Reinforcement Light Rays Path-Tracer Progress Report

Callum Pearce

cp15571@my.bristol.ac.uk

***Abstract*—This document contains my weekly progress of my thesis as part of my 4th year masters unit COMSM0111 in 2018/19. It is mainly useful to myself for reflecting on the decisions I have made throughout the project and why I chose them. However, I also hope it gives a detailed overview of how the project evolved with time for any other reader.**

I. Introduction

Each section of this document describes a single week of work. For each section I have decided to include the following break down:

- **Goals**: A few set goals I aimed to achieve in that week and the motivation behind them.
- **Research/Implementation Details**: What was done, and how it was achieved.
- **Resources**: Describes what resources were notably helpful during the week for research/implementation details.
- **Reflection**: What I believed went well in the week, what to avoid in the future, and were the goal outlined achieved? Finally, if necessary, what has changed for the project as a whole?

Week 1

Building off dome preliminary research I decided I had to build a ray-tracer in order to start any work with my project. This would be a good way of refreshing the basics of computer graphics.

A. Goals

1) Build a basic ray-tracer from scratch using only SDL and GLM as external libraries

B. Research/Implementation Goals

This week was fairly simple in terms of implementation. I mainly based my work on the ray-tracer I built with my project partner in my 3rd year of university. I used a similar project structure and followed the lab-sheets from `COMS30015` by Carl Henrik Ek. By the end of the week I had built a ray-tracer which simulated the following in real-time:

- Constructing surfaces from triangle primitives and projecting them onto a 2D pixel plane for camera viewing via ray-tracing.
- Supports camera movement
- Direct illumination

C. Resources

As mentioned the main resources used were those provided for `COMS30015` by Carl Henrik Ek in 2017. They gave me a good refresher on all the core concepts of ray-tracing and the mathematics behind it. I have based my rendering on the Cornell Box scene which is a classical scene for testing computer graphical renderings.

D. Reflection

I met my goal this week and built a well designed code-base to go with it.

Week 2

With a basic ray-tracer up and running, it was then time to add global illumination (indirect lighting) into the rendered scene and other features to make the basic ray-tracer complete for my purposes.

E. Goals

1) Implement Monte-Carlo global illumination within the ray-tracing pipeline (to go alongside direct light. It is Monte-Carlo global illumination I am planning to base my reinforcement learning technique presented by NVIDIA [1] on.
2) Create an object loader for the scene to test rendering in different scenes. I need a scene which has very low light levels in certain parts of the scene in order to show how reinforcement learning reduces noise in these areas.

F. Research/Implementation Goals

My implementation worked as follows; for every pixel in the image, calculate the colour by finding the direct light at that point combined with the indirect light. Where indirect light was sampled by shooting a ray into the scene and scattering it, then recursively finding the illumination at these points (via Monte Carlo).

For the object loader, I read in all vertices and then built triangles to form the defined surfaces in the file by using fan-triangulation. These triangles would be built into the scene by a call to the script with the file name of the `.obj` file to load.

I also introduced `openmp` to the project to speed things up by parallelising the ray-tracers pixel painting loop.

G. Resources

ScratchPixel 2.0 [7] provided an excellent description of Monte Carlo global illumination for a ray-tracer. As for the object loader, opengl gave a great tutorial for simple processing of `.obj` files [6].

H. Reflection

Monte Carlo global illumination was available for a custom scene as I had set out to achieve in this week. I also introduced an object loader which allows me to load in a scene of my choice, which will become especially useful when comparing different methods later on in the coursework. However, due to the introduction of global illumination the ray-tracer is far slower and takes a significant amount of time (nearly a day) to render a high quality image.

WEEK 3

With global illumination in hand and a custom scene to test it on the project moved on to begin working on reimplementing the *Learning Light transport the reinforced* way [1] paper.

I. Goals

1) Take detailed notes of the [1] paper and understand what needs to be changed with my current ray-tracer
2) Read through [9] Introduction, Markov Decision Processes and Temporal Difference learning chapters in order to understand the basis of Reinforcement learning.
3) Begin implementing the Radiance Volume data structure specified in [1].

J. Research/Implementation Goals

I began my week by first reading the Reinforcement Learning textbook [9] as to attain the second goal. I took many notes and gained a good understanding of temporal difference learning and how it builds on Markov Decision Processes as a model for reinforcement learning. I then read the NVIDIA paper [1] which I found out that I needed to use Expected Sarsa as my form of temporal difference learning for the path tracing algorithm created by NVIDIA.

