Parallel Spectral Renderer
TJHSST Senior Research Project
Computer Systems Lab 2009-2010

Stuart Maier

June 16, 2010

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 objects. 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.[5]

Standard ray tracing does have its drawbacks, however. Some effects seen in real images, such as caustics, the results of light refracting through an object, 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.[?]

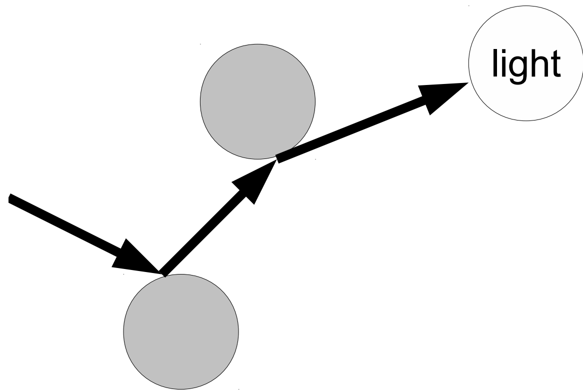

Figure 1: How the path tracer generates the rays.

The solution to this is 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 hits a light source. If that happens, 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. The objects color is based off of its transmittance of light rays as well as its proximity to light sources, creating a more realistic model than ray tracing.[4]

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.[3]

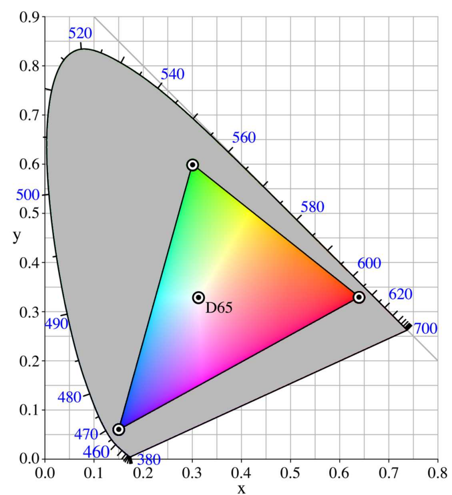

Figure 2: The part of the color space covered by the RGB system. Colors outside this range will render incorrectly on an RGB renderer.

For example, consider two objects with the same RGB color. Their reflectance spectra are also similar, except that one object

has a ”hole” at a certain wavelength. If a beam of monochromatic light at that wavelength is shot at the objects, spectral rendering will correctly render that object as dark, but RGB rendering will not distinguish between the objects.

2.2 Parallel

Unlike the graphics section of this project, the parallel section is much simpler. Unlike graphics, which can get more realistic essentially forever, there is a limit to how powerful parallelization can get. The most famous parallel architecture, and the one that this project will strive to emulate, is called BOINC (Berkeley 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]

The BOINC model is useful for this project because it can be easily adapted to it. A single computer can receive instruction to render a scene, and then it will act as a BOINC server. It will send the scene information to different clients, which can perform their work without any further communications with the server until the end, when the final packet of work is sent. This is the BOINC model, and it easily works for this project.

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. However, a rendering algorithm that would require more communication between the machines would be problematic for the current parallelization model, as the clients are not designed to interact with each other and cannot on a BOINC system. Thankfully, no such algorithm is being considered for this project.

4.1 Graphics

The graphics section is the spectral renderer. It runs, and appears to render images correctly. I see that it shows global illumination effects and well as reflection and refraction. The renderer uses 81 different wavelengths of light in order to accurately model essentially the entire color space without errors. This provides more accuracy than RGB in special situations, such as deep green colors and single wavelengths of light or peaks. However, the renderer itself is currently limited to modeling spheres and places. Additionally, there is some graniness in the images that is caused by not using enough light samples for the images.

4.2 Parallel

Reasonably good parallelism has 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.

5 Design

The actual code in this project is currently broken up into six components: Main, Object, Renderer, Spectrum, 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 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.

5.3 Renderer

This section holds all of the information about the objects in a scene. It performs the generation of rays and their conversion to spectral information. This part is highly important, as all 81 wavelengths in the spectrum data need to be processed in order for the ray to render correctly. This component does not calculate intersection code, however.

5.4 Spectrum

This section contains the spectral data. It contains code to retrieve the Cornell Box colors in their original spectral state. (The colors have to be downsampled from Cornell's 121 wavelengths, but this should make very little difference.) Additionally, it can also correctly convert spectral colors into their

RGB equivalents. This section is currently being used in the XML section in order to retrieve the Cornell Box colors, as well as transforming the output colors into RGB colors.

5.5 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.

5.6 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 spectral 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 method of evaluating the accuracy of a renderer, the Cornell Box image, cannot yet be rendered on the tracer because of the lack of advanced camera features. However, getting it to work is an important step in the evolution of the renderer.

