Anti-Aliased Ray Tracing with POSIX Threads
Michael Tichenor
David Naviaux
Oregon State University
ECE572 05/29/10

Abstract
Ray tracing is a technique developed in the late 1960s to visualize 3D objects with computers. With the advent of GPGPUs and multi-core CPUs, real-time rendering of ray-traced scenes is becoming a very real possibility. This paper discusses in detail the implementation details necessary for creating an anti-aliased ray tracer utilizing the POSIX thread standard. It discusses the challenges of implementing a ray-tracer, the strengths and weaknesses of several multi-threading APIs with regards to an object-oriented ray-tracing implementation, and the specific details of implementing parallel programming with POSIX threads.

KEY WORDS AND PHRASES: ray tracing, anti-aliasing, pthreads, POSIX, multithreading, GPGPU.

1. Introduction

The concepts behind Ray Tracing were originally introduced in 1968 by Arthur Appel in an AFIPS conference regarding techniques for rendering solids. Since then, the technique that he introduced for ray collision detection have been utilized in a variety of different applications, such as Ray Tracing, Radiosity Rendering and Photon Mapping. Creating digital images utilizing these techniques can produce remarkably realistic images, but are extremely computationally expensive. Rendering a single image by rendering each pixel of the image sequentially using any of these techniques can take hours, or days depending on the complexity of the scene.

However, the tasks required to render an image can be done separately, as each pixel can be rendered completely independently of the pixel next to it. This is where having multiple processors can be very handy: instead of having one processor work for ten hours on an image, we can have ten processors work for an hour on an image. With Multi-core CPUs and GPUs, it can take remarkably little time to render such an image. If a computer had sufficient processors, each pixel could be rendered in parallel.

2. Ray Tracing

What we see is a side effect of photon rays causing minute chemical changes in the back of our eyes. Most photons that enter our eyes have bounced from one surface to another, losing a portion of their energy at a given wavelength behind as heat. This is what provides color to what we see. One method of capturing an image is by capturing photons after they are emitted from a light source, bounce off several objects, and eventually find their way into the cameras viewing plane. Simulating billions of photons is rather computationally expensive, and it is far easier to simply approximate what the light values would be on a surface. This is where Ray Tracing comes in.

The concept of ray tracing is simple: shoot a ray into the scene, and have it accumulate light values as it bounces off objects. In practice, it takes quite a large number of formulas, and some decent organizational skills, in order to fully create a working ray-tracer.

2.1 Ray Generation

The first component to constructing a ray-tracing program is the ray, and how to construct the ray. Most graphics libraries provide a means of specifying the camera location and direction. Ray-tracers are no different: the starting position of the ray is set to be the same as the position of the camera. The ray direction going to be in the direction of the camera direction. Given an up-vector as well, indicating the orientation of the camera, we can utilize the cross-product of the up-vector and the reverse of the camera-direction vector to construct a right-vector. In an orthographic projection, we have all the rays project in the same direction but slightly change the origin. In a perspective projection, we have all the rays project in slightly different directions but have the same origin.

Given that:

- \(e\) Represents the camera position.
- \(w\) Represents the opposite of the camera direction.
- \(u\) Represents the camera up direction.
- \(v\) Represents the cross product of \(u\) and \(w\).
- \(d\) Represents the distance from the camera position to the point the camera is looking at.
- \(u\) Represents the offset of each vertical pixel from the camera position.
- \(v\) Represents the offset of each horizontal pixel from the camera position.
- \(i\) Represents the number of the horizontal pixel.
- \(j\) Represents the number of the vertical pixel.
- \(n_x\) Represents the total number of horizontal pixels.
- \(n_y\) Represents the total number of vertical pixels.
- \(l\) Represents the left of the scene.
- \(r\) Represents the right of the scene.
2.2 Ray Intersection

The collision of a ray with a sphere is done by utilizing the properties of the radius of the sphere and its position, and solving the linear equation that results.

Given that:

\[ e \quad \text{Represents the ray's origin.} \]
\[ d \quad \text{Represents the ray's direction.} \]
\[ c \quad \text{Represents the centerpoint of the sphere} \]
\[ R \quad \text{Represents the radius of the sphere} \]
\[ t \quad \text{Represents the distance the ray must travel to intersect with the sphere.} \]

The formula that determines the intersection point is:

\[
( e + td - c ) \cdot ( e + td - c ) - R^2 = 0
\]

Solving this equation for \( t \), using the quadratic equation, will result in the intersection of the ray. If the determinant of the quadratic equation is negative, then the ray will never touch the sphere. If the determinant is 0, then the ray touches the edge of the sphere. If the determinant is positive, the ray passes through the sphere and the smallest positive value of \( t \) represents the nearest point of intersection. Multiplying \( t \) by \( d \) and adding the result to \( e \) results in the actual point of intersection \( p \), and subtracting \( e \) from \( p \) gives the normal of the surface \( n \).

Determining the collision of a ray with a triangle is done using Barycentric coordinates. See *Fundamentals of Computer Graphics* by Shirley et al., 2009 for details regarding the implementation of Cramer's Rule and the computation of \( t, \gamma, \) and \( \beta \). Again, multiplying \( t \) by \( d \) and adding the result to \( e \) results in the actual point of intersection \( p \). We can take the cross product of the lines formed by two edges on the triangle to determine the normal \( n \) (if the normals of each vertex have not been provided with the triangle vertices). If the normals have been provided at each vertex, we can multiply \( \gamma, \beta, \) and \( 1-\gamma-\beta \) by the normal of each vertex and sum the results to calculate the normal at the point \( p \).

2.3 Ray Reflection

The color provided by each light is calculated utilizing the normal that is determined when the ray intersects with the nearest surface. A ray will need to be fired at each light in the scene from the intersection point. If the ray collides with any other surface before the ray can reach the light source, then the point is in shadow from that light source and will not receive any illumination. If the ray can reach the light source, then the object does receive illumination from that light source, which is calculated using the Phong illumination model, using the incoming ray direction as the eye direction, the ray to the light source as the light direction, and the surface normal calculated during the intersection.

The final calculation to be performed is the color determined by the reflection of other surfaces. This is a recursive process as the ray bounces from surface to surface. If the surface has no mirror component to it, then the mirror ray does not need to be generated, otherwise the color of the ray is determined in the same fashion as any other ray and added to the color of the surface at that point. Given the following values:

\[ e \quad \text{Represents the origin of the incoming ray.} \]
\[ d \quad \text{Represents the direction of the incoming ray.} \]
\[ p \quad \text{Represents the point of intersection of the surface.} \]
\[ n \quad \text{Represents the normal of the surface at the point} \]
\[ r \quad \text{Represents the direction of new reflection ray at point} \]

The formula to determine \( r \) is as follows:

\[
r = d - 2n( d \cdot n )
\]

The color added to the surface from the mirror component is calculated using the same algorithm that determines the pixel color, except that instead of utilizing \( e \) and \( d \) for the ray origin and ray direction, instead we use \( p \) as the ray origin and \( r \) as the ray direction. This value is multiplied by the amount of reflected color that the surface accumulates.

2.4 Color Evaluation

The total color of the ray as determined by the Phong illumination model is dependant upon the surface properties that the ray interacts with and the light sources within the scene. Given the following values:

\[ L \quad \text{Represents the final color} \]
With regards to ray-tracing, POSIX is useful in that it recognizes by the OS, which usually does not include multithreading usually will occur on the CPUs that are operating system. As a consequence, all the primary limitation of POSIX is that it relies upon the passing it a pointer to location in program memory. The execution is simply a matter of calling a function and executing this formula for every ray projected into the scene, as described in section 2.1. Since each of these rays can be calculated independently, they make for an excellent candidate for multi-threading. The question is, what do we use for multi-threading?

3. Multi-Threading APIs

There are several APIs designed for multi-threading applications. With the continually falling cost of multi-core CPUs, multi-threading APIs are beginning to become quite popular. There are easily a half dozen major multi-threading APIs: POSIX Threads, Win32, OpenMP, CUDA, Stream, and OpenCL. Keeping all these straight can be rather difficult, so I will cover each of these briefly.

