# Path

CS224 Project 4  
Out: February 17  
Due: Feb 26 11:59pm

## 1 Introduction

In this project you will implement path tracing, an algorithm proposed by Kajiya in his seminal 1986 paper *The Rendering Equation*.

Path tracing is a simple but powerful Monte Carlo rendering algorithm. Unlike naive Monte Carlo raytracing, path tracers just send out one recursive ray per intersection. Their final estimate for each pixel is the average of many runs of the same recursive tracing algorithm.

Path tracers are unbiased and consistent, capturing all effects modeled by the rendering equation with a single, straightforward algorithm. However, they can take a very long time to converge.

In this assignment you will implement a simple recursive path tracer, using the book, and the raytracer code, as your guide. Then you'll develop a feature for your path tracer on your own.

## 2 Requirements

This assignment is out of 90 points.

For a grade of a B (72/90):

1) Build a model world that should exhibit reflection, refraction, and color bleeding.

2) Implement a path tracer that handles:
   - (10pts) Recursive tracing, Reflection, Refraction
   - (18pts) Full global illumination (color bleeding, caustics)
   - (18pts) Path termination using Russian roulette
   - (18pts) Event-splitting (gather both direct and indirect contributions at each bounce in the path)
   - (8pts) Q&A in your README

3) Show a rendering of your world and others. Submit images of your renderings along with your code when handing in.

The output of your path tracer should match the demo: `/course/cs224/bin/path_demo`. Note that in Monte Carlo-based rendering algorithms, incorrect code can produce images that still look pretty good. Be careful with your probabilities!
2.1 Special Features

We will give you credit for each feature you implement. Accumulating 9pts gets you an (81/90); 18pts gets you an A (90/90).

- (Value: 3) Add attenuation to refracted paths, so that a path going through a gray glass crystal ball ends up less bright if it goes a long way through the glass rather than a short distance.

- (Value: 3) You’re going to trace many rays per pixel and average them. Rather than scattering them randomly over the sensor square, try “stratified” sampling, in which you divide the sensor area into a grid, and then pick one or two or twenty samples uniformly in each grid square. Evaluate the result in your README.

- (Value: 3 for absorption, 6 for emission, 9 for scattering) Implement participating media, so that your path tracer can render effects like fog, smoke and fire.

- (Value: 9) When during reflection you trace a recursive ray from a surface intersection, you can choose it randomly from the hemisphere; alternatively, you can choose it in a way that’s approximately proportional to the BRDF, and use importance sampling. G3D has functions to help you with this.

- (Value 12) Image-based lighting. Compose a mesh sitting on a plane into a live-action photograph, and use your path tracer to light it in such a way that the mesh looks like it’s part of the photographed scene. Include a copy of the scene and renders in your handin.

- (Value: 15) For the physics-savvy: Accurate prismatic effects, try implementing wavelength variation of the refractive index. With only RGB to work with you’ll need to find a way represent the entire light spectrum, and then remember what wavelength your ray has while tracing.

- (Value: 18) Another for the physics-savvy: try implementing a polarized path tracer, in which the polarization information is transported along the path, and apply Fresnel’s laws correctly. Then make a picture of a hall of mirrors.

This list isn’t exhaustive, so brainstorm your own ideas too! The point of this feature is for you to try something that isn’t already coded for you in the book. Some ideas to get you thinking:

- BRDF importance sampling is the most common technique, but there are different kinds of importance sampling (look at the terms in the rendering equation).
  

- Instead of fixing 0.5 as probability of terminating (returning the emissive term), what if you could intelligently vary this number? This is one of the basic ideas leading up to Metropolis light transport, which is good for discovering difficult-to-find light paths.

- Implemented the algorithm described in “Sorted Deferred Shading for Production Path Tracing” (Eisenacher et al. 2013).
  
  - [http://www.yiningkarlli.com/projects/takuarender/](http://www.yiningkarlli.com/projects/takuarender/) is a gorgeous renderer implemented by a Cornell grad student. It has a lot of nice features like multiple BSDF models, hierarchical adaptive sampling, and adaptive antialiasing. Check out the renders! Take a look at other fancy renderers like Mitsuba too.

If you’re not sure if your idea is feasible, ask a TA!
2.2 Community Contribution Scenes

Community contribution points are awarded for scenes that you contribute so that the rest of the class can use. Put your scenes in /contrib/projects/gkl/cs224/scenes/, and share an image on Piazza!

Make sure the scene is accessible to others by running:

```bash
chmod a+r <myscene>
```

2.3 Q&A

Please answer the following questions in your README. This is worth 8pts.

1. Does the variance in your image appear to change if you make the Russian Roulette termination probability larger or smaller? Explain why.

2. Each time you divide by a probability (PDF) in your code, list it in your README and justify why you did so. For example: “On line 1337, I divide by the PDF since the samples were drawn from a non-uniform distribution.”