Some testing has been performed, however. I have run parallel tests which show that there is lots of speedup when run on different machines. The tests have been performed on two different renderer complexities and up to 16 client computers. There are some bottlenecks, though. One is that there are network delays. These only come into play at the beginning and the end of the run, but I have noticed that they can take up some time. This is problematic because the data being sent in the end is image data, which takes up large amounts of space. Another is uneven work distribution. Some computers will take longer to finish than others simply because their part of the task is more complex.

One way to improve this would be to run this on a single multi-core machine. This would nearly eliminate network delays, as the only communication would take place inside the computer. As long as all the cores had a single memory source, they would just keep the info in memory and let the master thread compile the image together. Additionally, idle cores could quickly take over data from working cores, assuming the program was changed to let that happen. However, moving to this sort of setup would lock the user into using the one large machine instead of simply being able to use idle workstations.

A Code

A.1 main.c

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

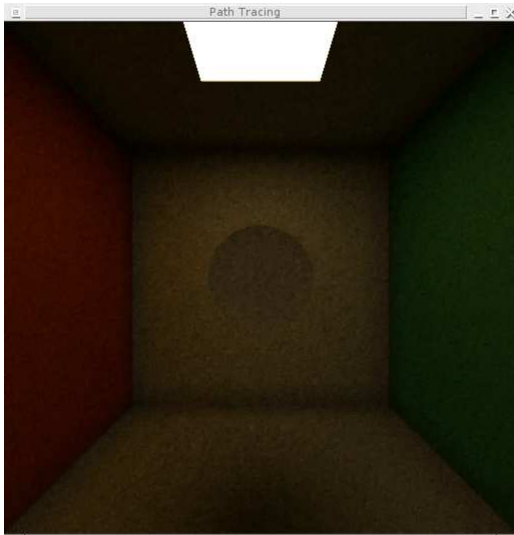

Figure 3: A sample output image of the tracer.

