*CS 371 Project 3:*

# Recursive Rays

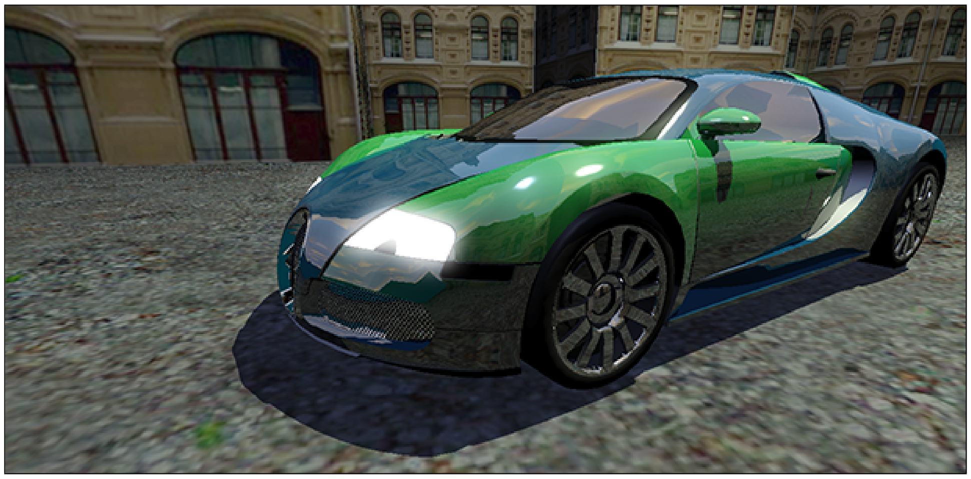

**Figure 1:** *A city scene reflected in the highly specular paint and chrome trim of a Bugatti Veyron. In this project you'll learn to simulate the reflections and shadows that make this image dramatic.*

## 1 Introduction

### 1.1 Overview

Turner Whitted's *ray tracing* algorithm recursively cast additional rays rays at each intersection point to create shadow, mirror reflection, and refraction phenomena. Thirty years later, his method remains the primary method for producing these effects in offline rendering in the film industry, and is beginning to gain traction in the real-time rendering community as well.

In this project, you'll extend your ray caster to be a full ray tracer with shadows, glossy reflection, mirror reflection, and emissive terms.

### 1.2 Educational Goals

On this project you'll learn how to:

1. Structure a recursive Whitted ray tracer
2. Work with both finite and infinite BSDF terms
3. Design a graphics scalability performance experiment
4. Manage asynchronous workflow between team members

### 1.3 Schedule

| | | | |
|---|---|---|---|
| **Out**: | Tuesday, | September 29 | 9:00 am |
| **Checkpoint**: | Thursday, | September 30, | 1:00 pm |
| **Due**: | Monday, | October 4, | 10:00 pm |

This is an easy, pair project. As with other projects, try to quickly cover the entire specification with stubbed out methods and provisional report text before refining any one area.

**Tip:** This project introduces experimentation after your software is complete. Plan your time accordingly.

As a reference, my solution required 300 statements and 300 comment lines including the reports (as reported by iCompile), plus several data files. If your codebase looks like it is going to be more than 1.5× larger or smaller, come talk to me because you may be on a bad path.

## 2 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 *last week's* project, with their permission.

On this project you may use any development style that you choose. You are not required to use pure side-by-side pair programming, although you may still find that an effective and enjoyable way of working.

During this project, you are not permitted to directly invoke the following classes and methods or look at their source code: the G3D `rayTrace` sample program.

## 3 Specification

Implement the following by extending the Eye Rays project.

1. Extend the `RenderSettings` class with:
   (a) a boolean to enable shadow rays
   (b) an integer for the number of recursive backwards tracing steps, where a setting of 1 gives direct illumination (the same results as last week) and 2 creates single-mirror reflections .
2. All GUI features from the previous project (Eye Rays), plus:
   (a) A checkbox to enable shadow rays
   (b) A number box with a slider to control the number of backwards steps
3. Restrict spot light illumination to the area within the spotlight cone.
4. Implement shadow ray casting.
5. Implement recursive mirror reflection ray tracing.
6. Implement emissive surfaces.

7. Implement glossy reflection.
8. Analysis of the runtime of the program as described in Sections 3.1 and 3.3.
9. A visually compelling scene of your own design that contains emissive surfaces, spot lights, shadows, multiple light sources, texture mapping, and mirror reflections.
10. Create the documentation reports specified in Sections 3.2 and 3.3.

