
System.getProperty("user.home") +
"/Documents/metro/Images/fire.png";
     private final static String pathToTexture3 =
System.getProperty("user.home") +
"/Documents/metro/Images/earth.jpg";
     private final static String pathToTexture4 =
System.getProperty("user.home") +
"/Documents/metro/Images/leather.jpg";
     private final static String pathToTexture5 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall4.jpg";
     private final static String pathToTexture6 =
System.getProperty("user.home") +
"/Documents/metro/Images/metal.jpg";
     private final static String pathToTexture7 =
System.getProperty("user.home") +
"/Documents/metro/Images/metal2.jpg";



     private final static String pathToWallImage =
System.getProperty("user.home") +
"/Documents/metro/Images/wall.jpg";
     private final static String pathToWallImage2 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall2.jpg";
     private final static String pathToWallImage3 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall3.jpg";
     private final static String pathToWallImage4 =
System.getProperty("user.home") +
```

```
"/Documents/metro/Images/sky.jpg";
     private final static String pathToWallImage5 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall5.png";
     private final static String pathToWallImage6 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall6.jpg";
     private final static String pathToWallImage7 =
System.getProperty("user.home") +
"/Documents/metro/Images/wall7.jpg";
     private final static String pathToFloorImage =
System.getProperty("user.home") +
"/Documents/metro/Images/floor.jpg";

     public ArrayList<Object3D> objects;
     public ArrayList<Lightsource> lights;
     public Camera3D camera;

     public World() {
          objects = new ArrayList<Object3D>();
          lights = new ArrayList<Lightsource>();
     }

     public static World getFireAndIce() {
          // create the world...
          World myWorld = new World();

          // camera time...
          // points of plane:
          // -3, 0, 0
          // -3, 3, 0
          //  3, 3, 0
          //  3, 0, 0

          // center of camera: 0.0,1.0,-3.0
          // radius: 1.0

          SphereCamera myCam = new SphereCamera(new
ThreeDPoint(-3.0, 0.0, 0.0), new ThreeDPoint(-3.0, 3.0, 0.0),
new ThreeDPoint(3.0,3.0,0.0), new ThreeDPoint(3.0,0.0,0.0), new
ThreeDPoint(0.0, 1.0,-3.0), 0.5);
          myWorld.camera = myCam;
```

```
          // now 2 spheres...

          LightProperties myProps = new LightProperties();

          myProps.isLight = false;
          myProps.objectColor = new Color(199,21,133);
          myProps.reflection = false;
          myProps.transparency = 0;
          myProps.indexOfRefraction = 1.0 + (1.0/3.0);

          try {
               ImageSphere mySphere = new ImageSphere(1.5, new
ThreeDPoint(-2.0,1.0,4.0), ImageIO.read(new
File(pathToTexture)), myProps, 200);
               myWorld.objects.add(mySphere);
          } catch (IOException e1) {
               e1.printStackTrace();
          }

          try {
               ImageSphere mySphere = new ImageSphere(1.5, new
ThreeDPoint(2.0,1.0,4.0), ImageIO.read(new
File(pathToTexture2)), myProps, 90);
               myWorld.objects.add(mySphere);
          } catch (IOException e1) {
               e1.printStackTrace();
          }


          myProps = new LightProperties();
          myProps.isLight = false;
          myProps.objectColor = new Color(238,213,183);
          myProps.reflection = true;
          myProps.transparency = 0;

          // a mirror sphere!
          try {
               SolidSphere mySphere = new SolidSphere(0.75, new
ThreeDPoint(0.0, 3.0, 4.0), myProps);
               myWorld.objects.add(mySphere);
          } catch (Exception e) {
```

```
            e.printStackTrace();
        }

        // now a light above the sphere...
        // center: 1.0, 4.0 3.0

        SphereLight myLight = new SphereLight(0.15, new
ThreeDPoint(0.0,1.0, 1.0), Color.white);
        myWorld.lights.add(myLight);

        // now the wall behind that sphere

        myProps = new LightProperties();
        myProps.isLight = false;
        myProps.objectColor = new Color(238,213,183);
        myProps.reflection = false;
        myProps.transparency = 0;

        // back wall
        // points:
        // -6,-6,8
        // -6, 6,8
        //  6, 6,8
        //  6,-6,8

        ImageRectangle myRect;
        try {
            myRect = new ImageRectangle(new ThreeDPoint(-6.0,
-6.0, 8.0), new ThreeDPoint(-6.0, 6.0, 8.0), new
ThreeDPoint(6.0, 6.0, 8.0), new ThreeDPoint(6.0, -6.0, 8.0),
ImageIO.read(new File(pathToWallImage2)), myProps);
            myWorld.objects.add(myRect);
        } catch (Exception e1) {
            e1.printStackTrace();
        }


        // right wall
        // points:
        // 6,-6,-2
        // 6, 6,-2
        // 6, 6, 8
```

```
          // 6,-6, 8

          ImageRectangle myRect3;
          try {
               myRect3 = new ImageRectangle(new ThreeDPoint(6.0,
-6.0, -2.0), new ThreeDPoint(6.0, 6.0, -2.0), new
ThreeDPoint(6.0, 6.0, 8.0), new ThreeDPoint(6.0, -6.0, 8.0),
ImageIO.read(new File(pathToWallImage3)), myProps);
               myWorld.objects.add(myRect3);
          } catch (IOException e1) {
               e1.printStackTrace();
          }


          // floor
          // points
          // -6, -6,-2
          // -6, -6, 8
          //  6, -6, 8
          //  6, -6,-2
          ImageRectangle myRect5;
          try {
               myRect5 = new ImageRectangle(new ThreeDPoint(-6.0,
-6.0, -2.0), new ThreeDPoint(-6.0, -6.0, 8.0), new
ThreeDPoint(6.0, -6.0, 8.0), new ThreeDPoint(6.0, -6.0, -2.0),
ImageIO.read(new File(pathToFloorImage)), myProps);
               myWorld.objects.add(myRect5);
          } catch (IOException e) {
               e.printStackTrace();
          }



          // left wall
          // points:
          // -6,-6,-2
          // -6, 6,-2
          // -6, 6, 8
          // -6,-6, 8

          ImageRectangle myRect2;
          try {
```

```
                myRect2 = new ImageRectangle(new ThreeDPoint(-6.0,
-6.0, -2.0), new ThreeDPoint(-6.0, 6.0, -2.0), new ThreeDPoint(-
6.0, 6.0, 8.0), new ThreeDPoint(-6.0, -6.0, 8.0),
ImageIO.read(new File(pathToWallImage)), myProps);
                myWorld.objects.add(myRect2);
           } catch (IOException e1) {
                e1.printStackTrace();
           }

           // ceiling
           // points
           // -6, 6,-2
           // -6, 6, 8
           //  6, 6, 8
           //  6, 6,-2

           myProps = new LightProperties();
           myProps.isLight = false;
           myProps.objectColor = new Color(238,213,183);
           myProps.reflection = false;
           myProps.transparency = 0;
           myProps.noShadow = true;

