CS 371 Project 2:

Eye Rays

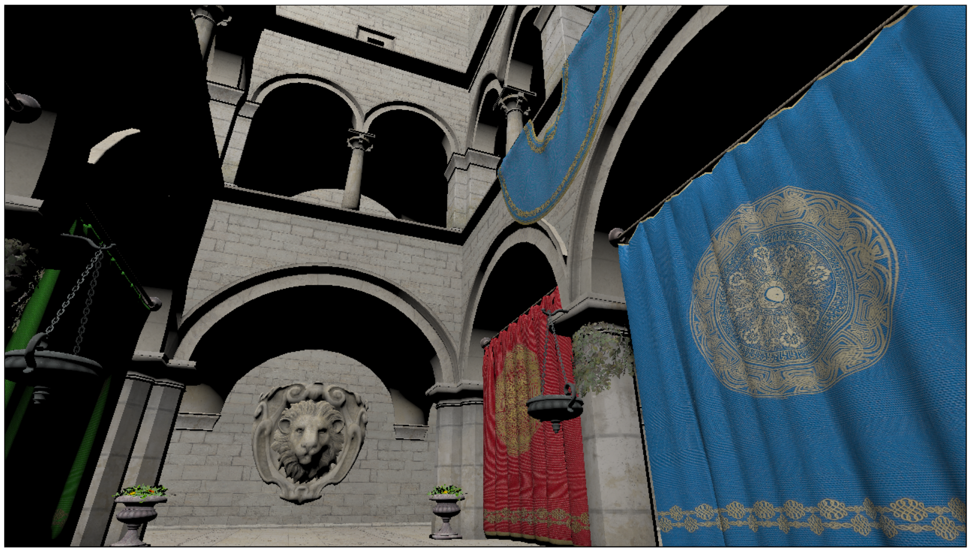


Figure 1: *Crytek's model of the atrium of the Sponza palace is a standard benchmark scene for 3D rendering algorithms. By the end of this lab, you'll be able to produce images of it yourself from first principles, without relying on the OpenGL/G3D renderer.*

1 Introduction

Ray casting is one of the core techniques for approximating photorealistic rendering. This algorithm casts a ray through each pixel to find the surface that colors that pixel. The most straightforward variant, which you will implement in this project, then shades the surface by iterating over the light sources. It was first investigated by Appel and others in the late 1960's, and quickly evolved into Whitted's **ray tracing** algorithm. Ray casting was also the basis for most real-time rendering until fairly recently. Many 3D games in the early 1990's such as *Wolfenstein*, *Doom*, *Heretic*, *Duke Nukem 3D*, and *Star Wars: Dark Forces* explicitly cast rays. Other games used a variant on ray casting called **rasterization** with **direct illumination** well into the 2000's. Later projects in this class will explore both ray tracing and rasterization using the physically-based framework that you build this week.

The previous two projects focused on modeling scenes and left the rendering to a simple black-box renderer built on OpenGL. In this project, you'll augment that preview with your own renderer so that you understand the entire system, from the data files that describe the scene to the way the value of an individual pixel is calculated. The images that your own program generates this week should exactly match the ones produced by the OpenGL preview renderer!

2 Table of Contents

3 Schedule

Out:	Tuesday,	September 18	
Checkpoints:	Thursday,	September 20,	1:00 pm
Due:	Monday,	September 24,	12:00 pm

This is an easy project for teams of three or four people. As with other projects, try to quickly cover the entire specification with stubbed out methods and provisional report text before refining any one area.

As a reference, my solution required about 260 statements and 350 comment lines, including the report Doxygen comments (as reported by iCompile), plus several data files. Note that I always line-wrap my comments with M-q; if you don't your comments may look ugly in my editor and they will report as many fewer lines. If your codebase looks like it is going to be more than 1.5× larger or smaller than my solution, come talk to me because you may be on a bad path.