### 3.1 Experiments

The goal of the performance experiments is to determine how algorithmic changes affect the runtime of the program.

While we can provide theoretical analysis of the algorithm, that is often useless in practice, which is why it seldom appears in computer graphics research papers. The scenes and resolutions that we care about may not be well-represented by the asymptotic behavior. Architectural issues such as cache behavior and branch coherence are hard to predict for real scenes, where the input statistics are unknown.

By this point in your computer science career, you should expect that exhaustively processing array will give roughly linear performance in the length of the array, that any kind of tree will give asymptotically logarithmic performance, and that increasing the number of threads for this kind of problem will give an asymptotically linear speedup. Your experiments should verify this; it is one of the first tests that your data structures are implemented correctly.

What you should be thinking about and attempting to test in your experiments are the non-asymptotic, real-world factors. At what point does the tree *actually* outperform the array? For small data sets, the constants might favor the array. Do you observe the impact of the data set no longer fitting in cache for a certain image or scene size? What impact does `TriTree` have on the size of the data in memory? Is it linear? What is the overhead of running multiple threads that prevents you from getting a purely linear speedup? What are scenarios where one thread per core would not increase performance over a single thread? What are scenarios where having *more* threads than cores would continue to improve performance.

**You should explain experiments that you would perform for important questions such as these, and actually perform a *few* of them.** I'm not giving you a specific check list of questions this week. I want you to take responsibility for the time and space performance of your program and investigate it following your own intuition. On future projects your development time will be limited by your program's performance, so it is important to start thinking about characterizing and measuring performance–that's the first and most essential step towards optimization!

### 3.2 Checkpoint (Thursday 1:00 pm)

For this checkpoint, commit to Subversion:

1. A template of your report, including a placeholder plot with temporary data but the formatting that you plan to use for final version.

2. An implementation of mirror and shadow rays, and use of the full `G3D::SuperBSDF` for shading. It need not work correctly or use the new `RenderSettings` options, but should represent your best effort to get it right in about an hour of programming.

You do not need to implement the GUI or perform experiments for the checkpoint.

## 3.3 Report

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

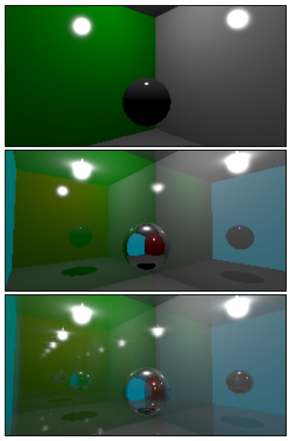

**Figure 2:** *The "Mirror Box" with 1, 2, and 7 backwards bounces.*

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.
3. Discuss any known errors in your program, and how you identified and attempted to correct them.
4. A plot depicting time to render vs. number of triangles for a heightfield using at least the following variants on the ray casting algorithm:
   - Using array search with a single thread at 320 × 200
   - Using array search with with multiple threads at 320 × 200
   - Using tree search with a single thread at 320 × 200
   - Using tree search with multiple threads at 320 × 200
   - Using tree search with multiple threads at 160 × 100
   - Using tree search with multiple threads at 640 × 400
   - Using tree search with multiple threads at 1280 × 800

   See the advice section for notes on how to perform an appropriate set of experiments. These graphs should have properly labelled axes (including units and the number of threads launched) and appropriate trendlines. Put all of the plots on the same image so that it is easy to compare them. Use colors/patterns that will be visually distinguished if printed in grayscale or viewed by a color-blind observer. Add a brief paragraph of concluding remarks based on the data.

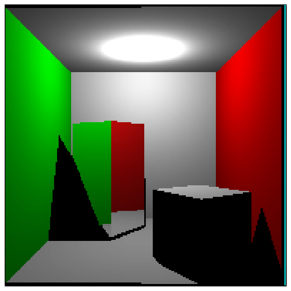

**Figure 3:** *The Cornell Box with reflections and shadows.*

5. Conclusions from your performance experiments.
6. Show pictures of the following scenes rendered with ray tracing:
   (a) The Cornell Box scene, as shown in Figure 3.
   (b) The Mirror Box scene with 1, 2, and 7 backwards bounces, as shown in Figure 2. This scene is available on the web page.

   (c) A visually compelling scene of your choice that demonstrates all of the features of your program, such as Figure 1. This need not be an entirely *new* scene; you can extend a scene that you or another student created for a previous project. Remember to commit the scene file and anything not in `cs-local` that it needs to run to the `data-files` directory.