           ImageRectangle myRect4;
           try {
                myRect4 = new ImageRectangle(new ThreeDPoint(-6.0,
6.0, -2.0), new ThreeDPoint(-6.0, 6.0, 8.0), new
ThreeDPoint(6.0, 6.0, 8.0), new ThreeDPoint(6.0, 6.0, -2.0),
ImageIO.read(new File(pathToWallImage4)), myProps);
                myWorld.objects.add(myRect4);
           } catch (IOException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
           }

           return myWorld;
      }

      /**
       * Gets a frame of the fire and ice tween. Frames range
from 0 to 100.
       * @param frame the frame to get
```

```
      * @return
      */
     public static World getFireAndIceTween(int frame) {
          final double maxFrames = 400;

          World toReturn = World.getFireAndIce();
          // the index of the mirror sphere is 2
          // let's start it at neg 2 and move it towards pos 2

          double newXpos = ((4 / maxFrames) * frame) - 2;
          SolidSphere ss = ((SolidSphere)
toReturn.objects.get(2));

          ThreeDPoint newPoint = ss.getCenter();
          newPoint.x = newXpos;
          newPoint.y = 3.5;

          ss = new SolidSphere(ss.getRadius(), newPoint,
ss.getLP(null));

          toReturn.objects.set(2, ss);

          double rot = (frame * 360.0) / maxFrames;
          ((ImageSphere)
toReturn.objects.get(0)).setRotation((int)rot);
          ((ImageSphere)
toReturn.objects.get(1)).setRotation((int)rot);



          return toReturn;
     }

     public static World getCylinderWorld() {
          // create the world...
          World myWorld = new World();

          // camera time...
          // points of plane:
          // -3, 0, 0
          // -3, 3, 0
          //  3, 3, 0
```

```
          //  3, 0, 0

          // center of camera: 0.0,1.0,-3.0
          // radius: 1.0

          SphereCamera myCam = new SphereCamera(new
ThreeDPoint(-3.0, 0.0, 0.0), new ThreeDPoint(-3.0, 3.0, 0.0),
new ThreeDPoint(3.0,3.0,0.0), new ThreeDPoint(3.0,0.0,0.0), new
ThreeDPoint(0.0, 1.0,-3.0), 2.0);
          myWorld.camera = myCam;


          // now two cylindars...
          LightProperties myProps = new LightProperties();

          myProps.isLight = false;
          myProps.objectColor = new Color(199,21,133);
          myProps.reflection = false;
          myProps.transparency = 0;
          myProps.indexOfRefraction = 1.0 + (1.0/3.0);

          SolidCylinder myCyl = new SolidCylinder(new
ThreeDPoint(-1.0, 2.0, 3.0), new ThreeDPoint(-4.0, 2.0, 6.0),
1.0, myProps);
          myWorld.objects.add(myCyl);

          //SolidCylinder myCyl2 = new SolidCylinder(new
ThreeDPoint(2.0, 0.0, 2.0), new ThreeDPoint(2.0, 2.0, 2.0), 1.0,
myProps);
          //myWorld.objects.add(myCyl2);

          SolidSphere mySh = new SolidSphere(1.0, new
ThreeDPoint(2.0, 0.0, 2.0), myProps);
          myWorld.objects.add(mySh);

          // a light between them...
          SphereLight myLight = new SphereLight(0.25, new
ThreeDPoint(1.0,2.0, 1.0), Color.white);
          myWorld.lights.add(myLight);

          return myWorld;
     }
```

```
     public static World getConeWorld() {
          // create the world...
          World myWorld = new World();

          // camera time...
          // points of plane:
          // -3, 0, 0
          // -3, 3, 0
          //  3, 3, 0
          //  3, 0, 0

          // center of camera: 0.0,1.0,-3.0
          // radius: 1.0

          SphereCamera myCam = new SphereCamera(new
ThreeDPoint(-3.0, 0.0, 0.0), new ThreeDPoint(-3.0, 3.0, 0.0),
new ThreeDPoint(3.0,3.0,0.0), new ThreeDPoint(3.0,0.0,0.0), new
ThreeDPoint(0.0, 1.0,-3.0), 2.0);
          myWorld.camera = myCam;


          // now a cone...
          LightProperties myProps = new LightProperties();

          myProps.isLight = false;
          myProps.objectColor = new Color(199,21,133);
          myProps.reflection = false;
          myProps.transparency = 0;
          myProps.indexOfRefraction = 1.0 + (1.0/3.0);

          SolidCone myCone = new SolidCone(new ThreeDPoint(-4.0,
2.0, 4.5), new ThreeDPoint(-1.0, 2.0, 4.0), 1.0, myProps);
          myWorld.objects.add(myCone);

          //SolidCylinder myCyl2 = new SolidCylinder(new
ThreeDPoint(2.0, 0.0, 2.0), new ThreeDPoint(2.0, 2.0, 2.0), 1.0,
myProps);
          //myWorld.objects.add(myCyl2);

          // a light between them...
          SphereLight myLight = new SphereLight(0.25, new
ThreeDPoint(1.0,2.0, 0.0), Color.white);
```

```
          myWorld.lights.add(myLight);

          return myWorld;
     }

     public static World getCubeWorld() {
          World myWorld = new World();

          LightProperties myProps = new LightProperties();
          myProps.isLight = false;
          myProps.reflection = false;

          // first, let's make a floor
          try {
               ImageRectangle myImgRect = new ImageRectangle(new
ThreeDPoint(0,0,0), new ThreeDPoint(0,0,5), new
ThreeDPoint(5,0,5), new ThreeDPoint(5,0,0), ImageIO.read(new
File(World.pathToFloorImage)), myProps);
               myWorld.objects.add(myImgRect);
          } catch (IOException e) {
               e.printStackTrace();
          }

          // next, a cube elevated off of the floor with
multicolored sides

          //SolidCube myCube = new SolidCube(new
ThreeDPoint(2.5,4,2.5), 3, Color.black, Color.blue, Color.cyan,
Color.darkGray, Color.green, Color.pink);
          //myWorld.objects.add(myCube);

          // now a light
          SphereLight myLight = new SphereLight(0.25, new
ThreeDPoint(5,6,5), Color.white);
          myWorld.lights.add(myLight);

          // now the camera
          //SphereCamera myCam = new SphereCamera(new
ThreeDPoint(-4, 0, 0), new ThreeDPoint(-4,8,0), new
ThreeDPoint(0, 8, -4), new ThreeDPoint(0, 0, -4), new
ThreeDPoint(-3, 6, -3), 0.25);
          SphereCamera myCam = new SphereCamera(new
```

