CPSC 490: OpenCL GPU Path Tracer

Brian Li
Advisor: Professor Holly Rushmeier

December 21, 2015

Abstract

The use of general purpose computing on graphics processing units (GPGPU) to accelerate ray tracing has become increasingly ubiquitous. I am proposing an implementation of a path tracer with OpenCL, a C framework which allows for the delegation of operations to the GPU. This provides acceleration to the typical, highly parallelizable path-tracing operations carried out on the CPU. A path tracer recursively fires rays from the camera, and follows their recursive, stochastic progression through the rest of the scene. As these rays progress, they accumulate radiance at every intersection with geometry. The image is then capable of simulating global (non-direct) illumination, provided enough samples are allowed to accumulate. The project builds upon my own work with C++, ray tracing, and global illumination, and will require me to become involved with parallel programming on the GPU and path tracing.

1 The Rendering Equation

1.1 Introduction

In 1986 Richard Kajiya introduced the rendering equation, a derivation which describes light transport in a form highly conducive to computer graphics.[2] The equation solves for the light at a point on a surface, given previous definitions for surrounding geometry and lighting.

\[ L_o(p, w_o) = L_e(p, w_o) + \int_{\Omega} f(p, w_o, w_i)L_i(p, w_i)|cos\theta_i|dw_i \]  

\( L_o(p, w_o) \) : Light at point \( p \) in direction \( w_o \).

\( L_e(p, w_o) \) : The emission produced by point \( p \) in direction \( w_o \).

\( \int_{\Omega} \) : The integral over the hemisphere around point \( p \).
\( f(p, w_o, w_i) \): The bidirectional reflection of point \( p \) with incoming light from direction \( w_i \), and leaving in direction \( w_o \).

\( L_i(p, w_i) \): The incoming light at point \( p \) from direction \( w_i \).

The equation is continuous, requiring an integral taken over the hemisphere about point \( p \). Solving the three-dimensional integral of light at all points in a scene would be too computationally intensive to be feasible. The aim of computer graphics then, in the most general sense, is to approximate an answer to this integral.

Rendering algorithms, broadly speaking, strive to approximate the manner in which light moves around in a prespecified scene relative to a virtual camera. *Unidirectional Monte Carlo Path Tracing* is one such solution which aims to solve the rendering equation through repeated random sampling, and it is the primary focus of this project.

### 1.2 Unidirectional Monte Carlo Path Tracing

Kajiya proposed a method for solving the rendering equation called *Path Tracing* that uses Monte Carlo methods to approximate the Rendering Equation.

#### 1.2.1 Monte Carlo Methods

Monte Carlo methods refer to a class of algorithms in which the distribution function of a variable is repeatedly sampled; these samples then gradually converge on the expected, real value, allowing us to approximate the real values of integrals.

It is helpful to think of this in opposition to more standard techniques. Traditionally, when evaluating an integral, we might approximate the area under the curve with a series of small rectangles, the width of which control the accuracy to which we are measuring the integral. The summed area of the rectangles is our answer.

However, we could also randomly “choose” points within a known domain and range which encompass the curve. The proportion of randomly chosen points captured by the function to those who escape it would provide a reasonable approximation of the area of the curve. [3]

Obviously, the former method is preferable in most situations. As the dimensionality of the integral climbs, however, Monte Carlo methods become more preferable in their predilection for cheaply calculating a solution where other estimates would be exceedingly convoluted and computationally intensive. The rendering equation is immensely complex, because when we consider the light at a point in the scene, we must assume that it is affected by every other point in the scene.

Therefore, in estimating the rendering equation, we will privilege Monte Carlo methodology over others, which will allow for cheaply and quickly approximating the answer, at the expense of noise (the primary weakness of Monte Carlo Path Tracing), as the number of samples must get quite large before converging. /citescratchapixel
The following is the Monte Carlo approximation of an integral, the estimator that will be used in my path tracing implementation. At each step of the path tracing algorithm, a random sample is taken, weighted by the probability of choosing that particular value, and averaged along with the rest of the samples by an accumulation buffer.

\[
\int f(x)dx \approx \frac{1}{N} \sum_{i=0}^{1-N} \frac{f(x_i)}{pdf(x_i)}
\]  

\(\frac{1}{N} \): We are taking \(N\) samples and accumulating them.

\(f(x_i)\): A single random sample.

\(pdf(x_i)\): The probability of choosing that particular random sample. We need to scale the value of the sample upwards by the inverse of the probability that we’ve chosen it. In our path tracing, we only deal with uniform pdfs, so this is always a constant.