7. **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)? Rate the educational value relative to the time invested from 1 (low) to 5 (high).

## 4 Evaluation Metrics

Recall that you are responsible for the entire project, including the correctness and clarity of code that you exported from Subversion to start the project, even if you were not the original author. So make sure that you know what code is in your project and that you understand it all!

To evaluate your project, I will check your project out from Subversion as of the deadline time. I will then run `icompile --doc` to generate the final report and documentation. I will read sections of your source code, the report in the `index.html` page generated by Doxygen, and sections of your documentation as generated by Doxygen. I may run your program, but I will primarily investigate its functionality by the description that you provide in the report. Note under this scheme, that the artifacts from your creation of and experimentation with the program are more important than the executable program itself. For many projects you can receive a favorable evaluation even if your program does not compile or execute.

As described in the *Welcome to Computer Graphics* document, I will evaluate your project in several categories:

- Mathematical correctness, including experimental methodology
- Adherence to the specification
- Program quality
- Report quality

Some questions I consider when evaluating the source code are: Is it possible for someone unfamiliar with it to find specific routines quickly? Is the code easy to understand? Does it make good tradeoffs between efficiency, clarity, and flexibility? Are data structures used effectively? Are the algorithms correct? Are the geometry and physics correct?

When evaluating the report, I consider: Do the experiments adequately explore the correctness, performance, robustness, and parameter space of the algorithm? Are known bugs made clear, along with how you tried to solve them? Are appropriate sources cited for algorithms and code? Does the overview documentation guide a reader to the relevant source code documentation? Is the architecture of the program clear?

The report and code should both be as concise as possible without compromising clarity. Use the papers we've read as examples of how to describe experiments compactly.

## 5 Implementation Advice

### 5.1 Getting Started

Start by exporting the previous week's code to new Subversion project. See the Tools handout from last week or refer to the Subversion manual for information about how to do this. Remember that you can use *anybody's* code from the previous week with their permission, so if you aren't happy with your own project as a starting point, just ask around.

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

```
svn co svn://graphics-svn.cs.williams.edu/3-RecursiveRays/recursive-<group>
```

Where you should replace `<group>` with your group name.

### 5.2 Workflow

Read the Programmer Workflow document again. Ask yourself and your partner if you are following the workflow style that it advocates. Don't be afraid to spend time talking about your workflow and schedule for the project–a little "meta" discussion up front could save you a few hours over the course of the project.

The shadow ray and mirror ray code are similar, so you will not miss anything if each partner only implements one of them. So I recommend that you perform that work simultaneously. Likewise, you can run many of the experiments in parallel and can write the report in parallel. Review the CS371 Tools document for advice when working with Subversion with a partner.

You should be able to complete the programming portion of this project in about 1/4 of the time that it took you to program the previous project by yourself. You should then be able to complete the experiments quickly and have a net development time of about half of what the previous project took. If you aren't seeing that kind of gain, stop and talk with your partner, me, or the game developers about what is going wrong.

### 5.3 Program Structure

I implemented this project by creating a series of helper methods for my `RayTracer::shade` method. These were `RayTracer::shadeDirect`, which included the shadow term and spotlight cone; and `RayTracer::shadeSpecularIndirect`, which cast recursive rays for mirror reflection. I chose that structure because it follows the math and anticipates a future need to simulate *diffuse* indirect illumination with another method. You may follow my design or create your own.

### 5.4 Glossy Reflection

This week you are allowed to use `G3D::SuperBSDF::evaluate`. So you can either alter your program to use that instead of implementing the scattering function evaluation on a surface sample, or you can extend your own scattering function implementation. Calling `evaluate` is easier and is likely to avoid introducing errors, but you may learn more if you write the scattering function yourself and doing so is enjoyable (for some people). Note that either way, you can look at the source code for `G3D::SuperBSDF::evaluate`.

Note that `evaluate` returns a `G3D::Color4`, and you need a `G3D::Color3`. The `G3D::Color4::rgb` method performs the conversion for you.

### 5.5 Emission

Perhaps the easiest way to render emission is to add it to the direct illumination term using the appropriate field of `G3D::SurfaceSample`.

If a pixel has a very bright value, you will simply see a bright pixel. Because of the way that lenses work, in a real camera or in your eye you would see a bright halo around that pixel. This effect is called **bloom** in computer graphics.