```
ThreeDPoint(0, 0, 0), new ThreeDPoint(0,8,0), new ThreeDPoint(6,
8, 0), new ThreeDPoint(6, 0, 0), new ThreeDPoint(2.5, 6, 2.5),
0.25);


          myWorld.camera = myCam;

          return myWorld;
     }


}

** CosineShadingProcessor.java

package tracing.local;

import globalStuff.Timekeeper;

import java.awt.Color;

import threeDWorld.raysAndVectors.Ray3D;
import tracing.RayProcessor;

/**
 * This class processes incoming rays, shades them using cosine
shading, and then forwards them to the passed RayProcessor.
 * @author ryan
 *
 */
public class CosineShadingProcessor implements RayProcessor {

     private double ambient;
     private RayProcessor toFor;

     /**
      * Creates a new cosine shading processor
      * @param amb the amount of ambient light (between 0 and 1)
      * @param toForward the proccesor to pass the shaded rays
to
      */
     public CosineShadingProcessor(double amb, RayProcessor
```

```
toForward) {
          ambient = amb;
          toFor = toForward;
     }


     @Override
     public void gotRay(Ray3D theRay, Color realColor) {
          long currentTime;
          if (Timekeeper.timingEnabled) { currentTime =
System.currentTimeMillis(); }

          Color theColor = theRay.calculateFinalColor();

          /*
           * Shading stuff, using cosine shading.
           *
           * The angle of every ray we get back will be between
0 and 1, so:
           *
           * each color componenet = (ambient * startColor) + (1
- ambient) * angle
           */

          double coeff = ambient + ((1.0 - ambient) *
theRay.getAngle());
          int newRed = (int) (theColor.getRed() * coeff);
          int newGreen = (int) (theColor.getGreen() * coeff);
          int newBlue = (int) (theColor.getBlue() * coeff);

          theColor = new Color(newRed, newGreen, newBlue);

          toFor.gotRay(theRay, theColor);

          //System.out.println(theRay.getAngle() + ": " +
theColor.getRed() + "," + theColor.getGreen() + "," +
theColor.getBlue());

          if (Timekeeper.timingEnabled)
{ Timekeeper.addTime(Timekeeper.timeCodeForCosineShading,
System.currentTimeMillis() - currentTime); }
     }
```

```
}

** LocationMappingProcessor.java

package tracing.local;

import globalStuff.Timekeeper;

import java.awt.Color;

import threeDWorld.raysAndVectors.Ray3D;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.threeDObjectsDefinitions.Camera3D;
import tracing.RayProcessor;

/**
 * This class sets the realX and realY values of incoming rays
and passes them on to the specified ray processor.
 * @author ryan
 *
 */
public class LocationMappingProcessor implements RayProcessor {

     private ThreeDPoint topLeft;
     private double scaleX;
     private double scaleY;
     private RayProcessor toFor;

     private Camera3D ourCamera;

     /**
      * Constructs a new location mapping processor
      * @param realWidth the width of the image
      * @param imageWidth the width of the camera plane
      * @param realHeight the height of the image
      * @param imageHeight the height of the cameraPlane
      * @param topLeftPoint the top left point of the camera
plane
      * @param toForward the processor to forward our results to
      */
     public LocationMappingProcessor(Camera3D theCamera, double
```

```
realWidth, double imageWidth, double realHeight, double
imageHeight, ThreeDPoint topLeftPoint, RayProcessor toForward) {
          topLeft = topLeftPoint;

          scaleX = (realWidth / imageWidth);
          scaleY = (realHeight / imageHeight);

          //System.out.println(scaleX + "," + scaleY);

          ourCamera = theCamera;

          toFor = toForward;
     }


     public void gotRay(Ray3D theRay, Color realColor) {
          long currentTime;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();

          /* old method:
           * To transform the point on the plane into a point in
our image:
           *
           * take the coordinates of the top left point
           * subtract the x coordinate from the x coordinates of
all the points
           * do the same for y
           * then make all the y values negative (which actually
makes them positive)
           *


          // for now, we're ignoring the Z point
          ThreeDPoint ourPoint;
          if (theRay.getFlag() ==
Ray3D.RAY_FLAG_COMPLETE_FROM_EYE) {
               ourPoint = theRay.getVector(0).startingPosition;
          } else {
               ourPoint =
theRay.getVector(theRay.getVectorCount() - 1).startingPosition;
          }
```

```
          //System.out.println(ourPoint);
          ourPoint = ourPoint.vectorSubtract(topLeft);

          ourPoint.y = (0.0 - ourPoint.y) * scaleY;
          ourPoint.x *= scaleX;

          //System.out.println(ourPoint.x + "," + ourPoint.y);

          theRay.setRealCoords((int) ourPoint.x, (int)
ourPoint.y);
          toFor.gotRay(theRay, realColor);

          //if (theRay.getAngle() != 0) {
          //   System.out.println(theRay.getRealX() + "," +
theRay.getRealY());
          //}

          */

          ThreeDPoint ourPoint;
          if (theRay.getFlag() ==
Ray3D.RAY_FLAG_COMPLETE_FROM_EYE) {
               ourPoint = theRay.getVector(0).startingPosition;
          } else {
               ourPoint =
theRay.getVector(theRay.getVectorCount() - 1).startingPosition;
          }

          ThreeDPoint mapped = ourCamera.getXYPointAt(ourPoint);

          int x = (int) (mapped.x * scaleX);
          int y = (int) (mapped.y * scaleY);

          theRay.setRealCoords(x, y);
          toFor.gotRay(theRay, realColor);

          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForMapping,
System.currentTimeMillis() - currentTime);
          return;
     }
```

```
}

** RayTraceRender.java

package tracing.local;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import threeDWorld.World;
import threeDWorld.raysAndVectors.Ray3D;
import tracing.RayPassback;
import tracing.RayProcessor;
/**
 * This class is essentially a full implementation of ray
tracing. It takes in a world and dimensions, and it will pass
every completed ray
 * onto the passed in RayProcessor.
 *
 * @author ryan
 *
 */
public class RayTraceRender implements RayPassback, Runnable {

     private static final int BLOCK_SIZE = 600;

     private World theWorld;
     private double ambient;
     private int threadCount;
     private int width;
     private int height;

     private double scaleX, scaleY;

     private ExecutorService myExe;

     private int neededVectors;
     private int returnedVectors;
     private int startedVectors;

     private int antialias ;
```

```
    private int shadowAA;

    private int widthCount = 0;
    private int heightCount = 0;
    private long totalRaysTraced = 0;

    private RayProcessor toProc;
    private CosineShadingProcessor ourShader;
    private LocationMappingProcessor ourMapper;