3.1 POSIX Threads

POSIX stands for the "Portable Operating System Interface for Unix", and is a family of related IEEE standards that define the API for the Unix shell. The POSIX Thread standard (IEEE Std 1003.1c-1995) define an API for manipulating program threads. There is also an implementation for pthreads for the Windows Operating System (pthreads-w32). Creating a thread of execution is simply a matter of calling a function and passing it a pointer to location in program memory. The primary limitation of POSIX is that it relies upon the operating system. As a consequence, all the multithreading usually will occur on the CPUs that are recognized by the OS, which usually does not include graphics cards or physics accelerators.

With regards to ray-tracing, POSIX is useful in that it requires very little additional code to implement: Take a segment of the program that can be parallelized, and make a pointer to a function that executes that piece of code, then have a POSIX thread execute that segment of code. The primary disadvantage to this is that it does not allow for execution of code on a graphics card or physics accelerator.

3.2 Win32 Threads

The Win32 Multi-threading API, created by Microsoft™, has much of the same functionality as the POSIX Threads API. New threads of execution can be created using a single function call and passing it a pointer to a location in program memory. Again, Win32 Threads are limited by the fact that threads of execution will only be executed on CPUs that are recognized by the OS.

The advantages and disadvantages to Win32 threads for ray-tracing are exactly the same, though they have the additional limitation of only being able to be executed in the Windows operating system.

3.3 OpenMP

OpenMP is a different kind of Multi-threading API. Instead of explicitly creating threads and handling the thread synchronization manually within the program, the programmer provides compiler hints using pragma commands to tell the compiler that a given section of code can be parallelized. The compiler is then given the task of performing code analysis and restructuring the code in a parallel fashion. OpenMP is one of the easiest way to implement parallel programming, but does not allow the programmer the explicit control over how multi-threading is implemented. At this time, OpenMP does not support GPGPUs, however there have been papers released over the past year regarding creating framework for OpenMP to GPGPU compilation.

OpenMP is even better for ray-tracing than POSIX or W32 Threads, as it does not require any changes to be made to the existing code, other than adding a few pragma comments. The primary disadvantage to this is that OpenMP must be supported by the compiler.

3.4 CUDA

CUDA (Compute Unified Device Architecture) is nVidia's parallel computing architecture that allows GPGPUs. CUDA code is stored in a string that is compiled and executed dynamically within a program, much the same way the Fragment Shaders are. These compiled CUDA programs are then provided inputs and passed off to the GPU to be executed, usually thousands at a time. After the CUDA programs have completed execution, the results can be compiled and utilized within the main program. This allows for incredibly massive
parallelization to be performed. The two primary problems with CUDA are that it can only be executed on nVidia GPUs, and that existing code has to be manually ported or re-written entirely in CUDA.

The advantage to ray-tracing in CUDA is that each ray can be independantly executed on a GPU. Considering that many GPUs have large numbers of processing units, this can mean enormous gains in computational power and greatly reduced rendering times. The primary disadvantage to this is that the code must be re-written from scratch in CUDA, and can only be executed on vendor specific hardware.

3.5 Stream

Stream is ATI's response to the CUDA initiative. It works in exactly the same fashion, and with similar syntax to that of CUDA. Stream also suffers from the same kind of limitations, in that it can only be executed on ATI GPUs, and that existing code has to be manually ported or re-written entirely. This has the same advantages and disadvantages with regards to ray-tracing that CUDA does.

3.6 OpenCL

OpenCL (Open Computing Language) is a framework for execution of program kernals on heterogeneous platforms, including CPUs, GPUs, and other processors. It includes a language for writing these program kernals, and the APIs necessary to define and control the platforms. The OpenCL initiative was started by Apple, Inc. and is currently being developed by Khronos Group, the developers behind OpenGL (Open Graphics Language). OpenCL works much the same way as CUDA, except that the kernal programs can be executed on either GPUs or CPUs.

Much like CUDA or Stream, ray-tracers must be rewritten from scratch in the OpenCL language before they can be parallelized using the OpenCL API. Unlike either CUDA or Stream, however, OpenCL is not vendor specific, and can be executed on either a GPU or a CPU.

3.7 DirectCompute

Microsoft has also developed an API for Windows Vista and Windows 7 called DirectCompute, which allows for programming of GPGPUs (General Purpose computation on Graphics Processing Units) as well as CPUs. It works similar to OpenCL, but is limited to only working on Windows Vista or Windows 7.

4. Multi-Threaded Anti-Aliasing

When a scene is rendered using ray-tracing with only one ray generated per pixel, there is a certain very distinct jaggedness to the edges of objects such as spheres, triangles, or shadows. This has to do with the fact that each ray calculates the color value for an exact point, and that color value is then distributed across the entire pixel. This problem is known as aliasing, and the solution for it is, quite logically, called anti-aliasing.

One method of anti-aliasing is done by rendering the scene at a much higher resolution, with far more pixels, then shrinking it. The smaller image pixels are composed of fractions of the larger images pixels, averaged together in a process called bilinear filtering.

Another method of anti-aliasing is done by shooting multiple rays per pixel into the scene at slightly different offsets, and then averaging the results together for a final composite picture result.

We chose the latter method for our anti-aliasing implementation, as it was very easy to implement with our existing ray-tracing implementation. This was accomplished by creating four additional frame buffers (image that would be rendered to screen) and averaging the results of each pixel of each of the four frame buffers into a single pixel of the original frame buffer. This is commonly referred to as 2x Anti-Aliasing.

When performing multi-threading during the rendering process, there are two main methods. The first method is to assign pixels to each thread to be rendered, usually by dividing the image into subsections, where each thread would be responsible for rendering the pixels in a specific subsection. This is best suited to anti-aliasing methods where the image is rendered at larger resolutions, or when it is possible to utilize a large number of threads or GPGPU programs during the rendering process.

The second method of multi-threading during the rendering process is specific to anti-aliasing. Each frame buffer that is rendered during the anti-aliasing process can be assigned to be rendered by a separate thread, or GPGPU program. This method works best when there is a limited number of threads that can be executing in tandem, especially when the number of threads or processors is equal to the number of frame buffers being rendered.

4.1 Multi-Threading API Decision

Due to the limitations of our system hardware, compiler, and existing code, we were rather limited in our selection of which multi-threading API we chose to utilize for our implementation.

The graphics card in the test machine was an ATI card, so we could not use CUDA as we had originally planned. The machine was operating Windows XP, so we couldn't
utilize DirectCompute. Because the ray-tracer had already been partially written, it would have been difficult and time consuming to rewrite the entire thing for Stream or OpenCL or either of the two previously mentioned APIs.

In addition, the compiler we were utilizing was Visual Studio 2008 Express Edition, which doesn't support OpenMP. This left us with either W32 or POSIX threads. Since the ray tracer had already been implemented in OpenGL for compatibility issues, we opted for to do the same for our multi-threading capability, and chose to go with POSIX.

4.2 Anti-Aliasing Implementation

Since we chose to go with the multiple frame-buffer approach for our anti-aliasing, we also chose the second method of multi-threading, which has a thread created for each frame-buffer that is being rendered. Recall that in section 2.1 we used the formulas:

\[ v = l + (r - l)(i + 0.5) / n_x \]
\[ u = b + (t - b)(j + 0.5) / n_y \]

to determine the ray offset. For the multiple frame-buffer approach, we substituted the 0.5, which represents the center of each pixel, with the following formulas:

\[ v_1 = l + (r - l)(i + 0.25) / n_x \]
\[ u_1 = b + (t - b)(j + 0.25) / n_y \]
\[ v_2 = l + (r - l)(i + 0.25) / n_x \]
\[ u_2 = b + (t - b)(j + 0.75) / n_y \]
\[ v_3 = l + (r - l)(i + 0.75) / n_x \]
\[ u_3 = b + (t - b)(j + 0.75) / n_y \]
\[ v_4 = l + (r - l)(i + 0.75) / n_x \]
\[ u_4 = b + (t - b)(j + 0.25) / n_y \]

to determine the ray offset. The 0.25 and 0.75 represent the bottom or left and top or right corners of the pixel, respectively. We then constructed an array of ray values for each frame-buffer using the above formulas.