4 Rules/Honor Code

You are encouraged to talk to other students and share strategies and programming techniques. You should not look at any other student's code for this project. You may look at and use anyone's code from previous projects, with their permission.

During this project, you are not permitted to look at the G3D `rayTrace` sample program.

5 Teams

- red: Lucky, Alex, April
- orange: Greg, Jonathan, Donny
- yellow: Owen, Qiao, Michael
- dan: Dan E., Dan F., Dan S.
- cyan: Lily, Tucker, James W.
- blue: Josh, Nico, Scott P-S., Scott S.
- ultraviolet: Parker, Cody, James R.

6 Individual Checkpoint (1pm Thu.)

Submit on paper...

1. Write one paragraph describing a project that you're considering for your mid-term or final. You may include pictures. Discuss how many people you think it would take to accomplish this and what you'll need to learn between now and then. Describe how you will present it–what is the visual takeaway or experience that the audience will have?

2. Implement `L_scatteredDirect` in legal C++ code (N.B. you won't be able to test it). Be very clear about the units for each step in comments and using appropriate types such as `Biradiance3`, `Radiance3`, `Power3`, etc.

3. Derive the intersection of ray $P + \hat{\omega}t$ with the cylinder centered at the origin that circles the $\hat{y}$ axis and extends from $y = 1$ to $y = -1$, with radius r in the xz plane. Assume that the cylinder has no top and bottom–it is like a toilet paper tube. Make the cylinder one-sided, so that rays that miss the outside never hit it (i.e., from the origin, it is invisible). Compute the normal to the point hit if there is an intersection. You will probably want to mix math and pseudocode in your final writeup.

7 Team Checkpoint (1:45pm Thu.)

1. Submit on paper a **schedule** describing your team's plan for completing the specification. This should list tasks that are individually no longer than 2 hours. Each task should have a team member's name next to it (or multiple ones, if they are pair programming), and be tied to a concrete time and date during which you anticipate the work will be performed.

   You may use a simple list, a calendar view, or a Gantt chart. I favor the latter for complex projects because it allows you to express dependencies.

Tip: You should probably keep a copy of your schedule when you hand it in.

2. Submit a printout of your **report** draft, and commit the relevant interfaces to SVN. The report should be the bare bones of everything described in section 8.1, but with a fully-fleshed out architectural overview. By reading the architectural overview and follow links from it to the referenced methods, I should be able to understand the structure of your program and the interfaces that the team members will later implement.

   I do not expect any implementation at this point. You are of course free to change your interfaces at any point during the project.

8 Specification

Implement a program with the following features:

Although we're implementing ray *casting* this week, next week we'll be ray *tracing*, which is why we are naming the class `RayTracer`.

1. A class named `RayTracer` that:

   (a) Supports at least the interface described in Section 8.2

   (b) Renders images by the ray casting algorithm (cast one primary ray per pixel into the scene)

   (c) Can operate in both single-threaded and concurrent multi-threaded modes.

   (d) Searches for intersections using both exhaustive array and pruning tree (`G3D::TriTree`) search, and can switch between them at run time.

   (e) Computes direct illumination under point lights as described in Equation 1:

$$\begin{aligned}\text{Let } \hat{\omega}_\mathrm{i} &= \mathbf{S}^2(Y_j - X)\\ \text{Let } r_j &= ||Y_j - X||\\ \text{Let } B_j &= \frac{\Phi_j}{4\pi r^2}\\ L_\mathrm{o}(X, \hat{\omega}_\mathrm{o}) &= \sum_{j=0}^{N-1}\left[B_j \, |\hat{\omega}_\mathrm{i} \cdot \hat{n}| \, f_{X,\hat{n}}(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})\right]\end{aligned} \tag{1}$$

   where light with index $0 \le j < N$ is described by power Φ_j and position Y_j, X is the intersection point, $\hat{n}$ is the **shading normal** at the intersection, and $\hat{\omega}_\mathrm{o}$ points back along the ray.