From this reading I found that I needed to first convert my ray-tracer into a path-tracer, where we simulate light paths (rays) bouncing round the room until they hit a light surface. Only when the ray intersects with an area light does it gain luminance which then dictates the irradiance of a given point in the room along with the diffuse surfaces it bounced off. This required me to remodel my point light and convert it into an area light as well as redesign my implementation of `Triangles` and create the child classes `Surfaces` and `AreaLight` which extend from it. Whilst also modifying the ray-tracing Monte Carlo global illumination method to be a path-tracing Monte Carlo global illumination method [10]. Instead of scattering the rays every bounce, I instead sample many rays for a single pixel to begin with and just trace that single ray until it reaches a light source (or other terminating conditions).

With the path-tracer ready, I was able to begin working on reimplementing the data structure which [1] relies on, The Irradiance Volume [2]. The irradiance volume data structure stores the irradiance for many sampled points in the room, for which when you intersect with some geometry in the scene, you can interpolate between the precomputed irradiance values stored at these points to find the predicted irradiance at any point in the scene. The more sample points you have in the room the more accurate your estimate will be. For every sampled point the irradiance is calculated by finding the radiance at the given point by calculating all light incoming from uniformly sampled discretized angles (a grid converted into a hemisphere around the sample point). I managed to implement a single Irradiance Hemisphere and visualise it in the scene before the end of the week.

K. Resources

The most useful sources this week were the NVIDIA paper [1] (which I will no longer mention in these sections as the thesis is essentially based around it) and the reinforcement learning book [9].

L. Reflection

This week was different from those so far, I really focused on research and have a lot of notes to show for it, as compared to the past two weeks which have essentially been based around implementation of a ray-tracer. The work is getting more challenging and open to thought and interpretation of what cutting edge work has already been done by NVIDIA. I can safely say I met my 3 goals I outlined for this week.

WEEK 4

I now have a basis point to start off with the Irradiance Volume [2] implementation as I have successfully simulated a single `RadianceVolume` (class). It was now time to build this structure into the rendering pipeline and using these sampled point estimates into my approximation of global illumination for given point in the scene.

M. Goals

1) Find a way to sample radiance volumes around the scene uniformly and use their estimates in the rendering pipeline to create a perceptually realistic image.
2) Create KD-Tree in order to quickly look up the closest Radiance Volumes around a given point in the scene. Also introduce the `icc` Intel compiler to speed up programs performance.
3) Use Trilinear interpolation between the radiance volumes in the scene to find the irradiance estimate for a given point in the scene.
4) Begin implementing the reinforcement learning approach with the Irradiance Volume data structure.

N. Research/Implementation Goals

As for the first goal, I decided to sample the radiance volumes uniformly by sampling n radiance volumes on a given surface (using that surfaces normal), where the value of n is determined by the area of that triangle. This meant the algorithm is able to adapt to different scenes. This is different to what is proposed by [2] where a bilevel grid is used. This would likely give a more accurate estimate on surfaces as will focus all of our radiance volumes on surfaces rather then in the middle of the scene. This technique seems to produce good quality images.

The KD-Tree was a fairly straightforward implementation based upon code from my 3rd year computer graphics unit we implemented for a Photon Map. This made it quick to get working for an irradiance volume data structure and it made a massive improvement on query time, $O(\log n)$ (on average) to find the closest irradiance volume to given point in the scene.

With an efficient way to lookup the radiance estimate at a point in the scene, I created a way for visualizing the scene using interpolation between the stored radiance values alone. This is similar to a photon mapper and it enabled me to render images in almost real-time (after the pre-compute of the radiance volumes of course). I also implemented a barycentric coordinate system to interpolate between radiance volumes location to find an irradiance estimate. However, all of this was more or less just to check if my radiance map was implemented correctly by visualizing the result, it is not the rendering approach [1] take.

O. Resources

Clearly the Irradiance Volume paper [2] by Peter Shirley (who is an important person in graphics research) was a huge help for implementing the Radiance Volume, even if my version is quite heavily modified in how it used.

As for the KD-Tree my Github repository for my 3rd years constructed Photon Map was very useful.