3 Getting Started

Path tracing is easy to implement if you have a solid understanding of the mathematics, so we recommend that you take a look at the Render HW first.

As previously mentioned, the book contains an implementation of a simple path tracer in G3D. You can find the walkthrough in the “Render Practice” chapter of the course textbook (Hughes, et al., Ch 32). Note that the code in the book is written for an older version of G3D (8.x) than the version the course is using (10.x), so some of the APIs you need may be different. Thus, you should use this chapter as a rough guide and understand the concepts instead of copying and pasting code.

Consider skimming Kajiya’s Rendering Equation paper, which introduces path tracing near the end (section 6). The paper is linked on the course web page.

To help come up with a special feature, you may find it useful to re-read (or skim through) the chapters in the textbook that deal with the physics of lighting. Google may also be your friend.

Finally, note that even though it may take tens of thousands of runs for a path tracer to converge, you can usually spot rendering bugs after about 30-50 passes, which takes only a few minutes. Start with a lower-resolution window to cut down iteration time.

4 Support Code

Support code can be copied from /course/cs224/asgn/path.

It makes use of G3D library, which gives you a lot of stuff for free, like scene intersection and returning the specular reflection and refraction impulses (vector and magnitude) at a given point. We have also made available the support code for the “Ray” assignment from last year, which can be copied from /course/cs224/asgn/ray. The Ray support code implements the majority of a basic raytracer so you may find it useful for figuring out the G3D API. For everything else, you can find G3D documentation online here:
You may develop using any editor you like, although we strongly suggest Qt Creator for its static completion ability. To run Qt Creator, run `qtcreator5.2.0` from a terminal and open the project `src/path.pro`. It's not too difficult to develop on your own machine by installing G3D and QT Creator, but make sure to test compile & run on a department machine before you hand in, because that's what we'll be grading on.

To compile via the command line, run `qmake` and then `make`.

You can run your path tracer in the command line. If you would like to use a scene other than the default, you can specify the path to a scene file as the first argument to the program. Several scene files (ending in `.scn.an`) are provided in `/contrib/projects/g3d/cs224/scenes/`.

We provide a simple GUI that you may find helpful in debugging your path tracer one piece at a time, and for saving images. The only function you need to modify for basic requirements is the `PathTracer::trace()` function. Feel free to modify other classes to suit your extra features.

In the World class, light sources are represented by light-emitting surfaces. Your path tracer will use emissive surfaces (which have a finite area) instead of virtual lights (which have infinitesimally small area) to light your scene. `World::emissivePoint()` will help you randomly select a point on a random light for direct lighting calculations. Assume all emissive points to emit light in a hemisphere rather than a full sphere.

To obtain the power of a Triangle that is an emitter, use the following code:

```cpp
shared_ptr<UniversalMaterial> material = dynamic_pointer_cast<UniversalMaterial>(
    tri.material());
material->emissive().mean(); // power of the light source
```

### 4.1 `shared_ptr<T>`

G3D makes extensive use a C++ smart pointer class called `shared_ptr`. We expect some students may not have run into smart pointers before, so we provide a brief overview here. If you’re familiar with smart pointers, you may skip this section.

A smart pointer is an object that can be used like a normal pointer, but provides some useful feature in addition. `shared_ptr`’s ‘useful feature’ is automatic memory management, using a technique called reference counting.

A `shared_ptr`’s reference count is the number of `shared_ptr`S that currently refer to the same thing. The refcount is incremented when one `shared_ptr` is copied to another; the refcount is decremented when a `shared_ptr` is destroyed or changed. When the refcount reaches zero, the `shared_ptr` deletes the memory it points to.

`shared_ptr` assumes that you only access the memory through a `shared_ptr`: therefore, if there are no `shared_ptrs` referring to the memory, then your program no longer knows about the memory, so it’s safe to delete it. This means you should never directly access the pointer wrapped by the `shared_ptr`, even if the `shared_ptr` API allows you to do so.

You can find more about `shared_ptr`s on Google. Note G3D (in particular, G3D9) aliases some `shared_ptr` types as `Type::Ref`.

**TL;DR - `shared_ptr<Type>` (or, sometimes, `Type::Ref`) acts like `Type*`, except you don’t need to call delete yourself. Never mix `shared_ptr<Type>` with a raw `Type*`.**

Finally, you may also get weird-looking errors if you accidentally use `someptr.foo` instead of `someptr->foo`. 
5 Handing In

You are required to hand in a README file along with your project. Please hand in your readme as a plain text (.txt) file.

You may submit your written answers as plain text or a PDF file. If you don’t place your written answers in your README, be sure to write in your README where the grader can find your answers.

**Do not hard code any paths in your program!**

To hand in your program, in a terminal type `cs224_handin path`. 