2. A scene containing the Cornell box with a single, **non-shadow casting** omni light source and no "environment" light.

3. A scene containing the Crytek Sponza atrium from a viewpoint and lighting conditions that approximately match Figure 1. Specifically, there should be a single, non-shadow casting omni light.

4. User interface functionality comparable to that shown in figure 2.

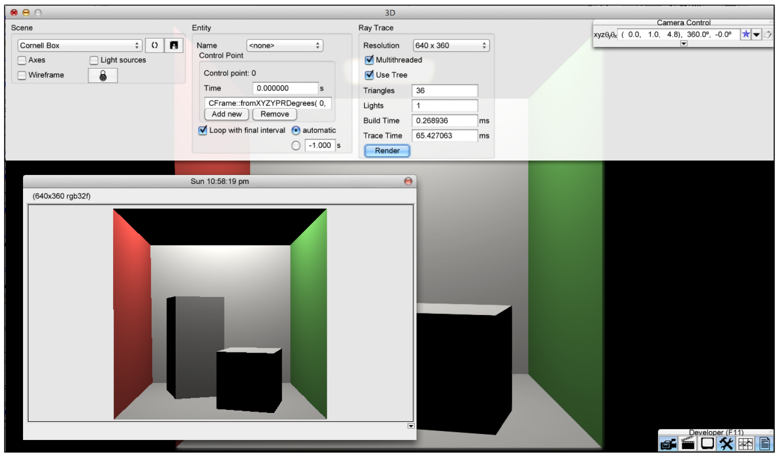
5. Create the documentation reports specified in section 8.1.

8.1 Report

Write an appropriately-formatted report that covers the following topics:

1. An architectural overview of your program.

2. Discuss significant design choices that you made, and argue why your choices were good for this project.



Cornell Box
[ ] Axes [ ] Light sources
[ ] Wireframe

Name <none>
Control Point
Control point: 0
Time 0.000000 s
CFrame::fromXYZYPRDegrees(0,
[✓] Loop with final interval (●) automatic
( ) -1.000 s

Resolution 640 x 360
[✓] Multithreaded
[✓] Use Tree
Triangles 36
Lights 1
Build Time 0.268936 ms
Trace Time 65.427063 ms

xyzθyθx (0.0, 1.0, 4.8), 360.0°, -0.0°

Sun 10:58:19 pm
(640x360 rgb32f)

Figure 2: *One possible user interface for the ray casting program.*

3. Discuss any known errors in your program, and how you identified and attempted to correct them.

4. Show pictures of the following scenes rendered with ray casting and by the preview renderer, side-by-side:

   (a) The Cornell Box scene

   (b) The Sponza scene

   (c) A visually compelling scene of your own design. This can re-use elements from previous assignments.

Tip: Your ray-cast images should match those from preview mode without wireframes when `environmentMapConstant` is zero, except for pixels colored by the sky box.

5. **Questions.** (*To calibrate your level of effort, all of these together should take you more than 10 minutes and less than one hour to complete.*)

   (a) The ray casting program that you wrote assumed that the only significant incident light was directly from the sources. Describe the errors contributed by this approach, a scene for which this error is significant, and briefly propose an algorithm for incorporating indirect light that has scattered from other surfaces.

   (b) Without performing a formal experiment, describe the performance impact of multithreading. Is it what you expected?

   (c) Briefly speculate on how `TriTree` might work. You may research this or read the source code, but I'm more interested in your own ideas about how you would design a data structure for ray-triangle intersection.

   (d) Briefly describe the different ways that you would have to change your program to incorporate another type of primitive, such as a true sphere. Consider everything from the scene data files through the shading algorithms.

6. **Feedback.** Your feedback is important to me for tuning the upcoming projects and lectures. Please report:

(a) How many hours you spent **outside** of class on this project on **required** elements, i.e., the minimum needed to satisfy the specification.

(b) How many additional hours you spent outside of class on this project on **optional** elements, such as polishing your custom scene or extreme formatting of the report.

