Parallel Ray Tracer
TJHSST Senior Research Project
Computer Systems Lab 2009-2010

Stuart Maier

October 28, 2009

Abstract

Computer generation of highly realistic images has been a difficult problem. Although there are algorithms that can generate images that look essentially real, they take large amounts of time to render. This project explores ways of distributing that onto multiple computers, in order to speed up the process.

1 Introduction

The generation of images via computer that look realistic is an important topic. There are many different methods that generate these images, but some of them are much more realistic than others. The most realistic systems are all based off of the concept of rays, which bounce off of objects in a scene and change their characteristics as they do so. These methods are the most realistic because they simulate individual photons, the packets of light that generate the images that we actually see.

2 Background

2.1 Graphics

The current standard used in many different programs that render images is the ray tracer. In this method of rendering, a screen is set up, and a camera is placed behind it. Rays are shot from the camera through the screen, and they hit obejcts. When they do so, they change their color depending on the color of the object and the amount of light that is visible from that point. This method generates reasonably realistic images, and they are not that expensive in terms of rendering cost.[6]

Standard ray tracing does have its drawbacks, however. Some effects seen in real images, such as caustics, do not appear in ray tracing. This occurs because the ray tracer assumes that if no light is visible from a point, the point should be dark. However, this is not true in real life. Light can bounce off an object and illuminate a point that would have otherwise been dark.[3]

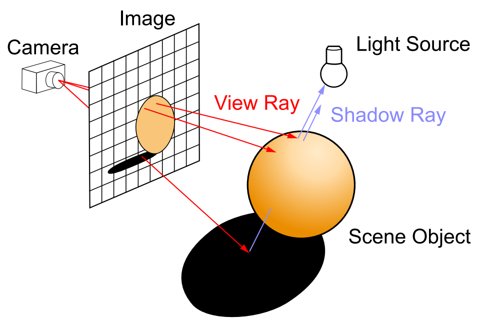

Figure 1: How the ray tracer generates the rays. Image from Wikipedia.

The solution to this is called photon mapping. It actually consists of two parts. In the first, rays are sent from light sources to the scene. They can bounce off of objects, and they create a photon map of different light intensity values at each part of the scene. In the second part, rays are sent from the camera to the scene, like standard ray tracing, but the photon map is used to calculate the light intensity at each point. This method allows for global illumination, and as such, it shows effects that ray tracing does not, such as caustics.[3]

However, photon mapping has some problems of its own. The biggest one is that it is a biased rendering algorithm. This brings us to the difference between unbiased and biased rendering algorthms. If an algorithm is unbiased, then the error bounds between it and the actual image are predictable. The more processing that you put into the render, the lower the error bounds go. However, in a biased algorithm, the error bounds jump around unpredictable. They tend to go down, but it is difficult to accurately estimate them.[2]

To solve this problem, we go to another algorithm, called path tracing. It sounds a lot like ray tracing, but differs from it in subtle ways. Rays are shot from the camera through the screen, like ray tracing, but when they hit objects, they either disappear, bounce, or transmit, depending on the characteristics of the object. This continues until the ray his a light source. If that happened, the pixel that the ray came from acquires the color of the light source. This is a very realistic model of how we actually see images, except that the rays run in reverse.[5]

Another problem that is shared by all of the techniques discussed so far are color problems. Most rendering systems treat light as RGB components. This works well for most scenes, as most of the color spectrum can be well-represented by RGB quantities. However, two materials with very different reflectance spectra, and thus very different colors, can be converted to exactly the same RGB colors. These are called metamers. They cause problems with rendering effects caused by certain parts of the spectrum.[4]

2.2 Parallel

Unlike the graphics section of this project, the parallel section is much simpler. There is simply a smaller ceiling on it. Unlike graphics, which can get more realistic essentially forever, but there is a limit to parallelism. The most famous parallel architecture, and the one that this project will strive to emulate, is called BOINC (Berke-

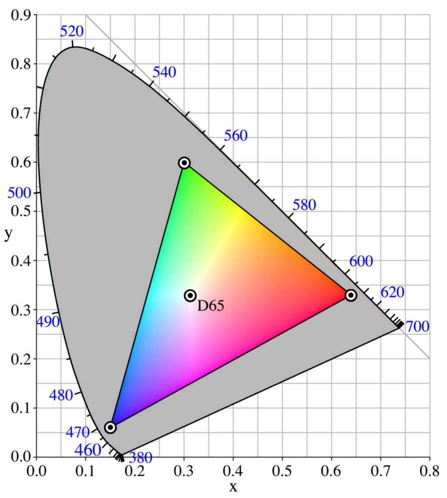

Figure 2: The part of the color space covered by the RGB system.

ley Open Infrastructure for Network Computing). BOINC is a parallel computing platform with a client-server model. The server sends packets of work to independent clients, and the clients return the finished work when they're done.[1]

3 Requirements

This project consists of two parts, both of which are important for the success of it. The first part is the rendering part. The renderer must be highly realistic. However, this part of the project has a very high ceiling. It can extend all the way up to spectral rendering, and even include rendering of images in parts of the spectrum that are not visible. It could show how IR waves travel and change the temperature of a scene, for example. The second part is the parallelism. This has a lower ceiling, because of the constraints placed upon it. I have based it off of the BOINC system, which means that all of the nodes processing are personal computers, and they can never interact with each other, and only interact with the server and the beginning and the end of the render.