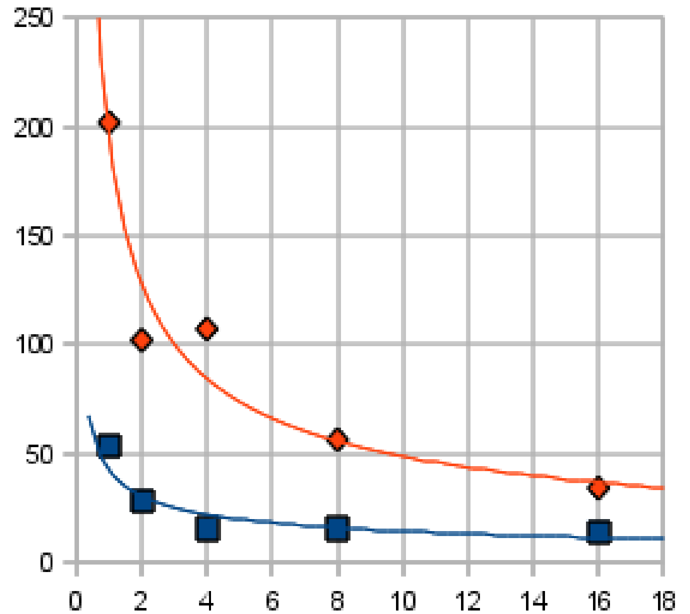

Figure 4: A graph of parallel speedup.

```
#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("Path 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));
    }
}
for (int pixel_y = 0; pixel_y < renderer
    .screen_height; pixel_y++) {
    for (int pixel_x = 0; pixel_x <
        renderer.screen_width; pixel_x
        ++) {
        GLdouble data[5][3];
        int pixel_num = pixel_y *
            renderer.screen_width * 3 +
             pixel_x * 3;
        if (pixel_x == 0) {
            data[0][0] = renderer.pixels
                [pixel_num];
            data[0][1] = renderer.pixels
                [pixel_num + 1];
            data[0][2] = renderer.pixels
                [pixel_num + 2];
        } else {
            data[0][0] = renderer.pixels
                [pixel_num - 3];
            data[0][1] = renderer.pixels
                [pixel_num - 2];
            data[0][2] = renderer.pixels
                [pixel_num - 1];
        }
        if (pixel_x == renderer.
            screen_width - 1) {
            data[1][0] = renderer.pixels
                [pixel_num];
            data[1][1] = renderer.pixels
                [pixel_num + 1];
            data[1][2] = renderer.pixels
```

```
                [pixel_num + 2];
        } else {
            data[1][0] = renderer.pixels
                [pixel_num + 3];
            data[1][1] = renderer.pixels
                [pixel_num + 4];
            data[1][2] = renderer.pixels
                [pixel_num + 5];
        }
        if (pixel_y == 0) {
            data[2][0] = renderer.pixels
                [pixel_num];
            data[2][1] = renderer.pixels
                [pixel_num + 1];
            data[2][2] = renderer.pixels
                [pixel_num + 2];
        } else {
            data[2][0] = renderer.pixels
                [pixel_num - 3 *
                renderer.screen_width];
            data[2][1] = renderer.pixels
                [pixel_num - 3 *
                renderer.screen_width +
                 1];
            data[2][2] = renderer.pixels
                [pixel_num - 3 *
                renderer.screen_width +
                 2];
        }
        if (pixel_y == renderer.
            screen_height - 1) {
            data[3][0] = renderer.pixels
                [pixel_num];
            data[3][1] = renderer.pixels
                [pixel_num + 1];
            data[3][2] = renderer.pixels
                [pixel_num + 2];
        } else {
            data[3][0] = renderer.pixels
                [pixel_num + 3 *
                renderer.screen_width];
            data[3][1] = renderer.pixels
                [pixel_num + 3 *
                renderer.screen_width +
                 1];
            data[3][2] = renderer.pixels
                [pixel_num + 3 *
                renderer.screen_width +
                 2];
        }
        data[4][0] = renderer.pixels[
            pixel_num];
        data[4][1] = renderer.pixels[
            pixel_num + 1];
        data[4][2] = renderer.pixels[
            pixel_num + 2];
        while (1) {
            int done = 1;
            for (int index = 0; index <
                4; index++) {
                if (data[index][0] >
                    data[index + 1][0])
                    {
                    done = 0;
                    GLdouble temp = data
                        [index][0];
                    data[index][0] =
                        data[index +
                        1][0];
                    data[index + 1][0] =
                         temp;
                }
                if (data[index][1] >
                    data[index + 1][1])
                    {
                    done = 0;
                    GLdouble temp = data
```

```
                                [index][1];
                            data[index][1] =
                                data[index +
                                1][1];
                            data[index + 1][1] =
                                temp;
                        }
                        if (data[index][2] >
                            data[index + 1
                            ][2]) {
                            done = 0;
                            GLdouble temp = data
                                [index][2];
                            data[index][2] =
                                data[index +
                                1][2];
                            data[index + 1][2] =
                                temp;
                        }
                    }
                    if (done == 1) {
                        break;
                    }
                }
                renderer.pixels[pixel_num] =
                    data[2][0];
                renderer.pixels[pixel_num + 1] =
                    data[2][1];
                renderer.pixels[pixel_num + 2] =
                    data[2][2];
            }
        }
    } 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 "matrix.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(plane.origin,
        plane.normal) - vector_dot_product(
        ray_origin, 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_add(ray_origin, vector_scale(
        ray_velocity, t));
    struct vector diff = vector_subtract(*int1,
        plane.origin);
    diff.x = fabs(diff.x) - plane.width.x - plane.
        height.x;
    diff.y = fabs(diff.y) - plane.width.y - plane.
        height.y;
    diff.z = fabs(diff.z) - plane.width.z - plane.
        height.z;
    if (diff.x > 0.1 || diff.y > 0.1 || diff.z >
        0.1) {
        return ZERO_INTERSECTIONS;
    }
    return ONE_INTERSECTION;
}
```

