15-462 Project 2: Ray Tracer

Release Date: Thursday, October 22, 2009
Due Date: Tuesday, November 24, 2009

Starter Code: http://www.cs.cmu.edu/~15462/proj/p2.tar.gz

1 Overview

In the first two projects, you learned how to use OpenGL to render simple scenes. For this project, we will be moving away from OpenGL (finally!) and asking you to implement a basic ray tracer that can handle shadows, reflections and refractions. All rendering will be done with software, only using OpenGL to display the final image to the screen (we provide that part for you).

As a warning, this assignment is far more code intensive than either of the previous two. However, it should be more straightforward since there is not any OpenGL involved and since the textbook is a very excellent resource for this topic. Even so, start early. Do not wait until the last week to start. There is a lot of code and debugging can take a fairly long time.

Chapter 10 of the Shirley textbook will be the most useful resource for this assignment, so we strongly recommend that you look at it before starting this assignment. Nearly all topics covered in this handout are also covered in the textbook (though sometimes we present slightly different mathematics). Any references to the Shirley text, unless noted otherwise, are in chapter 10.

2 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at
   /afs/cs.cmu.edu/academic/class/15462-f09/handin/andrewid/p2/.
   All your files should be placed here. Please make sure you have a directory
   and are able to write to it well before the deadline. We are not responsible
   if you wait until 10 minutes before the deadline and run into trouble. Also,
   remember that you must run aklog cs.cmu.edu every time in order to
   read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any tex-
   tures, models, or screenshots that you used or created. Your deliverables
   include:
3. Please do not include:
   • The bin/ folder or any .o or .d files.
   • Executable files

4. Do not add levels of indirection when submitting. For example, your makefile should be at .../andrewid/p2/Makefile, not ...
   /andrewid/p2/myp2/Makefile or ...
   /andrewid/p2/p2.tar.gz. Please use the same arrangement as the handout.

5. We will enter your handin directory, add the include and lib directories, and run make, and it should build correctly. The code must compile and run on the WeH 5336 cluster machines. Be sure to check to make sure you submit all files and that it builds correctly.

   Note that there is a good chance that the 5336 cluster may disappear during the course of this assignment. Course staff will provide you with directions on what to do in the event that this occurs.

3 Required Tasks

A very general overview of the implementation requirements is as follows. Refer to subsequent sections of the handout for more details.

Input: We provide you with a loaded scene and an OpenGL renderer.
Output: Your must produce an imaged created by raytracing the scene.
Requirements:
   • Implement the Raytracer class as defined by the spec to ray trace scenes.
   • Write intersection tests for all types of geometric objects in the scene.
   • Use bounding volumes and at least one spatial data structure to speed up raycasting.
   • Properly handle arbitrary scaling, rotation, and translation of geometries.
   • Implement the basic ray tracing algorithm by sending a ray from the eye through all objects in the scene, up to a recursion depth of at least 4.
   • Add direct illumination and shadows by sending rays to point lights.
   • Add specular reflections by sending reflected rays into the scene.
   • Add refractions by sending transmission rays through dielectric materials.
   • Compute colors as specified in section 10.
• Correctly render all provided scenes.
• Submit a few screenshots of your program’s renderings.
• Use good code style and document well. We will read your code.
• Fill out p2.txt with details on your implementations.

There is also opportunity for up to 10% extra credit by implementing things above the minimum requirements. See section 11 for details.

p2.txt should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you should put it in this file. You should also note which source files you edited.

Examples of things to put in p2.txt:

• Mention parts of the requirements that you did not implement and why.
• Describe any complicated algorithms used or algorithms that are not described in the book/handout.
• Justify any major design decisions you made, such as why you chose a particular algorithm.
• List any extra work you did on top of basic requirements of which the grader should be aware.

4 Starter Code

It is recommended that you begin by first reviewing the starter code as provided. Most is the same as the previous project, but there are some major adjustments this time. The README gives a breakdown of each source file.

In particular, you should have a close look at raytracer.hpp/cpp and all the headers in scene/.

4.1 Building and Running the Code

The code is designed to run and build on the SCS Linux machines. The makefile builds an executable. Consult the README for more detailed build instructions.

We have also provided a Visual Studio 2008 solution, though it will take a bit of effort to get working since the programs have required command-line arguments. More details are in the starter code. If you Windows, your project still must build and run on the 5336 machines, so you will still have to go there to test it before submitting.