POSIX is designed so that a new thread is given a memory location within the program to begin execution. The syntax utilized is as follows:

```c
void *osuRaytraceSceneThread(void *t) {
    // TODO: Insert Code Here.
    pthread_exit( (void *) t );
}
```

Since this function is not a real function, but rather a location in memory, the function name must be preceded by a *, indicating that the name of the function is simply a location in memory. The "void *t" is a second pointer in memory to values that will be utilized by the function.

This can be a null value, a single data type, or a struct containing multiple data types. In our case, it was an integer value indicating the thread id.

Inside the function we simply created a new function that consisted of a nested pair of loops, the outer loop representing the horizontal pixels and the inner loop representing the vertical pixels. A switch statement inside both loops choose the frame buffer and ray values depending upon which thread of execution was being executed, and then perform the ray tracing algorithm for that pixel using that framebuffer and ray values.

The creation of the threads was performed immediately after the scene construction in the main portion of the program. Each thread was passed an integer value representing its thread id, which was used inside the function definition that was at the memory location the threads were told to begin execution at. The syntax for this was:

```c
for( t = 0; t < NUM_THREADS; t++ )
{
    rc = pthread_create(&thread[t],
                        osuRaytraceSceneThread,
                        (void *) t);
}
```

The main thread of execution then utilized a small loop which waited for each thread of execution to finish and rejoin the main program. The syntax for this was:

```c
for( t = 0; t < NUM_THREADS; t++ )
{
    rc = pthread_join(thread[t],
                      &status);
}
```

As an aside, it should be noted that the return value of these should be checked to verify that the threads have been successfully created or have successfully rejoined the parent program.

5. Multi-Thread Evaluation

In order to evaluate the effectiveness of multithreading within our program, a few simple timing statements were added to the program utilizing the standard ctime libraries, and outputting the result to console at each stage of execution. We then ran the program at various resolutions: 100x100, 200x200, 300x300, 400x400, and 500x500. The results were recorded for both a non-multi-threaded implementation of anti-aliasing, and the multi-threaded implementation of anti-aliasing described above.

5.1 Test Hardware
The machine that this was evaluated upon was a custom built machine running an Intel Core2 Duo 6300 @ 1.86 GHz with 2.00 GB RAM running Windows XP Service Pack 3 with an ATI Radeon 4890 GPU with 1.00 GB of RAM.

Since the POSIX threads are not designed to take advantage of the GPU, the only multi-threading speed-up should occur because of the multiple cores within the CPU. The Intel Core2 Duo has 2 cores, meaning that the program should theoretically finish in exactly half the time, assuming no time delay for context switching.

### 5.2 Test Times

We first rendered the anti-aliased scene without multi-threading. In order, we rendered the scene at resolutions of 100, 200, 300, 400, and 500 pixels to a side. The results are listed below, in seconds.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Setup Time</th>
<th>Raytrace Time</th>
<th>Render Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.2650</td>
<td>0.4380</td>
<td>0.0150</td>
<td>0.7180</td>
</tr>
<tr>
<td>200</td>
<td>0.3120</td>
<td>1.7970</td>
<td>0.0310</td>
<td>2.1400</td>
</tr>
<tr>
<td>300</td>
<td>0.3120</td>
<td>4.1900</td>
<td>0.0630</td>
<td>4.4840</td>
</tr>
<tr>
<td>400</td>
<td>0.3430</td>
<td>7.2660</td>
<td>0.0780</td>
<td>7.7030</td>
</tr>
<tr>
<td>500</td>
<td>0.3590</td>
<td>11.2810</td>
<td>0.1100</td>
<td>11.7500</td>
</tr>
</tbody>
</table>

As can be seen, the scene setup time was almost constant, which makes sense as that part consisted primarily of creating the initial ray values and setting up the scene. The render time was also fairly small, considering that all that was being done was copying values from the frame buffer to the graphics card for display. The ray tracing time far outweighs the setup and rendering time, as expected.

We then rendered the anti-aliased scene using multi-threading. Again, we rendered the scene at resolutions of 100, 200, 300, 400, and 500 pixels to a side. The results are listed below, in seconds.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Setup Time</th>
<th>Raytrace Time</th>
<th>Render Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.2810</td>
<td>0.2190</td>
<td>0.0150</td>
<td>0.5150</td>
</tr>
<tr>
<td>200</td>
<td>0.2960</td>
<td>0.9220</td>
<td>0.0320</td>
<td>1.2500</td>
</tr>
<tr>
<td>300</td>
<td>0.3120</td>
<td>2.0000</td>
<td>0.0470</td>
<td>2.3590</td>
</tr>
<tr>
<td>400</td>
<td>0.3280</td>
<td>3.5620</td>
<td>0.0940</td>
<td>3.9840</td>
</tr>
<tr>
<td>500</td>
<td>0.3590</td>
<td>5.5160</td>
<td>0.1250</td>
<td>6.0000</td>
</tr>
</tbody>
</table>

### 5.3 Time Analysis

The values for Setup Time of the scene were practically identical for both the multi-threaded and non-multi-threaded versions of the application, as expected. This portion of the application was not parallelized, so no speedup was expected. The same applies to the render times, which were not parallelized either. The speedup for the raytracing time, however, is quite clearly apparent. The time required was very close to half the time taken for rendering. In some cases it took slightly less than half the time to raytrace the same sized image, and in others it was slightly more than half the time. Considering that there are several background processes running on the test machine, it is entirely possible that some of them may have skewed the test results a very small degree.

According to Amdahl's Law, which states that the overall speed-up of applying the improvement is

\[
1 / \left( \left( 1 - P \right) + \left( P / S \right) \right)
\]

Where \( P \) represents fraction of the program that is parallelizable, and \( S \) represents the speed at which that fraction will be running. For example, with the 500x500 rendering, the total time was 11.750 seconds. The time that was rendered that could be parallelized was 11.281 seconds. The speed at which the parallelized code would be running on a dual-core CPU would be 2.0 (twice as fast. Thus we can plug in the following values for \( S \) and \( P \):

\[
P = 11.281 / 11.750 = -0.96 \\
S = 2.0
\]

Thus Amdahl's Law states that the speed-up should be:

\[
1 / \left( \left( 1 - 0.96 \right) + \left( 0.96 / 2.0 \right) \right) = 1 / 0.52 = -1.923
\]

The total time of execution for our multi-threaded application with the same resolution was 6.0 seconds. If we divide the execution time of the non-multi-threaded by the speed-up, the result should be close to the time of execution of our multi-threaded program.

\[
11.750 / 1.923 = -6.11
\]

Since the estimated execution time of 6.11 was close to the actual execution time of 6.0, it is fairly safe to assume that Amdahl's Law held true in this instance. It would be unreasonable to expect that the results would be perfectly correct on a machine that is running multiple background processes which could throw either the multi-threaded or non-multi-threaded results off.

### 6 Conclusions

The implementation of multi-threading for anti-aliasing turned out to be remarkably easy to implement, partially because we chose not to re-write the ray-tracing algorithms to utilize GPGPUs, and secondarily because the implementation of anti-aliasing that we chose lent itself well to the POSIX multithreading implementation. The results of implementing multi-threading were almost exactly what were expected: almost double the speed of a non-multi-threaded implementation.
Figure 1: A modified cornell box rendered with 2x Anti-aliasing.