    /**
     * Creates a new RayTraceRender that will perform a ray
trace on the passed light. This particular implementation using
cosine shading,
     * so ambLight is the amount of ambient light in the scene
not from a light source.
     *
     * @param myWorld the world to render
     * @param ambLight the amount of background light
     * @param thread the number of threads
     * @param w the width of the desired image
     * @param h the height of the desired image
     * @param aa the anti-aliasing factor (should probably be
set around 4 or 6.)
     * @param saa the anti-aliasing factor for shadows (should
be higher for nicer shadows. Must be at least 1 to have any
shadow.
     * @param toProces the RayProcessor to send completed rays
to
     */
    public RayTraceRender(World myWorld, double ambLight, int
thread, int w, int h, int aa, int saa, RayProcessor toProces) {
        theWorld = myWorld;
        ambient = ambLight;
        threadCount = thread;
        width = w;
        height = h;
        antialias = aa;
        shadowAA = saa;
```

```
          // first, calculate our scaling factors
          scaleX = (theWorld.camera.getPlaneWidth() /
Double.valueOf(width)) / (antialias);
          scaleY = (theWorld.camera.getPlaneHeight() /
Double.valueOf(height)) / (antialias);

          // figure out how many pixels we need
          neededVectors = width * height * antialias *
antialias;

          toProc = toProces;

          ourMapper = new
LocationMappingProcessor(myWorld.camera, w,
myWorld.camera.getPlaneWidth(), h,
myWorld.camera.getPlaneHeight(),
myWorld.camera.getTopLeftPoint(), toProc);
          ourShader = new CosineShadingProcessor(ambient,
ourMapper);
     }

     /**
      * This method is intended for use with distributing -- as
in over mutltiple platforms -- the ray tracing procedure.
      *
      * This method returns the number of rays the tracer needs
to calculate a complete image. After constructing a
RayTraceRender object
      * normally, one can use this method to determine the total
amount of work that needs to be done.
      * @return the number of rays needed
      */
     public int getNeededRays() { return neededVectors; }

     /**
      * This method is intended for use with distributing -- as
in over mutltiple platforms -- the ray tracing procedure.
      *
      * This method sets the number of rays the tracer should
look for. You should use this method in combination with
setWidthCount
      * to make the tracer trace only part of the image.
```

```
      * @param stopAt the number of rays to stop tracing at
      */
     public void setNeededRays(int stopAt) { neededVectors =
stopAt; }

     /**
      * This method is intended for use with distributing -- as
in over mutltiple platforms -- the ray tracing procedure.
      *
      * This method gets the current width count, or where the
tracer is starting at. Mostly here for convience, never really
      * needed.
      * @return the current width count
      */
     public int getWidthCount() { return widthCount; }

     /**
      * Returns the highest valid value for width count
      * @return the highest valid value
      */
     public int getMaxWidthCount() { return width * antialias; }

     /**
      * This method is intended for use with distributing -- as
in over mutltiple platforms -- the ray tracing procedure.
      *
      * Sets the width count to start tracing at. You should use
this method to start the tracer at some given width into the
image
      * so that you can distribute the ray tracer's work over
multiple computers.
      *
      * @param startingWidthCount the width count to start at
      */
     public void setWidthCount(int startingWidthCount)
{ widthCount = startingWidthCount; }

     @Override
     public synchronized void gotRay(Ray3D theRay, boolean
requested) {
          totalRaysTraced++;
          if (requested) {
```

```
                returnedVectors++;

                // try to get the next chunk
                // it'll figure it out if we don't need it / over
shoot
                this.tracePixels(1);
           }
           if (theRay.getFlag() !=
Ray3D.RAY_FLAG_COMPLETE_FROM_EYE) return;



           if (toProc == null) { return; }

           // do the shading and forward the ray. Keep in mind
that ourShader will pass the ray onto a LocationMappingProcessor
           ourShader.gotRay(theRay, null);
      }

      @Override
      public void run() {
           myExe =
Executors.newFixedThreadPool(this.threadCount);

           // start tracing some pixels
           // give a decent buffer size
           tracePixels(RayTraceRender.BLOCK_SIZE * 2);
      }

      private synchronized void tracePixels(int toTrace) {
           if (toTrace == 0) { return; }

           int i = toTrace;
           while (i != 0) {
                if (startedVectors == neededVectors) { return; }

                if (heightCount == height * antialias)
{ heightCount = 0; widthCount++; }

                double x = scaleX * Double.valueOf(widthCount);
                double y = scaleY * Double.valueOf(heightCount);
                //System.out.println(x + "," + y);
```

```
                myExe.execute(new RayTraceRenderThread(theWorld,
theWorld.camera.getVectorThroughPoint(x, y), this, shadowAA));
                heightCount++;
                i--;
                startedVectors++;
                //System.out.println(startedVectors);
           }




     }

     /**
      * Returns true if the tracer is done, false otherwise
      * @return the status
      */
     public boolean isFinished() {
          return (neededVectors == returnedVectors);
     }

     /**
      * Returns a number between 0 and 1 (finished) indicating
the progress of the tracer
      * @return
      */
     public double getProgress() {
          //System.out.println("Missing: " + (neededVectors -
returnedVectors) + " out of " + neededVectors);
          return Double.valueOf(returnedVectors) /
Double.valueOf(neededVectors);
     }

     /**
      * Gets the total number of rays traced by the tracer,
including non-requested (shadow AA and reflection) rays
      * @return the total number of rays traced
      */
     public long getTotalRays() { return totalRaysTraced; }

     public void shutdown() {
```

```
          myExe.shutdown();
     }

}

** RayTraceRenderThread.java

package tracing.local;

import globalStuff.Timekeeper;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

import threeDWorld.World;
import threeDWorld.raysAndVectors.LightProperties;
import threeDWorld.raysAndVectors.Ray3D;
import threeDWorld.raysAndVectors.ThreeDPoint;
import threeDWorld.raysAndVectors.Vector3D;
import threeDWorld.threeDObjectsDefinitions.Lightsource;
import threeDWorld.threeDObjectsDefinitions.Object3D;
import tracing.RayPassback;

/**
 * This class is responsible for tracing a single passed ray and
returning it. This class differs from the other ray tracing
thread class
 * because it calculates the angle between the normal of the
object and the light ray, which is helpful for cosine shading.
Only works with
 * one light source.
 * @author ryan
 *
 */
public class RayTraceRenderThread implements Runnable {
     private int numShadowAA;

     private World theWorld;
     private Ray3D ourRay;
     private RayPassback thePB;
```

```
     private ThreeDPoint smallTDP;

     public RayTraceRenderThread(World world, Vector3D toStart,
RayPassback pb, int shadowAA) {
          theWorld = world;
          thePB = pb;

          ourRay = new Ray3D();
          ourRay.addVector(toStart);
          numShadowAA = shadowAA;
     }

     public void run() {
          Vector3D currentVector =
ourRay.getVector(ourRay.getVectorCount() - 1);