There are two ways to run the code. The first is interactively, where you can see an OpenGL rendering of the scene, and navigate the scene by moving the camera with the mouse and keyboard. Then you can raytrace the scene with a keypress.

The second simply loads a scene, raytraces it, and saves the image without opening a window. This is ideal if you want to raytrace on computers without
a display or over ssh. The mode is controlled via command line arguments. See the README for more details.

4.2 What You Need to Implement

The code that you are required to implement is located in raytracer.cpp. The specification for each function is in the source file, and relevant types are generally in the corresponding header file. You may additionally edit any other source files in the handout, though you must keep the basic program behavior the same. To add additional source files, edit the lists in sources.mk.

The starter code provides you with an implementation of Raytracer::raytrace that iterates over each pixel for you. It will invoke Raytracer::trace_pixel, which you should implement to compute the color at each pixel. You do not have to use our implementation of Raytracer::raytrace, as long as it meets the described specification.

4.3 Scene Files

Scenes are described in an XML format. All scenes that you must support are in the scenes/ folder. We encourage you to create your own, as well.

5 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects. See section 11 for more extra credit opportunities.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not. As before, we will be looking for correct usage of the C/C++ language.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes.

6 Scene Layout

In this section we describe how the scene that you must raytrace is represented and all of its components. Consult the corresponding header files (in the scene/ folder) for even more detail.
6.1 Scene

A scene is composed of several parts:

1. Geometries
2. Lights
3. Materials
4. Meses
5. Background color
6. Refractive index of air

Geometries, materials, and meshes are detailed later in the section.

There are two different kinds of lights: an ambient light term and a list of point lights. Ambient light applies to all opaque objects, and point lights are used for computing direct illumination and shadows. Each point light consists of a position, a color, and a set of attenuation factors. These attenuation factors behave the same as in OpenGL. See section 10.2.2 for more details.

The background color is to be used anywhere a ray goes off to infinity. You can replace this with some kind of environment map (e.g. skydome or skybox) for extra credit.

The refractive index of air simply specifies the initial refractive index at the eyes location. Your raytracer may assume that the camera is always in air.

6.2 Materials

Materials define all the properties of a surface, including the ambient, diffuse, specular colors, the texture, and the refractive index of the material. The starter code we give you loads the texture into memory for you, but you have to do the texture sampling yourself. Objects share materials in order to share texture data.

A refractive index of 0 is a special case to mean the object is opaque. The interpretation of the colors depends on if the object is opaque or not. Consult section 10 for details.

Your raytracer may assume that any solid object pieced together from smaller geometries has the same refractive index all over.

6.3 Geometries

There is a base geometry class that contains the position, orientation, scale, and material to be used for the geometry. There are 3 subclasses of geometry.

The transformations should be applied in the following order:

1. Scaling
2. Rotation
3. Translation

The material should be used as the material for the entire geometry (except triangles, see below).
**Sphere**  A perfect sphere. Becomes an ellipsoid if scaled.

**Triangle**  A triangle, with a different material for each vertex. The member `Geometry::material` should be ignored for triangles. See section 10.1.1 for more information.

**Model**  Very similar to what you saw in project 0. Each model contains a pointer to a mesh in the list of meshes. Each mesh is made up of a set of triangles. Models can share the same mesh, but with different transformations and materials.

Note that we suggest using virtual functions to accomplish tasks on different kinds of geometries without the need for casting or switch statements, much as you would in a language like Java. Most of the operations you need to do that depend on the type of geometry can be easily expressed as a function of the base `Geometry` class, such as intersection tests. There is a virtual function, `render`, already there as an example. As with all C++ idioms, you can consult the TAs for help.

### 7 Ray Casting and Intersection Tests

#### 7.1 Ray Casting

The primary ability needed by the ray tracer is the ray cast function, which sends out a given ray $p(t) = e + dt$ into the scene and returns the first object intersected by the ray and the time at which the intersection occurs. This basic function will be used by all other parts of the ray tracer to perform such tasks as casting eye rays, shadow rays, reflected rays, and transmission rays. Note that you also have to deal with bounds on the ray. For example, when sending out eye rays, you should only consider intersections that occur within the viewing frustum.