4 Procedure

This project has both the graphics part and the parallel part, and they are mostly independent of each other. Improvements can be made in the graphics section without changing the parallelization algorithm, and vice versa. This is only possible because the rendering algorithms are embarassingly parallel, that is, it is trivial it parallelize.

4.1 Graphics

The part of the graphics section that has already been completed is the initial ray tracer. The ray tracer is running, and it displays the standard ray tracing effects. It can model diffuse and specular reflections, mirror reflectances, and refractions. However, it is currently limited to modeling spheres and infinite planes.

Future advances in the graphics section will hopefully include much of what was written above. The renderer can be scaled up to a path tracer that rendes the entire spectrum without RGB components. However, the progress here will depend on the simplicity of rewriting the existing graphics code to accomodate all this.

4.2 Parallel

Reasonably good parallelism has already been achieved in the project. MPI provides a system that automatically starts up programs on slave computers and runs the renderer. However, it is not perfect. Currently, the program can only run inside the SysLab. Technically, it has to be started on a certain computer, but that can be easily changed.

This mechanism is fine for testing purposes, but if there is enough time, I plane to transition to a more flexible parallelization system. Ideally, it would run on the popular BOINC framework.

5 Design

The actual code in this project is currently broken up into five components: Main, Renderer, Object, Vector, and XML.

5.1 Main

Main is, of course, the main component. It handles all MPI and OpenGL work that needs to be done, and it handles the initial startup of the renderer. The, it hands things off to Renderer for processing if it's a slave computer, or checks MPI if it's the master computer.

5.2 Renderer

This section holds all of the information about the objects in a scene. It performs the generation of rays and their conversion to pixel colors. However, it does not calculate intersection code.

5.3 Object

This section calculates information about objects. It's main function is to find intersections between a ray and an object, one of the more time-consuming steps in the rendering process.

5.4 Vector

Vector is a pure calculation library. It provides 3-D vector for the other components to use, and can perform calculation on them. It's purpose is to make calculations in the rest of the program easier, as they often work with 3-D vectors, with spatial ones or color one.

5.5 XML

XML Only runs when the program is first started, but it has an important function. It reads in the scene XML data, processes the contained informaption, and hands it off to Renderer to be converted into a scene.

6 Results

The current project has a ray tracer renderer that can run on many different machines at once. No detailed analysis of the output has yet been performed. However, output images appear to be reasonably realistic. No rigorous analysis has yet been done because the renderer is not yet advanced enough. The best metghod of evaluating the accuracy of a renderer, the Cornell Box image, cannot yet be rendered on the ray tracer. However, getting it to work is an important step in the evolution of the renderer.

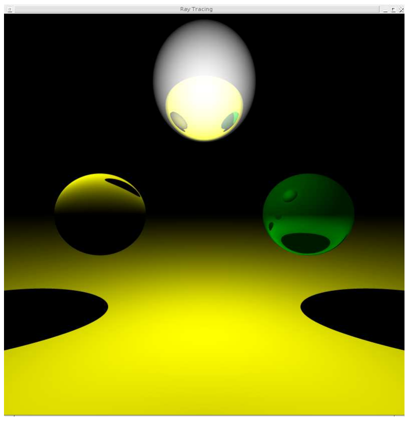

Figure 3: A sample output image of the tracer.