(c) Rate the difficulty of this project for this point in a 300-level course as: too easy, easy, moderate, challenging, or too hard. What made it so?

(d) What did you learn on this project (very briefly)? In addition to the algorithm, consider the workflow lessons, programming and design experience, and the process of thinking about questions.

8.2 RayTracer

Your `RayTracer` class should follow the interface below to ensure compatibility between each other's projects going forward. You may extend the interface as you see fit. Please re-type this code to ensure that at least one person on your team has read it closely. As with all code in your program, you are responsible for understanding, documenting, and debugging this code.

```
#ifndef RayTracer_h
#define RayTracer_h

#include <G3D/G3DAll.h>

class RayTracer : public ReferenceCountedObject {
public:

    class Settings {
    public:
        int                     width;
        int                     height;
        bool                    multithreaded;
        bool                    useTree;
        Settings();
    };

    class Stats {
    public:
        int                     lights;

        int                     triangles;

        /** width x height */
        int                     pixels;

        float                   buildTriTreeTimeMilliseconds;
        float                   rayTraceTimeMilliseconds;
        Stats();
    };

protected:
    /** Array of random number generators so that each threadID may
        have its own without using locks. */
    Array< shared_ptr<Random> > m_rnd;

    // The following are only valid during a call to render()
    shared_ptr<Image>         m_image;
    shared_ptr<Lighting>      m_lighting;
    shared_ptr<Camera>        m_camera;
    Settings                  m_settings;
    TriTree                   m_triTree;

    RayTracer();

    /** Called from GThread::runConcurrently2D(), which is invoked
         in traceAllPixels() */
    void traceOnePixel(int x, int y, int threadID);

    /** Called from render().  Writes to m_image. */
    void traceAllPixels(int numThreads);

    /**
       \param The ray in world space
       \param maxDistance Don't trace farther than this
       \param anyHit If true, return any surface hit, even if it is
          not the first
       \return The surfel hit, or NULL if none was hit
     */
    shared_ptr<Surfel> castRay
        (const Ray&                ray,
         float                     maxDistance = finf(),
         bool                      anyHit = false) const;

    Radiance3 L_scatteredDirect
       (const shared_ptr<Surfel>& surfel,
        const Vector3&            wo,
        Random&                   rnd) const;

public:

    static shared_ptr<RayTracer> create();

    /** Render the specified image */
    shared_ptr<Image> render
    (const Settings&                       settings,
     const Array< shared_ptr<Surface> >&  surfaceArray,
     const shared_ptr<Lighting>&          lighting,
     const shared_ptr<Camera>&            camera,
     Stats&                               stats);

};
```

```
#endif
```

9 Implementation Advice

9.1 Getting Started

Note that this week I am not specifically requiring you to extend the Meshes project, although you are welcome to. I chose a different route, myself. I began this project with an empty directory. I ran icompile in that directory with no arguments, and when prompted, told it that I wanted a new G3D starter project. This built a basic GUI for me that handled, and provided some convenient scenes, like Sponza and the Cornell Box. I then deleted the "Alpha Base" scene, which contains animation that I wasn't prepared to deal with this week.

Tip: Use the –opt flag to icompile to run faster when you aren't actively tracking down a bug.

The Subversion command to check out your project this week is:

```
svn co svn://graphics-svn.cs.williams.edu/371/2-EyeRays/eyerays-<teamname>
```

in which you should replace `<teamname>` with your team name.

9.2 `RayTracer.cpp`

I provide a partial implementation below to help get you started. As always, you're responsible for understanding and documenting this code–and fixing it if it (unintentionally) contains bugs.

