15418 Project Proposal

Parallel Monte Carlo Path Tracing

Cassidy Diamond (cdiamond) and Akintayo Salu (asalu)

URL: https://dialmond.github.io/15418-project/

Summary

We will be implementing Path Tracing using Monte Carlo methods to render photo-realistic images. We will parallelize using both CUDA (with NVIDIA GPUs) and OpenMP, and compare results to a sequential version. With promising progress, we will extend to combine these parallelization methods. Finally, we will define a standard of "correctness" for the images produced and compare scores to sequential, benchmark renderings.

Background

The inspiration for this project comes from another course I'm taking this semester, 21-387 Monte Carlo Methods and Applications, which regularly showcases examples of Monte Carlo algorithms being used for various real-world applications, including photo realistic image generation. In Image generation, the goal is to render a 2D image of a 3D scene by simulating the physical properties of light, given information about the materials of the scene, the camera, and the illumination sources.

Path tracing is one such method for performing this. In a naive implementation, for each pixel in an image, we sample *n* rays and trace the path of the ray as it reflects off of various surfaces in a scene. A single ray contributes colors from where it initially hits, as well as from where it reflects. The rendering equation (standard to image generation) is used to produce the final ray color. Finally, we then take the average color returned among all rays for the final pixel color. (Pseudocode included in the appendix, figure 1). It's a Monte Carlo method as we calculate the pixel colors based on history-independent samples of rays, which lends itself well to parallelization.

Path tracing differs from traditional ray tracing in that we follow the paths of our rays, rather than simply stop at their endpoints. This allows us to produce more accurate renderings that are more responsive to complex, multi-source illumination and object materials.

The naive algorithm gives us two possible dimension to parallelize over: pixels, as well as samples. However, certain scenes can not be accurately modeled by this algorithm, or are more antagonistic to uniform random sampling. Consider, for example, scenes in which light comes from a very small point souce, like a crack in a door. Uniformly random rays are unlikely to bounce into illumination. Other problems in accuracy arise too, when we consider extremely reflective surfaces, or other complex illumination sources.

A more complex version of this algorithm, Metropolis light transport, takes this further, requiring fewer samples to achieve less noise. Here, new sampling paths are modifications of previous, "successful"/colorful paths, which introduces an additional dependency on parallelization in that samples are not all unique. An example distinguishing these two results is shown in the appendix (figure 2).

Challenges

As stated before, naive Path Tracing can be at first an effective method of photo realistic image generation, but falls short for more complicated scenes. We therefore seek to use multiple variations of this algorithm, including Metropolis light transport and importance sampling, while noting that this introduces additional dependencies between samples and makes parallelization more complicated. Furthermore, we should consider how we access the memory for objects in our scene, as small changes to our random rays may produce significant changes in which objects – and therefore, regions of memory – need to be accessed.

We consider the tradeoff between accuracy of the image, computation cost in rays, and ability to parallelize. It is likely we will explore multiple strategies as opposed to purely "random" sampling.

Finally, different architectures give us different challenges. With CUDA, we have the ability to use more threads, but we need to pay closer attention to ensuring uniform behavior in pixels or samples so that we can effectively utilize warps. With OpenMP, we may be able to have our rays exhibit more distinct behavior, but we then need to consider the tradeoff between having fewer samples to do work.

Resources

We will use the GHC machines to access their NVIDIA GPUs, and for uniform benchmarking and testing. We will likely start by modifying the code from Scotty3D, which is provided to students taking the Computer Graphics course at CMU, or use an initially sequential version. However, neither of us have taken this course before, so we will need to familiarize ourselves with it initially. Additionally, we will follow papers like Metropolis Light Transport and Optimally combining sampling techniques for Monte Carlo rendering as guides to implement variations on path tracing ourselves. We will certainly find and utilize more resources as the project continues, which will be noted.

Goals and Deliverables

Our primary goal is to showcase photo realistic image generation, and, a considerable performance improvement in computation time, which will be achieved through the use of CUDA and OpenMP implementations of path tracing we will create.

Specifically, we will:

- Implement a CUDA and OpenMP version of the path tracing
- Introduce several benchmark scenes and create benchmark images that we will compare our implementations against. A notion of "correctness" is itself interesting to define, as Monte Carlo image generation can introduce multiple kinds of noise to an image. We will decide on precise, measurable quantities that we can benchmark against that mark a successful, or accurate rendering.
- Explore different variations in the algorithm, such as Metropolis Light Transport and Importance Sampling, to produce better results with fewer samples, or better results for more complicated scenes. We will explore tradeoffs between correctness and parallelizability.
- Document our findings, produce and present renderings of scenes using various techniques, and show speedup graphs to report our gains due to parallelization.
- Compare results across CUDA and OpenMP and determine what the strengths of each implementation strategy are in benchmark tests and developer experience.

