15-462 Project 4: Raytracing

Release Date: Thursday, October 25, 2012

Due Date: Tuesday, November 15, 2012, 23:59:59

Starter Code: http://www.cs.cmu.edu/afs/cs/academic/class/15462-f12/www/project/p4.tar.gz

1 Overview

In the first three projects, you learned how to use OpenGL to render simple scenes. For this project, we will be moving away from OpenGL (huzzah!) 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 (and we provide that part for you). So you don't need to touch any OpenGL.

As a warning, this assignment is far more code intensive than the previous. 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. This is a 3-weeks assignment after all.

Raytracer chapter 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).

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-f12-users/`*andrewid*`/p4/`. 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 you login 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 textures, models, shaders, or screenshots that you used or created. Your deliverables include:

   - `src/` folder with all `.cpp` and `.hpp` files.
   - Makefile and all `*.mk` files
   - `writeup.txt`
   - Any models/textures/shaders needed to run your code.

3. Please **do not** include:

   - The `bin/` folder or any `.o` or `.d` files.
   - Executable files
   - Any other binary or intermediate files generated in the build process.

   Run `make clean` before submitting. If you were using Visual Studio, be sure to clean the solution before submitting.

4. Do not add levels of indirection when submitting. For example, your makefile should be at `.../`*andrewid*`/p4/Makefile`, **not** `.../`*andrewid*`/p4/myproj/Makefile` or `.../`*andrewid*`/p4/p4.tar.gz`. Please use the same arrangement as the handout.

5. We will enter your handin directory, and run `make clean && make`, and it should build correctly. **The code must compile and run on the GHC 5xxx cluster machines**. Be sure to check to make sure you submit all files and that it builds correctly.

6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to `.../`*andrewid*`/p4-late1/`. These will be locked in turn on each subsequent late day.

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 several scenes make up of various geometries, a loader, and an OpenGL renderer that approximates the lighting of the scene. We also provide code to move the camera and save screenshots.

Output: Your must produce an image created by raytracing the scene by implementing the `Raytracer::raytrace` function to its in-code specification.

Requirements:

- Implement the `Raytracer` class as defined by the spec to raytrace scenes.

- Write intersection tests for all types of geometric objects in the scene.
- 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 3.
- 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 screen shots of your program's renderings, from various camera view-points.
- Fill out `writeup.txt` with details on your implementation.
- Use good code style and document well. We *will* read your code.

At a minimum, you must modify `project.cpp` and `project.hpp` in the folder `raytracer/` and `writeup.txt`, though you may modify or add additional source files. `writeup.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 methods and 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 and any additional ones you have added.

Examples of things to put in `writeup.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 or method.
- List any extra work you did on top of basic requirements of which the grader should be aware.

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

We provide you with reference renderings for each scene we give you. These renderings were taken from the default camera view. As such, at least a few of your screenshots must be taken from different angles than the reference shots.

Note: The writeup and/or handout may be updated during the course of the project. You are responsible for monitoring the bboard, where notice of such updates will be posted. It is suggested you often grab the newest copy of the writeup, as minor writeup fixes will likely not be announced.

4 Starter Code

It is recommended that you begin by first reviewing the starter code as provided. Most of it is the same as the previous project. However, in this project you'll have to look at a lot more of it, particularly the files that define the structure of the scene. The `README` gives a breakdown of each source file.2

4.1 Building and Running the Code

The code is designed to run and build on the Gates 5xxx machines and comes with a makefile. Consult the `README` for more detailed build and running instructions.

We have also provided a Visual Studio 2008 solution (works in Visual Studio 2010 Express too) , though it will take a bit of effort to get working since the programs have required command-line arguments. Note that there are differences in the compilers and graphics card across machines and thus the Windows solution might not be thoroughly tested for all possible machines. More details are in the `README`. If you use Windows, your project still **must** build and run on GHC Linux machines, so you will still have to test it on them before submitting. There are some differences in the compilers, so **code that compiles and works with Visual Studio may not compile or run correctly with GCC**. Make sure you test it well before the deadline. Be sure not to submit Windows binaries, either.

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 an implementation of `Raytracer::raytrace` that iterates over each pixel of the screen. For each pixel, it invokes an empty function, `Raytracer::trace_pixel`, which you should implement to compute the color of that pixel. If you wish, you can change the 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.

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.

We will be looking for correct and clean usage of the C language, such as making sure memory is freed and many other common pitfalls. These can impact your grade. Additionally, we will comment on your C++-specific usage, though we will generally be more lenient with points (at least earlier in the semester). More general style and C-specific style (i.e., rules that apply in both C and C++) will, however, affect your grade.

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 or assign partial credit. Good documentation is a requirement.

