

MPCS 51087
Problem Set 5
GPU Ray Tracing with CUDA

Spring 2018

Due Date: Friday, May 25, 2018 by 11:59 pm

1 Introduction to Ray Tracing

Ray tracing is a powerful method for rendering three-dimensional objects with complex light interactions. Many other physical phenomena are also analogous to ray tracing – for example, simulating neutrons passing through matter. Figure 1 shows the basic idea for a reflective object. An observer (the eyeball) is viewing an object through a window (the rectangle). The object is illuminated by a light source, which is emitting rays of light (the arrows) in many directions. The observer will see rays that reflect off of the object.

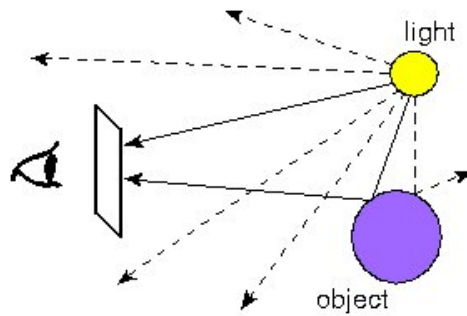


Figure 1: Illustration of Ray-Tracing. Image credit: [1]

Our task is to render the image seen by the observer through the window. To do so, the simulation can do one or both of the following:

- Simulate light rays starting at the light source and ending at the observer.
- Simulate light rays starting from the observer and going backwards to the light source.

We will implement the latter scheme in this assignment. We will simulate a reflective sphere in black-and-white, illuminated by a single light source. A serial version can be implemented in about 100 lines of code.

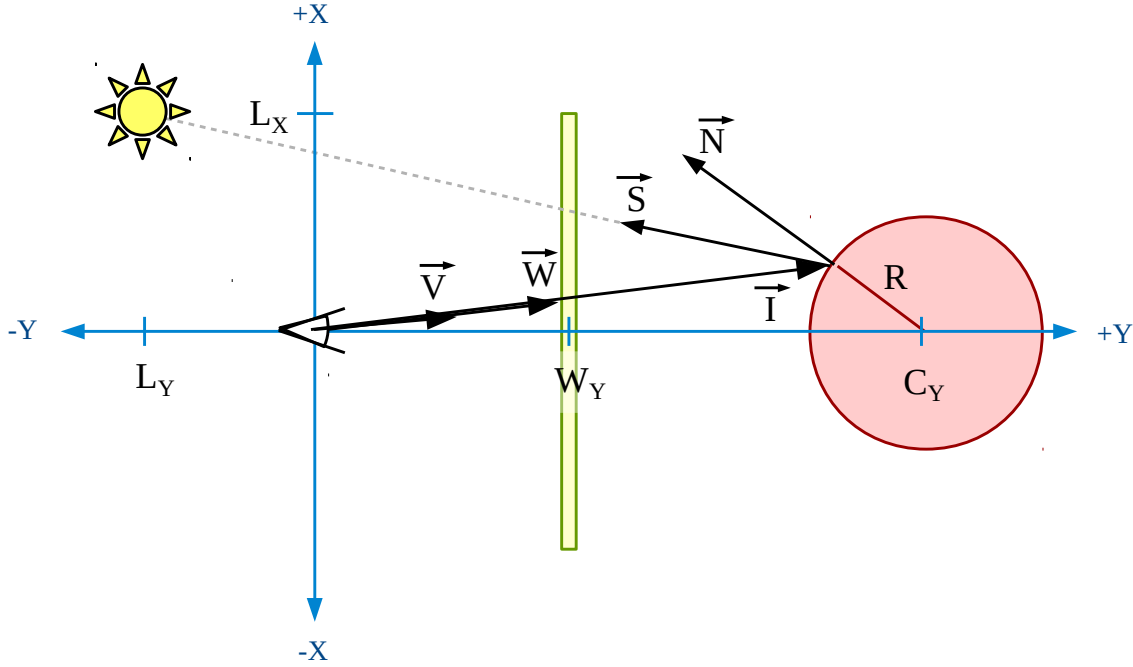


Figure 2: 2D Diagram for Ray-Tracing

2 Vector Notation

Solving the problem involves the use of 3D vectors. A vector \vec{V} has scalar components, $\vec{V} = (V_X, V_Y, V_Z)$. Adding or subtracting two vectors yields a vector, $\vec{V} - \vec{U} = (V_X - U_X, V_Y - U_Y, V_Z - U_Z)$. Multiplying a scalar and a vector yields a scalar, $t\vec{V} = (tV_X, tV_Y, tV_Z)$. Dividing a vector by a scalar yields a scalar, $\vec{V}/t = (V_X/t, V_Y/t, V_Z/t)$. The dot product (a multiplication between two vectors) yields a scalar, $\vec{V} \cdot \vec{U} = V_X U_X + V_Y U_Y + V_Z U_Z$. The norm (or magnitude or length) of the vector is a scalar defined by $|\vec{V}| = \sqrt{V_X^2 + V_Y^2 + V_Z^2}$.