          // get the object closest
          Object o = getClosestObject(currentVector, null);

          if (o == null) {
               // we didn't hit anything
               ourRay.setFlag(Ray3D.RAY_FLAG_COMPLETE_KILLED);
               thePB.gotRay(ourRay, true);
               return;
          }

          if (o instanceof Lightsource) {
               //System.out.println("Light: " +
ourRay.getVectorCount());
               // we hit a light! this ray is totally complete.
               ourRay.addVector(((Lightsource)
o).getNullVector());
               ourRay.setAngle(1);
               ourRay.setFlag(Ray3D.RAY_FLAG_COMPLETE_FROM_EYE);
               thePB.gotRay(ourRay, true);
               return;
          }

          LightProperties ourLP = ((Object3D)
o).getLP(smallTDP);
```

```
          // at this point, we know that the object hit is an
Object3D

          // see if we need to refract or refect...
          if (ourLP.transparency != 0 || ourLP.reflection) {
               //System.out.println("Found trans. Depth: " +
ourRay.getVectorCount());
               // we need to shot a ray through
               ourRay.addVector(((Object3D)
o).getNewVector(currentVector, smallTDP));
               // now recurse...
               this.run();
               return;
          }


          // see if a ray drawn from the hitting point hits a
light
          Vector3D testVec = new Vector3D();
          testVec.startingPosition = smallTDP;
          testVec.theProps = ourLP;

          for (Lightsource LS : theWorld.lights) {
               ArrayList<ThreeDPoint> toTest =
LS.getPossibleHittingPoints(numShadowAA);
               boolean haveSentFirst = false;

               for (ThreeDPoint tdp : toTest) {
                    testVec.direction =
tdp.vectorSubtract(testVec.startingPosition);
                    testVec.direction.makeUnitVector();
                    Ray3D modRay = ourRay.clone();

                    Object oo = getClosestObject(testVec, o);
                    // don't need to check for nothing because we
will hit the light source

                    if (oo instanceof Lightsource) {

                         modRay.addVector(testVec);
                         modRay.addVector(LS.getNullVector());
```

```
                        // put in the angle for cosine
shading...
                        modRay.setAngle(((Object3D)
o).getCosineOfAngleFromNormal(testVec.startingPosition,
testVec.direction));

                   } else {

                        modRay.addVector(testVec);
                        modRay.addVector(LS.getNullVector());

                        Object3D theObj = (Object3D) o;

                        if
(theObj.getLP(testVec.startingPosition).noShadow) {


modRay.setAngle(theObj.getCosineOfAngleFromNormal(testVec.starti
ngPosition, testVec.direction));
                        } else {
                             modRay.setAngle(0.5 *
theObj.getCosineOfAngleFromNormal(testVec.startingPosition,
testVec.direction));
                             //modRay.setAngle(0.0);
                        }

                   }


     modRay.setFlag(Ray3D.RAY_FLAG_COMPLETE_FROM_EYE);
                   thePB.gotRay(modRay, !(haveSentFirst));
                   if (haveSentFirst == false) { haveSentFirst =
true; }
              }
              return;
          }


          ourRay.setFlag(Ray3D.RAY_FLAG_COMPLETE_KILLED);
          thePB.gotRay(ourRay, true);
          return;
```

```
     }

     /**
      * Finds the closet object that the passed vector will hit,
excluding (for efficency) the object it is coming from.
      * @param theVec the vector
      * @param startingObject the starting object
      * @return the closet object we hit
      */
     private Object getClosestObject(Vector3D theVec, Object
startingObject) {
          long currentTime = 0;
          if (Timekeeper.timingEnabled) currentTime =
System.currentTimeMillis();
          HashMap<ThreeDPoint, Object> hit = new
HashMap<ThreeDPoint, Object>(theWorld.objects.size());

          // add in all the objects we hit
          for (Object3D theObj : theWorld.objects) {
               ThreeDPoint hitOrMiss =
theObj.checkForCollision(theVec);
               if (hitOrMiss != null) {
                    // make sure we didn't add ourselves
                    if (theObj == startingObject) { continue; }
                    hit.put(hitOrMiss, theObj);
               }
          }

          // add in all the lights we hit
          for (Lightsource LS : theWorld.lights) {
               ThreeDPoint hitOrMiss = LS.testVector(theVec);
               if (hitOrMiss != null) {
                    hit.put(hitOrMiss, LS);
               }
          }

          // did we actually hit anything?
          if (hit.size() == 0) {
               if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForGetClose,
System.currentTimeMillis() - currentTime);
               return null;
```

```
          }

          // if we are still alive, we hit at least something.
          // find the smallest distance.

          Iterator<ThreeDPoint> toIt = hit.keySet().iterator();
          smallTDP = toIt.next();
          double smallest =
smallTDP.getComparitveDistance(theVec.startingPosition);


          while (toIt.hasNext()) {
               ThreeDPoint next = toIt.next();
               double newDist =
next.getComparitveDistance(theVec.startingPosition);
               if (newDist < smallest) {
                    smallTDP = next;
                    smallest = newDist;
               }
          }

          // now get the object we hit
          // it is either an Object3D or a Lightsource
          if (Timekeeper.timingEnabled)
Timekeeper.addTime(Timekeeper.timeCodeForGetClose,
System.currentTimeMillis() - currentTime);
          return hit.get(smallTDP);
     }

}

** PixelPassback.java

package tracing.network;

import threeDWorld.raysAndVectors.ThreeDPoint;

/**
 * A simple interface that allows one to passback a single
pixel. A pixel, for all intents and purposes, is a ThreeDPoint
where the X value is
 * the x value, the y value is the y value, and the z value is
```

```
the color.
 * @author ryan
 *
 */
public interface PixelPassback {
     /**
      * Passing back a pixel
      * @param pixel the pixel
      */
     public void gotPixel(ThreeDPoint pixel);

     /**
      * We do not have any more pixels to send.
      */
     public void done();
}

** RayTraceClientSideSocket.java

package tracing.network;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.net.Socket;

import threeDWorld.World;
import tracing.local.RayTraceRender;

public class RayTraceClientSideSocket implements Runnable {

     // these fields are duplicated in RayTraceClientSideSocket
     private final int CODE_PIXEL = 1;
     private final int CODE_DONE = 0;

     private String ipAddress;
     private int port;
     private int threads;
     private World ourWorld;

     private RayTraceRender myRender;
```

```
     public RayTraceClientSideSocket(String ip, int thePort,
World theWorld, int threadCount) {

          ourWorld = theWorld;
          ipAddress = ip;
          port = thePort;
          threads = threadCount;

     }

     @Override
     public void run() {
          try {
               Socket ourSock = new Socket(ipAddress, port);
               System.out.println("Starting to connect...");
               // block while we're connecting..
               while (!ourSock.isConnected() ||
ourSock.isClosed()) {
                    Thread.sleep(500);
               }
               System.out.println("Connected. Making
streams...");

               // create our streams
               DataInputStream in = new
DataInputStream(ourSock.getInputStream());
               DataOutputStream out = new
DataOutputStream(ourSock.getOutputStream());

               System.out.println("Streams made. Waiting for
data...");

               // wait until we have some kind of data...
               while (in.available() == 0) {
                    Thread.sleep(500);
               }