Beyond this, with more time we will continue to explore different variations in sampling strategies in more depth (expand on MLT, Importance Sampling, and other methods). With less time these variations will be perhaps only superficially covered. With more time we will also explore combining both CUDA and OpenMP as Professor Jia suggested to further divide our workload and explore other dependency structures, or render images with a moving camera.

Platform Choice

We will be developing our code in C++ due to its prevalence in image generation and our experience with it in the class. We will use CUDA and OpenMP as natural extensions of the concepts we've learned during the first part of the semester. NVIDIA GPUs are the industry standard for image rendering, and will likely give us the best performance and speedup.

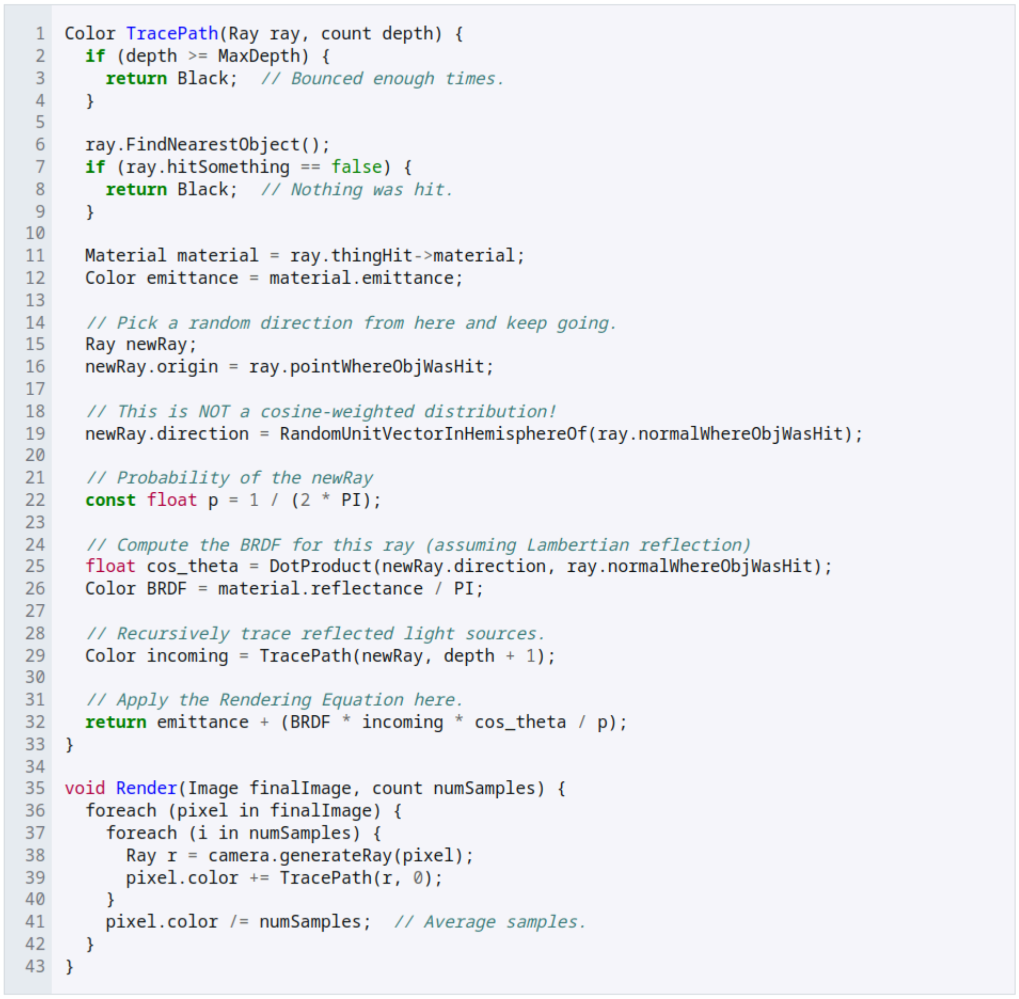
However, OpenMP will provide us unique challenges in parallelization strategies (as well as strengths compared to CUDA, like independence of threads), an opportunity to explore additional architectures and support machines without GPUs, and a source for comparison in our speedup graphs.