#### 7.2 Intersection Tests

You must write intersection tests for each of the objects. You may wish to write this code in conjunction with the basic ray tracing algorithm outlined in the next section so that you can test your intersection tests along the way. Consult the Shirley text for sphere-ray and triangle-ray intersection tests.

##### 7.2.1 Model-Ray Intersection

The simplest method for a mesh is to perform an intersect test on every triangle in the mesh and return the one with the minimum time (if such an intersection exists). Of course, this can be prohibitively slow, and so it would be much better to have a sub-linear method that involved some kind of spatial optimization, which we require you to do.
7.3 Instancing

In addition to handling intersections of simple spheres, triangles, and models, you are required to handle arbitrary rotations, translations, and scaling of these geometries. This requires a bit of care, since an ellipsoid-ray intersection test is much harder than a sphere-ray intersection. So we want to perform intersection tests in the object’s local space rather than world space.

You must handle arbitrary affine transformations for all of the geometries in the assignment. For more details on this, you can refer to section 10.8 of the Shirley text.

Note: Transforming normals into world space from local space requires a special normal matrix. We provide some code to help you with this, in `math/matrix.hpp`.

7.4 Optimizations

We require you to implement some optimizations for your raycasting algorithm. In particular, you must implement bounding volumes and a spatial data structure, as described below.

Each of the above describe a narrow-phase intersection test with a ray. However, some of them (particularly the model intersection test) are computationally expensive. We can mitigate this by providing a less accurate but much cheaper broad-phase intersection test. A broad phase test would be something very cheap to compute that may return false positives (never false negatives). If the broad-phase check returns a hit, we run the narrow-phase check to get a definite answer. This way, you can do far fewer narrow-phase checks for each ray.

7.4.1 Bounding Volumes

One simple broad-phase optimization is to surround each object with a bounding box or sphere. You would first check for intersection with the box or sphere, and if this succeeds, you would then perform the narrow-phase check. Ray-box and ray-sphere intersection tests are also in the text. You may use either (or both) in your raytracer.

7.4.2 Spatial Data Structures

A more sophisticated broad-phase is using spatial data structures to achieve sub-linear performance. Similar to many search algorithms, ray casting can be optimized by a “divide and conquer” approach, whereby we create a suitable data structure to store the geometries in the scene and to optimize searching for intersections. Such data structures include octrees, KD trees, BSP trees, and uniform spatial subdivision.

You may implement any of these spatial data structures that you wish. Course staff recommends either octrees or KD trees. You may also find “loose octrees” a useful data structure to implement, instead of a regular octree.
Note that you will get a lot more mileage out of your spatial data structure if you treat each triangle of each mesh as an individual object rather than treating the mesh as a single object.

### 7.4.3 Other Optional Optimizations

Consult section 12 for advice on more general performance optimizations.

## 8 Basic Ray Tracing and Eye Rays

### 8.1 The Ray Tracing Function

Now that you have methods to intersect objects, you can use these methods to begin building the basic ray tracing algorithm. We use our ray casting function to create the basic recursive ray tracing function that, given a ray \( p(t) = e + dt \), returns the color of that ray. This will be used by eye rays, reflected rays, and transmission rays.

Basically, the ray trace function invokes ray cast to determine if an object is hit within the time bounds. If so, it computes the color on the object at that point. Otherwise, it returns the color of the background.

### 8.2 Eye Rays

You can use the ray tracing function to create the basics of your ray tracer. The idea is simple: for every pixel on the screen, you will want to compute the “eye ray” coming out of that pixel and cast it into the scene. If a ray intersects an object, you will want to return the color of the pixel at that point of intersection.

At first, you probably want to make a very simple color computation. For example, return only the diffuse color at that point, or perhaps return the color as computed by Phong illumination. Of course, the actual computation is much more involved (see section 10), but this will allow you to test you eye rays.

## 9 Computing the Recursive Rays

The basic ray tracing algorithm you will have written so far simply returns the color of the first object that it intersects. If your intersection tests are correct, then your code should currently return a scene with no shading and only flat colors.

The next step is to compute the color correctly. However, this requires your ray tracer to be able to correctly send out the remaining 3 types of rays: shadow, reflected, and transmission.

