GPU Path Tracer Documentation

Savon Brown

December 7, 2020

Contents

1 Intro to GPU Path Tracing

1.1 What is Path Tracing

Path Tracing starts with Ray Tracing. Ray tracing is a process by which you simulate light rays coming from a light source, and how it would interact with objects in a scene. You then find all the light rays that hit your eye, and render the scene.

Ray Tracing requires a lot of computational power. Path Tracing attempts to optimize this by ray tracing backwards. Instead of tracing light rays from the light to the camera, and thus computing the path of rays that never hit us. We, instead, track rays from our eye to the light source as shown in Figure 1. When a ray travels from our eye, and hits objects, if it comes in contact with a light source it would be rendered. This presents us with a lot less rays to compute the path of.

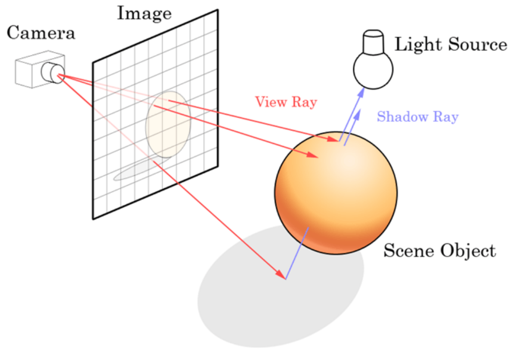

Figure 1: Diagram of a Path Tracer casting rays from a camera.

1.2 Why do we need a GPU?

We don't. A GPU serves as a further optimization. Our algorithm for determining what gets rendered in a scene says that for every pixel on the screen send rays through it. Then for every ray that comes from the camera, have that ray travel until we say stop, changing direction each time it hits an object. If it has hit a light source then color the first object we hit with all the color we collected from all the other objects along the path of the ray. Our function takes a canvas (matrix representing pixels), a background color, and a scene of objects to render. We also limit the number of bounces per ray (NB) as to not iterate a ray for every and return the background when we exceed the bounce limit. The following pseudocode describes this process.

Algorithm 1 Serial Path Tracing

```
1: function COLOR($canvas, bg, scene$)    ▷ We will paint the canvas
2:     for pixel in canvas do
3:         for each ray through the pixel do
4:             $origin \leftarrow camera.location$
5:             $direction \leftarrow pixel.location - camera.location$
6:             $ray = createRay(origin, direction)$    ▷ ray is just a 3d vector
7:             for i=0; i<NB; i++ do
8:                 if ray hits scene then
9:                     $ray.color* = scene.object.color$ ▷ ray is collecting color
10:                    $ray.direction \leftarrow newReflectedRay.direction$
11:                else
12:                    $ray.color \leftarrow background.color$
13:                end if
14:            end for
15:            $pixel.color* = ray.color$    ▷ pixel collects color from ray
16:        end for
17:    end for
18:    return canvas
19: end function
```

This algorithm is a lot to digest. Searching the Scene of objects for a hit takes O(n) time where n is the number of objects, each time. It can be improves to O(log(n)) if we use a tree to hold the objects, but that's not the GPU's optimization. With p representing the number of pixels and r representing the number of rays per pixel, our total runtime is

$$O(p*r*n) = \underbrace{O(p)}_{\text{for each pixel}} * \underbrace{O(r)}_{foreachray} * \underbrace{O(n)}_{foreachobject}$$

That's obscene when you think about resolutions as high as the hundred thousands or millions of pixels, images with over 10000 rays per pixel (which is necessary for any hint of realism) and scenes ranging from simple spheres to complex meshes with thousands of triangles. Things can get out of hand quite quickly.

A GPU helps solve this problem with parallelism. Instead of each pixel having to be done in a serial manner, the color for each pixel is computed in parallel on GPU threads. Thus decreasing our runtime to O(r*n) with the computation of all pixels being constant with respect to one another.

1.3 Overview of remainder of the Documentation

The remainder of the documentation will iterate through the files used in this project starting from our main and working outward. It will follow the implementation of all my relevant classes as well as at the end, having citations for all relevant sources that my code base was both constructed from and inspired by. My code resides on my github repository *HERE* If you use any code, please credit me and the sources I mention in my documentation in hopes someone else may benefit from my work and the work of others in the same way you have.

As this project is ongoing, these docs aren't fully complete until the project is complete. Proceed with that in mind when considering my code.

- Main.cu
- Vec3.h
- Ray.h
- Camera.h

- Hitable.h/Hitable list.h
    - Sphere.h
    - Moving Sphere.h
    - Aarect.h
    - Box.h
- Texture.h/Material.h
- Scenes.h

2 Main.cu

This file invokes the rest of the api and handles GPU error checking (not native). We use this file to actually build worlds, cameras, and objects that will be rendered on the gpu then passed back here to be put into an image file.

2.1 check cuda(result, function,file,line)

This is the function that handles error and takes any error passed by the CUDA and tells us where it was in our program. That way the program doesn't only die when there is an error, but instead tells us which kernerl (gpu function) threw it and on which line.

2.2 render(frameBuffer, samples per pixel, screensize, ray, world, random state, camera)

This function performs the ray casting. It takes rays and casts the to objects in the scene. It then takes the returned color and put it in a frame buffer (image array). The randomstate is generated by taking each pixel as a seed to a random number generator and passing the random number to the pixel that gave us the seed to produce it. We need that random number for a lot of our materials.

