CS 371 Project 3:

Recursive Rays

Figure 1: *A city scene reflected in the 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.*

Contents

1 Introduction **2**
- 1.1 Overview . . . 2
- 1.2 Schedule . . . 2

2 Rules/Honor Code **3**
- 2.1 Teams . . . 3

3 Individual Checkpoint **4**

4 Team Checkpoint **5**

5 Group Specification **6**
- 5.1 Experiments . . . 7
- 5.2 Report . . . 7

6 Implementation Advice **9**
- 6.1 Getting Started . . . 9
- 6.2 Workflow . . . 9
- 6.3 Recursive Rays . . . 9
- 6.4 Experiments . . . 9
- 6.5 Previewing Model Files . . . 10
- 6.6 Creating Compelling Scenes . . . 10

1 Introduction

1.1 Overview

Turner Whitted's *ray tracing* algorithm [Whitted 1980] recursively cast additional rays rays at each intersection point to create shadow, mirror reflection, and refraction phenomena. This is the core idea behind the family if ray-based techniques that dominate realistic rendering today.

In this project, you'll extend your direct-illumination ray caster to be a full ray tracer with shadows, mirror reflection, and specular refraction. You'll then design and perform scalability experiment. In doing so, you'll increase your ability to translate mathematics into concrete software implementations and gain new skills for software and algorithm analysis through experimentation.

1.2 Schedule

Available:	Monday,	September 25	8:00 pm
Individual Checkpoint:	Wednesday,	September 27,	12:00 pm
Team Checkpoint:	Wednesday,	September 27,	12:00 pm
Due:	Monday,	October 3,	12:00 pm

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

This is a moderately challenging project. You'll probably only spend about 20 minutes implementing new features. However, beware that performing experiments and documenting their results in the report can take substantial time–this includes time adding infrastructure for the experiment, time for the experiment to execute, and time analyzing the results.

As a reference, my solution required 300 statements and 300 comment lines including the reports (as reported by iCompile), plus several data files. Most of this was implemented for the previous week's project. 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

Do not discuss material related to the individual checkpoint with any other student until the deadline is past. You may discuss it with me and other faculty, and use *any* other resources available except where prohibited below.

After the individual checkpoint, I encourage you to students in other groups to share strategies, programming techniques, and data files. You should not look at any group's code for this project. You may look at and use anyone's code from *last week's* project, with their permission.

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.

2.1 Teams

Tip: Note that teams *and* specifications are growing, so you'll need to devote more time to planning and synchronization.

1. hanrahan: James R., Cody, Lily, Qiao, Donny
2. jensen: Scott P., Josh, Tucker, Parker
3. schlick: Jonathan, Lucky, April, Dan E.
4. kajiya: Alex, Nico, Michael, Dan F.
5. veach: Owen, Dan S., James W., Greg

3 Individual Checkpoint (Wed 12pm)

Submit on paper a design for the important elements of the `RayTrace` class by extending the pseudocode in listing 1. Pay careful attention to structural types (e.g., name points with capital letters, indicate vectors with explicit types or hats and arrows, distinguish between a surfel and its location) and units. Abstract uninteresting details such as iteration mechanisms and pointer syntax. This may be hand-written if you print clearly. You can follow any clear and consistent pseudocode notation that you wish, it doesn't have to be the same as mine. Note that you may need to alter some of the code from my example.

I'll evaluate the technical correctness, clarity, and design. A good design for a mathematically-intense program should draw a clear correspondence between the math and the program through appropriate method boundaries and variable names. It should also reveal details that affect the computation but which may not appear in the mathematics, such as how we handle impulses.

I expect that this will take you about half an hour if you have completed the reading and taken good notes in class. If this looks like it will take you more than an hour, please just hand in whatever you accomplish in an hour and make an appointment to see me so that we can review your approach.

Keep a copy of your design to discuss with your team after the checkpoint.

```
render() : Image
   iterate over pixels (maybe concurrently), launching traceOnePixel

traceOnePixel(x : integer, y : integer) : void
   generate ray (P, w) through the pixel center
   set the pixel to Li(P, w)

Li(P : R^3m,  w : S^2) : W/(m^2 sr)
   trace P + tw to the first intersection, described by surfel
   wo = -w
   return Lo(surfel, wo)

Lo(surfel, wo : S^2) : W/(m^2 sr)
   X = surfel.location
   for each light in lighting.lightArray:
       Y = light.cframe().translation
       wi = (Y - X).direction()
       ...
   return surfel.emittedRadiance(wo) + ...

...
```

Listing 1: *Starter pseudocode for this week's renderer.*

4 Team Checkpoint (Thu 1pm)

Submit on paper a schedule for your development, experimentation, and documentation process. Recall that a schedule ties together team members, times at which work is done, and what the tasks are, as shown in figure 2. The individual elements should be no longer than two hours. Keep a copy up to date throughout the project (it need not be in SVN–you can use any scheduling tools that you wish).