Note that all of these are covered extensively in Shirley, with full derivations for the math involved. You should consult the text for more detail.
9.1 Using the Ray Cast Function

Each of the recursive rays will also use the ray cast functionality. However, unlike eye rays, which are fired from the camera, all of these rays are fired from the point of intersection \( p \) with an object in the scene. This point is given to us by the eye ray’s intersection tests, and so all we need to do is compute the direction of the new recursive ray.

9.1.1 Recursion Depth

Two of these rays will be used in recursive calls to your ray tracing function. However, this leaves open the possibility for infinite recursion, as rays bounce and refract around the scene forever. The ray tracer must be stopped somewhere. You want this to happen once the contribution has become small, so it is not noticeable.

There are a few ways to accomplish this, but one simple way is to simply cap the maximum recursion depth of the ray tracer. Once the max depth is reached, reflection and refraction are not considered. We require your ray tracer to support up to at least a depth of 4, though you may use a more sophisticated method.

9.1.2 Slop Factor

The other major issue with recursive ray tracing is the fact that the ray’s origin is on the surface of a geometry. This means that the intersection test will likely return \( t = 0 \), since the ray is colliding with an object at time 0.

One easy way to correct for this is to introduce a slop factor \( \epsilon > 0 \) as the minimum time bound, to prevent the collision with the ray origin from occurring. \( \epsilon \) should be a very small positive number.

9.2 Shadow Rays

Shadow rays are rays that are cast from the intersection point to a light source to determine the visibility of that light source. When computing direct illumination (see section 10.2.2), one must determine whether a light source is even visible from the intersection point. For this we can simply use our ray cast function to determine if there is another object in the way. If a shadow ray hits any object, then there is no contribution from that light.

Note: This actually breaks down in the face of refraction, since a transparent object doesn’t actually block light rays from reaching a point. It in fact can concentrate them more, resulting in effects like caustics. For this project, you can simply ignore this fact when casting a shadow ray. That is, you may have transparent objects cast complete shadows.
9.3 Reflected Rays

Computing reflected rays is straightforward. We take the incoming ray, bounce it off the normal, and invoke the ray tracer on this reflected ray.

9.4 Transmission Rays

Certain materials allow the transport of light through them. These materials are known as dielectrics and allow for the refraction of light. In our scene, dielectrics are represented using the Material::refraction_index attribute. We use 0 as a special case for opaque objects, and any non-zero value to represent the refraction index of that material.

We utilize Snell’s Law to compute the angle of a refractive ray. Consult the Shirley text for a full derivation.

9.4.1 Tracking the Current Refraction Index

Tracing a scene with dielectrics can be a bit tricky since we must track the current refraction index so we know which values to put into the equation. We assume that the ray trace starts in the scene’s background refraction index, which is given by the Scene class. From there, any time you enter a dielectric, the current index changes. Once you leave, the index goes back to what it was before.

We suggest using a small stack to track this information. You can determine whether you’re entering or leaving a dielectric based on the direction of the normal vector. The normal points out, so if the dot product of the normal and the incoming ray is negative, the ray is entering. Otherwise, the ray is exiting. Be careful, since rays can reflect in between refractions (via total internal reflection, etc.).

One other small issue to consider is that of floating point error. It may be the case that your ray casting, for reasons caused by errors inherent in floating point, missed an entrance/exit from a dielectric. This can cause your stack to become corrupt/invalid. It may be impossible to avoid this, so your best bet is to have code to handle the case where the stack becomes invalid.

10 Computing the Color

Once we have determined when and where an intersection occurs, we must compute the color at that point. Your code must utilize the recursive ray tracing calls as described in section 9.

Note that all equations in the section dealing with colors are done on a component-wise basis. That is, you compute the red, green, and blue individually. The Color3 class overloads the multiplication operation to be component-wise, so you should be able to do this for all 3 components simultaneously.
10.1 Computing the Needed Values

First you must determine a few things at the point \( p \). Of chief interest are the material, the normal \( N \), the texture coordinates \((u, v)\), the viewing ray \( V \), and each light ray \( L \). \( V \) and \( L \) can be easily computed. The others may be computable directly (as in the case of a sphere), but may need to be interpolated.