### 1.2.2 Path Tracing

It is from the aforementioned techniques that we reach path tracing. Path tracing derives heavily from the more general ray tracing algorithm, in which "rays" are fired from the eye into a scene, returning information as they branch through the scene.

In both basic ray and path tracing, because the ray originates at the eye, the algorithm is considered "forward" and unidirectional (because the rays are traced in one direction). A separate type of path tracer, the bidirectional path tracer, (which is not discussed in this project) follows rays from the lights as well.

In both ray and path tracers, the propagation of light through the scene is defined by the material assigned to each surface. The material holds information on the BRDF, with which the renderer determines the behavior of the ray as it travels through the scene. In a traditional, Whitted style ray tracer, a ray may break off into its separate components, specular, diffuse, etc... However, in a path tracer, we only follow a single path through the scene and the ray never splits.

Instead, at each intersection with the scene, the ray "chooses," so-to-speak, what happens next via a random sample. At each bounce, the light beam accumulates information on its path in the form of a "weight," which is scaled according to the previous surfaces it has struck.

The primary ray fired from the eye gives us \(p\), and the direction of interest, \(w_o\). From then on, we evaluate the integral iteratively, following the single ray through the scene. We essentially reduce the propagation of light through a scene to many, many Markov Chains, sequences of states which are transitioned to and from with certain probabilities. Each progressive state is defined by randomly sampling the material’s BRDF.

At a certain point, the radiance is so small that it makes no sense to continue bouncing. However, we also do not want to cut off arbitrarily, leading to biased results. Instead, we use
a Monte Carlo technique called Russian roulette in which, below a certain weight, a ray can terminate prematurely based on a randomly selected number. Rays that do not terminate are weighted by their probability of continuing.

1.3 The Bidirectional Reflectance Distribution Function

In a scene, the materials specify the manner in which light moves, and the function that describes how light, entering from a certain view direction, behaves, is given by the BRDF.

1.3.1 Lambertian

A Lambertian surface is one in which light is scattered equally about the hemisphere. This is the equation for the BRDF of a Lambertian surface. [6]

\[ L_o = \rho_{brdf} = \frac{1}{\pi} \rho_d \]  

(3)

\( \rho_d \) is Albedo, or incoming light at the point multiplied by the color of the surface. [6]

\[ \rho_d = \int_{2\pi} cL_i \cos \theta dw \]  

(4)

Recall our previous Monte Carlo estimator, in which we evaluate an integral by averaging random samples weighted by the inverse of their probability of occurrence. Given that, we can solve for this integral.

The cosine of \( \theta \) is equal to the dot product of the normal, \( N \), and the direction of the light, \( L \).

We know that the probability of choosing a given direction in a hemisphere is \( \frac{1}{2\pi} \). In this case, the \( \pi \) in the pdf will cancel with the \( \pi \) in the original equation, giving us a multiplier of 2.

This gives us:

\[ L_o \approx \frac{1}{N} \sum_{i=0}^{1-N} 2(N \cdot L)c \]  

(5)

So at every bounce in which we choose to evaluate the diffuse component of a material, all we need to do is multiply the color by 2 and the dot product of \( N \) and \( L \).
1.3.2 Specular

A specular surface is a reflective one. Where Lambertian reflection is independent of direction (the outgoing path will be chosen independent of the incoming one), specular reflection is very much dependent on the angle of entry. Perfect reflection would have the angle of the light ray between normal be the same, but in the opposite direction.

Because the outgoing light in a certain direction is entirely dependent on light coming in from a single direction, there is no integral to evaluate and we can calculate it without concerning ourselves with the pdf.

However, few materials are truly reflective, or pure mirrors. Often, surfaces have any combination of specular, diffuse, or transmissive components. This behavior is partly modeled by the refractive index of a material, a number which indicates how light behaves when entering the medium from another medium.

I implemented the Schlick approximation to calculate the Fresnel factor between two mediums with different indices of refraction. The factor determines the proportion of light absorbed, reflected, or refracted at a certain angle of interaction:

\[
R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5
\]  \hspace{1cm} (6)

\[
R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2
\]  \hspace{1cm} (7)

In this approximation \(n_1\) and \(n_2\) describe the index of refraction, and \(\cos\theta\) is the dot product of the normal and view vectors.

Because we only follow a single path and cannot split rays, we sample this proportion to determine the behavior of the light in a certain path.

In the case of a refractive material, the factor determines the likelihood of reflectance versus transmission (movement through the medium). In the case of a specular material, the factor determines the likelihood of specular reflectance versus Lambertian diffuse reflectance.