               System.out.println("Got data. Sleeping...");

               // we have data! Let's wait a bit just to make
sure all of our data gets here...
```

```
                Thread.sleep(2000);

                System.out.println("Done sleeping.");

                // alright, time to read in what we want!
                // ambient width height antialias numRays
startingWidth shadowAA
                double amb = in.readDouble();
                int w = in.readInt();
                int h = in.readInt();
                int aa = in.readInt();
                int numRays = in.readInt();
                int startingWidth = in.readInt();
                int shadowAA = in.readInt();

                System.out.println("Loaded values: " + amb + "," +
w + "," + h + "," + aa + "," + numRays + "," + startingWidth +
"," + shadowAA);

                System.out.println("Creating the renderer");
                myRender = new RayTraceRender(ourWorld, amb,
threads, w, h, aa, shadowAA);

                myRender.setNeededRays(numRays);
                myRender.setWidthCount(startingWidth);

                System.out.println("Running the renderer");
                // run the renderer (vroom vroom).
                (new Thread(myRender)).start();

                while (!myRender.isFinished()) {
                     // we want to let it run as long as it needs
to.
                     System.out.println("Renderer progress: " +
myRender.getProgress());
                     Thread.sleep(5000);
                }

                System.out.println("Have all data.");
                // alright! we're done! time to grab all of our
data and ship it out!
                BufferedImage result = myRender.getImage();
```

```
                int i = 0;
                int ii = 0;

                while (i != result.getWidth()) {
                     ii = 0;
                     while (ii != result.getHeight()) {
                          try {
                               int theColor = result.getRGB(i,
ii);

                               // verify that it isn't black.
                               // if you are wondering why we can
get ride of all black pixels, it is because all pixels on the
host side
                               // start out as black.
                               Color toCheck = new
Color(theColor);
                               if (toCheck.getRed() == 0 &&
toCheck.getGreen() == 0 && toCheck.getBlue() == 0) {
                                    ii++;
                                    continue;
                               }

                               // we are still here, we need to
send this pixel.
                               //System.out.println("Going to
write: " + this.CODE_PIXEL + "," + i + "," + ii + "," +
theColor);
                               out.writeInt(this.CODE_PIXEL);
                               out.writeInt(i);
                               out.writeInt(ii);
                               out.writeInt(theColor);


                          } catch (Exception e) {
                               // ignore out of bounds. We just
won't send it.
                               // probably black (dead) anyway
                          }
                          ii++;
                     }
```

```
                    i++;
               }

               System.out.println("Sent all data. Sending
termination sequence...");

               // ok, we've sent all of our pixels. Tell the
server we're done...
               out.writeInt(this.CODE_DONE);
               out.writeInt(this.CODE_DONE);
               out.writeInt(this.CODE_DONE);
               out.writeInt(this.CODE_DONE);

               System.out.println("Sent. Cleaning up...");

               out.close();
               in.close();
               ourSock.close();

               System.out.println("Done.");
               return;

          } catch (Exception e) {
               e.printStackTrace();
          }
     }

     /**
      * Returns the results of the internal ray tracer's
getProgress method
      * @return the progress of the tracer
      */
     public double getProgress() {
          return myRender.getProgress();
     }

     /**
      * Runs the client socket
      * @param args none
      */
     public static void main(String[] args) {
          final int threadCount = 7;
```

```
          World theWorld = World.getFireAndIce();
          final String ip = "172.17.114.211";
          final int port = 6002;

          RayTraceClientSideSocket mySock = new
RayTraceClientSideSocket(ip, port, theWorld, threadCount);

          mySock.run();

     }


}
```

** RayTraceServer.java

```
package tracing.network;

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import threeDWorld.World;
import threeDWorld.raysAndVectors.ThreeDPoint;
import tracing.local.RayTraceRender;

/**
 * This class is responsible for running a central server that
will divide up the work needed to render an image and give the
work
 * out EQUALLY to a finite number of connected computers.
 *
 * @author ryan
 *
 */
public class RayTraceServer implements PixelPassback {
```

```
    private int portToListen;
    private ArrayList<Socket> theSockets;
    private int socketsReturned = 0;

    private World toRender;
    private double ambient;
    private int width;
    private int height;
    private int antialias;
    private int shadowAA;

    private HashMap<String, Integer> speedMap;

    private BufferedImage toReturn;

    /**
     * Constructs a new RayTraceServer that will listen on the
passed port when instructed.
     * @param port the port to listen on
     * @param myWorld the world to render
     * @param ambLight the amount of ambient light
     * @param w the width of the image
     * @param h the height of the image
     * @param aa the amount of anti-aliasing to use
     * @param saa the amount of shadow AA
     */
    public RayTraceServer(int port, World myWorld, double
ambLight, int w, int h, int aa, int saa) {
        portToListen = port;
        theSockets = new ArrayList<Socket>();
        toRender = myWorld;
        ambient = ambLight;
        width = w;
        height = h;
        antialias = aa;
        shadowAA = saa;

        speedMap = new HashMap<String, Integer>();

        toReturn = new BufferedImage(w, h,
java.awt.image.BufferedImage.TYPE_INT_RGB);
    }
```

```
     /**
      * Starts listening for connections, and keeps listening
for connections until milis (in miliseconds) has elapsed.
      * @param milis the time to wait
      * @return the number of connections
      */
     public int listenForIncomingConnection(int milis) {
          try {
               ServerSocket ss = new ServerSocket(portToListen);
               ss.setSoTimeout(milis);

               long timeNow = System.currentTimeMillis();
               while (System.currentTimeMillis() - timeNow <
milis) {
                    try {
                         theSockets.add(ss.accept());
                    } catch (java.net.SocketTimeoutException e) {
                         // don't even bother doing anything
                    }
               }

          } catch (IOException e) {
               e.printStackTrace();
          }

          return theSockets.size();
     }

     /**
      * Starts listening from all the connected sockets and
begins to render the image.
      */
     public void start() {
          // first, we make a dummy renderer just to figure out
how many rays we're dealing with
          RayTraceRender myRender = new RayTraceRender(toRender,
ambient, 1, width, height, antialias, shadowAA);

          int totalRaysNeeded = myRender.getNeededRays();
          int totalWidthCount = myRender.getMaxWidthCount();
          int totalShares = theSockets.size();
```

```
          if (speedMap.size() != 0) {
               // calculate the real number of shares
               totalShares = 0;
               for (Integer i : speedMap.values()) {
                    totalShares += i.intValue();
               }
          }

          totalRaysNeeded /= totalShares;
          totalWidthCount /= totalShares;

          int lastDistribWidthCount = 0;