Schedule

Our schedule will almost certainly change as we explore image generation – a topic neither of us have prior experience in – but we initially expect to follow a similar schedule to the one below:

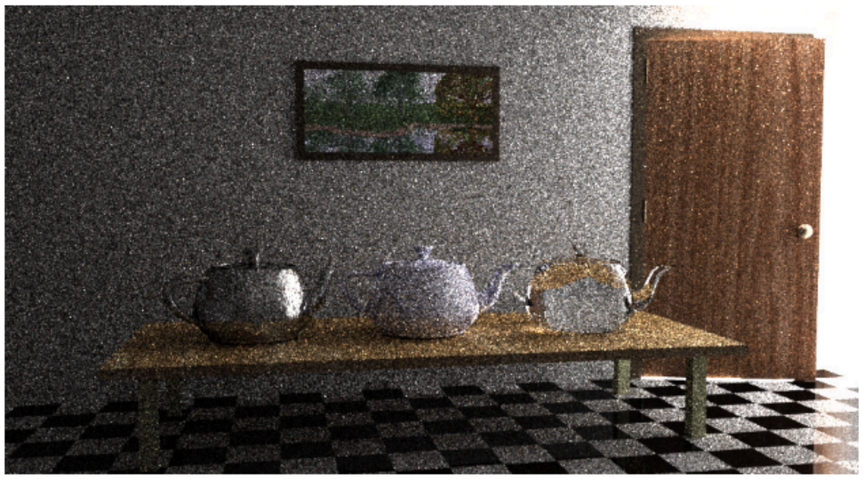
- Week 1: Research path tracing more and explore other projects and papers that have done GPU and OpenMP parallelization before – again, we're both new to image generation. We may not even use the Scotty3D code.
    - Create an initial sequential path tracing algorithm to benchmark against.
    - Begin work on parallelizing using OpenMP
- Week 2: Finish parallelization using OpenMP. Create initial scenes we can use for benchmarking and compare sequential and parallel versions. Start working on using CUDA
- Week 3: Finish parallelization using CUDA. Create more in depth comparisons of our methods in preparation for our milestone report. This will likely include a testing suite and more adversarial scenes.
- Week 4: Milestone report. Explore changes in our algorithm that increases performance, or performs better on adversarial scenes.
- Week 5: Buffer time for previous weeks, start working on on nice to have features if time.
- Week 6: Finalize report.

Appendix

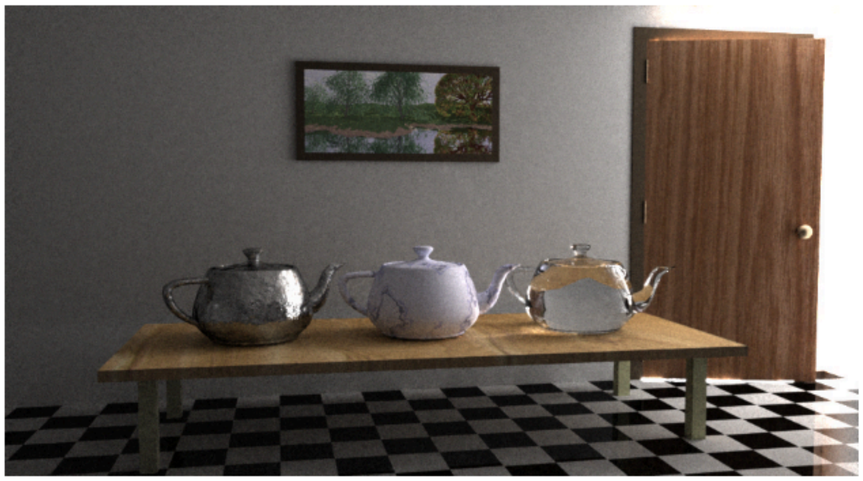


```
Color TracePath(Ray ray, count depth) {
  if (depth >= MaxDepth) {
    return Black;  // Bounced enough times.
  }

  ray.FindNearestObject();
  if (ray.hitSomething == false) {
    return Black;  // Nothing was hit.
  }

  Material material = ray.thingHit->material;
  Color emittance = material.emittance;

  // Pick a random direction from here and keep going.
  Ray newRay;
  newRay.origin = ray.pointWhereObjWasHit;

  // This is NOT a cosine-weighted distribution!
  newRay.direction = RandomUnitVectorInHemisphereOf(ray.normalWhereObjWasHit);

  // Probability of the newRay
  const float p = 1 / (2 * PI);

  // Compute the BRDF for this ray (assuming Lambertian reflection)
  float cos_theta = DotProduct(newRay.direction, ray.normalWhereObjWasHit);
  Color BRDF = material.reflectance / PI;

  // Recursively trace reflected light sources.
  Color incoming = TracePath(newRay, depth + 1);

  // Apply the Rendering Equation here.
  return emittance + (BRDF * incoming * cos_theta / p);
}

void Render(Image finalImage, count numSamples) {
  foreach (pixel in finalImage) {
    foreach (i in numSamples) {
      Ray r = camera.generateRay(pixel);
      pixel.color += TracePath(r, 0);
    }
    pixel.color /= numSamples;  // Average samples.
  }
}
```

Figure 1: Path tracing pseudo code (from Wikipedia)



(a) Bidirectional path tracing with 40 samples per pixel.



(b) Metropolis light transport with an average of 250 mutations per pixel [the same computation time as (a)].

Figure 2: MLT vs bidirectional path tracing results (from this paper, Metropolis Light Transport)