If you wish to implement bloom in your own project after you have completed the rest of the specification, look at the `G3D::Film` class, which is the same one that the preview renderer uses by default. It even contains a helper method for making a GUI to control the bloom and gamma effects. This is purely optional and you will not receive points for adding it to your program.

### 5.6 Spot Lights

The code that I showed briefly in lecture for determining if a point was outside a spot light's cone of influence was:

```
const bool outside =
    ((light.spotHalfAngle < halfPi()) &&
    (-dot(w_i, light.spotDirection) < cos(light.spotHalfAngle)));
```

Read the documentation to understand what the variables involved mean, and then walk through an example of this to make sure that you understand why this code works.

### 5.7 Shadow Rays

Compute shadow rays by casting a ray from the surface towards the light. You can cast the ray the other way around, but this makes it a little easier when you go to implement mirror reflection. Stop tracing when you reach the light–objects *behind* the light don't cast shadows!

If you cast from exactly at the surface position, sometimes you will intersect the surface itself due to finite precision roundoff in floating point representation. To avoid this, **bump** the ray slightly away from the surface by offsetting the starting point. I prefer to bump the ray by a small distance (such as 0.0001 m) along the geometric normal of the surface. Others bump the ray along the shading normal or the ray direction. Explain why you chose one of these over the others.

## 5.8 Mirror Reflection

I implemented mirror reflection by invoking `G3D::SuperBSDF::getImpulses` with a `G3D::SmallArray<SuperBSDF::Impulse, 3>`. A `SmallArray` has a similar interface to an `Array`, but it preallocates a fixed number of elements on the stack. This allows it to avoid heap allocation if the actual size of the array is within that fixed bound. This design trades a small amount of time when referencing elements and a small amount of heap space for a relatively large performance gain versus heap allocation for a large array.

I then cast a new backwards ray along the impulse direction, bumping the ray origin as I did for shadow rays.

Note that according to the documentation, you should not scale the light contribution by a projected area term for the impulses.

## 5.9 Experiments

When you perform experiments, it is important to hold all parameters constant except for the one you are intentionally varying. That means that you should render images with the camera in the same location relative to the heightfield, the heightfield filling the same portion of the screen, and at the same resolution (unless you are intentionally varying the resolution). You can generate heightfields at various resolutions by using the Photoshop Image/Image Size... command. It doesn't matter for this project whether you start at the largest or smallest.

**Tip:** Make sure you run your performance experiments on an optimized build!

What heightfield sizes should you use? That depends on the speed of your implementation and processor. Using small heightfields will minimize your experiment time, which is important because the ray casting algorithm can take hours to run. But for heightfields that are too small, random fluctuations in the testing environment (e.g., background processes), and the overhead of thread launch and startup memory behavior will drown out the actual long-term performance characteristics of the algorithm. You should therefore perform some preliminary trials to find a reasonable medium between these. The longest run will be for the single-threaded array implementation. Target a 15-minute render time for that algorithm.

Add some data points for tree tracing that are beyond those that would be reasonable to test with array access. That is, your graph should show a trend for both array and tree methods until the array performance seems to be well understood and is getting unwieldy to continue measuring. You can then extrapolate the array trend and continue to measure the tree trend explicitly.

**Tip:** Tools like Excel are convenient but not required for making plots. You can also draw them in Illustrator or PowerPoint if you need more control over formatting.

Assumingyou are reasonably confident that you have two computers with the same performance characteristics, you may run experiments in parallel on multiple machines. A good way to do this is to duplicate a few experiments to verify that they take the same amount of time on the different machines, and then run unique experiments from then on. This can substantially reduce the amount of time that you spend in lab.

### 5.10 Previewing Model Files

G3D comes with a program called "viewer" that can load most of the data files that G3D classes can load, including images, movies, 3D models, cube maps, fonts, and GUI themes. The program is in the `bin/viewer/viewer`.

### 5.11 Creating Compelling Scenes

I used several tricks to model the scene in Figure 1. You need not employ the same tricks that I did. But you should use your newfound modeling skills and bold exploration of the G3D documentation of the various G3D Specification classes to push your renderer beyond its nominal capabilities using clever scenes.

For example, The shadows under the car aren't perfectly dark–I achieved that effect using a shadow casting sun light and a second, non-shadow casting light to emulate the sky. Even though we don't know how to ray trace a cubemap, my car is still reflecting a cloudy sky. I did this by loading a cube map's individual faces and building a giant emissive cube around the scene.

# Index