2.3 color(ray,world,randomstate,camera)

This function takes the casted ray from the camera to the objects within the scene. It takes in a random state as well because all of our GPU functions need it to simulate randomness.

2.4 createWorld(shapeList, cameraOptions)/freeWorld(shapeList, camera)

These 2 functions work together to build and destroy the world from memory. The first takes the shapes we define and put them into a scene so that we can cast rays of light at them from a predetermined camera. The second function is a garbage collector that empties the scene when we are done rendering the image.

2.5 run(image,camera,filename,settings)

This function sets up all the information/states for the GPU, then takes all of that information and outputs the frame buffer as an image file.

3 Vec3.h

We use this class for creating and operating on 3 dimensional vectors. This class is simply full of operator overloading for vector operations and indexing.

3.1 vec3::clamp(low,high,value)

This function simply takes a number and forces it within the range of low to high by returning min(max(low,value),high)

4 Ray.h

The ray is just a 3d vector we use to simulate the direction of a light ray.

4.1 ray::point at parameter(t)

This is the only function and it returns a point in space that we later use to see is that point returned lies on the surface of our shape and should be rendered.

5 Camera.h

The camera determines the orientation of our image (or vieweing orientation) and certain effects that have to do with the cameras properties. Cameras are created with a lens radius, an up direction and a point to lookat. It then calculates a bunch of other information in relation to it and the scene to properly orient itself.

5.1 camera::get ray(horizontal,vertical,randomState)

This function is how we actually go about casting a ray to the screen. Using the size of the screen for reference on where to send the ray.

6 Hitable.h/Hitable list.h

A hittable is an abstract class that I use to define a surface that we can intersect with, then put these items into a list. Things that can be hit include shapes, but can also include mediums like water and fog. It also defines an abstract ”hit” function to be implemented by each hitable and a hit record that tells each hitable where they were hit as well as properties of the point such as the normal and whether we intersected from the inside or outside the shape.

6.1 Sphere.h

Spheres are created with a vector representing its center, a radius, and a surface material.

6.1.1 sphere::hit(ray,min,max,rec)

The hit function for the sphere uses the quadratic equation and checks the point returned by the ray as a solution to the equation. ”rec” is the record containing the point. If it is a solution then it lies on the circle thus we render. We make sure to take the solution closest to us as to not render things we can’t see. ”min” and ”max” are the min and max viewing distances

6.1.2 sphere::get sphere uv(point,u,v)

We map the surface coordinates of a sphere to a rectangle for the purposing of image mapping to a round shape. This function returns the point on a rectangle corresponding to a point on a sphere.

6.2 Moving Sphere.h

Moving spheres are the same as spheres, but center and radius are both functions of time.

6.2.1 moving Sphere::hit(ray,min,max,rec)

Much like the sphere this also uses the quadratic equation, but radius and position of the sphere are both functions of time. We still only render rays that hit the object first.

6.3 Aarect.h

6.3.1 aarect::hit(ray,min,max,rec)

This function takes an axis aligned rectangle and verifies whether or not the ray has crossed over its boundaries in any axis. If it has then we render. This can be rotated on all axis.

6.4 Box.h

6.4.1 box::hit(ray,min,max,rec)

This function essentially assembles a box out of axis aligned rectangles by rotating and scaling as needed.

7 Texture.h/Material.h

This is where a lot of the simulation of our environment occurs. To accurately simulate the look of realistic objects, it only suffices you have realistic looking materials on surfaces. The more accurately your materials imitate the interactions their real counterparts have with light AND we give the scene enough light, our image should look realistic.

Texture is an abstract class that wraps material so that we can also do image mapping or other non realistic textures. Textures need only have a color value.

Materials is also an abastract class that requires objects of the class to determine their color value using incident light rays and the color we gave the shape by implementing a "scatter" function. This is how we achieve the effect of Global illumination as well. Each material's constructor is just a color or previously defined texture value.

An example to more clearly differentiate between texture and materials. A red ball illuminated by a blue light: Each part of the ball has a color we already decided, red. But because we are using a material, the blue light will cause some parts of the ball to be purple, whereas a normal texture wouldn't necessarily react to the light ray.

7.1 Lambertian

7.1.1 lambertian::scatter(ray in, rec, color

For a lambertian material our scatter function chooses a random direction in a sphere around the point we intersected with and casts a ray to a random point in the sphere. For this to be accurate the distribution of possible points around our point must be equal in all direction.

7.2 Metal

7.2.1 metal::scatter(ray in, rec, color

For a metal our scatter function takes the hit record and grabs the surface normal from the hit record and reflects our incident ray round that normal and send it back out. The fuzz parameter adds a random bias to each reflected ray so that our surface isn't perfectly smooth if we don't want it to be.

7.3 Dielectric

7.3.1 dielectric::scatter(ray in, rec, color)

For the dielectric our function reflects some rays and refracts others. We define the index of refraction so our dielectrics can look like foggy crystal balls or transparent panes.

7.4 Diffuse Light

7.4.1 dielectric::scatter(ray in, rec, color)

Our light does not scatter incoming rays but instead implements another abstract class "emmitted". It has a default value of 0 or false because no other material besides light will emit anything. Our light emits whatever color we defined and cam be used to illuminate the scene with any color of our choice instead of simple white light.

List of Figures