A Code

A.1 main.c

```
#include <GL/glut.h>
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "object.h"
#include "renderer.h"
#include "vector.h"
#include "xml.h"

struct renderer renderer;

void glut_display_func()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRasterPos2i(-1, -1);
    glDrawPixels(renderer.screen_width, renderer.
          screen_height, GL_RGB, GL_FLOAT, renderer.
          pixels);
    glutSwapBuffers();
}

int main(int argc, char** argv)
{
    int mpi_num_computers;
    int mpi_id;
    MPI_Status mpi_status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_num_computers
          );
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_id);

    xml_process_file(&renderer);

    if (mpi_id == 0) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(renderer.screen_width,
              renderer.screen_height);
        glutCreateWindow("Ray␣Tracing");
        glClearColor(1.0, 1.0, 1.0, 0.0);
        glutDisplayFunc(glut_display_func);
    }

    if (mpi_num_computers == 1) {
        printf("Booting␣the␣serial␣renderer.\n");
        renderer_render(&renderer);
    } else {
        printf("Booting␣the␣parallel␣renderer.\n");
        MPI_Barrier(MPI_COMM_WORLD);
        int num_slaves = mpi_num_computers - 1;
        if (renderer.screen_width % num_slaves != 0)
             {
            printf("Invalid␣number␣of␣slaves.\n");
            MPI_Finalize();
            return 1;
        }
        if (mpi_id == 0) {
            int done = 0;
            int ticks = 0;
            while (1) {
                char command;
                MPI_Recv(&command, 1, MPI_CHAR,
                      MPI_ANY_SOURCE, 0,
                      MPI_COMM_WORLD, &mpi_status);
                if (command == 'c') {
                    int current_slave = mpi_status.
                         MPI_SOURCE;
                    GLfloat pixels[renderer.
                         screen_height][renderer.
                         screen_width][3];
                    MPI_Recv(pixels, renderer.
                         screen_height * renderer.
                         screen_width * 3, MPI_FLOAT
                         , current_slave, 0,
                         MPI_COMM_WORLD, &mpi_status
                         );
                    for (int pixel_x = (
                         current_slave - 1) * (
                         renderer.screen_width /
                         num_slaves); pixel_x <
                         current_slave * renderer.
                         screen_width / num_slaves;
                         pixel_x++) {
                        for (int pixel_y = 0;
                             pixel_y < renderer.
                             screen_height; pixel_y
                             ++) {
                            renderer.pixels[pixel_y
                                 * renderer.
                                 screen_width * 3 +
                                pixel_x * 3 + 0] =
                                pixels[pixel_y][
                                pixel_x][0];
                            renderer.pixels[pixel_y
                                * renderer.
                                screen_width * 3 +
                                pixel_x * 3 + 1] =
                                pixels[pixel_y][
                                pixel_x][1];
                            renderer.pixels[pixel_y
                                * renderer.
                                screen_width * 3 +
                                pixel_x * 3 + 2] =
                                pixels[pixel_y][
                                pixel_x][2];
                        }
                    }
                    done++;
                    if (done == mpi_num_computers -
                         1) {
                        break;
                    }
                } else if (command == 't') {
                    ticks++;
                    printf("Progress: %f\r", ticks *
                          100.0 / (renderer.
                         screen_width * renderer.
                         screen_height));
                }
            }
        } else {
            renderer.screen_render_start_x = (mpi_id
                 - 1) * (renderer.screen_width /
                num_slaves);
            renderer.screen_render_width = renderer.
                screen_width / num_slaves;
            renderer_render(&renderer);
            char command = 'c';
            MPI_Send(&command, 1, MPI_CHAR, 0, 0,
                MPI_COMM_WORLD);
            MPI_Send(renderer.pixels, renderer.
                screen_height * renderer.
                screen_width * 3, MPI_FLOAT, 0, 0,
                MPI_COMM_WORLD);
        }
    }

    if (mpi_id == 0) {
        glutMainLoop();
    }

    MPI_Finalize();
    return 0;
}
```

A.2 object.c

```
#include <GL/glut.h>
#include <math.h>
#include "object.h"
#include "vector.h"

int ray_sphere_intersection(struct vector ray_origin
    , struct vector ray_velocity, struct object
    sphere, struct vector *int1, struct vector *
    int2, int *hit_type)
{
    GLdouble det = pow(vector_dot_product(
        ray_velocity, vector_subtract(sphere.origin
        , ray_origin)), 2) - vector_mag_squared(
        vector_subtract(sphere.origin, ray_origin))
         + pow(sphere.radius, 2);
    *hit_type = HIT_OUTSIDE;
    if (det < 0.0) {
        return ZERO_INTERSECTIONS;
    }
    if (det == 0.0) {
        *int1 = vector_scale(ray_velocity,
            vector_dot_product(ray_velocity,
            vector_subtract(sphere.origin,
            ray_origin)));
        return ONE_INTERSECTION;
    }
    GLdouble scale = vector_dot_product(ray_velocity
        , vector_subtract(sphere.origin, ray_origin
        ));
    if (scale < 0.2 && scale > -0.2) {
        return ZERO_INTERSECTIONS;
    }
    if (scale - sqrt(det) < 0.0) {
        if (scale + sqrt(det) < 0.0) {
            return ZERO_INTERSECTIONS;
        }
        *int1 = vector_scale(ray_velocity, scale +
            sqrt(det));
        *hit_type = HIT_INSIDE;
        return ONE_INTERSECTION;
    }
    *int1 = vector_scale(ray_velocity, scale - sqrt(
        det));
    *int2 = vector_scale(ray_velocity, scale + sqrt(
        det));
    return TWO_INTERSECTIONS;
}

int ray_plane_intersection(struct vector ray_origin,
     struct vector ray_velocity, struct object
    plane, struct vector *int1)
{
    GLdouble num = vector_dot_product(
        vector_subtract(plane.origin, ray_origin),
        plane.normal) - vector_dot_product(
        ray_velocity, plane.normal);
    GLdouble den = vector_dot_product(ray_velocity,
        plane.normal);
    if (num == 0.0) {
        return INFINITE_INTERSECTIONS;
    }
    if (den == 0.0) {
        return ZERO_INTERSECTIONS;
    }
    GLdouble t = num / den;
    if (t < 0.2 && t > -0.2) {
        return ZERO_INTERSECTIONS;
    }
    if (t < 0.0) {
        return ZERO_INTERSECTIONS;
    }
    *int1 = vector_scale(ray_velocity, t);
    return ONE_INTERSECTION;
}
```