          ExecutorService myExe =
Executors.newFixedThreadPool(theSockets.size() + 1);
          for (Socket s : theSockets) {
               if (speedMap.size() == 0) {
                    // constructor: Socket theSocket, double amb,
int w, int h, int aa, int raysToTrace, int startingWidthCount,
PixelPassback toPB
                    myExe.execute(new RayTraceServerSideSocket(s,
ambient, width, height, antialias, totalRaysNeeded,
lastDistribWidthCount, this, shadowAA));
                    lastDistribWidthCount += totalWidthCount;
               } else {
                    if (!
speedMap.containsKey(s.getInetAddress().getHostAddress()))
{ continue; }
                    int shares =
speedMap.get(s.getInetAddress().getHostAddress());
                    myExe.execute(new RayTraceServerSideSocket(s,
ambient, width, height, antialias, totalRaysNeeded * shares,
lastDistribWidthCount, this, shadowAA));
                    lastDistribWidthCount += totalWidthCount *
shares;
               }
          }
     }

     /**
      * Returns the overall progress of the render, between 0
```

```
and 1.
      * @return the progress
      */
     public double getProgress() {
          return Double.valueOf(socketsReturned) /
Double.valueOf(theSockets.size());
     }

     /**
      * Returns the image rendered by the server
      * @return the image
      */
     public BufferedImage getImage() {
          return toReturn;
     }

     @Override
     public synchronized void gotPixel(ThreeDPoint pixel) {
          /*
           * Remember:
           *
           * x is x
           * y is y
           * z is the color
           */

          toReturn.setRGB((int) pixel.x, (int) pixel.y, (int)
pixel.z);
     }

     public void done() {
          socketsReturned++;
     }

     /**
      * Adds an entry to the speed map. The speed map controls
the number of shares of the work that each computer gets. If the
      * speed map is left empty, everyone will get one share of
the work. If the speed map is not empty, and a computer that is
not in the
      * speed map connects, they will not be given any work.
      * @param ip the ip
```

```
      * @param shareCount the number of shares
      */
     public void addSpeedMapEntry(String ip, int shareCount) {
          speedMap.put(ip, new Integer(shareCount));
     }

}

** RayTraceServerSideSocket.java

package tracing.network;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.ArrayList;

import threeDWorld.raysAndVectors.ThreeDPoint;

public class RayTraceServerSideSocket implements Runnable {

     // these fields are duplicated in RayTraceClientSideSocket
     private final int CODE_PIXEL = 1;
     private final int CODE_DONE = 0;

     private Socket ourSocket;

     private double ambient;
     private int width;
     private int height;
     private int antialias;
     private int shadowAA;

     private int numRays;
     private int startingWidth;

     private PixelPassback thePB;

     private ArrayList<Integer> theBuffer;
```

```
     /**
      * Constucts a new server side socket to talk to a already-
connected client
      *
      * @param theSocket the socket bearing a connection to the
worker client
      * @param amb the amount of ambient light
      * @param w the width
      * @param h the height
      * @param aa the amount of antialiasing
      * @param raysToTrace the number of rays to trace
      * @param startingWidthCount the starting width count for
the worker
      * @param toPB the place to pass our data back to.
      * @param shadowAntiA the amount of shadow AA
      */
     public RayTraceServerSideSocket(Socket theSocket, double
amb, int w, int h, int aa, int raysToTrace, int
startingWidthCount, PixelPassback toPB, int shadowAntiA) {
          ourSocket = theSocket;

          ambient = amb;
          width = w;
          height = h;
          antialias = aa;
          shadowAA = shadowAntiA;

          numRays = raysToTrace;
          startingWidth = startingWidthCount;
          thePB = toPB;

          theBuffer = new ArrayList<Integer>();

     }

     @Override
     public void run() {
          // first things first: make sure the socket is
connected. If it isn't, we're done here.
          if (!ourSocket.isConnected()) { return; }

          // ok, so we're connected. Let's get ready to read and
```

```
write...
          try {
               DataOutputStream myDOS = new
DataOutputStream(ourSocket.getOutputStream());
               DataInputStream in = new
DataInputStream(ourSocket.getInputStream());

               // alright, now send the data that the client
needs to do its job.
               // Format: ambient width height antialias numRays
startingWidth shadowAA

               myDOS.writeDouble(ambient);
               myDOS.writeInt(width);
               myDOS.writeInt(height);
               myDOS.writeInt(antialias);
               myDOS.writeInt(numRays);
               myDOS.writeInt(startingWidth);
               myDOS.writeInt(shadowAA);

               // make sure it all actually gets sent..
               myDOS.flush();

               // now, loop for data.
               while (ourSocket.isConnected()) {

                    if (theBuffer.size() >= 4) {
                         // the first int is a code.
                         //System.out.println(theBuffer.get(0) +
"," + theBuffer.get(1) + "," + theBuffer.get(2) + "," +
theBuffer.get(3));
                         if (theBuffer.get(0).intValue() ==
this.CODE_DONE) {
                              //System.out.println("Got done
code");
                              thePB.done();
                              in.close();
                              myDOS.close();
                              ourSocket.close();
                              return;
                         }
```

```
                        if (theBuffer.get(0).intValue() !=
this.CODE_PIXEL) {
                             // a hardcore error happened... bad
code!
                             //System.out.println("BAD CODE!");
                             System.exit(0);
                        }

                        // if we're still alive, we have a pixel
to process!

                        ThreeDPoint ourPixel = new
ThreeDPoint();
                        // remember, x is x, y is y, and z is
the color.

                        ourPixel.x =
theBuffer.get(1).intValue();
                        ourPixel.y =
theBuffer.get(2).intValue();
                        ourPixel.z =
theBuffer.get(3).intValue();

                        thePB.gotPixel(ourPixel);

                        // remove the four bits we just dealt
with
                        theBuffer.remove(0);
                        theBuffer.remove(0);
                        theBuffer.remove(0);
                        theBuffer.remove(0);
                   }


                   if (in.available() >= 1) {
                        //System.out.println("data");
                        theBuffer.add(new
Integer(in.readInt()));
                   } else {
                        // we don't want to block up
everything...
                        Thread.sleep(50);
```

```
                    }
                }

                //System.out.println("No conn");
                // we are not connected.
                thePB.done();
                in.close();
                myDOS.close();
                // no need to close the socket: it is already
closed.
                return;

           } catch (IOException e) {
                e.printStackTrace();
           } catch (InterruptedException e) {
                e.printStackTrace();
           }

      }


}

** ImageCreator.java

package tracing.processors;

import globalStuff.Timekeeper;

import java.awt.Color;
import java.awt.image.BufferedImage;

import threeDWorld.raysAndVectors.Ray3D;
import tracing.RayProcessor;

/**
 * This is a ray processor that will render an image (in real
time) based on rays passed back to it. It uses weighted
averaging, i.e., the
 * first ray given back has the most significant impact on that
pixel.
 *
```

```
 * @author ryan
 *
 */
public class ImageCreator implements RayProcessor {

     private BufferedImage myImage;