I'm not requiring your report to be drafted at a specific time any more, but by now you should recognize the team workflow advantages of establishing your report and journal early in the week. Likewise, I recommend that you get in the habit of reaching the crux of the project before the scheduled lab session. If you achieve this, I'll be available to answer questions on the most challenging aspects as you reach them, and you may be able to entirely avoid working on the weekend.

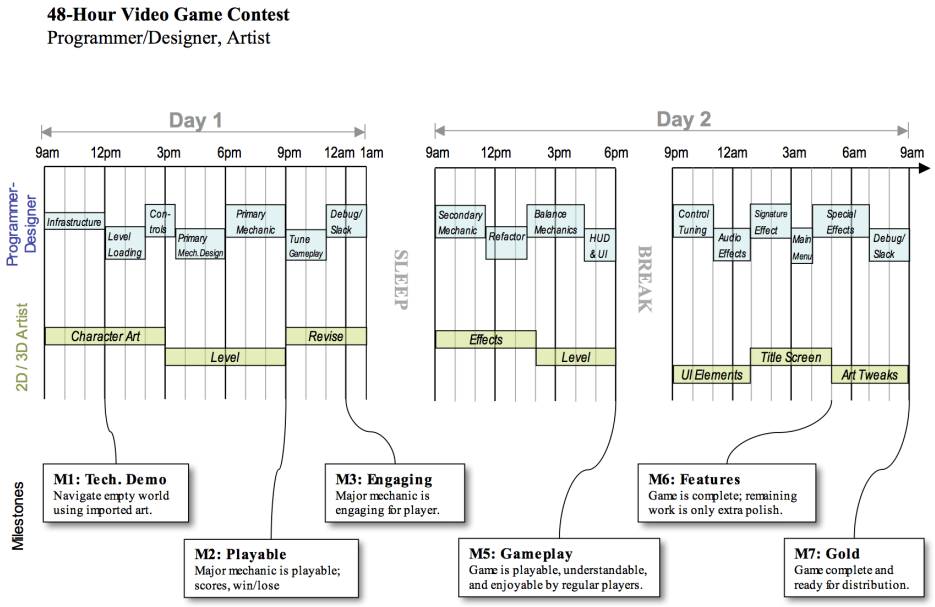

Figure 2: *Reference schedule for a two-person team working on a weekend game jam project, such as Ludum Dare. Note that tasks are assigned to a specific team member, at a specific time. This allows close coordination and ensures that there are no schedule conflicts. On a larger team, the team leader can also check after each task to make sure that the process is in control. Note that you should not be working this much on a 371 project!*

5 Group Specification (Mon 10pm)

Complete the following by extending the *Eye Rays* project from last week:

1. Extend the `RayTrace::Settings` class and its GUI representation with:
   (a) a boolean to enable shadows
   (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.
   (c) an integer for the number of primary rays per pixel
2. Extend the `RayTrace::Stats` class and its GUI representation with ray counts for:
   (a) Primary rays
   (b) Indirect (impulse) rays
   (c) Shadow rays
   (d) Total rays
3. Restrict spot light illumination to the area within the spotlight cone.
4. Shadow ray casting.
5. Recursive BSDF impulse (reflection and refraction) ray tracing.
6. Emissive surfaces (if you did not do so last week).
7. If there are multiple rays per pixel, distribute them uniformly at random within a pixel and average the result (this is not optimal sampling, but is easy). If there is one ray per pixel, cast it through the center.
8. Analysis of the runtime of the program, as described in Sections 5.1 and 5.2.
9. A set of simple images that demonstrate that spot light, emissive, shadow rays, and impulses are working correctly individually (e.g., a Cornell Box with a mirror sphere). These are targeted experiments for showing that features exist, and for debugging them. This is one form of scientific visual communication (the plots in the report are another!)
10. A visually compelling image (or set of images) of your own design that demonstrate that spot light, emissive, shadow rays, and impulses are working correctly. This should be an image that shows that the features scale and interact well. The image needs to contain specific features, but should stand on its own visually. A layperson should be impressed by the image, and anyone looking at it should immediately understand what is shown and that the rendering is good.
11. Perform the experiments described in section 5.1 and create the documentation report specified in section 5.2.

Tip: Consider the whole composition and not just the scientific elements. For example, if you render a room, then you'll need not only lights, reflectors, and shadow casters, but other elements to tie it all together such as a rug and some chairs.