A.3 object.h

```
#ifndef OBJECT_H
#define OBJECT_H

#include "vector.h"

#define ZERO_INTERSECTIONS 0
#define ONE_INTERSECTION 1
#define TWO_INTERSECTIONS 2
#define INFINITE_INTERSECTIONS 3

#define HIT_OUTSIDE 0
#define HIT_INSIDE 1

enum object_type {
    SPHERE,
    PLANE
};

struct object {
    enum object_type type;
    struct vector origin;
    GLdouble radius;
    struct vector normal;
    struct vector emmitivity;
    struct vector ambient_reflectivity;
    struct vector diffuse_reflectivity;
    struct vector specular_reflectivity;
    GLdouble shininess;
    struct vector reflectivity;
    struct vector refractivity;
    GLdouble index_refraction;
};

// Calculates the intersection between a ray and a
     sphere.
int ray_sphere_intersection(struct vector, struct
     vector, struct object, struct vector*, struct
     vector*, int*);

// Calculates the intersection between a ray and a
     plane.
int ray_plane_intersection(struct vector, struct
     vector, struct object, struct vector*);

#endif
```

A.4 renderer.c

```
#include <math.h>
#include <mpi.h>
#include <stdlib.h>
#include "object.h"
#include "renderer.h"
#include "vector.h"

void renderer_init(struct renderer *renderer, int
     screen_width, int screen_height, struct vector
     camera, int sampling_ratio)
{
    renderer->screen_width = screen_width;
    renderer->screen_height = screen_height;
    renderer->pixels = malloc(sizeof(GLfloat) *
         renderer->screen_height * renderer->
         screen_width * 3);
    for (int x = 0; x < renderer->screen_height; x
         ++) {
        for (int y = 0; y < renderer->screen_width;
             y++) {
            renderer->pixels[x * renderer->
                 screen_width * 3 + y * 3 + 0] =
                 0.0;
            renderer->pixels[x * renderer->
                 screen_width * 3 + y * 3 + 1] =
                 0.0;
            renderer->pixels[x * renderer->
                 screen_width * 3 + y * 3 + 2] =
                 0.0;
        }
    }
    renderer->screen_render_start_x = 0;
    renderer->screen_render_start_y = 0;
    renderer->screen_render_width = renderer->
         screen_width;
    renderer->screen_render_height = renderer->
         screen_height;
    renderer->camera = camera;
    renderer->screen_lowerleft.x = -1.0;
    renderer->screen_lowerleft.y = -1.0;
    renderer->screen_lowerleft.z = 1.0;
    renderer->screen_lowerright.x = 1.0;
    renderer->screen_lowerright.y = -1.0;
    renderer->screen_lowerright.z = 1.0;
    renderer->screen_upperleft.x = -1.0;
    renderer->screen_upperleft.y = 1.0;
    renderer->screen_upperleft.z = 1.0;
    renderer->sampling_ratio = sampling_ratio;
    renderer->object_count = 0;
    renderer->objects = NULL;
}

void renderer_add_object_common(struct renderer *
     renderer, struct vector emmitivity, struct
     vector ambient_reflectivity, struct vector
     diffuse_reflectivity, struct vector
     specular_reflectivity, GLdouble shininess,
     struct vector reflectivity, struct vector
     refractivity, GLdouble index_refraction)
{
    renderer->object_count++;
    renderer->objects = realloc(renderer->objects,
         sizeof(struct object) * renderer->
         object_count);
    renderer->objects[renderer->object_count - 1].
         emmitivity = emmitivity;
    renderer->objects[renderer->object_count - 1].
         ambient_reflectivity = ambient_reflectivity
         ;
    renderer->objects[renderer->object_count - 1].
         diffuse_reflectivity = diffuse_reflectivity
         ;
    renderer->objects[renderer->object_count - 1].
         specular_reflectivity =
         specular_reflectivity;
    renderer->objects[renderer->object_count - 1].
         shininess = shininess;
    renderer->objects[renderer->object_count - 1].
         reflectivity = reflectivity;
    renderer->objects[renderer->object_count - 1].
         refractivity = refractivity;
    renderer->objects[renderer->object_count - 1].
         index_refraction = index_refraction;
}

void renderer_add_sphere(struct renderer *renderer,
     struct vector origin, GLdouble radius, struct
     vector emmitivity, struct vector
     ambient_reflectivity, struct vector
     diffuse_reflectivity, struct vector
     specular_reflectivity, GLdouble shininess,
     struct vector reflectivity, struct vector
     refractivity, GLdouble index_refraction)
{
    renderer_add_object_common(renderer, emmitivity,
          ambient_reflectivity, diffuse_reflectivity
         , specular_reflectivity, shininess,
         reflectivity, refractivity,
         index_refraction);
    renderer->objects[renderer->object_count - 1].
         type = SPHERE;
    renderer->objects[renderer->object_count - 1].
         origin = origin;
    renderer->objects[renderer->object_count - 1].
         radius = radius;
}

void renderer_add_plane(struct renderer *renderer,
     struct vector origin, struct vector normal,
     struct vector emmitivity, struct vector
     ambient_reflectivity, struct vector
     diffuse_reflectivity, struct vector
     specular_reflectivity, GLdouble shininess,
     struct vector reflectivity, struct vector
     refractivity, GLdouble index_refraction)
{
    renderer_add_object_common(renderer, emmitivity,
          ambient_reflectivity, diffuse_reflectivity
         , specular_reflectivity, shininess,
         reflectivity, refractivity,
        index_refraction);
    renderer->objects[renderer->object_count - 1].
        type = PLANE;
    renderer->objects[renderer->object_count - 1].
        origin = origin;
    renderer->objects[renderer->object_count - 1].
        normal = normal;
}

int renderer_find_intersection(struct renderer *
    renderer, struct vector origin, struct vector
    current_vector, struct vector *
    current_intersection_point, int *hit_type)
{
    int current_intersected_object = -1;
    for (int test_object = 0; test_object < renderer
        ->object_count; test_object++) {
        struct vector int1;
        int hit_type_temp = HIT_OUTSIDE;
        int num_intersections;
        if (renderer->objects[test_object].type ==
            SPHERE) {
            struct vector int2;
            num_intersections =
                ray_sphere_intersection(origin,
                current_vector, renderer->objects[
                test_object], &int1, &int2, &
                hit_type_temp);
        } else if (renderer->objects[test_object].
            type == PLANE) {
            num_intersections =
                ray_plane_intersection(origin,
                current_vector, renderer->objects[
                test_object], &int1);
        }
        if (num_intersections > ZERO_INTERSECTIONS
            && (current_intersected_object == -1 ||
             vector_mag(int1) < vector_mag(*
            current_intersection_point))) {
            current_intersected_object = test_object
                ;
            *current_intersection_point = int1;
            *hit_type = hit_type_temp;
        }
    }
    return current_intersected_object;
}

void renderer_illuminate_point(struct renderer *
    renderer, struct vector lit_point, int
    lit_object, struct vector current_vector,
    struct vector normal_ray, struct vector *color,
     struct vector scale)
{
    for (int current_light_object = 0;
        current_light_object < renderer->
        object_count; current_light_object++) {
        if (lit_object == current_light_object) {
            continue;
        }
        if (renderer->objects[current_light_object].
            type == SPHERE) {
            struct vector shadow_ray =
                vector_normalize(vector_subtract(
                renderer->objects[
                current_light_object].origin,
                lit_point));
            struct vector blank;
            int blank2;
            if (renderer_find_intersection(renderer,
                 lit_point, shadow_ray, &blank, &
                blank2) != current_light_object) {
                continue;
            }

            GLdouble diffuse_product =
                vector_dot_product(normal_ray,
                shadow_ray);
            if (diffuse_product > 0.0) {
                struct vector combined_color =
                    vector_multiply(renderer->
                    objects[lit_object].