P. Reflection

This week my work was heavily focused around creating this Radiance Map and visualizing it. I have managed to get some good evidence of its correctness and this is essential as it is key that it is implemented correctly for the reinforcement learning approach undertaken by [1] to work.

WEEK 5

With the radiance map created and validated, this week brings together a lot of what I have previously implemented and mainly involves implementing the reinforcement learning for light paths rendering approach.

Q. Goals

1) Implement an importance sampling approach based on the pre-computed data from the radiance map
2) Implement an initial version of the reinforcement learning approach and attempt to get similar results to that seen in [1]

R. Research/Implementation Goals

I quickly managed to get an importance sampling version of the path-tracer working by converting the radiance-volume values into a cumulative distribution which I could sample from the inverse of, in order to fire rays more likely towards light sources. This worked well at increasing the number of light rays which intersect with the area light source however some artefacts and strange patterns are present in the image, could this be to do with what [1] suggests about this not being a complete approximation?

I then moved on to implementing the reinforcement learning path tracer which was also fairly quick to implement. This does not perform the pre-computation step and clearly traces light paths towards the light far more effectively then randomly sampling, however once again there are artefacts present. I believe this is to do with my implementation of the radiance map, therefore I am going to plot the Voronoi render of my scene for the radiance volumes as shown in [1] to validate I have a similar layout. This will be effective in the future for debugging and visualizing the density of my samples. I have also realised that [1] might use a different sampling method for radiance volume locations using a low-discrepancy sequence (Hammersley) which I am planning on researching as well. I need to get this base datastructure correct before I can validate my reinforcement learning approach.

I have managed to implement the initial version of the Reinforcement Learning Path Tracer. This version is based upon the CPU and is currently very slow. I now have 4 different rendering methods available to me (although they are not easy to switch between right now), all of which produce different results in different time periods. I would currently say the pre-compute importance sampling approach gives the best results, likely due to the number of rays initially shot however, the pre-computation is very computationally intensive.

I have begun to research into Cuda in order to run my code on a GPU.

S. Resources

Primarily this week I was just implementing the details in the following papers [1] [2] [8].

T. Reflection

The reinforcement learning approach seems to be learning where the light, however it si difficult to validate it without rendering an image which is near convergence (i.e. has very little noise). With a converged default path tracer render and a converged reinforcement path tracer render, I would be able to compare the two (for the same scene) and figure out if the reinforcement path tracer is working correctly just by visual observation.

The issue I am faced with at the end of this week is that I need to compare the methods against one another and validate their correctness via comparing the renders to a default forward path tracer. However, the render time is far too long and switching between the current implemented methods is not very easy. It would takes days to compute a high quality

converged render for all methods, therefore I look to the GPU now to parallelize my code.

WEEK 6

Cuda will be the story for this week. I have never written code for a GPU and I am unfamiliar with their architecture. Luckily, in my own PC I have a 1070Ti which is one of the most powerful 10 series cards (an 20 RTX series would be nice for my future deep learning). I decided to research Cuda because simply I had heard about it more then OpenCL and saw there were a few tutorials online for getting it to work with a standard path tracer, compared to OpenCL.

I want to stress that the main aim of my project is to not make a path tracer as fast as possible. I believe that I would never win this game in the time period available. There are so many tips and tricks and great experience built up in industry it is useless to compete on this playing field. I instead look to reduce memory consumption and potentially improve the accuracy of the method introduced by [1]. Therefore, the GPU code is only being written so I can validate my results in a couple of minutes compared to a couple of days!

U. Goals

1) Research into Cuda and understand the basic architecture of a GPU and how it compares to a CPU. This will help dictate how I optimally code for the GPU.
2) Transfer the Default path tracer to run on the GPU and correctly work with Cuda. This will require class modifications of my entire code base!
3) Transfer the Reinforcement Path Tracer and Voronoi plot to work on the GPU.

Research/Implementation

I began my research into Cuda by taking a look at some notes on the architecture of the Cuda GPU, however I could not find a lot describing it awfully well, but I did find some useful explanations of the differences on Stack Overflow and Quora which gave me enough confidence to start following the tutorial for implementing a default path tracer in Cuda.