Figure 2: A modified cornell box rendered without anti-aliasing. Notice that the edges of the shadows and spheres are jagged compared to the anti-aliased image.
/**** FILE INFORMATION ****
FILENAME: raytracer.cpp
AUTHOR: Michael Tichenor
DATE: 5/16/2010

#include "osuGraphics.h"
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>

#define IMAGE_HEIGHT 100
#define IMAGE_WIDTH 100
#define MAX_LIGHTS 8
#define MAX_OBJECTS 32
#define ANTI_ALIASING 1
#define MULTI_THREADING 0
#define NUM_THREADS 4
#define DEBUG 0

struct color
{
    double r, g, b;
};

struct trace
{
    double direction[3];
    double origin[3];
};

struct material
{
    struct color kd;
    struct color ks;
    double p;
    struct color ka;
    struct color km;
};

struct pointlight
{
    double r, g, b;
    double point[3];
};

struct hitrecord
{
    trace ray;
    double t;
double point[3];
double normal[3];
struct color kd;
struct color ks;
double p;
struct color ka;
struct color km;
}

struct box
{
    double min[3];
    double max[3];
};

///////////////
// PROTOTYPES //
///////////////

void osuOrtho(double left, double right, double bottom, double top, double nearp, double farp);
void osuPerspective(double fovy, double nearp, double farp);
void osuLookAt(double from[3], double at[3], double up[3]);
void osuCrossProduct(double b[3], double c[3], double a[3]);
double osuDotProduct(double a[3], double b[3]);
double osuLength(double a[3]);
void osuNormalize(double a[3]);
double max(double, double);
double min(double, double);
void osuRaytraceScene(struct color *pixel, struct trace ray, class scene myscene);
void *osuRaytraceSceneThread(void *t);

///////////////
// CLASSES //
///////////////

// CLASS: surface
// PURPOSE: parent class for any 3D surface.
class surface
{
public:
    virtual bool hit(trace ray, double t0, double t1, struct hitrecord *rec)
    {
        return false;
    }
    virtual box bounding_box()
    {
        struct box retval;
        for(int i = 0; i < 3; i++)
        {
            retval.min[i] = 0.0;
            retval.max[i] = 0.0;
        }
        return retval;
    }
private:
};

// CLASS: triangle
// PURPOSE: a type of 3D surface.
class triangle : public surface
{
public:
    triangle(double p1[3], double p2[3], double p3[3], struct material *mat)
    {
        for(int i = 0; i < 3; i++)
        {
            a[i] = p1[i];
            b[i] = p2[i];
            c[i] = p3[i];
        }
        this->mat = mat;
    }
    virtual bool hit(trace ray, double t0, double t1, struct hitrecord *rec)
// Cramer's rule!

int x = 0;
int y = 1;
int z = 2;

double a, b, c, d, e, f, g, h, i, j, k, l, M, beta, gamma, t;

da = this->a[x] - this->b[x];
b = this->a[y] - this->b[y];
c = this->a[z] - this->b[z];
d = this->a[x] - this->c[x];
e = this->a[y] - this->c[y];
f = this->a[z] - this->c[z];
g = ray.direction[x];
h = ray.direction[y];
i = ray.direction[z];
j = this->a[x] - ray.origin[x];
k = this->a[y] - ray.origin[y];
l = this->a[z] - ray.origin[z];
M = a*((e*i)-(h*f)) + (b*((g*f)-(d*i))) + (c*((d*h)-(e*g)));

t = 0.0 - ((f*((a*k)-(j*b)))+((j*c)-(a*l))+(d*((b*l)-(k*c))))/M;
if ( t < t0 || t > t1 ) return false;
gamma = ((i*((a*k)-(j*b)))+((j*c)-(a*l))+(g*((b*l)-(k*c))))/M;
if( gamma < 0.0 || gamma > 1.0 ) return false;
beta = ((j*((e*i)-(h*f)))+((g*f)-(d*i))+(l*((d*h)-(e*g))))/M;
if( beta < 0.0 || beta > 1.0 - gamma ) return false;

double ab[3];
double ac[3];
for( int i = 0; i < 3; i++ )
{
    ab[i] = this->b[i] - this->a[i];
    ac[i] = this->c[i] - this->a[i];
}
double n[3];
osuNormalize( ab );
osuNormalize( ac );
osuCrossProduct( ab, ac, n );
osuNormalize( n );
double p[3];
for( int i = 0; i < 3; i++ )
{
    p[i] = ray.origin[i] + t * ray.direction[i];
}

// TODO: copy values to rec.

for( int i = 0; i < 3; i++ )
{
    rec->normal[i] = n[i];
    rec->point[i] = p[i];
    rec->ray.direction[i] = ray.direction[i];
    rec->ray.origin[i] = ray.origin[i];
}

rec->t = t;
rec->ka.r = this->mat->ka.r;
rec->ka.g = this->mat->ka.g;
rec->ka.b = this->mat->ka.b;
rec->kd.r = this->mat->kd.r;
rec->kd.g = this->mat->kd.g;
rec->kd.b = this->mat->kd.b;
rec->km.r = this->mat->km.r;
rec->km.g = this->mat->km.g;
rec->km.b = this->mat->km.b;
rec->ks.r = this->mat->ks.r;
rec->ks.g = this->mat->ks.g;
rec->ks.b = this->mat->ks.b;
rec->p = this->mat->p;
return true;
}

box bounding_box()
{
  struct box retval;
  for( int i = 0; i < 3; i++ ){
    retval.min[i] = min( a[i], b[i] );
    retval.min[i] = min( c[i], retval.min[i] );
    retval.max[i] = max( a[i], b[i] );
    retval.max[i] = max( c[i], retval.max[i] );
  }
  return retval;
}

private:
  double a[3];
  double b[3];
  double c[3];
  struct material *mat;
};

// CLASS: sphere
// PURPOSE: a type of 3D surface.
class sphere : public surface
{
public:
  sphere( double center[3], double radius, struct material *mat )
  {
    this->center[0] = center[0];
    this->center[1] = center[1];
    this->center[2] = center[2];
    this->radius = radius;
    this->mat = mat;
  }

bool hit( trace ray, double t0, double t1, struct hitrecord *rec )
{
  double n[3];
  double p[3];
  double t;
  double e[3];
  double d[3];
  double _d[3];
  double c[3];
  double e_minus_c[3];
  double r = this->radius;
  for( int i = 0; i < 3; i++ )
  {
    c[i] = this->center[i];
    e[i] = ray.origin[i];
    d[i] = ray.direction[i];
    e_minus_c[i] = e[i] - c[i];
    _d[i] = 0.0 - ray.direction[i];
  }
  osuNormalize( d );
  osuNormalize( _d );
  double d_dot_d = osuDotProduct( d, d );
  double d_dot_e_minus_c = osuDotProduct( d, e_minus_c );
  double _d_dot_e_minus_c = osuDotProduct( _d, e_minus_c );
  double discriminant = ( d_dot_e_minus_c * d_dot_e_minus_c ) - ( d_dot_d * ( e_minus_c_dot_e_minus_c - ( r * r ) ) );
  if( discriminant < 0.0 )
  {
    // Zero Solutions
    return false;
  }
  if( discriminant == 0.0 )
  {
    // One Solution
    t = _d_dot_e_minus_c / d_dot_d;
if ( DEBUG )
{
    printf( "SPHERE COLLISION! t: %5.2f\n", t );
}

if ( t < t0 || t > t1 ) return false;

if ( discriminant > 0.0 )
{
    // Two Solutions
    double t_minus = ( ( _d_dot_e_minus_c ) - sqrt( discriminant ) ) / d_dot_d;
    double t_plus = ( ( _d_dot_e_minus_c ) + sqrt( discriminant ) ) / d_dot_d;

    bool valid_t_minus = true;
    bool valid_t_plus = true;

    if ( t_minus < t0 || t_minus > t1 ) valid_t_minus = false;
    if ( t_plus < t0 || t_plus > t1 ) valid_t_plus = false;

    if ( !valid_t_minus && !valid_t_plus )
    {
        return false;
    }
    if ( !valid_t_minus && valid_t_plus )
    {
        t = t_plus;
    }
    if ( valid_t_minus && !valid_t_plus )
    {
        t = t_minus;
    }
    if ( valid_t_minus && valid_t_plus )
    {
        t = min( t_plus, t_minus );
    }
}