3 Theory

This section describes the theory used in our implementation. The implementation itself is brief and is described in Algorithm 1 below, so you may skip this section as needed.

Figure 2 shows a 2D cross-section of the problem with the information we need. The observer is at the origin and faces the positive y-direction. The sphere is located at $\vec{C} = (C_X, C_Y, C_Z)$ and has radius R . The window is parallel to the (x,z)-plane at $y = W_Y$ and has bounds $-W_{max} < W_X < W_{max}$ and $-W_{max} < W_Z < W_{max}$. In the simulation, we will represent the window as an $n \times n$ grid, G . The light source is located at $\vec{L} = (L_X, L_Y, L_Z)$.

Our task is to simulate many rays originating from the observer (so-called “view rays”) with randomly-selected directions. The steps are as follows:

- **Select the direction of view ray (\vec{V}).** Let \vec{V} be a unit vector representing the direction of the view ray. In spherical coordinates, we will randomly select its component angles (θ, ϕ) .

Then we will get \vec{V} in Cartesian coordinates:

$$\begin{aligned} V_X &= \sin \theta \cos \phi \\ V_Y &= \sin \theta \sin \phi \\ V_Z &= \cos \theta \end{aligned}$$

One important point to note is that we cannot sample both ϕ and θ from uniform random distributions between 0 and 2π without biasing the distribution. To sample the unit sphere correctly, we will sample only ϕ from a uniform distribution between 0 and 2π . We will then sample the *cosine* of θ from a uniform distribution between -1.0 and 1.0. E.g.,

Algorithm 1 Direction Sampling Algorithm

- 1: Sample ϕ from uniform distribution $(0, 2\pi)$
 - 2: Sample $\cos(\theta)$ from uniform distribution $(-1, 1)$
 - 3: $\sin(\theta) = \sqrt{1 - \cos^2(\theta)}$
 - 4: $V_X = \sin(\theta) \cos(\phi)$
 - 5: $V_Y = \sin(\theta) \sin(\phi)$
 - 6: $V_Z = \cos(\theta)$
-

- **Find the intersection of the view ray with the window (\vec{W}).** Knowing that the window is at W_Y , the window's point-of-intersection with the view ray is given by the vector \vec{W} :

$$\vec{W} = \frac{W_Y}{V_Y} \vec{V}$$

If the view ray is outside the window ($|W_X| > W_{max}$ or $|W_Z| > W_{max}$), we reject it and chose a new \vec{V} .

- **Find the intersection of view ray with sphere (\vec{I}).** Let \vec{I} be the sphere's point-of-intersection with the view ray. To find \vec{I} , we solve the following system of equations:

$$\begin{aligned} \vec{I} &= t\vec{V} \\ |\vec{I} - \vec{C}|^2 &= R^2 \end{aligned}$$

These are the equations of the view ray and the sphere, respectively. Solving for t yields:

$$t = (\vec{V} \cdot \vec{C}) - \sqrt{(\vec{V} \cdot \vec{C})^2 + R^2 - \vec{C} \cdot \vec{C}}$$

which can be back-substituted to get \vec{I} . If t does not have a real solution ($((\vec{V} \cdot \vec{C})^2 + R^2 - \vec{C} \cdot \vec{C}) < 0$), then view ray does not intersect the sphere and we choose a new \vec{V} .

- **Find the observed brightness of the sphere (b).** Next, we want to find the brightness of the sphere that is observed at \vec{I} . To do so, we:

- Find the unit normal vector (\vec{N}). The unit normal vector \vec{N} is perpendicular to the sphere's surface at \vec{I} .

$$\vec{N} = \frac{\vec{I} - \vec{C}}{|\vec{I} - \vec{C}|}$$

- Find the direction to the light source (\vec{S}). The direction to light source (sometimes called the “shadow ray”) is represented by the unit vector \vec{S} .

$$\vec{S} = \frac{\vec{L} - \vec{I}}{|\vec{L} - \vec{I}|}$$

- Find the brightness (b). The brightness can be found from \vec{S} and \vec{N} using “Lambertian shading”.

$$b = \begin{cases} 0 & \vec{S} \cdot \vec{N} < 0 \\ \vec{S} \cdot \vec{N} & \vec{S} \cdot \vec{N} \geq 0 \end{cases}$$

- **Add the brightness to the window’s grid.** We find (i, j) such that $\vec{G}(i, j)$ is the position of \vec{W} on the window’s grid G and let:

$$G(i, j) = G(i, j) + b$$

4 Implementation

Algorithm 2 describes a ray-tracing implementation. As described above, the observer is at the origin and is facing the positive- y direction. The sphere is located at $\vec{C} = (C_X, C_Y, C_Z)$ and has radius R . The light source is located at $\vec{L} = (L_X, L_Y, L_Z)$. The window is parallel to the (x,z)-plane at $y = W_Y$, has bounds $-W_{max} < W_X < W_{max}$ and $-W_{max} < W_Z < W_{max}$, and is represented by an $n \times n$ grid, G .