A.3 object.h

```
#ifndef OBJECT_H
```

```
#define OBJECT_H

#include "spectrum.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 width;
    struct vector height;
    struct spectrum emmitivity;
    struct spectrum diffuse_reflectivity;
    struct spectrum reflectivity;
    struct spectrum 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 <time.h>
#include "object.h"
#include "renderer.h"
#include "spectrum.h"
#include "vector.h"

void renderer_init(struct renderer *renderer, int
    screen_width, int screen_height, struct vector
    camera, struct vector screen_origin, struct
    vector screen_width_vector, struct vector
    screen_height_vector, int sampling_ratio, int
    depth)
{
    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 = screen_origin;
    renderer->screen_lowerright = vector_add(
        screen_origin, screen_width_vector);
    renderer->screen_upperleft = vector_add(
        screen_origin, screen_height_vector);
    renderer->sampling_ratio = sampling_ratio;
    renderer->depth = depth;
    renderer->object_count = 0;
    renderer->objects = NULL;
    srand(time(NULL));
}

void renderer_add_object_common(struct renderer *
    renderer, struct spectrum emmitivity, struct
    spectrum diffuse_reflectivity, struct spectrum
    reflectivity, struct spectrum 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].
        diffuse_reflectivity = diffuse_reflectivity
        ;
    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
    spectrum emmitivity, struct spectrum
    diffuse_reflectivity, struct spectrum
    reflectivity, struct spectrum refractivity,
    GLdouble index_refraction)
{
    renderer_add_object_common(renderer, emmitivity,
        diffuse_reflectivity, 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 width,
    struct vector height, struct spectrum
    emmitivity, struct spectrum
    diffuse_reflectivity, struct spectrum
    reflectivity, struct spectrum refractivity,
    GLdouble index_refraction)
{
    renderer_add_object_common(renderer, emmitivity,
```

```
            diffuse_reflectivity , 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].
        width = width;
    renderer->objects[renderer->object_count - 1].
        height = height;
    renderer->objects[renderer->object_count - 1].
        normal = vector_normalize(
        vector_cross_product(width, height));
}

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
            && num_intersections <
            INFINITE_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_ray(struct renderer *renderer, struct
     vector origin, struct vector current_vector,
     GLdouble index_refraction, struct spectrum *
     color, int depth, struct spectrum scale)
{
    if (spectrum_mag(scale) < 0.001) {
        return;
    }

    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;
    }
    GLdouble new_index_refraction = renderer->
        objects[current_intersected_object].
        index_refraction;
    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);
        new_index_refraction = 1.0;
    }

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

    struct vector new_ray;
    new_ray.x = 0.0;
    new_ray.y = 0.0;
    new_ray.z = 0.0;
    new_ray = vector_subtract(new_ray, normal_ray);
    while (vector_dot_product(normal_ray, new_ray) <
         0.0) {
        new_ray.z = rand() * 1.0 / RAND_MAX * 2.0 -
            1.0;
        double theta = rand() * 1.0 / RAND_MAX * 2.0
             * 3.141592653;
        double r = sqrt(1.0 - pow(new_ray.z, 2));
        new_ray.x = r * cos(theta);
        new_ray.y = r * sin(theta);
    }
    renderer_ray(renderer,
        current_intersection_point, new_ray,
        index_refraction, color, depth - 1,
        spectrum_multiply(renderer->objects[
        current_intersected_object].
        diffuse_reflectivity, scale));
    renderer_ray(renderer,
        current_intersection_point, vector_subtract
        (current_vector, vector_scale(normal_ray, -
        dot_product_view_normal * 2.0)),
        index_refraction, color, depth - 1,
        spectrum_multiply(renderer->objects[
        current_intersected_object].reflectivity,
        scale));
    GLdouble cosine = sqrt(1 - pow(index_refraction
        / new_index_refraction, 2) * (1 - pow(
        dot_product_view_normal, 2)));
    struct vector scaled_light_vector = vector_scale
        (current_vector, index_refraction /
        new_index_refraction);
    struct vector scaled_normal;
    if (dot_product_view_normal > 0.0) {
        scaled_normal = vector_scale(normal_ray,
            dot_product_view_normal *
            index_refraction / new_index_refraction
```

```
            - cosine);
    } else {
        scaled_normal = vector_scale(normal_ray,
            dot_product_view_normal *
            index_refraction / new_index_refraction
             + cosine);
    }
    renderer_ray(renderer,
        current_intersection_point, vector_add(
        scaled_light_vector, scaled_normal),
        new_index_refraction, color, depth - 1,
        spectrum_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 spectrum color;
                    color = spectrum_black();
                    struct spectrum scale;
                    scale = spectrum_normal();
                    renderer_ray(renderer, renderer
                        ->camera, current_vector,
                        1.0, &color, renderer->
                        depth, scale);
                    pixel_color = vector_add(
                        pixel_color,
                        spectrum_to_RGB(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 "spectrum.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 depth;
    int object_count;
    struct object *objects;
};

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

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

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

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

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

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

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

#endif
```