                    diffuse_reflectivity, renderer
                    ->objects[current_light_object
                    ].emmitivity);
                *color = vector_add(*color,
                    vector_multiply(vector_scale(
                    combined_color, diffuse_product
                    ), scale));
            }

            GLdouble specular_product =
                vector_dot_product(current_vector,
                vector_subtract(shadow_ray,
                vector_scale(normal_ray, 2.0 *
                diffuse_product)));
            if (specular_product > 0.0) {
                specular_product = pow(
                    specular_product, renderer->
                    objects[lit_object].shininess);
                struct vector combined_color =
                    vector_multiply(renderer->
                    objects[lit_object].
                    specular_reflectivity, renderer
                    ->objects[current_light_object
                    ].emmitivity);
                *color = vector_add(*color,
                    vector_multiply(vector_scale(
                    combined_color,
                    specular_product), scale));
            }
        }
    }
}

void renderer_ray(struct renderer *renderer, struct
    vector origin, struct vector current_vector,
    struct vector *color, int depth, struct vector
    scale)
{
    if (depth == 0) {
        return;
    }

    struct vector current_intersection_point;
    int hit_type;
    int current_intersected_object =
        renderer_find_intersection(renderer, origin
        , current_vector, &
        current_intersection_point, &hit_type);
    if (current_intersected_object == -1) {
        return;
    }

    struct vector normal_ray;
    if (renderer->objects[current_intersected_object
        ].type == SPHERE) {
        normal_ray = vector_normalize(
            vector_subtract(
            current_intersection_point, renderer->
            objects[current_intersected_object].
            origin));
    } else if (renderer->objects[
        current_intersected_object].type == PLANE)
        {
        normal_ray = renderer->objects[
            current_intersected_object].normal;
    }
    if (hit_type == HIT_INSIDE) {
        struct vector zero_vector;
        zero_vector.x = 0.0;
        zero_vector.y = 0.0;
        zero_vector.z = 0.0;
        normal_ray = vector_subtract(zero_vector,
            normal_ray);
    }
    current_intersection_point = vector_add(
        current_intersection_point, vector_scale(
        normal_ray, 0.0));

    GLdouble dot_product_view_normal = -
        vector_dot_product(current_vector,
        normal_ray);
    *color = vector_add(*color, vector_multiply(
        vector_scale(renderer->objects[
        current_intersected_object].emmitivity,
        dot_product_view_normal), scale));

    *color = vector_add(*color, vector_multiply(
        renderer->objects[
        current_intersected_object].
        ambient_reflectivity, scale));

    renderer_illuminate_point(renderer,
        current_intersection_point,
        current_intersected_object, current_vector,
         normal_ray, color, scale);

    if (vector_mag(renderer->objects[
        current_intersected_object].reflectivity) >
         0.0) {
        renderer_ray(renderer,
            current_intersection_point,
            vector_subtract(current_vector,
            vector_scale(normal_ray,
            vector_dot_product(current_vector,
            normal_ray) * 2.0)), color, depth - 1,
            vector_multiply(renderer->objects[
            current_intersected_object].
            reflectivity, scale));
    }

    if (vector_mag(renderer->objects[
        current_intersected_object].refractivity) >
         0.0) {
        GLdouble cosine = sqrt(1 - pow(1.0 /
            renderer->objects[
            current_intersected_object].
            index_refraction, 2) * (1 - pow(
            dot_product_view_normal, 2)));
        struct vector scaled_light_vector =
            vector_scale(current_vector, 1.0 /
            renderer->objects[
            current_intersected_object].
            index_refraction);
        struct vector scaled_normal = vector_scale(
            normal_ray, dot_product_view_normal /
            renderer->objects[
            current_intersected_object].
            index_refraction - cosine);
        renderer_ray(renderer,
            current_intersection_point, vector_add(
            scaled_light_vector, scaled_normal),
            color, depth - 1, vector_multiply(
            renderer->objects[
            current_intersected_object].
            refractivity, scale));
    }
}

void renderer_render(struct renderer *renderer)
{
    struct vector pixel_diff_x = vector_scale(
        vector_subtract(renderer->screen_lowerright
        , renderer->screen_lowerleft), 1.0 /
        renderer->screen_width);
    struct vector pixel_diff_y = vector_scale(
        vector_subtract(renderer->screen_upperleft,
         renderer->screen_lowerleft), 1.0 /
        renderer->screen_height);
    struct vector ray_diff_x = vector_scale(
        pixel_diff_x, 1.0 / renderer->
        sampling_ratio);
    struct vector ray_diff_y = vector_scale(
        pixel_diff_y, 1.0 / renderer->
        sampling_ratio);

    for (int pixel_y = renderer->
        screen_render_start_y; pixel_y < (renderer
        ->screen_render_start_y + renderer->
        screen_render_height); pixel_y++) {
        for (int pixel_x = renderer->
            screen_render_start_x; pixel_x < (
            renderer->screen_render_start_x +
            renderer->screen_render_width); pixel_x
            ++) {
            struct vector screen_position = renderer
                ->screen_lowerleft;
            screen_position = vector_add(
                screen_position, vector_scale(
                pixel_diff_y, pixel_y));
            screen_position = vector_add(
                screen_position, vector_scale(
                pixel_diff_x, pixel_x));
            struct vector pixel_color;
            pixel_color.x = 0.0;
            pixel_color.y = 0.0;
            pixel_color.z = 0.0;
            for (int ray_y = 0; ray_y < renderer->
                sampling_ratio; ray_y++) {
                for (int ray_x = 0; ray_x < renderer
                    ->sampling_ratio; ray_x++) {
                    struct vector current_vector =
                        vector_normalize(
                        vector_subtract(
                        screen_position, renderer->
                        camera));
                    struct vector origin;
                    origin.x = 0.0;
                    origin.y = 0.0;
                    origin.z = 0.0;
                    struct vector color;
                    color.x = 0.0;
                    color.y = 0.0;
                    color.z = 0.0;
                    struct vector scale;
                    scale.x = 1.0;
                    scale.y = 1.0;
                    scale.z = 1.0;
                    renderer_ray(renderer, origin,
                        current_vector, &color, 16,
                         scale);
                    pixel_color = vector_add(
                        pixel_color, color);
                    screen_position = vector_add(
                        screen_position, ray_diff_x
                        );
                }
                screen_position = vector_subtract(
                    screen_position, pixel_diff_x);
                screen_position = vector_add(
                    screen_position, ray_diff_y);
            }
            pixel_color = vector_scale(pixel_color,
                1.0 / pow(renderer->sampling_ratio,
                 2));
            renderer->pixels[pixel_y * renderer->
                screen_width * 3 + pixel_x * 3 + 0]
                 = pixel_color.x;
            renderer->pixels[pixel_y * renderer->
                screen_width * 3 + pixel_x * 3 + 1]
                 = pixel_color.y;
            renderer->pixels[pixel_y * renderer->
                screen_width * 3 + pixel_x * 3 + 2]
                 = pixel_color.z;
            char command = 't';
```

```
                MPI_Send(&command, 1, MPI_CHAR, 0, 0,
                    MPI_COMM_WORLD);
            }
        }
}
```