Algorithm 2 Ray Tracing Algorithm

```

1: allocate  $G[1 \dots n][1 \dots n]$  ▷ The window is represented on the grid  $G$ 
2:  $G[i][j] = 0$  for all  $(i, j)$ 
3: for  $n = 1 \dots N_{rays}$  do
4:   repeat
5:     Sample random  $\vec{V}$  from unit sphere ▷ Algorithm 1
6:      $\vec{W} = \frac{W_Y}{V_Y} \vec{V}$  ▷ The intersection of the view ray and the window
7:     until  $|W_X| < W_{max}$  and  $|W_Z| < W_{max}$  and  $(\vec{V} \cdot \vec{C})^2 + R^2 - \vec{C} \cdot \vec{C} > 0$ 
8:      $t = (\vec{V} \cdot \vec{C}) - \sqrt{(\vec{V} \cdot \vec{C})^2 + R^2 - \vec{C} \cdot \vec{C}}$ 
9:      $\vec{I} = t\vec{V}$  ▷ The intersection of the view ray and the sphere
10:     $\vec{N} = \frac{\vec{I} - \vec{C}}{|\vec{I} - \vec{C}|}$  ▷ The unit normal vector at  $\vec{I}$ 
11:     $\vec{S} = \frac{\vec{L} - \vec{I}}{|\vec{L} - \vec{I}|}$  ▷ The direction of the light source at  $\vec{I}$ 
12:     $b = \text{MAX}(0, \vec{S} \cdot \vec{N})$  ▷ The brightness observed at  $\vec{I}$ 
13:    find  $(i, j)$  such that  $G(i, j)$  is the gridpoint of  $\vec{W}$  on  $G$ 
14:     $G(i, j) = G(i, j) + b$  ▷ Add brightness to grid (must be thread safe!)
15: Output grid  $G$  to file

```

Note that in Algorithm 2, all rays are independent from one another, except for the fact that they are all tallying their results to a single common grid (Line 14). As the rays are randomized, it may be the case that multiple rays try to update the value at the same grid location at the same

time, resulting in undefined behavior. Therefore, some sort of synchronization will be necessary to ensure correctness.

Figure 3 shows a sample image where $\vec{L} = (4, 4, -1)$, $W_Y = 10$, $W_{max} = 10$, $C = (0, 12, 0)$, $R = 6$.