A.6 spectrum.c

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

GLdouble spectrum_wavelength_rgb_lookup[81][3] = {
    {0.0014, 0.0000, 0.0065}, {0.0022, 0.0001,
         0.0105}, {0.0042, 0.0001, 0.0201},
    {0.0076, 0.0002, 0.0362}, {0.0143, 0.0004,
         0.0679}, {0.0232, 0.0006, 0.1102},
    {0.0435, 0.0012, 0.2074}, {0.0776, 0.0022,
         0.3713}, {0.1344, 0.0040, 0.6456},
    {0.2148, 0.0073, 1.0391}, {0.2839, 0.0116,
         1.3856}, {0.3285, 0.0168, 1.6230},
    {0.3483, 0.0230, 1.7471}, {0.3481, 0.0298,
         1.7826}, {0.3362, 0.0380, 1.7721},
    {0.3187, 0.0480, 1.7441}, {0.2908, 0.0600,
         1.6692}, {0.2511, 0.0739, 1.5281},
    {0.1954, 0.0910, 1.2876}, {0.1421, 0.1126,
         1.0419}, {0.0956, 0.1390, 0.8130},
    {0.0580, 0.1693, 0.6162}, {0.0320, 0.2080,
         0.4652}, {0.0147, 0.2586, 0.3533},
    {0.0049, 0.3230, 0.2720}, {0.0024, 0.4073,
         0.2123}, {0.0093, 0.5030, 0.1582},
    {0.0291, 0.6082, 0.1117}, {0.0633, 0.7100,
         0.0782}, {0.1096, 0.7932, 0.0573},
    {0.1655, 0.8620, 0.0422}, {0.2257, 0.9149,
         0.0298}, {0.2904, 0.9540, 0.0203},
    {0.3597, 0.9803, 0.0134}, {0.4334, 0.9950,
         0.0087}, {0.5121, 1.0000, 0.0057},
    {0.5945, 0.9950, 0.0039}, {0.6784, 0.9786,
         0.0027}, {0.7621, 0.9520, 0.0021},
    {0.8425, 0.9154, 0.0018}, {0.9163, 0.8700,
         0.0017}, {0.9786, 0.8163, 0.0014},
    {1.0263, 0.7570, 0.0011}, {1.0567, 0.6949,
         0.0010}, {1.0622, 0.6310, 0.0008},
    {1.0456, 0.5668, 0.0006}, {1.0026, 0.5030,
         0.0003}, {0.9384, 0.4412, 0.0002},
    {0.8544, 0.3810, 0.0002}, {0.7514, 0.3210,
         0.0001}, {0.6424, 0.2650, 0.0000},
    {0.5419, 0.2170, 0.0000}, {0.4479, 0.1750,
         0.0000}, {0.3608, 0.1382, 0.0000},
    {0.2835, 0.1070, 0.0000}, {0.2187, 0.0816,
         0.0000}, {0.1649, 0.0610, 0.0000},
    {0.1212, 0.0446, 0.0000}, {0.0874, 0.0320,
         0.0000}, {0.0636, 0.0232, 0.0000},
    {0.0468, 0.0170, 0.0000}, {0.0329, 0.0119,
         0.0000}, {0.0227, 0.0082, 0.0000},
    {0.0158, 0.0057, 0.0000}, {0.0114, 0.0041,
         0.0000}, {0.0081, 0.0029, 0.0000},
    {0.0058, 0.0021, 0.0000}, {0.0041, 0.0015,
         0.0000}, {0.0029, 0.0010, 0.0000},
    {0.0020, 0.0007, 0.0000}, {0.0014, 0.0005,
         0.0000}, {0.0010, 0.0004, 0.0000},
    {0.0007, 0.0002, 0.0000}, {0.0005, 0.0002,
         0.0000}, {0.0003, 0.0001, 0.0000},
    {0.0002, 0.0001, 0.0000}, {0.0002, 0.0001,
         0.0000}, {0.0001, 0.0000, 0.0000},
    {0.0001, 0.0000, 0.0000}, {0.0001, 0.0000,
         0.0000}, {0.0000, 0.0000, 0.0000}
};

GLdouble spectrum_cornell_box_white_data[121] = {
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.343,
    0.445,
    0.551,
    0.624,
    0.665,
    0.687,
    0.708,
    0.723,
    0.715,
    0.710,
    0.745,
    0.758,
    0.739,
    0.767,
    0.777,
    0.765,
    0.751,
    0.745,
    0.748,
    0.729,
    0.745,
    0.757,
    0.753,
    0.750,
    0.746,
    0.747,
    0.735,
    0.732,
    0.739,
    0.734,
    0.725,
    0.721,
    0.733,
    0.725,
    0.732,
    0.743,
    0.744,
    0.748,
    0.728,
    0.716,
    0.733,
    0.726,
    0.713,
    0.740,
    0.754,
    0.764,
    0.752,
    0.736,
    0.734,
    0.741,
    0.740,
    0.732,
    0.745,
    0.755,
    0.751,
    0.744,
    0.731,
    0.733,
    0.744,
    0.731,
    0.712,
    0.708,
    0.729,
    0.730,
    0.727,
    0.707,
    0.703,
```

```
    0.729,
    0.750,
    0.760,
    0.751,
    0.739,
    0.724,
    0.730,
    0.740,
    0.737,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000
};

GLdouble spectrum_cornell_box_green_data[121] = {
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.092,
    0.096,
    0.098,
    0.097,
    0.098,
    0.095,
    0.095,
    0.097,
    0.095,
    0.094,
    0.097,
    0.098,
    0.096,
    0.101,
    0.103,
    0.104,
    0.107,
    0.109,
    0.112,
    0.115,
    0.125,
    0.140,
    0.160,
    0.187,
    0.229,
    0.285,
    0.343,
    0.390,
    0.435,
    0.464,
    0.472,
    0.476,
    0.481,
    0.462,
    0.447,
    0.441,
    0.426,
    0.406,
    0.373,
    0.347,
    0.337,
    0.314,
    0.285,
    0.277,
    0.266,
    0.250,
    0.230,
    0.207,
    0.186,
    0.171,
    0.160,
    0.148,
    0.141,
    0.136,
    0.130,
    0.126,
    0.123,
    0.121,
    0.122,
    0.119,
    0.114,
    0.115,
    0.117,
    0.117,
    0.118,
    0.120,
    0.122,
    0.128,
    0.132,
    0.139,
    0.144,
    0.146,
    0.150,
    0.152,
    0.157,
    0.159,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000
};