5.1 Experiments

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

We can provide theoretical analysis of the Whitted ray tracing algorithm; it is clearly linear in the ray depth, number of pixels, and rays per pixel, and for suitable assumptions logarithmic in the number of triangles in the scene. The constants might matter significantly at even the relatively large number of triangles and rays considered and the assumptions needed for the analysis might not be satisfied in practice. 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. This will be the case for this project. Your experiments should trivially verify this; it is one of the first tests that your data structures are implemented correctly. (Although make sure that you're using a big enough scene and image, such as Sponza at 640×400.)

What you should be thinking about and attempting to test in your experiments are the non-asymptotic, real-world factors. For example, 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 observing 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?

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.

5.2 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.
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 heigh tfield using *at least* the following variants on the ray casting algorithm:

- Using array search with a single thread at 320×180
- Using array search with with multiple threads at 320×180
- Using tree search with a single thread at 320×180
- Using tree search with multiple threads at 320×180
- Using tree search with multiple threads at 160×90
- Using tree search with multiple threads at 640×360
- Using tree search with multiple threads at 1280×720

These plots 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. Note that you do not need to use a spreadsheet to draw plots, and often that is the worst way to accomplish visualization of data (see Tufte's *Visual Display of Quantitative Information* [Tufte 1986]), particularly if you use the default settings. I frequently draw mine "freehand" in presentation or illustration software (being careful to measure distances correctly).

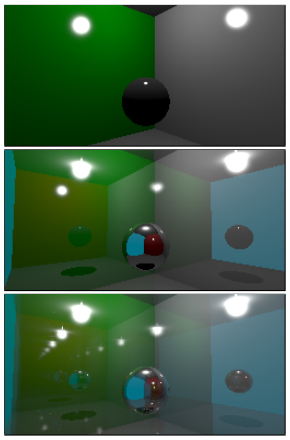

Figure 3: *The "Mirror Box" with 1, 2, and 7 backwards bounces.*

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 4.
   (b) The Cornell-like Mirror Box scene with 1, 2, and 7 backwards bounces, as shown in Figure 3.
   (c) Individual targeted visual experiments showing that each illumination term is implemented correctly.
   (d) 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 `mac-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 (per-person, on average), including scheduled lab, on **required** elements, i.e., the minimum needed to satisfy the specification.
   (b) How many additional hours you spent on **optional** elements, such as polishing your custom scene or adding new features.
   (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) A list of what you learned while working on this project. Make sure all team members contribute to this list.

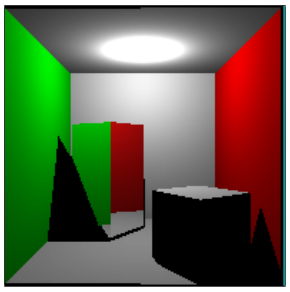

Figure 4: *A version of the Cornell Box with reflections, shadows, and a specularly reflective object. Your results may differ slightly depending on where you put the light source.*

6 Implementation Advice

6.1 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/371/3-RecursiveRays/recursive-<group>
```

Replace `<group>` with your group name.

6.2 Workflow

Read the Padraic Hennessy's 2010 Programmer Workflow document (which was assigned reading last week) again. Ask yourself and your partners if you are following the workflow style that it advocates. Spend time talking about your workflow and coordination (not just schedule) for the project–a little "meta" discussion up front could save you a few hours over the course of the project. Actively critique each other's workflow. Suggestions such as "let me show you how to do that without taking your hands off the keyboard," "this would be more effective to debug using gdb," and "we're stuck in a rathole–let's patch around this bug and move on for now," can save us *hours* over the course of a project.

You already completed the key design elements for this project individually. So consider programming in parallel, with each team member working on a different method or part of the report. Likewise, you can run many of the experiments in parallel and can write the report in parallel.

6.3 Recursive 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 move on to implement impulse scattering. Stop tracing when you reach the light–objects *behind* the light don't cast shadows!

Note that there is an optional argument for `TriTree::intersect` to stop tracing at any intersection, not just the first one. Consider when you might use this, and why it is useful.

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.

6.4 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

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

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.

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 plot 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.

Assuming that you 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.

6.5 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`.

6.6 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.

References

COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. 2008. Subversion complete reference. In *Version Control with Subversion*. O'Reilly, ch. 9. http://svnbook.red-bean.com/en/1.5/svn.

ref.html.

TUFTE, E. R. 1986. *The visual display of quantitative information*. Graphics Press, Cheshire, CT, USA. 8

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June), 343–349. 2

Index

asymptotic performance, 7

bump, 9

checkpoint, 4, 5
Cornell Box, 8

experiment, 7, 9

height field, 7

Mirror Box, 8

RayTrace::Settings, 6
RayTrace::Stats, 6
render, 4
report, 7

shadow, 6
Sponza, 7
spot light, 6
Subversion export, 9

teams, 3
thread, 7
traceOnePixel, 4
TriTree, 7, 8

viewer, 10

Whitted ray tracing, 7
workflow, 9