for ( int i = 0; i < 3; i++ )
{
    p[i] = e[i] + ( t * d[i] );
    n[i] = ( p[i] - c[i] ) / r;
}
rec->t = t;
for ( int i = 0; i < 3; i++ )
{
    rec->normal[i] = n[i];
    rec->point[i] = p[i];
}
rec->ka.r = this->mat->ka.r;
rec->ka.g = this->mat->ka.g;
rec->ka.b = this->mat->ka.b;
rec->kd.r = this->mat->kd.r;
rec->kd.g = this->mat->kd.g;
rec->kd.b = this->mat->kd.b;
rec->km.r = this->mat->km.r;
rec->km.g = this->mat->km.g;
rec->km.b = this->mat->km.b;
rec->ks.r = this->mat->ks.r;
rec->ks.g = this->mat->ks.g;
rec->ks.b = this->mat->ks.b;
rec->p = this->mat->p;
return true;
}

box bounding_box()
{
    struct box retval;
    for ( int i = 0; i < 3; i++ )
    {
        retval.min[i] = this->center[i] - this->radius;
        retval.max[i] = this->center[i] + this->radius;
    }
    return retval;
}
private:
    double center[3];
    double radius;
    struct material *mat;
};

// CLASS: scene
class scene
{
public:
    scene()
    {
        this->objects = 0;
        this->background.r = 0.0;
        this->background.g = 0.0;
        this->background.b = 0.0;
        this->Ia.r = 0.2;
        this->Ia.g = 0.2;
        this->Ia.b = 0.2;
        this->epsilon = 0.00000000001;
        this->max_reflections = 8;
    }

    bool hit( trace ray, double t0, double t1, struct hitrecord *rec )
    {
        bool retval = false;
        double t = t1;
        for( int i = 0; i < objects; i++ )
        {
            hitrecord temp_rec;
            if( this->object[i]->hit( ray, t0, t1, &temp_rec ) )
            {
                if( t > temp_rec.t )
                {
                    retval = true;
                    rec->ka.r = temp_rec.ka.r;
                    rec->ka.g = temp_rec.ka.g;
                    rec->ka.b = temp_rec.ka.b;
                    rec->kd.r = temp_rec.kd.r;
                    rec->kd.g = temp_rec.kd.g;
                    rec->kd.b = temp_rec.kd.b;
                    rec->km.r = temp_rec.km.r;
                    rec->km.g = temp_rec.km.g;
                    rec->km.b = temp_rec.km.b;
                    rec->ks.r = temp_rec.ks.r;
                    rec->ks.g = temp_rec.ks.g;
                    rec->ks.b = temp_rec.ks.b;
                    rec->p = temp_rec.p;
                    rec->t = temp_rec.t;
                    for( int j = 0; j < 3; j++ )
                    {
                        rec->normal[j] = temp_rec.normal[j];
                        rec->point[j] = temp_rec.point[j];
                        rec->ray.direction[j] = temp_rec.ray.direction[j];
                        rec->ray.origin[j] = temp_rec.ray.origin[j];
                    }
                    t = temp_rec.t;
                }
            }
        }
        return retval;
    }

    box bounding_box()
    {
        struct box retval;
        if( objects >= 1 )
        {
            retval = this->object[0]->bounding_box();
        }
        else
        {
for( int i = 0; i < this->objects; i++ )
{
        struct box object_box;
        object_box = this->object[i]->bounding_box();
        for( int j = 0; j < 3; j++ )
        {
                if( object_box.min[j] < retval.min[j] ) retval.min[j] = object_box.min[j];
                if( object_box.max[j] > retval.max[j] ) retval.max[j] = object_box.max[j];
        }
    }

return retval;
}

color ray_color( trace ray, double t0, double t1 )
{
    struct color c;
    struct hitrecord rec, srec;
    if( this->hit( ray, t0, t1, &rec ) )
    {
        // Calculate Ambient Lighting
        c.r = rec.ka.r * Ia.r;
        c.g = rec.ka.g * Ia.g;
        c.b = rec.ka.b * Ia.b;
        // Calculate Diffuse and Specular Lighting
        for( int i = 0; i < this->lights; i++ )
        {
            trace sray;
            for( int j = 0; j < 3; j++ )
            {
                sray.origin[j] = rec.point[j];
                sray.direction[j] = this->I[i]->point[j] - rec.point[j];
            }
            if( !this->hit( sray, this->epsilon, 1.0, &srec ) )
            {
                double h[3];
                double l[3];
                double d[3];
                for( int j = 0; j < 3; j++ )
                {
                    d[j] = ray.direction[j];
                    l[j] = sray.direction[j];
                }
                osuNormalize( d );
                osuNormalize( l );
                for( int j = 0; j < 3; j++ )
                {
                    h[j] = l[j] - d[j];
                }
                osuNormalize( h );
                c.r += ( rec.kd.r * this->I[i]->r * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.r * this->I[i]->r * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
                c.g += ( rec.kd.g * this->I[i]->g * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.g * this->I[i]->g * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
                c.b += ( rec.kd.b * this->I[i]->b * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.b * this->I[i]->b * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
            }
        }
        // Calculate Mirror Lighting
        double r[3];
        double d[3];
        double n[3];
        for( int i = 0; i < 3; i++ )
        {
            r[i] = ( rec.kd.r * this->I[i]->r * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.r * this->I[i]->r * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
            c.g += ( rec.kd.g * this->I[i]->g * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.g * this->I[i]->g * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
            c.b += ( rec.kd.b * this->I[i]->b * max( 0.0, osuDotProduct( rec.normal, l ) ) + ( rec.ks.b * this->I[i]->b * pow( osuDotProduct( rec.normal, h ), rec.p ) ) );
        }
    }
}

// Calculate Mirror Lighting
double d[3];
double n[3];
for( int i = 0; i < 3; i++ )
{
d[i] = ray.direction[i];
n[i] = rec.normal[i];
}

osuNormalize( d );
osuNormalize( n );
for( int i = 0; i < 3; i++ )
{  
r[i] = d[i] - ( 2.0 * osuDotProduct( d, n ) * n[i] );
}

osuNormalize( r );
trace mray;
for( int i = 0; i < 3; i++ )
{  
mray.origin[i] = rec.point[i];
mray.direction[i] = r[i];
}

struct color reflection;
reflection.r = 0.0;
reflection.g = 0.0;
reflection.b = 0.0;

if( ( rec.km.r + rec.km.g + rec.km.b ) > 0.0 )
{
  reflection = this->mirror_color( mray, t0 + this->epsilon, t1, 1 );
}

c.r += rec.km.r * reflection.r;
c.g += rec.km.g * reflection.g;
c.b += rec.km.b * reflection.b;

// Calculate Transparency and Refraction
// TODO: Insert Code Here.
else
{
  // Calculate Background Color
  c.r = background.r;
  c.g = background.g;
  c.b = background.b;
}
return c;

color mirror_color( trace ray, double t0, double t1, int bounces )
{
  struct color c;
  struct hitrecord rec, srec;
  if( this->hit( ray, t0, t1, &rec ) )
  {
    // Calculate Ambient Lighting
    c.r = rec.ka.r * Ia.r;
    c.g = rec.ka.g * Ia.g;
    c.b = rec.ka.b * Ia.b;