GLdouble spectrum_cornell_box_red_data[121] = {
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.040,
    0.046,
    0.048,
    0.053,
    0.049,
    0.050,
    0.053,
    0.055,
    0.057,
    0.056,
    0.059,
    0.057,
    0.061,
```

```
0.061,
0.060,
0.062,
0.062,
0.062,
0.061,
0.062,
0.060,
0.059,
0.057,
0.058,
0.058,
0.058,
0.056,
0.055,
0.056,
0.059,
0.057,
0.055,
0.059,
0.059,
0.058,
0.059,
0.061,
0.061,
0.063,
0.063,
0.067,
0.068,
0.072,
0.080,
0.090,
0.099,
0.124,
0.154,
0.192,
0.255,
0.287,
0.349,
0.402,
0.443,
0.487,
0.513,
0.558,
0.584,
0.620,
0.606,
0.609,
0.651,
0.612,
0.610,
0.650,
0.638,
0.627,
0.620,
0.630,
0.628,
0.642,
0.639,
0.657,
0.639,
0.635,
0.642,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
0.000,
```

```
    0.000,
    0.000,
    0.000,
    0.000,
    0.000,
    0.000
};

struct vector spectrum_to_RGB(struct spectrum
     spectrum) {
    struct vector XYZ;
    XYZ.x = 0.0;
    XYZ.y = 0.0;
    XYZ.z = 0.0;
    for (int wave = 0; wave < 81; wave++) {
        XYZ.x += spectrum.waves[wave] *
             spectrum_wavelength_rgb_lookup[wave
             ][0];
        XYZ.y += spectrum.waves[wave] *
             spectrum_wavelength_rgb_lookup[wave
             ][1];
        XYZ.z += spectrum.waves[wave] *
             spectrum_wavelength_rgb_lookup[wave
             ][2];
    }
    XYZ = vector_scale(XYZ, 1.0 / 21.371);
    struct vector RGB;
    RGB.x = 3.2410 * XYZ.x - 1.5374 * XYZ.y - 0.4986
         * XYZ.z;
    RGB.y = -0.9692 * XYZ.x + 1.8760 * XYZ.y +
         0.0416 * XYZ.z;
    RGB.z = 0.0556 * XYZ.x - 0.2040 * XYZ.y + 1.0570
         * XYZ.z;
    if (RGB.x <= 0.00304) {
        RGB.x *= 12.92;
    } else {
        RGB.x = 1.055 * pow(RGB.x, 1.0 / 2.4) -
             0.055;
    }
    if (RGB.y <= 0.00304) {
        RGB.y *= 12.92;
    } else {
        RGB.y = 1.055 * pow(RGB.y, 1.0 / 2.4) -
             0.055;
    }
    if (RGB.z <= 0.00304) {
        RGB.z *= 12.92;
    } else {
        RGB.z = 1.055 * pow(RGB.z, 1.0 / 2.4) -
             0.055;
    }
    return RGB;
}

struct spectrum spectrum_cornell_box_white() {
    struct spectrum spectrum;
    int cornell_point = 0;
    for (int wave = 0; wave < 81; wave++) {
        int inner_loop = cornell_point % 4;
        spectrum.waves[wave] =
             spectrum_cornell_box_white_data[
             cornell_point] + (
             spectrum_cornell_box_white_data[
             cornell_point + 1] -
             spectrum_cornell_box_white_data[
             cornell_point]) * inner_loop / 4;
        if (inner_loop == 3) {
            cornell_point++;
        }
        cornell_point++;
    }
    return spectrum;
}

struct spectrum spectrum_cornell_box_green() {
    struct spectrum spectrum;
    int cornell_point = 0;
```

```
    for (int wave = 0; wave < 81; wave++) {
        int inner_loop = cornell_point % 4;
        spectrum.waves[wave] =
             spectrum_cornell_box_green_data[
             cornell_point] + (
             spectrum_cornell_box_green_data[
             cornell_point + 1] -
             spectrum_cornell_box_green_data[
             cornell_point]) * inner_loop / 4;
        if (inner_loop == 3) {
            cornell_point++;
        }
        cornell_point++;
    }
    return spectrum;
}

struct spectrum spectrum_cornell_box_red() {
    struct spectrum spectrum;
    int cornell_point = 0;
    for (int wave = 0; wave < 81; wave++) {
        int inner_loop = cornell_point % 4;
        spectrum.waves[wave] =
             spectrum_cornell_box_red_data[
             cornell_point] + (
             spectrum_cornell_box_red_data[
             cornell_point + 1] -
             spectrum_cornell_box_red_data[
             cornell_point]) * inner_loop / 4;
        if (inner_loop == 3) {
            cornell_point++;
        }
        cornell_point++;
    }
    return spectrum;
}