10.1.1 Interpolation

For triangles, the way to compute the values at a given point is by interpolation. This requires the barycentric coordinates \( \alpha, \beta, \gamma \) computed in the intersection test. To get the value of a vector, color, or float at any given point \( p = \alpha a + \beta b + \gamma c \) where \( a, b, c \) are the vertices of the triangle, we simply interpolate the value. So, for example, to compute the diffuse color \( k_d \) at \( p \), where \( c_i \) is the diffuse color at vertex \( i \), we have

\[
k_d = \alpha c_a + \beta c_b + \gamma c_c.
\]

This computation works identically for all vectors, floats, or colors. So we can interpolate the normal, the texture coordinates, and every value of the material.

Note, the Triangle class has a different material on each vertex, and so you must interpolate all the values of the material to get the correct effect. This includes textures. So you sample each texture using the interpolated texture coordinates, then interpolate these samples. Ignore the Geometry::material member for triangles.

For the Model class, you only need to interpolate normals and texture coordinates, not materials.

10.2 The Three Components

We require that your ray tracer support direct illumination, specular reflection, and refraction. Note that the color computations vary based on the type of object. In our simple model, we support only fully opaque objects and fully transparent objects. In the former, only direct illumination and specular reflection contribute to the color. In the latter, only specular reflection and refraction contribute to the color. The exact ways in which these are computed and combined are described in this section.

10.2.1 Texture Color

First we need the texture color, \( t_p \), at our point. We provide you with textures loaded into arrays. You must first write a texture lookup function based on these. Nearest sampling is sufficient, though linear sampling is preferred. You must also provide the texture coordinates, whose computation is described in 10.1.
10.2.2 Direct Illumination

Your ray tracer should support the basic Phong illumination model for its direct illumination that we have been using for the past two projects. However, we do not need to use the Phong specular component since we have a more accurate specular computation. Therefore, direct illumination consists of ambient and diffuse colors.

Ambient, as always, is the ambient color of light, \( c_a \), multiplied by the material’s ambient color, \( k_a \). The color \( c_a \) is given by \( \text{Scene}::\text{ambient} \cdot \text{light} \).

Diffuse is computed for each light \( i \) in the set of lights \( I \). We multiply the color of the light at the point \( p \), \( c_i \), by the diffuse material \( k_d \) and the dot product of the normal \( N \) and the light vector \( L \). However, we must first use a shadow ray to determine whether the light actually contributes at that point. If the shadow ray hits an object between the point and the light, then there is no contribution from that light.

Lights also have attenuation, so the color \( c_i \) isn’t exactly the color of the light. There are 3 attenuation terms: constant, \( a_c \); linear, \( a_l \); and quadratic, \( a_q \). The color \( c_i \) of the light with color \( c \) at distance \( d \) from the light is

\[ c_i = \frac{c}{a_c + da_l + d^2a_q} \]

This is also one place where the object’s texture comes into play. The entire direct illumination component should be multiplied by the texture at that point, \( t_p \).

So, all together, the color at a point \( p \) is

\[ c_p = t_p \left( c_a k_a + \sum_{i \in I} b_i c_i k_d \max\{N \cdot L, 0\} \right) \]

where \( b_i \) is 0 if the shadow ray from \( p \) to \( i \) intersects an object, 1 otherwise.

10.2.3 Reflection and Refraction

The color contributions from specular reflection and refraction are from recursive calls to the ray trace function. Using the computed reflection/transmission rays, you compute the color of that ray.

In the case of reflection, you must multiply the returned color by the material’s specular color, which is given by \( \text{Material}::\text{specular} \), and also by the texture color \( t_p \).

In the case of refraction, you must use Beer’s Law to compute the color attenuation through the material. For transparent objects, treat the diffuse color as the color of the material. Details can be found in the Shirley text.

10.3 Putting It All Together

For opaque surfaces, we simply sum the two components. You compute the direct illumination and specular terms, then sum them to get the final color. The story is a little more complex for dielectrics.
10.3.1 The Fresnel Effect

For dielectrics, we must consider the Fresnel equations, which describe how much light reflects and how much refracts on a given surface. We will actually use an approximation, called the Schlick approximation.

The Schlick approximation of the Fresnel effect is described on page 214 of Shirley. You should compute the Fresnel coefficient $R$. Given that and the values of specular reflection $c_r$ and refraction $c_f$, the final color is

$$c_p = R c_r + (1 - R) c_f.$$  

Note: if there was no refraction component (due to total internal reflection), then just use $R = 1$.