The tutorial links to sources for an introduction to Cuda and was overall very helpful and self contained [3]. From this, it took about 3 days of solid programming and redesigning of code-base in order to support the Default path tracer on Cuda. Not only did this require the modification of all my classes syntactically, but semantically a lot of the program had to be written in order to run on the device. For example, Cuda has no knowledge of the `std` library, hence there is no easy way (unless using thrust) to work with a vector type. Most of my code relied on this! As mentioned I could have used thrust, but I wanted to be safe and stick with what there is plenty of support for, basic C++ Heap arrays. This gave me a much better understanding of the memory structure of my program. The only one thing I wish I could keep were smart pointer in C++ (being part of the `std` library) due to their automatic memory management.

While converting the default path-tracer was difficult, it was much harder to convert the reinforcement path tracer, as I had no tutorial holding my hand the way through it. As with the natural progression of things, I now understood Cuda so I had the confidence to apply to my own problem. Now the major difference between the default and reinforcement path tracer in terms of concurrency is its writing/reading from a global array in heap memory during rendering. This means race conditions need to be taken care of when performing the update rule, Cuda atomic operation constructs made this relatively easy. My major difficulty was redesigning the K-D Tree I implemented for quick nearest neighbour search. Initially I got the reinforcement path tracer to work with linear traversal to find the closest radiance volume, but this was slow with thousands of radiance volumes in the room. The K-D Tree was not trivial to implement on the GPU, unlike your standard interview question regarding binary trees, I was not able to define the K-D Tree recursively to be stored on the GPU as I had to `cudaMemcpy` the entire structure to the device with all correct pointers relating to one another within the tree. I remembered that from my algorithms classes any tree structure could be rewritten to be in the form of an array, and that is exactly what I did in order for it to work on the GPU. I implemented this in a couple of days in Cuda and tested it, finding over a 10x speedup from my linear search implementation!. Note, never define any device code in Cuda recursively it's super slow! For example, writing the tree traversal algorithm for finding the nearest neighbour recursively was actually slower then the linear search algorithm. Instead, I implemented my own stack class of Cuda and created an iterative search in the tree using this, and this is where my 10x speed-up came from.

Note, I also implemented the voronoi plot for debugging nearest neighbour search on the GPU.

V. Resources

Nvidia documentation and tutorial were massively helpful this week [5] [4], and of course stack overflow saved me multiple times.

W. Reflection

I now have 2 different path tracers, the default and reinforcement approach both written on the GPU. Both can render an image which has relatively low noise (low enough to tell if the algorithm is converging correctly) in just a few minutes compared to a few days! From this I managed to validate my reinforcement learning implementation and gain the similar benefits to those seen in [1]. I would not call myself a 'good' Cuda programmer, however I know my way around and feel confident using it now.

With this complete I have essentially implemented everything I need from [1]. Now everything I work on is new territory to my knowledge! I will need to do a lot of research on how to move forward!

WEEK 7

With the [1] simple path tracer method implemented, I now need to research non-linear function approximators and see which form applies best for my problem; Given a position in 3D space, output a sampled direction for the ray to scatter in, where the distribution to sample from is the learned irradiance distribution for each x, y, z in space. There are a lot of question that come with this which I will need to research.

My other task is to complete my research poster to go with my project, which is due in on Wednesday this week. So, I will be spending a couple of days doing that this week.

X. Goals

1) Complete the research poster
2) Research a way of modelling all irradiance distributions in the room as a non-linear function (or may have to model each one individually as a non-linear function, this I am currently uncertain about)

Y. Research/Implementation

I mainly worked on the poster for the first 2 days of the week and reflected on what I have done so far. Not much to report on this.

REFERENCES

[1] Ken Dahm and Alexander Keller. Learning light transport the reinforced way. *arXiv preprint arXiv:1701.07403*, 2017.

[2] Gene Greger, Peter Shirley, Philip M Hubbard, and Donald P Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[3] Nvidia. Accelerated ray tracing in one weekend in cuda. https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/.

[4] Nvidia. Cuda programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[5] Nvidia. An even easier introduction to cuda. https://devblogs.nvidia.com/even-easier-introduction-cuda/.

[6] opengl. Model loading. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/.

[7] ScratchPixel. Scratchpixel2.0. http://www.scratchapixel.com/.

[8] Peter Shirley and Kenneth Chiu. Notes on adaptive quadrature on the hemisphere. Technical report, Technical Report 411, Department of Computer Science, Indiana University , 1994.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2017.

[10] Wikipedia. Path tracing. https://en.wikipedia.org/wiki/Path_tracing.