A.5 renderer.h

```
#ifndef RENDERER_H
#define RENDERER_H

#include <GL/glut.h>
#include "object.h"
#include "vector.h"

struct renderer {
    int screen_width;
    int screen_height;
    GLfloat *pixels;
    int screen_render_start_x;
    int screen_render_start_y;
    int screen_render_width;
    int screen_render_height;
    struct vector camera;
    struct vector screen_lowerleft;
    struct vector screen_lowerright;
    struct vector screen_upperleft;
    int sampling_ratio;
    int object_count;
    struct object *objects;
};

// Initializes the rendering structure.
void renderer_init(struct renderer*, int, int,
     struct vector, int);

// Common code for creating an object.
void renderer_add_object_common(struct renderer*,
     struct vector, struct vector, struct vector,
     struct vector, GLdouble, struct vector, struct
     vector, GLdouble);

// Create a sphere.
void renderer_add_sphere(struct renderer*, struct
     vector, GLdouble, struct vector, struct vector,
      struct vector, struct vector, GLdouble, struct
      vector, struct vector, GLdouble);

// Create a plane.
void renderer_add_plane(struct renderer*, struct
     vector, struct vector, struct vector, struct
     vector, struct vector, struct vector, GLdouble,
      struct vector, struct vector, GLdouble);

// Find the intersected object of a ray.
int renderer_find_intersection(struct renderer*,
     struct vector, struct vector, struct vector*,
     int*);

// Illuminate a single point.
void renderer_illuminate_point(struct renderer*,
     struct vector, int, struct vector, struct
     vector, struct vector*, struct vector);

// Trace a ray.
void renderer_ray(struct renderer*, struct vector,
     struct vector, struct vector*, int, struct
     vector);

// Render the scene.
void renderer_render(struct renderer*);

#endif
```