2 Programming on the GPU

The aforementioned equations are all that are needed to implement a simple path tracing system. However, this project aims to implement the algorithm on the GPU, a process which will now be covered.
2.1 The GPU

The GPU, or graphics processing unit, is a piece of hardware optimized for handling large amounts of arithmetic, featuring many more cores designed to process data in parallel at the expense of flow control and caching, core tenets of the CPU, the central processing unit. Adapting massively parallel tasks for computation on the GPU is the primary goal of GPGPU, or General Purpose programming on the GPU.

The path tracing algorithm, because each ray is calculated independently of one another, is highly optimized for parallelization. What is calculated in $\text{pixel}_{i,j}$ has no bearing on the rays being followed for $\text{pixel}_{i+n,j+n}$. The algorithm is iterative, so the same set of steps is executed for each state in the chain. Furthermore, all rays will share the same information about the scene. We might say this conforms to the SIMD framework, or "single instruction, multiple data," a class of problem the GPU is uniquely suited for handling. Therefore, the path tracing algorithm should be incredibly conducive to the setup on the GPU. [1]

To interface with the GPU, we will be using a "language" called OpenCL. The term language is a loose one; OpenCL is really a collection of data structures and libraries extending C and C++ that allow one to delegate kernels (small programs) to (theoretically) any processing unit in a computer (although our focus remains on GPUs).

2.2 OpenCL

OpenCL, the Open Computing Language, extends C and C++, both of which traditionally target CPUs. The first iteration was developed in 2008, as an interface to program on different devices from different vendors. Prior to OpenCL, Nvidia, IBM, and the like all had their own proprietary languages, making programs designed for cross-device execution exceedingly difficult. I am programming on my Mid 2012 MacBook Pro, with an Intel HD Graphics 4000 graphics card. Note that although OpenCL can run on many devices, not limited to GPUs, I will discuss GPUs primarily, as they are the focus of this project. [7]

I programmed an OpenCL manager which organizes and manages the GPU pertinent data; the manager is programmed with the C++ wrapper for OpenCL. C++’s inherent OOP capabilities make it very easy for us to organize the various data structures necessary for GPGPU within a single class instance.

2.2.1 Programming in OpenCL

In OpenCL, programs for the GPU take the form of a "kernel," essentially a function that is delegated to the GPU by the host application, a program running in C or C++ on the CPU. The group of devices being managed by the host application is called the context of the program, also managed by its own object. When the host application wants to execute the kernel on the OpenCL device, it dispatches the kernel, and associated host-side (CPU) arguments to a command queue, the pipeline by which the kernel is loaded up on the OpenCL
device. The kernel is marked void, so to read back the processed data, we dispatch a buffer
to the command queue, then brings the processed information back to the host application.

The GPU, can, given a single instruction, operate on a variegated set of data simultaneously. This is called vector computing, and one of the core capabilities of OpenCL is the introduction of vector data types, such as float3, float4, char16 etc... C and C++ have no such capabilities, as the CPU works primarily on scalar values. Note that vectors are not to be confused with the C++ STL vector, a wrapper for a dynamic array. While the STL vector stores multiple elements, when we operate on it, we still do so element by element. This is different from the OpenCL; every value in the vector can be operated on simultaneously, resulting in faster computation than in a corresponding scalar data types.

2.2.2 Work Partitioning

Our instructions must be broken up and delegated to the different cores on the GPU. The primary units of "work" being performed are:

**Work Item (a single ray)**: While the kernel is the instruction delegated to the GPU, each separate instance of the kernel operates on different arguments. A work item is a single instance of the kernel. Every work item is assigned a unique global id. In our path tracer, each work item is a separate path.

**Work Group (a group of rays)**: The work group is a set of work items which has access to the same resources; each work group exists on a single compute unit, or core, of the GPU. We want to put as many ray paths into a work group as possible, because every ray path uses the same scene information.

Crucially, work items are only synchronizable when they exist in the same work group. It is important to know that the GPU speedups rely on performing the same tasks in parallel; if multiple kernels within a work group branch in different directions, the execution is significantly slowed because each work item is performing a different instruction (and how can there be parallelization when the instructions are different?). When we program on the GPU, we want to minimize flow control and branching, because divergence among work items in a group nullify the optimization we might otherwise gain.

2.2.3 Memory Partitioning

When we deal with code running on a non CPU device, we need to consider more than just the heap and stack for memory management. Every argument being delegated to the GPU must have a qualifier that lets the device know where the memory for the argument comes from.