     /**
      * Makes a new image creator that will render an image with
the passed width and passed height
      * @param w the width
      * @param h the height
      */
     public ImageCreator(int w, int h) {
          myImage = new BufferedImage(w, h,
BufferedImage.TYPE_INT_RGB);
     }

     @Override
     public void gotRay(Ray3D theRay, Color realColor) {
          long currentTime = 0;
          if (Timekeeper.timingEnabled) { currentTime =
System.currentTimeMillis(); }

          try {
               int currentRGB = myImage.getRGB(theRay.getRealX(),
theRay.getRealY());
               Color myColor = new Color(currentRGB);
               if (myColor.getRed() == 0 && myColor.getGreen() ==
0 && myColor.getBlue() == 0) {
                    // currently, it is black, so we can ignore
it and just fill in our image.
                    myImage.setRGB(theRay.getRealX(),
theRay.getRealY(), realColor.getRGB());
               } else {
                    // we have to average
                    int newRed = (myColor.getRed() +
realColor.getRed()) / 2;
                    int newGreen = (myColor.getGreen() +
realColor.getGreen()) / 2;
                    int newBlue = (myColor.getBlue() +
realColor.getBlue()) / 2;
```

```
                    myColor = new Color(newRed, newGreen,
newBlue);
                    myImage.setRGB(theRay.getRealX(),
theRay.getRealY(), myColor.getRGB());
               }
          } catch (Exception e) {
               // something went wrong. This pixel must've been
out of bounds. Ignore it.
          }

          if (Timekeeper.timingEnabled)
{ Timekeeper.addTime(Timekeeper.timeCodeForRayRender,
System.currentTimeMillis() - currentTime); }
     }

     /**
      * Returns the image created by this image creator.
      * @return the image
      */
     public BufferedImage getImage() {
          return myImage;
     }

}

** MySQLStorage.java

package tracing.processors;

import globalStuff.MySQL;
import globalStuff.Timekeeper;

import java.awt.Color;
import java.net.InetAddress;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import threeDWorld.World;
import threeDWorld.raysAndVectors.Ray3D;
import tracing.RayProcessor;
```

```
import tracing.local.RayTraceRender;

/**
 * This class can be used to store either the pixels or the
complete rays into a MySQL database. This class independently
handles
 * all of the database connection mumbo-jumbo, and thus you only
have to pass in the information needed to connect.
 *
 * @author ryan
 *
 */
public class MySQLStorage implements RayProcessor {

     private final int maxBufferSize = 10000000;

     private boolean storePixels;

     private Connection theConn;
     private StringBuilder theSQL;
     private Statement theStatement;

     private String localHostname;


     /**
      * Creates a new MySQLStorage object and gets it ready to
go.
      *
      * @param pixels if true, we'll save the pixels. If false,
we'll save the whole ray.
      */
     public MySQLStorage( boolean pixels) {
          storePixels = pixels;

          theSQL = new StringBuilder();
          theSQL.append("INSERT DELAYED INTO " + (storePixels ?
"pixels" : "rays") + "(CoordX, CoordY, Color, CellFrom)
VALUES");
```

```
          try {
               localHostname =
InetAddress.getLocalHost().getHostName();

               // connect to the database...
               theConn = MySQL.getSQLConnection();
               //theConn.setAutoCommit(false);
               theStatement = theConn.createStatement();

               // clear out any data in the database now
               theStatement.execute("delete from pixels");
               theStatement.execute("delete from rays");

          } catch (Exception e) {
               e.printStackTrace();
          }

     }



     @Override
     public void gotRay(Ray3D theRay, Color realColor) {

          try {
               //System.out.println(threadsOut);
               // see how full our SQL statement is...
               if (theSQL.length() > maxBufferSize) {
                    // better send it in and set it up for a new
one...
                    // do substring thing to cut out the last
comma
                    String toSend = theSQL.substring(0,
theSQL.length() - 1);

                    theStatement.execute(toSend);

                    theSQL.delete(0, theSQL.length());
                    theSQL.append("INSERT DELAYED INTO " +
(storePixels ? "pixels" : "rays") + "(CoordX, CoordY, Color,
CellFrom) VALUES");
               }
```

```
                if (storePixels) {
                     theSQL.append("(");
                     theSQL.append(theRay.getRealX());
                     theSQL.append(",");
                     theSQL.append(theRay.getRealY());
                     theSQL.append(",");
                     theSQL.append(realColor.getRGB());
                     theSQL.append(",'");
                     theSQL.append(localHostname);
                     theSQL.append("'),");
                } else {
                     // TODO this stuff for ray storage
                }
           } catch (Exception e) {
                e.printStackTrace();
           }
     }

     /**
      * Commits all the data we've gotten ready to the database.
This method should be called when the tracer completes in order
      * to make sure everything went in...
      */
     public void commit() {
          try {
                String toSend = theSQL.substring(0,
theSQL.length() - 1);
                theStatement.execute(toSend);
                if (!theConn.getAutoCommit()) theConn.commit();
          } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
          }
     }


     /**
      * Test out the processor, tries to store some REAL data
into the database.
      * @param args none
      */
```

```
     public static void main(String[] args) {
          boolean pixelsOnly = true;

          long currentTime = System.currentTimeMillis();

          Timekeeper.init();
          World ourWorld = World.getFireAndIce();

          MySQLStorage myStore = new MySQLStorage(pixelsOnly);

          // World myWorld, double ambLight, int thread, int w,
int h, int aa, int saa, RayProcessor toProces
          RayTraceRender myRender = new RayTraceRender(ourWorld,
0.5, 7, 800, 400, 4, 2, myStore);

          (new Thread(myRender)).start();

          while (!myRender.isFinished()) {
               System.out.println(myRender.getProgress());
               try {
                    Thread.sleep(10000);
               } catch (InterruptedException e) {
                    e.printStackTrace();
               }
          }

          System.out.println("Done in: " +
(System.currentTimeMillis() - currentTime));
          myStore.commit();
          myRender.shutdown();
     }
}

** RayPassback.java

package tracing;

import threeDWorld.raysAndVectors.Ray3D;

/**
 * A simple interface for threads to passback processed rays
 * @author ryan
```

```
 *
 */
public interface RayPassback {
     /**
      * Passing a ray back
      * @param theRay the ray to pass back
      * @param requested if the ray was requested (additional
data optional)
      */
     public void gotRay(Ray3D theRay, boolean requested);
}

** RayProcessor.java

package tracing;

import java.awt.Color;

import threeDWorld.raysAndVectors.Ray3D;

/**
 * This interface should be implemented by classes that process
rays that have been completed.
 *
 * @author ryan
 *
 */
public interface RayProcessor {
     /**
      * Passing a ray back. You may assume that rays that have
been passed back are mapped (i.e., their realX and realY
properties have
      * been set.)
      *
      * @param theRay the ray to pass back
      * @param realColor the real color that should be rendered
for this ray. Do not use the ray's calculateFinalColor method.
      */
     public void gotRay(Ray3D theRay, Color realColor);
}
```