Tip: Use `Surfel::reflectivity` as your shading value when you're debugging the ray cast. It will produce the "color" of the surface. Another team member can then work on shading in parallel to you.

```

RayTracer::Settings::Settings() :
    width(160),
    height(90),
    multithreaded(true),
    useTree(false) {

#   ifdef G3D_DEBUG
        // If we're debugging, we probably don't want threads by default
        multithreaded = false;
#   endif
}


RayTracer::Stats::Stats() :
    lights(0),
    triangles(0),
    pixels(0),
    buildTriTreeTimeMilliseconds(0),
    rayTraceTimeMilliseconds(0) {}


RayTracer::RayTracer() {
    m_rnd.resize(System::numCores());
    for (int i = 0; i < m_rnd.size(); ++i) {
        // Use a different seed for each and do not be threadsafe
        m_rnd[i] = shared_ptr<Random>(new Random(i, false));
    }
}


shared_ptr<RayTracer> RayTracer::create() {
    return shared_ptr<RayTracer>(new RayTracer());
}


shared_ptr<Image> RayTracer::render
(const Settings&                           settings,
 const Array< shared_ptr<Surface> >&  surfaceArray,
 const shared_ptr<Lighting>&          lighting,
 const shared_ptr<Camera>&            camera,
 Stats&                               stats) {

    RealTime start;
    debugAssert(notNull(lighting) && notNull(camera));

    // TODO: store member pointers to the arguments
    // so that they can propagate inside the callbacks

    // Build the TriTree
    start = System::time();
    m_triTree.setContents(surfaceArray);
    stats.buildTriTreeTimeMilliseconds =
       float((System::time() - start) / units::milliseconds());

    // Allocate the image
    m_image = Image::create(settings.width, settings.height,
        ImageFormat::RGB32F());

    // Render the image
    start = System::time();
    const int numThreads =
        settings.multithreaded ? GThread::NUM_CORES : 1;
    traceAllPixels(numThreads);
    stats.rayTraceTimeMilliseconds =
        float((System::time() - start) / units::milliseconds());

    // TODO: Fill out other stats
    shared_ptr<Image> temp(m_image);

    // TODO: Reset pointers to NULL to allow garbage collection

    return temp;
}
```

```
void RayTracer::traceAllPixels(int numThreads) {
    GThread::runConcurrently2D(...);
}


shared_ptr<Surfel> RayTracer::castRay
  (const Ray& ray,
   float      maxDistance,
   bool       anyHit) const {

    // Distance from P to X
    float distance(maxDistance);
    shared_ptr<Surfel> surfel;

    if (m_settings.useTree) {

        // Treat the triTree as a tree
        surfel = m_triTree.intersectRay(ray, distance, anyHit);

    } else {

        // Treat the triTree as an array
        Tri::Intersector intersector;
        for (int t = 0; t < m_triTree.size(); ++t) {
            const Tri& tri = m_triTree[t];
            intersector(ray, m_triTree.cpuVertexArray(), tri,
                        anyHit, distance);
        }

        surfel = intersector.surfel();
    }

    return surfel;
}
```

9.3 The Rendering GUI

Your `RayTracer::render` method will take a relatively long time to compute an image, perhaps several minutes. If you invoke it from `App::onGraphics3D`, then it will run every time that the screen needs to refresh. That will make it appear that your program has crashed because it will be extremely slow. So you should only invoke `render` when the user presses the "Render" button.

I built the required GUI for this project by adding the following to `App.h`:

```
protected:
    GuiDropDownList*      m_resolutionList;
    RayTracer::Settings   m_rayTracerSettings;
    RayTracer::Stats      m_rayTracerStats;

public:
    void onResolutionChange();
    void onRenderButton();
```

and code including the following in `App.cpp`:

```
void App::onResolutionChange() {
    TextInput ti(TextInput::FROM_STRING,
         m_resolutionList->selectedValue().text());

    m_rayTracerSettings.width = ti.readNumber();
    ti.readSymbol("x");
    m_rayTracetSettings.height = ti.readNumber();
}


void App::makeGUI() {

     ...
    m_resolutionList = rtPane->addDropDownList
        ("Resolution", Array<std::string>("16 x 9", "160 x 90",
            "320 x 180", "640 x 360", "1280 x 720"),
         NULL, GuiControl::Callback(this, &App::onResolutionChange));
    rtPane->addCheckBox("Multithreaded", &m_rayTracerSettings.multithreaded);
    ...
}
```

You can time infrequent tasks by measuring the difference in `G3D::System::time` calls. For tasks that run every frame the `G3D::Profiler` and `G3D::StopWatch` classes are more appropriate, but you don't need that on this project.