struct spectrum spectrum_cornell_box_light() {
    GLdouble spectrum_cornell_box_light_data[121];
    for (int wave = 0; wave < 121; wave++) {
        if (wave < 5) {
            spectrum_cornell_box_light_data[wave] =
                 0.000;
        } else if (wave < 30) {
            spectrum_cornell_box_light_data[wave] =
                 8.000 * (wave - 5) / 25;
        } else if (wave < 55) {
            spectrum_cornell_box_light_data[wave] =
                 (15.600 - 8.000) * (wave - 30) / 25
                  + 8.000;
        } else if (wave < 80) {
            spectrum_cornell_box_light_data[wave] =
                 (18.400 - 15.600) * (wave - 55) /
                 25 + 15.600;
        } else {
            spectrum_cornell_box_light_data[wave] =
                 0.000;
        }
    }
    struct spectrum spectrum;
    int cornell_point = 0;
    for (int wave = 0; wave < 81; wave++) {
        int inner_loop = cornell_point % 4;
        spectrum.waves[wave] =
             spectrum_cornell_box_light_data[
             cornell_point] + (
             spectrum_cornell_box_light_data[
             cornell_point + 1] -
             spectrum_cornell_box_light_data[
             cornell_point]) * inner_loop / 4;
        spectrum.waves[wave] *= 10.0;
        if (inner_loop == 3) {
            cornell_point++;
        }
        cornell_point++;
    }
    return spectrum;
}

struct spectrum spectrum_black() {
    struct spectrum spectrum;
    for (int wave = 0; wave < 81; wave++) {
        spectrum.waves[wave] = 0.0;
    }
    return spectrum;
}

struct spectrum spectrum_normal() {
    struct spectrum spectrum;
    for (int wave = 0; wave < 81; wave++) {
        spectrum.waves[wave] = 1.0;
    }
    return spectrum;
}

struct spectrum spectrum_add(struct spectrum s1,
     struct spectrum s2) {
    struct spectrum s3;
    for (int wave = 0; wave < 81; wave++) {
        s3.waves[wave] = s1.waves[wave] + s2.waves[
             wave];
    }
    return s3;
}

struct spectrum spectrum_multiply(struct spectrum s1
     , struct spectrum s2) {
    struct spectrum s3;
    for (int wave = 0; wave < 81; wave++) {
        s3.waves[wave] = s1.waves[wave] * s2.waves[
             wave];
    }
    return s3;
}

GLdouble spectrum_mag(struct spectrum s1) {
    GLdouble mag = 0.0;
    for (int wave = 0; wave < 81; wave++) {
        mag += s1.waves[wave] * s1.waves[wave];
    }
    return sqrt(mag);
}
```

A.7 spectrum.h

```
#ifndef SPECTRUM_H
#define SPECTRUM_H

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

// Holds info in a range of wavelengths in the
     visible spectrum.
struct spectrum {
    GLdouble waves[81];
};

// Converts a spectrum color to RGB.
struct vector spectrum_to_RGB(struct spectrum);

// Generates Cornell Box white.
struct spectrum spectrum_cornell_box_white();

// Generates Cornell Box green.
struct spectrum spectrum_cornell_box_green();

// Generates Cornell Box red.
struct spectrum spectrum_cornell_box_red();

// Generates Cornell Box light.
struct spectrum spectrum_cornell_box_light();
```

```
// Generates black.
struct spectrum spectrum_black();

// Generates normal.
struct spectrum spectrum_normal();

// Adds two spectrums together.
struct spectrum spectrum_add(struct spectrum, struct
      spectrum);

// Multiplies two spectrums together.
struct spectrum spectrum_multiply(struct spectrum,
     struct spectrum);

// Gets the magnitude of a spectrum.
GLdouble spectrum_mag(struct spectrum);

#endif
```

A.8 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;
}

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

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.9 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);

// Cross product of two vectors.
struct vector vector_cross_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.10 xml.c