    // Calculate Diffuse and Specular Lighting
    for( int i = 0; i < this->lights; i++ )
    {
      trace sray;
      for( int j = 0; j < 3; j++ )
      {
        sray.origin[j] = rec.point[j];
        sray.direction[j] = this->I[i]->point[j] - rec.point[j];
      }
      if( !this->hit( sray, this->epsilon, 1.0, &srec ) )
      {
        double h[3];
        double l[3];
        double d[3];
        for( int j = 0; j < 3; j++ )
        {
          d[j] = ray.direction[j];
          l[j] = sray.direction[j];
        }
osuNormalize( d );
ouNormalize( l );
for( int j = 0; j < 3; j++ )
{
    h[j] = l[j] - d[j];
}
osuNormalize( h );
ocDotProduct( rec.normal, l ) ) ) + ( rec.ks.r * this->I[i]->r * pow( osuDotProduct( rec.normal, h ), rec.p ) );
c.g += ( rec.kd.g * this->I[i]->g * max( 0.0, osuDotProduct( rec.normal, l ) ) ) + ( rec.ks.g * this->I[i]->g * pow( osuDotProduct( rec.normal, h ), rec.p ) );
c.b += ( rec.kd.b * this->I[i]->b * max( 0.0, osuDotProduct( rec.normal, l ) ) ) + ( rec.ks.b * this->I[i]->b * pow( osuDotProduct( rec.normal, h ), rec.p ) );
}

// Calculate Mirror Lighting
double r[3];
double d[3];
double n[3];
for( int i = 0; i < 3; i++ )
{
    d[i] = ray.direction[i];
    n[i] = rec.normal[i];
}
osuNormalize( d );
osuNormalize( n );
for( int i = 0; i < 3; i++ )
{
    r[i] = d[i] - ( 2.0 * osuDotProduct( d, n ) * n[i] );
}
osuNormalize( r );
trace mray;
for( int i = 0; i < 3; i++ )
{
    mray.origin[i] = rec.point[i];
    mray.direction[i] = r[i];
}
struct color reflection;
reflection.r = 0.0;
reflection.g = 0.0;
reflection.b = 0.0;
if( ( rec.km.r + rec.km.g + rec.km.b ) > 0.0 && bounces < this->max_reflections )
{
    reflection = this->mirror_color( mray, t0 + this->epsilon, t1, bounces + 1
};
}
c.r += rec.km.r * reflection.r;
c.g += rec.km.g * reflection.g;
c.b += rec.km.b * reflection.b;

// Calculate Transparency and Refraction
// TODO: Insert Code Here.
else
{
    // Calculate Background Color
    c.r = background.r;
c.g = background.g;
c.b = background.b;
    return c;
}
void add_object( surface *object )
{
    if( objects >= MAX_OBJECTS ) return;
    this->object[objects] = object;
    objects++;
}
if ( DEBUG )
{
    struct box objectbox = object->bounding_box();
    double objectsize = 1.0;
    objectsize *= objectbox.max[0] - objectbox.min[0];
    objectsize *= objectbox.max[1] - objectbox.min[1];
    objectsize *= objectbox.max[2] - objectbox.min[2];
    printf( "OBJECT %i (SIZE %5.2f) ADDED TO SCENE.\n", this->objects, objectsize );
}

// Allow modification of Ia, I, and background.
void set_background_color( struct color intensity )
{
    this->background.r = intensity.r;
    this->background.g = intensity.g;
    this->background.b = intensity.b;
}

void set_ambient_luminosity( struct color intensity )
{
    this->Ia.r = intensity.r;
    this->Ia.g = intensity.g;
    this->Ia.b = intensity.b;
}

void add_light( double position[3], struct color intensity )
{
    if ( lights >= MAX_LIGHTS ) return;
    struct pointlight *light = new pointlight;
    light->r = intensity.r;
    light->g = intensity.g;
    light->b = intensity.b;
    light->point[0] = position[0];
    light->point[1] = position[1];
    this->I[lights] = light;
    this->lights++;
    if ( DEBUG )
    {
        printf( "LIGHT %i ADDED TO SCENE.\n", this->lights );
    }
}

void set_epsilon( double val )
{
    this->epsilon = val;
}

void set_max_reflections( int reflections )
{
    this->max_reflections = reflections;
}

private:
int lights;
struct pointlight *I[MAX_LIGHTS];
struct color Ia;
struct color background;
int objects;
surface *object[MAX_OBJECTS];
double epsilon;
int max_reflections;
};

///////////
// GLOBALS //
///////////

double u[3] = { 0.0, 0.0, 0.0 };
double v[3] = { 0.0, 0.0, 0.0 };
double w[3] = { 0.0, 0.0, 0.0 };
double e[3] = { 0.0, 0.0, 0.0 };  
struct trace pray[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct trace pray1[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct trace pray2[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct trace pray3[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct trace pray4[IMAGE_WIDTH][IMAGE_HEIGHT];  
scene cornellbox;  
struct color framebuffer[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct color framebuffer1[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct color framebuffer2[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct color framebuffer3[IMAGE_WIDTH][IMAGE_HEIGHT];  
struct color framebuffer4[IMAGE_WIDTH][IMAGE_HEIGHT];

//////////  
// MAIN //  
//////////

void main( int argc, char **argv ) {
    clock_t StartClock, SetupClock, TraceClock, RenderClock, EndClock;
    double SetupTime, TraceTime, RenderTime, EndTime;

    // Timing Functions
    StartClock = clock();

    osuBeginGraphics( IMAGE_WIDTH, IMAGE_HEIGHT );

    // Primary Camera
    double from[3] = { 0.0, 0.0, 2.0 };
    double at[3] = { 0.0, 0.0, 1.0 };
    double up[3] = { 0.0, 1.0, 2.0 };

    // Secondary Camera
    double top[3] = { 0.0, 2.0, 0.0 };
    double front[3] = { 0.0, 2.0, -1.0 };
    double center[3] = { 0.0, 1.0, 0.0 };

    // Perspective Settings
    double fovy = 60.0;
    double nearp = 1.0;
    double farp = 100.0;

    // Cornell Box Corners
    double corner0[3] = { -1.0, -1.0,  1.0 };
    double corner1[3] = {  1.0, -1.0,  1.0 };
    double corner2[3] = {  1.0, -1.0, -1.0 };
    double corner3[3] = { -1.0, -1.0, -1.0 };
    double corner4[3] = { -1.0,  1.0,  1.0 };
    double corner5[3] = {  1.0,  1.0,  1.0 };
    double corner6[3] = {  1.0,  1.0, -1.0 };
    double corner7[3] = { -1.0,  1.0, -1.0 };

    // Spheres
    double s1p[3] = { -0.33, -0.75, -0.33 };
    double s1r = 0.25;
    double s2p[3] = {  0.33, -0.75,  0.33 };
    double s2r = 0.25;
    double s3p[3] = {  0.00,  0.25,  0.00 };
    double s3r = 0.25;
    double s4p[3] = { -0.25, -0.25,  0.00 };
    double s4r = 0.25;
    double s5p[3] = {  0.25, -0.25,  0.00 };
    double s5r = 0.25;

    // Lights
    double l1p[3] = {  0.00,  0.70,  0.0 };
    struct color l1c;
    l1c.r = 0.5;
    l1c.g = 0.5;
    l1c.b = 0.5;
    double l2p[3] = { -0.5,  0.5,  2.0 };
    struct color l2c;
    l2c.r = 0.3;
    l2c.g = 0.3;
    l2c.b = 0.3;

    // Materials
    struct material s1m;
    struct material s2m;
    struct material s3m;
struct material s4m;
struct material s5m;
struct material s6m;

// Specify an orange surface.
s1m.ka.r = 1.00;
s1m.ka.g = 0.30;
s1m.ka.b = 0.00;
s1m.kd.r = 1.00;
s1m.kd.g = 0.30;
s1m.kd.b = 0.00;
s1m.km.r = 0.05;
s1m.km.g = 0.05;
s1m.km.b = 0.05;
s1m.ks.r = 0.80;
s1m.ks.g = 0.80;
s1m.ks.b = 0.80;
s1m.p = 100.0;

// Specify a mirror surface.
s2m.ka.r = 0.01;
s2m.ka.g = 0.01;
s2m.ka.b = 0.01;
s2m.kd.r = 0.01;
s2m.kd.g = 0.01;
s2m.kd.b = 0.01;
s2m.km.r = 0.60;
s2m.km.g = 0.60;
s2m.km.b = 0.60;
s2m.ks.r = 0.50;
s2m.ks.g = 0.50;
s2m.ks.b = 0.50;
s2m.p = 100.0;