A.6 vector.c

```
#include <GL/glut.h>
#include <math.h>
#include "vector.h"

struct vector vector_add(struct vector v1, struct
     vector v2)
{
    struct vector v3;
    v3.x = v1.x + v2.x;
    v3.y = v1.y + v2.y;
    v3.z = v1.z + v2.z;
    return v3;
}

struct vector vector_subtract(struct vector v1,
     struct vector v2)
{
    struct vector v3;
    v3.x = v1.x - v2.x;
    v3.y = v1.y - v2.y;
    v3.z = v1.z - v2.z;
    return v3;
}

struct vector vector_scale(struct vector v1,
     GLdouble scalar)
{
    struct vector v2;
    v2.x = v1.x * scalar;
    v2.y = v1.y * scalar;
    v2.z = v1.z * scalar;
    return v2;
}

struct vector vector_multiply(struct vector v1,
     struct vector v2)
{
    struct vector v3;
    v3.x = v1.x * v2.x;
    v3.y = v1.y * v2.y;
    v3.z = v1.z * v2.z;
    return v3;
}

GLdouble vector_dot_product(struct vector v1, struct
      vector v2)
{
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}

GLdouble vector_mag(struct vector v1)
{
    return sqrt(vector_dot_product(v1, v1));
}

GLdouble vector_mag_squared(struct vector v1)
{
    return vector_dot_product(v1, v1);
}

struct vector vector_normalize(struct vector v1)
{
    return vector_scale(v1, 1.0 / vector_mag(v1));
}
```

A.7 vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <GL/glut.h>
```

```
// Holds three doubles in a single structure and
     allows for their manipulation.
struct vector {
    GLdouble x;
    GLdouble y;
    GLdouble z;
};

// Vector addition.
struct vector vector_add(struct vector, struct
     vector);

// Vector subtraction.
struct vector vector_subtract(struct vector, struct
     vector);

// Vector scalar multiplication.
struct vector vector_scale(struct vector, GLdouble);

// Vector multiplication by components.
struct vector vector_multiply(struct vector, struct
     vector);

// Dot product of two vectors.
GLdouble vector_dot_product(struct vector, struct
     vector);

// Magnitude of a vector.
GLdouble vector_mag(struct vector);

// Squared magnitude of a vector.
GLdouble vector_mag_squared(struct vector);

// Normalized version of a vector.
struct vector vector_normalize(struct vector);

#endif
```

A.8 xml.c

```
#include <libxml/parser.h>
#include "renderer.h"
#include "xml.h"