There are many ways to display your image on screen. You could convert the CPU-image that you rendered into a GPU-image using one of the many `G3D::Texture` constructors, and then write code in `App::onGraphics2D` that draws a rectangle filled with that texture. For this approach, you may wish to use `G3D::RenderDevice::push2D` and `G3D::Draw::rect2D`.

Alternatively, you can use the `G3D::GApp::show` method to create a pop-up window with your image inside it (that's what I did, since it was really easy; see figure 2). That method just creates a new `G3D::GuiWindow` with a single `G3D::GuiTextureBox` inside it, so you could also create your own style of pop-up window, or embed the display within another part of your UI.

The drawback of using the built-in G3D GUI controls to display your image is that it is hard to add your own debugging handlers. For example, if you explicitly render your own UI in `onGraphics2D`, then you can write an `onEvent` handler that detects mouse clicks on them. It is often handy when debugging a ray caster to launch a 1×1 window, i.e., single-ray, render job when the user clicks on a pixel. This allows you to render the whole scene, and then set a breakpoint and re-cast the ray through one pixel while watching it in the debugger. It is more challenging to set up that kind of infrastructure if you are using a GUI control.

There are many ways to display the number of triangles in the scene. For example, you can print on-screen using a `G3D::screenPrintf` from `onGraphics3D`, an explicit call to `G3D::GFont::draw2D` from `onGraphics2D`, create a disabled `G3D::GuiNumberBox` or `G3D::GuiTextBox`, or create a `G3D::GuiLabel`. As with your other UI choices, you must decide how much you value ease of implementation, ease of use for the end-user, attractiveness, performance, and functionality. Mine looked like:

```
    rtPane->addNumberBox("Triangles",    &m_rayTracerStats.triangles);
```

Remember to briefly describe and support your UI choices in the report, just as you would for your other implementation decisions.

9.4 Functors (Closures)

The **functor** design pattern is a C++ class that acts like a C++ function. They are an approximation of the general programming language feature of a **closure**, which is just a function that has a persistent parent environment in which to retain state.

A regular C++ function can retain state between invocations. Local variables marked with the `static` keyword are initialized once, the first time that the function is invoked, and then retain their value on subsequent evaluations. This is convenient for memoizing results, for example. However, it has several drawbacks. One drawback is that the programmer has little control over the order in which variables are destroyed when the program shuts down. Another drawback is that it is hard for other parts of the program to access the state stored in these local variables.

C++ allows overloading of several operators. For example, we can write

```
Vector3 a;
Vector3 b;
...
a = a + b;
```

because `Vector3::operator+` overloads the default + operator to work on `Vector3` as well as numbers. In addition to the expected arithmetic operators, some surprising operators can be overloaded. These include the dereference operator, `->`, the array operator, `[]`, and the **function application operator**, `()`. This means that we can create a class that supports the application syntax of a function. For example:

```
class FakeFunction {
public:
    float operator()(int x, bool y);
};

...

FakeFunction f;
float z = f(3, true);
```

Such a class is called a **functor**. Because it is a class, we can add member variables and other utility methods. That allows the "function" to retain state between invocations, and for access to that state from other parts of the program. (This is how you create a **closure** in C++).

The `G3D::Tri::Intersector` class that you will use in the ray caster is a functor. It exposes the triangle and the barycentric coordinates of the closest intersection as public member variables. It provides additional information about the intersection through some helper methods. It also has some member variables that we won't use in this project that control how the `operator()` works.

Index