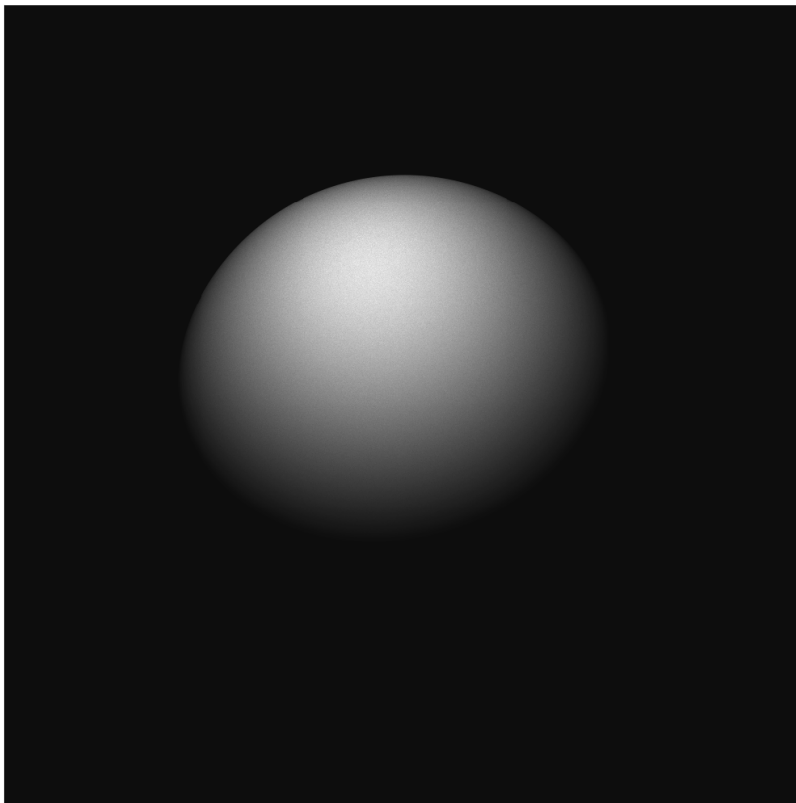


Figure 3: Ray-Traced Render of a Sphere Illuminated from Top Left

4.1 Random Number Generation on GPU

There are a wide variety of methods for generating pseudo random number sequences on computers. The simplest methods are known as Linear Congruential Generators (LCGs), and are very fast and only require a single variable (typically a 64-bit unsigned integer) to track the state of the system. As such, they are very efficient in terms of speed and memory usage, though are subject to several limitations. One limitation is that some generators have a small period, where a period is the quantity of numbers that it can generate before the same number gets generated again and the cycle begins to repeat itself. For instance, the standard C function `rand()` has a period of roughly 2^{32} , or roughly 2 billion. Another limitation is that for some applications (e.g., security, gambling) it may be problematic to have a stream of random numbers be totally identifiable by only a single state variable.

There are more robust methods such as Mersenne Twister algorithm that are capable of both much higher period random number streams, and less predictable outputs. This is accomplished by having a much larger state vector, sometimes composed of hundreds of numbers, that work with the algorithm to define where the generator is in the stream.

On GPUs, the standard CUDA method of generator random numbers is by use of the `curand`

functions. This is a heavy generator with a large state vector, which can cause some issues if each of millions of threads needs to initialize a unique generator stream to just generate a handful of random numbers. Typically, this will mean that it will be necessary to reduce the total number of random number generator streams that are created to be far below the number of threads being launched, or otherwise make re-use of each random number stream. For instance, if you want to launch 1 billion threads, it is best to either reduce the number of threads by packing multiple tasks into each thread, or set up a sharing system for the RNG streams.

In this assignment, our use case allows us to use a simpler LCG, and just allow all threads to seed the LCG with a number based on their unique thread ID. Therefore, for this assignment, you will have the following two options for generating random numbers:

1. Use the `curand` functions, but make sure that each thread samples and traces multiple rays, so as to amortize the costs of initializing the generator and reducing the overall memory cost of the generator.
2. -OR- Launch each ray on its own thread, and use the below LCG:

```

1 // Standard Modulus 48 LCG (from java.util.Random)
2 // Returns double uniform between 0.0 and 1.0
3 double rn(unsigned long long * seed)
4 {
5     double ret;
6     unsigned long long n1;
7     unsigned long long a = 25214903917;
8     unsigned long long m = 281474976710656; //2^48
9     unsigned long long c = 11;
10    n1 = ( a * (*seed) + c ) % m;
11    *seed = n1;
12    ret = (double) n1 / m;
13    return ret;
14 }

```

5 Questions

1. (20 pts) Make and test a serial CPU version of Algorithms 1 and 2. Take user input for the number of rays and gridpoints; the other parameters can be hardcoded. (Hint: Define structs and functions for the vectors and vector operations. With these simple abstractions, the code can be written in about 100 lines.)
2. (30 pts) Use CUDA to parallelize Algorithms 1 and 2 for GPUs. Note that, while the rays are independent, updating the window could result in write conflicts. Therefore, some synchronization is necessary.
3. (10 pts) Experiment with different block and thread configurations of your code and determine an (approximately) optimal configuration for launching your kernel on a Midway GPU of your choice. Report your optimal settings in your PDF writeup.
4. (20 pts) Compare the runtime of the serial CPU version and CUDA GPU versions as functions of problem size (the number of rays). For the CUDA version, use the maximum occupancy of the device.
5. (10 pts) Show a sample image produced by your CUDA ray-tracing implementation.

6 Compiling, Code Cleanliness, and Documentation (10 pts)

We plan on compiling and running your code as part of grading it. You must include a makefile capable of compiling your code into a runnable executable. Also, while we are not grading to any rigid coding standard, your code should be human readable and well commented so we can understand what is going on.

7 What to Submit

Your submission should be in the form of a zip or tar file containing the following items:

- Your source code for your serial and CUDA versions should be in different directories, each with their own makefiles.
- Working makefiles that compile each version of your code
- Your plotting script(s)
- A single writeup (in PDF form) containing all plots and discussions asked for in the assignment

8 References

1. Rademacher, Paul. *Ray Tracing: Graphics for the Masses*. <https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html>
2. *Ray tracing (graphics)*. [http://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](http://en.wikipedia.org/wiki/Ray_tracing_(graphics))