**Global**: Data stored on the entire GPU. We rarely use this in my path tracer implementation, because it is the slowest; however, it is also the largest of the three data spaces. If we
were operating on a scene with thousands of triangles, we would be unable to store them in local memory; in that case, we would store the triangle data in global memory.

**Local**: Data shared by a work group. At the beginning of each path, my path tracer copies all the global scene information to local memory, which is orders faster to access than global memory. Because every path needs to access the entire scene, this works out very well.

**Private**: Data used by a single work item. This is the default (no qualifier). We use the private memory to store intermediate data specific to each work item.

### 2.2.4 Limitations

There are some limitations associated with programming on the GPU.

First of all, the GPU is not optimized for flow control. OpenCL does not offer support for recursion. Luckily, path can both be converted to an iterative solution, since the amount of light at a point is additive. We simply keep track of the last position of the ray, and the new direction in which it is being fired.

As the kernel is a C based function, we don’t get object oriented programming outside of structs. Most notably, we don’t have polymorphism, which is typically essential to path and ray tracing. One of the strong points of ray tracing is that different primitives and geometry can be added to a scene so long as they implement certain functions, such as an intersection function or bounding box function. The fix I have implemented is storing all primitives in a `Primitive` class, with the exception of triangles.

With a cursory knowledge of path tracing and OpenCL, we can now implement a fully functioning path tracer on the GPU.

### 3 Code

### 3.1 Infrastructural Code

Although I wanted to code as much of the path tracer on my own as possible, certain codes were taken from other sources for the sake of spending as much time with the core concepts of path tracing and gpu programming as possible. These are the GUI, and the PNG writer. The GUI was taken from Peter Kutz and Karl Li’s CUDA path tracer implementation, and the PNG writer from an online tutorial.

The rest of the code is primarily my own, or otherwise derived from tutorials and papers online.
3.2 Random Number Generator

I used William Langdon’s Pseudo Random Number Generator for NVidia CUDA as my pRNG [5]. I keep an array of random number seeds for each pixel on host memory; random numbers are generated on the fly within the kernel with the aforementioned algorithm, and the final number becomes the new seed, read back to host memory, at the end of each path.

3.3 The Kernel

The kernel is where all the path tracing resides. First, materials, primitives, and triangles are copied to the local memory. As of right now, we only support hard coded values for scene information. There is, however, an OBJ reader which imports triangle meshes.

A ray is fired from the pixel assigned to the work-item, determined by the global id assigned to the kernel. The pixel is further subdivided and the position of the ray slightly jittered (with the aforementioned pRNG) to provide anti-aliasing.

Once the primary ray is generated, the ray begins to “bounce” through the scene. The iterative loop first checks for intersections. If an intersection is found in the scene, the material of the surface is queried and its properties evaluated. If the material is refractive or specular, the fresnel coefficient is calculated to determine the next step of the ray.[4]

The ray is terminated when it hits a light or when the Russian roulette condition is met. When the ray ends, the final, accumulated color of the ray is written to a buffer which is copied back to the host memory.

3.4 The Accumulation Buffer

When a full set of paths has been traced, the accumulation buffer adds its contribution to the previously generated image. The image can be downloaded at any point, and the progressive accumulation of samples is visible on a GUI.

4 Results

Some path traced images!
Figure 1: Two Specular Spheres, One Diffuse
641 Iterations

Figure 2: Specular Sphere, Specular Triangle Mesh Box, and Spherical Light
1312 Iterations
Figure 3: Refractive Sphere, Specular Sphere, and Spherical Light
1060 Iterations

Figure 4: Specular Sphere and Background Plane
503 Iterations
5 Future Work

**GPU Refactoring** : I used a single kernel to program this entire path tracer. While simple, this is hardly the most effective use of the GPU to speed up path tracing. More effective would be having multiple kernels handing similar tasks, such as primary ray generation, acceleration structure construction, and image reconstruction. That way we wouldn’t be penalized so heftily for branching, and we could add much needed acceleration support to the path tracer.

**Acceleration** : We need a KD-Tree to speedup triangle acceleration.

**Texture Mapping** : We have simple BRDFs, but they don’t support texture maps so we are limited to solid colors for each primitive. I would like to implement texture maps in the future.

**Scene** : All scenes are hard coded into the path tracer (except OBJs, which are read). I would like to code a scene parser that reads the specifications for the scene from a predetermined file format.

**Bidirectional Tracing** : Tracing beams from the lights as well, could greatly aid our endeavor for realistic images.
6 Acknowledgements

Special thanks to Professor Holly Rushmeier for advising me on this project, and to Wendy Chen for listening to my weekly updates.
References