// specify a white surface
s3m.ka.r = 1.00;
s3m.ka.g = 1.00;
s3m.ka.b = 1.00;
s3m.kd.r = 0.75;
s3m.kd.g = 0.75;
s3m.kd.b = 0.75;
s3m.km.r = 0.00;
s3m.km.g = 0.00;
s3m.km.b = 0.00;
s3m.ks.r = 0.25;
s3m.ks.g = 0.25;
s3m.ks.b = 0.25;
s3m.p = 10.0;

// specify a red surface
s4m.ka.r = 1.00;
s4m.ka.g = 0.30;
s4m.ka.b = 0.30;
s4m.kd.r = 1.00;
s4m.kd.g = 0.30;
s4m.kd.b = 0.30;
s4m.km.r = 0.10;
s4m.km.g = 0.10;
s4m.km.b = 0.10;
s4m.ks.r = 0.25;
s4m.ks.g = 0.25;
s4m.ks.b = 0.25;
s4m.p = 10.0;

// specify a green surface
s5m.ka.r = 0.30;
s5m.ka.g = 1.00;
s5m.ka.b = 0.30;
s5m.kd.r = 0.30;
s5m.kd.g = 1.00;
s5m.kd.b = 0.30;
s5m.km.r = 0.10;
s5m.km.g = 0.10;
s5m.km.b = 0.10;
s5m.ks.r = 0.25;
s5m.ks.g = 0.25;
s5m.ks.b = 0.25;
s5m.p = 10.0;

// specify a gold surface
s6m.ka.r = 0.01;
s6m.ka.g = 0.01;
s6m.ka.b = 0.01;
s6m.kd.r = 0.60;
s6m.kd.g = 0.60;
s6m.kd.b = 0.30;
s6m.km.r = 0.60;
s6m.km.g = 0.60;
s6m.km.b = 0.30;
s6m.ks.r = 1.00;
s6m.ks.g = 1.00;
s6m.ks.b = 0.50;
s6m.p = 100.0;

osuLookAt( from, at, up );
// osuLookAt( top, center, front );
osuPerspective( fovy, nearp, farp );
// osuOrtho( -1.0, 1.0, -1.0, 1.0, -1.0, 1.0 );

// Set up the scene
sphere s1( s1p, s1r, &s1m );
sphere s2( s2p, s2r, &s2m );
triangle t1( corner0, corner1, corner2, &s3m );
triangle t2( corner2, corner3, corner0, &s3m );
triangle t3( corner3, corner2, corner6, &s3m );
triangle t4( corner6, corner7, corner3, &s3m );
triangle t5( corner6, corner5, corner4, &s3m );
triangle t6( corner4, corner5, corner6, &s3m );
triangle t7( corner0, corner3, corner7, &s4m );
triangle t8( corner7, corner4, corner0, &s4m );
triangle t9( corner2, corner1, corner5, &s5m );
triangle t10( corner5, corner6, corner2, &s5m );

double triforce0[3] = { 0.00, 0.50, -1.00 };
double triforce1[3] = {-0.25, 0.00, -0.95 };
double triforce2[3] = { 0.25, 0.00, -0.95 };
double triforce3[3] = {-0.25, 0.00, -0.95 };
double triforce4[3] = {-0.50, -0.50, -1.00 };
double triforce5[3] = { 0.00, -0.50, -0.95 };
double triforce6[3] = { 0.25, 0.00, -0.95 };
double triforce7[3] = { 0.00, -0.50, -0.95 };
double triforce8[3] = { 0.50, -0.50, -1.00 };
triangle t11( triforce0, triforce1, triforce2, &s6m );
triangle t12( triforce3, triforce4, triforce5, &s6m );
triangle t13( triforce6, triforce7, triforce8, &s6m );
cornellbox.add_light( l1p, l1c );
cornellbox.add_light( l2p, l2c );
cornellbox.add_object( &s1 );
cornellbox.add_object( &s2 );
cornellbox.add_object( &t1 );
cornellbox.add_object( &t2 );
cornellbox.add_object( &t3 );
cornellbox.add_object( &t4 );
cornellbox.add_object( &t5 );
cornellbox.add_object( &t6 );
cornellbox.add_object( &t7 );
cornellbox.add_object( &t8 );
cornellbox.add_object( &t9 );
cornellbox.add_object( &t10 );
cornellbox.add_object( &t11 );
cornellbox.add_object( &t12 );
cornellbox.add_object( &t13 );

// Threading Code.
pthread_t thread[NUM_THREADS];
pthread_attr_t attr;
int rc;

20
int t;
void *status;
if( MULTI_THREADING && ANTI_ALIASING )
{
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
}

// Timing Functions
SetupClock = clock();
SetupTime = (double) (SetupClock - StartClock) / (double) CLOCKS_PER_SEC;
printf("Setup Time: %5.5f.\n", SetupTime);
if( MULTI_THREADING && ANTI_ALIASING )
{
    for ( t = 0; t < NUM_THREADS; t++ )
    {
        rc = pthread_create(&thread[t], &attr, osuRaytraceSceneThread, (void *) t);
        if( rc )
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
}
else
{
    if( ANTI_ALIASING )
    {
        for ( int x = 0; x < IMAGE_WIDTH; x++ )
        {
            for ( int y = 0; y < IMAGE_HEIGHT; y++ )
            {
                osuRaytraceScene( &framebuffer1[x][y], pray1[x][y], cornellbox );
                osuRaytraceScene( &framebuffer2[x][y], pray2[x][y], cornellbox );
                osuRaytraceScene( &framebuffer3[x][y], pray3[x][y], cornellbox );
                osuRaytraceScene( &framebuffer4[x][y], pray4[x][y], cornellbox );
            }
        }
    }
    else
    {
        for ( int x = 0; x < IMAGE_WIDTH; x++ )
        {
            for ( int y = 0; y < IMAGE_HEIGHT; y++ )
            {
                osuRaytraceScene( &framebuffer[x][y], pray[x][y], cornellbox );
            }
        }
    }
}

// Threading Code.
if( MULTI_THREADING && ANTI_ALIASING )
{
    pthread_attr_destroy(&attr);
    for ( t = 0; t < NUM_THREADS; t++ )
    {
        rc = pthread_join(thread[t], &status);
        if( rc )
        {
            printf("ERROR; return code from pthread_join() is %d\n", rc);
            exit(-1);
        }
    }
}

// Timing Functions
TraceClock = clock();
TraceTime = (double) (TraceClock - SetupClock) / (double) CLOCKS_PER_SEC;
printf("Raytrace Time: %5.5f.\n", TraceTime);

int r, g, b;
for ( int x = 0; x < IMAGE_WIDTH; x++ )

21
for( int y = 0; y < IMAGE_HEIGHT; y++ )
{
    if( !ANTI_ALIASING ){
        r = (int) ( framebuffer[x][y].r * 255.0 );
        g = (int) ( framebuffer[x][y].g * 255.0 );
        b = (int) ( framebuffer[x][y].b * 255.0 );
        osuWritePixel( x, y, r, g, b );
    } else {
        r = (int) (( framebuffer1[x][y].r + framebuffer2[x][y].r + framebuffer3[x][y].r + framebuffer4[x][y].r ) * 255.0 * 0.25 );
        g = (int) (( framebuffer1[x][y].g + framebuffer2[x][y].g + framebuffer3[x][y].g + framebuffer4[x][y].g ) * 255.0 * 0.25 );
        b = (int) (( framebuffer1[x][y].b + framebuffer2[x][y].b + framebuffer3[x][y].b + framebuffer4[x][y].b ) * 255.0 * 0.25 );
        osuWritePixel( x, y, r, g, b );
    }
}

// Timing Functions
RenderClock = clock();
RenderTime = (double) ( RenderClock - TraceClock ) / (double) CLOCKS_PER_SEC;
printf( "Render Time: %5.5f.\n", RenderTime );