11 Extra Credit

Any improvements/optimizations to the ray tracer above the minimum requirements can be cause for extra credit. Note that no matter what you do, your raytracer must correctly support all given scenes, exactly as they are. Therefore, you may need to edit some of the application code so your extra stuff doesn’t run unless something additional is defined at the command line.

Some possibilities are:

- Make your ray tracer distributed by adding any number of the following:
  - Anti-aliasing
  - Soft shadows
  - Depth of field
  - Glossy reflection
  - Motion blur (requires an animated scene, see below)

- Add something extra to handle global illumination, such as photon mapping to create caustics.

- Ray trace your own geometry by writing an intersect test for it and building a scene demonstrating it. You do not need to add it to the loader unless you really want to. Instead, you can just hard-code in a few scenes and add some command-line flag or something to use them.

- Change the background of your scene by using an environment map (e.g. skydome or skybox) Again, you can hard code this if you don’t want to edit the loader, and just add some kind of flag.

- Modify a scene to be an animation by updating geometries each frame and implementing some kind of physical simulation. This can be very simple or rather complex.

- Make the scene interactive with the mouse in some way.

- Add more sophisticated materials and effects such as subsurface scattering
12 Words of Advice

12.1 General Advice

Writing a ray tracer is a substantial undertaking, which is why we have allotted you a substantial amount for this project. We are giving you four and a half weeks to complete this assignment, so you will want to take advantage of this. There is a lot of code to write, a lot of math to think through, and a lot of time needed to render your scenes, so you will not want to wait until the deadline is close to start. This project is also more substantial than the previous two assignments in that there are many design decisions to make about your implementation and how you choose to the structure your code can have effects on the efficiency of your final result.

The Shirley text is a very valuable resource for raytracing, and we heavily suggest you start by reading all relevant sections of the text and consulting the text during the course of the assignment.

Rendering scenes with your raytracer is expensive, and can take anywhere from several seconds to hours depending on your implementation and the scene complexity. You will want to set aside at least a day or two just for rendering. Remember that other people will need to use the cluster, so we strongly advise you to not render your scenes on the 5336 machines when other people need to work locally. Please schedule your rendering time carefully and do not monopolize the machines.

12.2 Programming Hints

Since raytracing takes a lot of time, paying attention to writing efficient code is important. Of course, efficiency is most certainly not the most important thing. Correctness, maintainability and good code organization are your most important concerns. However, you should avoid writing obviously unnecessarily slow code. Here are a few hints:

- Avoid recomputing values than can be cached and use many times. For example, you can precompute matrices for geometries’ transformations and inverse transformations once, rather than for every ray cast.

- **Do not** allocate memory in performance sensitive areas. Memory allocation is really, really slow. Do any necessarily allocations in an initialization step, or, even better, use the stack or add members to existing structs or classes to avoid additional allocations at all.

- Avoid trigonometric and square root functions when you can do without them, as they are rather expensive. Note that a lot of vector operations such as normalization, magnitude, and distance use square root, so use squared magnitude and squared distance where possible, and avoid normalizing vectors unnecessarily.
• Avoid virtual functions if a non-virtual function will suffice, since virtual functions are more expensive to call. Note that this does not mean to use switch statements or casting instead of virtual functions, but rather, don’t make a function virtual if you can leave it non-virtual.

Some more general programming hints:

• Orientations are stored as quaternions, with which you may be unfamiliar. Basically, they store a 3D rotation in a compact format. If you’d rather just work with matrices instead, the quaternion class has a function to convert it to a rotation matrix.

• Different parts of the raytracer require a lot of the same functionality, which means you can have a lot of code reuse. We highly suggest that you carefully consider how to organize the code to reduce code repetition. Remember that part of your grade is dependent on code organization.

• Don’t be afraid to edit the starter code we give you to keep it modular and organized. We highly suggest, for example, adding functions and members to the Geometry class (or at least the same source file) for functions closely related to geometries.

• We provide a lot of useful starter code for you, so you don’t have to bother writing a lot of basic routines. Not everyone is taking advantage of all the vector math routines that we provide. Take a look at the headers, for if you need some vector or matrix operation, it is likely already there.

• Any Linux computer on campus may be used to build the project. However, you must still make sure that it compiles on the 5336 cluster machines.

• If you use Windows to implement the project, be sure to test on the Linux machines. The compilers are not quite the same, and certain things that compile with MSVC do not compile or behave differently with GCC.