```
#include <libxml/parser.h>
```

```c
#include "renderer.h"
#include "spectrum.h"
#include "vector.h"
#include "xml.h"

struct spectrum xml_get_spectrum(xmlChar* string)
{
    struct spectrum spectrum;
    if (xmlStrcmp(string, (const xmlChar*)"white")
         == 0) {
        spectrum = spectrum_cornell_box_white();
    } else if (xmlStrcmp(string, (const xmlChar*)"
         red") == 0) {
        spectrum = spectrum_cornell_box_red();
    } else if (xmlStrcmp(string, (const xmlChar*)"
         green") == 0) {
        spectrum = spectrum_cornell_box_green();
    } else if (xmlStrcmp(string, (const xmlChar*)"
         light") == 0) {
        spectrum = spectrum_cornell_box_light();
    } else if (xmlStrcmp(string, (const xmlChar*)"
         black") == 0) {
        spectrum = spectrum_black();
    } else if (xmlStrcmp(string, (const xmlChar*)"
         normal") == 0) {
        spectrum = spectrum_normal();
    }
    return spectrum;
}

void xml_process_file(struct renderer *renderer)
{
    xmlDocPtr doc = xmlParseFile("data.xml");
    xmlNodePtr cur = xmlDocGetRootElement(doc);
    struct vector camera;
    struct vector screen_origin;
    struct vector screen_width;
    struct vector screen_height;
    int sampling = atoi(xmlGetProp(cur, "sampling"))
         ;
    int width = atoi(xmlGetProp(cur, "width"));
    int height = atoi(xmlGetProp(cur, "height"));
    int depth = atoi(xmlGetProp(cur, "depth"));
    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);
        }
        if (xmlStrcmp(cur->name, (const xmlChar*)"
             screen") == 0) {
            xmlNodePtr object_att = cur->
                 xmlChildrenNode;
            while (object_att != NULL) {
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"origin") == 0)
                     {
                    screen_origin.x = strtof(
                         xmlGetProp(object_att, "x")
                         , NULL);
                    screen_origin.y = strtof(
                         xmlGetProp(object_att, "y")
                         , NULL);
                    screen_origin.z = strtof(
                         xmlGetProp(object_att, "z")
                         , NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"width") == 0) {
                    screen_width.x = strtof(
                         xmlGetProp(object_att, "x")
                         , NULL);
                    screen_width.y = strtof(
                         xmlGetProp(object_att, "y")
                         , NULL);
                    screen_width.z = strtof(
                         xmlGetProp(object_att, "z")
                         , NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"height") == 0)
                     {
                    screen_height.x = strtof(
                         xmlGetProp(object_att, "x")
                         , NULL);
                    screen_height.y = strtof(
                         xmlGetProp(object_att, "y")
                         , NULL);
                    screen_height.z = strtof(
                         xmlGetProp(object_att, "z")
                         , NULL);
                }
                object_att = object_att->next;
            }
            renderer_init(renderer, width, height,
                 camera, screen_origin, screen_width
                 , screen_height, sampling, depth);
        }
        if (xmlStrcmp(cur->name, (const xmlChar*)"
             object") == 0) {
            struct vector origin;
            GLfloat radius;
            struct vector width;
            struct vector height;
            struct spectrum emmitivity;
            struct spectrum diffuse;
            struct spectrum reflectivity;
            struct spectrum 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*)"width") == 0) {
                    width.x = strtof(xmlGetProp(
                         object_att, "x"), NULL);
                    width.y = strtof(xmlGetProp(
                         object_att, "y"), NULL);
                    width.z = strtof(xmlGetProp(
                         object_att, "z"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"height") == 0)
                     {
                    height.x = strtof(xmlGetProp(
                         object_att, "x"), NULL);
                    height.y = strtof(xmlGetProp(
                         object_att, "y"), NULL);
                    height.z = strtof(xmlGetProp(
                         object_att, "z"), NULL);
                }
                if (xmlStrcmp(object_att->name, (
                     const xmlChar*)"emmitivity") ==
                      0) {
                    emmitivity = xml_get_spectrum(
                         xmlGetProp(object_att, "
                         color"));
```

```
                }
                if (xmlStrcmp(object_att->name, (
                    const xmlChar*)"diffuse") == 0)
                    {
                    diffuse = xml_get_spectrum(
                        xmlGetProp(object_att, "
                        color"));
                }
                if (xmlStrcmp(object_att->name, (
                    const xmlChar*)"reflectivity")
                    == 0) {
                    reflectivity = xml_get_spectrum(
                        xmlGetProp(object_att, "
                        color"));
                }
                if (xmlStrcmp(object_att->name, (
                    const xmlChar*)"refractivity")
                    == 0) {
                    refractivity = xml_get_spectrum(
                        xmlGetProp(object_att, "
                        color"));
                }
                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, diffuse,
                    reflectivity, refractivity,
                    index_refraction);
            }
            if (xmlStrcmp(xmlGetProp(cur, "type"), (
                const xmlChar*)"plane") == 0) {
                renderer_add_plane(renderer, origin,
                     width, height, emmitivity,
                    diffuse, reflectivity,
                    refractivity, index_refraction)
                    ;
            }
        }
        cur = cur->next;
    }
}
```

A.11 xml.h

```
#ifndef XML_H
#define XML_H

#include <libxml/parser.h>
#include "renderer.h"
#include "spectrum.h"

// Gets a spectrum from the string.
struct spectrum xml_get_spectrum(xmlChar*);

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

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

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