// Timing Functions
EndClock = clock();
EndTime = (double) ( EndClock - StartClock ) / (double) CLOCKS_PER_SEC;
printf( "Total Time: %5.5f.\n", EndTime );

osuFlush();
osuWaitOnEscape();
osuEndGraphics();

if( MULTI_THREADING ){
    pthread_exit( NULL );
}

/*******/
// FUNCTIONS //
/*******/

// FUNCTION: osuOrtho
// PURPOSE: Arranges the rays in orthographic projection.
void osuOrtho(double left, double right, double bottom, double top, double nearp, double farp) {
    double r = right;
    double l = left;
    double t = top;
    double b = bottom;
    double nx = (double) IMAGE_WIDTH;
    double ny = (double) IMAGE_HEIGHT;
    for( int i = 0; i < IMAGE_WIDTH; i++ )
    {
        for( int j = 0; j < IMAGE_HEIGHT; j++ )
        {
            double _u = l + ( ( r - l ) * ( (double) i + 0.50000001 ) / ( nx ) );
            double _v = b + ( ( t - b ) * ( (double) j + 0.50000002 ) / ( ny ) );
            double _u1 = l + ( ( r - l ) * ( (double) i + 0.25000001 ) / ( nx ) );
            double _v1 = b + ( ( t - b ) * ( (double) j + 0.25000002 ) / ( ny ) );
            double _u2 = l + ( ( r - l ) * ( (double) i + 0.25000001 ) / ( nx ) );
            double _v2 = b + ( ( t - b ) * ( (double) j + 0.75000002 ) / ( ny ) );
            double _u3 = l + ( ( r - l ) * ( (double) i + 0.75000001 ) / ( nx ) );
            double _v3 = b + ( ( t - b ) * ( (double) j + 0.75000002 ) / ( ny ) );
            double _u4 = l + ( ( r - l ) * ( (double) i + 0.75000001 ) / ( nx ) );
            double _v4 = b + ( ( t - b ) * ( (double) j + 0.75000002 ) / ( ny ) );
            for( int k = 0; k < 3; k++ )
            {
                // Code here
            }
        }
    }
}
// FUNCTION: osuPerspective
// PURPOSE: Arranges the rays in perspective projection.
void osuPerspective(double fovy, double nearp, double farp) {
    fovy *= 3.14159265 / 180.0;
    double aspect = (double) IMAGE_HEIGHT / (double) IMAGE_WIDTH;
    double d = nearp;
    double r = d * tan(fovy / 2.0);
    double l = 0.0 - r;
    double t = r * aspect;
    double b = 0.0 - t;
    double nx = (double) IMAGE_WIDTH;
    double ny = (double) IMAGE_HEIGHT;
    for (int i = 0; i < IMAGE_WIDTH; i++) {
        for (int j = 0; j < IMAGE_HEIGHT; j++) {
            double _u = l + ( (r - l) * ( (double) i + 0.50000001 ) / (nx) ) ;
            double _v = b + ( (t - b) * ( (double) j + 0.50000002 ) / (ny) ) ;

            pray[i][j].direction[k] = ( -d * w[k] ) + ( _u * u[k] ) + ( _v * v[k] ) ;
            pray[i][j].origin[k] = e[k];

            pray1[i][j].direction[k] = ( -d * w[k] ) + ( _u1 * u[k] ) + ( _v1 * v[k] ) ;
            pray1[i][j].origin[k] = e[k];

            pray2[i][j].direction[k] = ( -d * w[k] ) + ( _u2 * u[k] ) + ( _v2 * v[k] ) ;
            pray2[i][j].origin[k] = e[k];

            pray3[i][j].direction[k] = ( -d * w[k] ) + ( _u3 * u[k] ) + ( _v3 * v[k] ) ;
            pray3[i][j].origin[k] = e[k];

            pray4[i][j].direction[k] = ( -d * w[k] ) + ( _u4 * u[k] ) + ( _v4 * v[k] ) ;
            pray4[i][j].origin[k] = e[k];
        }
    }
}

// FUNCTION: osuLookat
// PURPOSE: Arranges the rays to point in a specific direction.
void osuLookAt(double from[3], double at[3], double up[3]) {
    for (int i = 0; i < 3; i++) {
        
    }
}
e[i] = from[i];
v[i] = up[i] - from[i];
w[i] = from[i] - at[i];
}
osuNormalize( v );
osuNormalize( w );
osuCrossProduct( v, w, u );

if ( DEBUG ) {
    printf( "e(%5.2f,%5.2f,%5.2f). v(%5.2f,%5.2f,%5.2f). w(%5.2f,%5.2f,%5.2f). u(%5.2f,%5.2f,%5.2f)\n", 
e[0], e[1], e[2], v[0], v[1], v[2], w[0], w[1], w[2], u[0], u[1], u[2] );
}

// FUNCTION: osuCrossProduct
// PURPOSE: Calculates the outer product of two vectors.
void osuCrossProduct( double b[3], double c[3], double a[3] )
{
    int x = 0;
    int y = 1;
    int z = 2;
    a[x] = ( b[y] * c[z] ) - ( b[z] * c[y] );
    a[y] = ( b[z] * c[x] ) - ( b[x] * c[z] );
    a[z] = ( b[x] * c[y] ) - ( b[y] * c[x] );
}

// FUNCTION: osuDotProduct
// PURPOSE: Calculates the inner product of two vectors.
double osuDotProduct( double a[3], double b[3] )
{
    double retval = 0.0;
    for( int i = 0; i < 3; i++ )
    {
        retval += a[i] * b[i];
    }
    return retval;
}

// FUNCTION: osuLength
// PURPOSE: Calculates the length of a vector.
double osuLength( double a[3] )
{
    double retval = 0.0;
    for( int i = 0; i < 3; i++ )
    {
        retval += a[i] * a[i];
    }
    retval = sqrt( retval );
    return retval;
}

// FUNCTION: osuNormalize
// PURPOSE: normalizes a vector.
void osuNormalize( double a[3] )
{
    double length = osuLength( a );
    for( int i = 0; i < 3; i++ )
    {
        a[i] /= length;
    }
}

// FUNCTION: max
// PURPOSE: returns the larger of two values.
double max( double a, double b )
{
    double retval;
    if( a > b ) retval = a;
    else if( a <= b ) retval = b;
    return retval;
}

// FUNCTION: min
// PURPOSE: returns the smaller of two values.
double min( double a, double b )
{
    double retval;
    if( a > b ) retval = b;
    if( a <= b ) retval = a;
    return retval;
}

// FUNCTION: osuRaytraceScene
// PURPOSE: traces a ray into a scene and returns the pixel value.
void osuRaytraceScene( struct color *pixel, struct trace ray, class scene myscene )
{
    struct color temp;
    temp = myscene.ray_color( ray, 0.0, 100000.0 );
    temp.r = max( 0.0, temp.r );
    temp.g = max( 0.0, temp.g );
    temp.b = max( 0.0, temp.b );
    pixel->r = temp.r;
    pixel->g = temp.g;
    pixel->b = temp.b;
}

// FUNCTION: osuRaytraceSceneThread
// PURPOSE: traces a ray into a scene and returns the pixel value.
void *osuRaytraceSceneThread( void *t )
{
    long thread_id = (int) t;
    for( int x = 0; x < IMAGE_WIDTH; x++ )
    {
        for( int y = 0; y < IMAGE_HEIGHT; y++ )
        {
            if( ANTI_ALIASING )
            {
                switch( thread_id )
                {
                case 0:
                    osuRaytraceScene( &framebuffer1[x][y], pray1[x][y], cornellbox );
                    break;
                case 1:
                    osuRaytraceScene( &framebuffer2[x][y], pray2[x][y], cornellbox );
                    break;
                case 2:
                    osuRaytraceScene( &framebuffer3[x][y], pray3[x][y], cornellbox );
                    break;
                case 3:
                    osuRaytraceScene( &framebuffer4[x][y], pray4[x][y], cornellbox );
                    break;
                }
            }
        }
    }
    pthread_exit( (void *) t );
    return NULL;
}