void xml_process_file(struct renderer *renderer)
{
    xmlDocPtr doc = xmlParseFile("data.xml");
    xmlNodePtr cur = xmlDocGetRootElement(doc);
    struct vector camera;
    int sampling = atoi(xmlGetProp(cur, "sampling"))
         ;
    int width = atoi(xmlGetProp(cur, "width"));
    int height = atoi(xmlGetProp(cur, "height"));
    cur = cur->xmlChildrenNode;
    while (cur != NULL) {
        if (xmlStrcmp(cur->name, (const xmlChar*)"
             camera") == 0) {
            camera.x = strtof(xmlGetProp(cur, "x"),
                 NULL);
            camera.y = strtof(xmlGetProp(cur, "y"),
                 NULL);
            camera.z = strtof(xmlGetProp(cur, "z"),
                 NULL);
            renderer_init(renderer, width, height,
                 camera, sampling);
        }
        if (xmlStrcmp(cur->name, (const xmlChar*)"
             object") == 0) {
            struct vector origin;
            GLfloat radius;
            struct vector normal;
            struct vector emmitivity;
            struct vector ambient;
            struct vector diffuse;
            struct vector specular;
            GLfloat shininess = strtof(xmlGetProp(
                 cur, "shininess"), NULL);
            struct vector reflectivity;
            struct vector refractivity;
            GLfloat index_refraction = strtof(
                 xmlGetProp(cur, "index_refraction")
                 , NULL);
            xmlNodePtr object_att = cur->
                 xmlChildrenNode;
            while (object_att != NULL) {
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"origin") == 0)
                     {
                    origin.x = strtof(xmlGetProp(
                         object_att, "x"), NULL);
                    origin.y = strtof(xmlGetProp(
                         object_att, "y"), NULL);
                    origin.z = strtof(xmlGetProp(
                         object_att, "z"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"normal") == 0)
                     {
                    normal.x = strtof(xmlGetProp(
                         object_att, "x"), NULL);
                    normal.y = strtof(xmlGetProp(
                         object_att, "y"), NULL);
                    normal.z = strtof(xmlGetProp(
                         object_att, "z"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"emmitivity") ==
                      0) {
                    emmitivity.x = strtof(xmlGetProp
                         (object_att, "r"), NULL);
                    emmitivity.y = strtof(xmlGetProp
                         (object_att, "g"), NULL);
                    emmitivity.z = strtof(xmlGetProp
                         (object_att, "b"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"ambient") == 0)
                      {
                    ambient.x = strtof(xmlGetProp(
                         object_att, "r"), NULL);
                    ambient.y = strtof(xmlGetProp(
                         object_att, "g"), NULL);
                    ambient.z = strtof(xmlGetProp(
                         object_att, "b"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"diffuse") == 0)
                      {
                    diffuse.x = strtof(xmlGetProp(
                         object_att, "r"), NULL);
                    diffuse.y = strtof(xmlGetProp(
                         object_att, "g"), NULL);
                    diffuse.z = strtof(xmlGetProp(
                         object_att, "b"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"specular") ==
                     0) {
                    specular.x = strtof(xmlGetProp(
                         object_att, "r"), NULL);
                    specular.y = strtof(xmlGetProp(
                         object_att, "g"), NULL);
                    specular.z = strtof(xmlGetProp(
                         object_att, "b"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"reflectivity")
                     == 0) {
                    reflectivity.x = strtof(
                         xmlGetProp(object_att, "r")
                         , NULL);
                reflectivity.y = strtof(
                    xmlGetProp(object_att, "g")
                    , NULL);
                reflectivity.z = strtof(
                    xmlGetProp(object_att, "b")
                    , NULL);
            }
            if (xmlStrcmp(object_att->name, (
                const xmlChar*)"refractivity")
                == 0) {
                refractivity.x = strtof(
                    xmlGetProp(object_att, "r")
                    , NULL);
                refractivity.y = strtof(
                    xmlGetProp(object_att, "g")
                    , NULL);
                refractivity.z = strtof(
                    xmlGetProp(object_att, "b")
                    , NULL);
            }
            object_att = object_att->next;
        }
        if (xmlStrcmp(xmlGetProp(cur, "type"), (
            const xmlChar*)"sphere") == 0) {
            radius = strtof(xmlGetProp(cur, "
                radius"), NULL);
            renderer_add_sphere(renderer, origin
                , radius, emmitivity, ambient,
                diffuse, specular, shininess,
                reflectivity, refractivity,
                index_refraction);
        }
        if (xmlStrcmp(xmlGetProp(cur, "type"), (
            const xmlChar*)"plane") == 0) {
            renderer_add_plane(renderer, origin,
                 normal, emmitivity, ambient,
                diffuse, specular, shininess,
                reflectivity, refractivity,
                index_refraction);
        }
    }
    cur = cur->next;
  }
}
```

A.9 xml.h

```
#ifndef XML_H
#define XML_H

#include "renderer.h"

// Processes the input file.
void xml_process_file(struct renderer*);

#endif
```

References

[1] David P. Anderson, "BOINC: A System for Public-Resource Computing and Storage", *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pp. 4-10, 2004.

[2] Keenan Crane, "Bias in Rendering".

[3] Henrik Wann Jensen, "Global Illumination using Photon Maps", *Rendering Techniques*, pp. 21-30, 1996.

[4] Garrett M. Johnson and Mark D. Fairchild, "Full-Spectral Color Calculations in Realistic Image Synthesis", *IEEE Computer Graphics and Applications*, 1999.

[5] James T. Kajiya, "The Rendering Equation", *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986.

[6] Turner Whitted, "An Improved Illumination Model for Shaded Display", *Communications of the ACM 23(6)*, pp. 343-349, 1980.