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 `src/scene/` folder) for even more detail.

6.1 Scene

A scene is composed of several parts:

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

In the provided scenes, multiple geometries that are part of the same overall solid (e.g., a tetrahedron made from triangles) will all have the same refractive index. However, they can (and will) vary the other material properties across the solid. Again, see section 10 on color computation.

6.3 Geometries

There is a base geometry class that contains the position, orientation, and scale to be used as the transformation for that geometry.

The transformations should be applied in the following order:

1. Scaling
2. Rotation
3. Translation

Each specific kind of geometry is represented as a subclass, in its own header/source file pair.

Sphere A perfect sphere. Becomes an ellipsoid if scaled.

Triangle A triangle, with a different material for each vertex. See section 10.1.2 for more information.

Model Very similar to what you saw in the first project. 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 have 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 already a virtual function, `Geometry::render`, as an example. As with all C++ idioms, you can consult the TAs for help.

6.4 Other Stuff

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 camera's location. Your raytracer may assume that the camera is always in air.

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. Optimization is not required for this assignment, but you may find it useful to consider.[1]

7.3 Instancing

In addition to handling intersections of simple spheres, triangles, and models, you are required to 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. Details are in the textbook, but the basic process is as follows. For each intersection test with a ray and object:

1. Acquire the transformation matrix M and its inverse M^{-1}.
2. Transform the ray by M^{-1} to put it in the object's local space.
3. Do the intersection test in the local space using this new ray. The result is the time of intersection.
4. Use M, the object, and the time of intersection to compute the location of intersection in world space.

[1] Note that the scenes with complex models will take much longer to render than those with just triangles and/or spheres. So don't be surprised by this.

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," which is different than the regular transformation matrix. Specifically, if M is the transformation matrix, $(M^{-1})^T$ is the corresponding normal matrix. We provide some routines to help you with this, in `math/matrix.hpp`. However, when there is scaling, the vectors obtained by multiplying by the normal matrix are no longer unit length. Therefore, you **must** re-normalize after multiplying by the normal matrix if there was scaling.[2]

8 Basic Ray Tracing and Eye Rays

Consult the shirley text for more deatail on these items.

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 some constant color for objects and another for background. Of course, the actual computation is much more involved (see section 10), but this will allow you to test your 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,

[2] For the curious, this is the reason you need `GL_NORMALIZE` enabled with OpenGL when you have scaling. When enabled OpenGL re-normalizes normals after multiplying by the normal matrix.

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 3, 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. ϵ 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 computation or the slop factors, 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, even if it means the color won't be completely correct. Your raytracer should not crash or fail to halt on any inputs.

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 Spheres

The normal and texture coordinates are directly computable on a sphere. The normal is obvious, pointing directly away from the center. For texture coordinates, we wrap a sphere using the latitude and longitude lines as the basis of texture coordinates. That is, one texture component corresponds to the latitude, and the other corresponds to the longitude. You may consult the OpenGL rendering code in the starter code to see exactly how these texture coordinates correspond to position. The Shirley text also has details.

10.1.2 Interpolation

For triangles, the way to compute the values at a given point is by interpolation. This requires the barycentric coordinates α, β, γ 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, which must actually be done with two interpolations. First, the texture coordinates are interpolated. Then, the texture from each material must be sampled at those coordinates. Finally, these texture colors must themselves be interpolated.

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 and a function to return the color of a specific pixel. You must first write a texture lookup function based on these, which should behave similarly to OpenGL texture lookups.[3] Nearest sampling is sufficient, though better sampling can be grounds for extra credit. 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 Blinn-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 `Scene::ambient_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, which we'll call 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

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

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

[3] That is, $(0, 0)$ is the bottom-left corner of the texture, $(1, 1)$ is the top-right.

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 `Material::specular`, and also by the texture color t_p.

In the case of refraction, you may assume the light has no attenuation through the dielectric, and so the color is unchanged.[4]

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

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

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,

[4] For extra credit, have a look at Beer's Law in the text, which discusses how light actually attenuates through dielectrics.

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 some sort of data structure for optimizations. The Shirley text provides some ideas for bounding volumes and spatial data structures. If you are considering spatial data structures, you may wish to look at loose octrees.
- 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.
- Create additional, interesting scenes to show off your new features. These may be hard-coded (activated by flag) if you like.

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. 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 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 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. Note that the since the program can be launched without OpenGL initialization, it is possible to create renderings on a remote machine via ssh.

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 consideration. 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 before you start raytracing, 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 already-allocated 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. Of course, a lot of algorithms require unit-length vectors, so only avoid it when possible.
- 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. Take a look at the headers, for if you need some basic vector or matrix operation, it is likely already there.
- 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.