METROPOLIS LIGHT TRANSPORT

Ivan Iglesias

April 2020

1 Introduction

Metropolis light transport is a procedure that constructs paths from the eye to a light source using a path tracer, then with the information of the previous path constructs modifications. Once a path to a light source has been found, the algorithm is going to explore nearby paths. In scenes with difficulty accessing light sources, the MLT is going to focus on effective paths to the light and sample these paths resulting in a faster solution to the final image.

Figure 1: Metropolis Scene Example [1]

2 MLT Paper

The original paper of Metropolis Light Transport was proposed by Eric Veach in 1997. Veach illustrated the idea of storing the paths of the light and applying modifications to sample paths that have a higher probability of reaching the light. However, the mutation strategy proposed by this first approach was complicated and hard to implement. After

Veach published the Metropolis Light Transport new mutation strategies have appeared that demand to be easier to implement and equally robust. In this paper, we are going to follow one of them that is call Primary Sample Space Metropolis Light Transport. This paper presents a new mutation strategy that works in the space of uniform random numbers used to build up paths. Instead of mutating in the path space, mutations are realized in an infinite-dimensional unit cube of pseudo-random numbers.

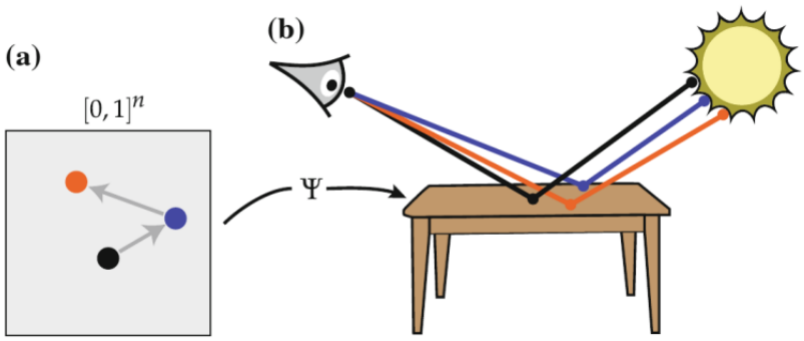


Figure 2: Mapping from Random Space to Path Space [4]

3 My implementation

I have implemented a Regular Monte Carlo path tracer and added on top the PSSMLT following the equations proposed by the paper. In the following explanations, I am going to go straight to the Metropolis implementation steps.

3.1 Generating the paths

The algorithm starts by generating random paths and approximating the luminance of the paths to select the first sample-path. In my implementation, I started with this approach but then I moved to a simpler and faster one that gives almost identical results. Instead of sampling a hardcoded number of paths, I am going to sample until a path returns a luminance value higher than zero and start mutating from this path.

```
while (1)
{
    mlt.ResetRandomCoords();
    //Compute new path
    PathSample sample = generate_new_path(info.cam, info.m_width,
                                          info.m_height, mlt);
    mlt.global_time++;

    //Clear the stack
    while (!mlt.primary_samples_stack.empty())
        mlt.primary_samples_stack.pop();

    auto value = luminance(sample.F);
    //Check if valid path
    if (value > 0.0f)
    {
        b = value;
        p_large = 0.5f;
        old_path = sample;
        break;
    }
}
```

Figure 3: Generate Initial Path

3.1.1 Generating New Path

The function "generate_new_path" is going to sample a new path as a regular path tracer would but, instead of calling a new random value each time we are going to call the function "NextSample"

```
int x, y;
x = mlt.NextSample() * width;
y = mlt.NextSample() * height;
```

Figure 4: Choose Pixel to Sample

3.1.2 Storing and Computing Modifications

The function "NextSample" will be in charge of storing and returning the random values that will be used to generate new paths. It follows the code proposed by the paper.

```
float PrimarySample(int i) {
  if (u[i].modify < time) {
    if (large_step) {          // large step
      Push(i, u[i]);           // save state
      u[i].modify = time;
      u[i].value = random();
    } else {                   // small step
       if (u[i].modify < large_step_time) {
         u[i].modify = large_step_time;
         u[i].value = random();
       }
            // lazy evaluation of mutations
       while (u[i].modify < time-1) {
         u[i].value = Mutate(u[i].value);
         u[i].modify++;
       }
       Push(i, u[i]);          // save state
       u[i].value = Mutate(u[i].value);
       u[i].modify++;
    }
  }
  return u[i].value;
}
```

Figure 5: Choose Pixel to Sample [4]

The paper divides the modifications into large and small steps. The large ones will directly compute a new random for the modification while the small ones will mutate the previously stored value as we can appreciate in the code above. The small modifications will have more probability of being accepted but the large ones will help to sample new zones.

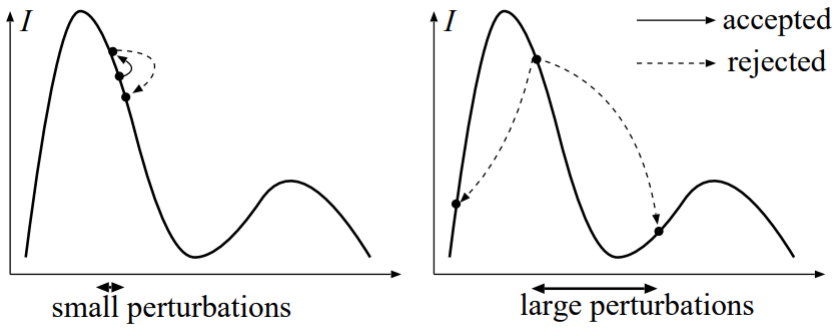


Figure 6: Small and Large modifications [4]

The values that are being mutated are the ones we stored from the previous path and then we store and return the modification for future paths.

3.2 Sample a Tentative Path

Every mutation we are going to sample a new path based on the previous information.

3.2.1 Accept or Reject

Metropolis method constructs a Markovian process whose stationary distribution is p(z) to generate samples according to p(z) = 1/b*I(z). This sample is going to be accepted or rejected based on an acceptance probability if this acceptance probability is greater than 1, then the sample is directly accepted.

$$a(\mathbf{z}_i \rightarrow \mathbf{z}_t) = \min\left\{\frac{I(\mathbf{z}_t)\cdot T(\mathbf{z}_t \rightarrow \mathbf{z}_i)}{I(\mathbf{z}_i)\cdot T(\mathbf{z}_i \rightarrow \mathbf{z}_t)}, 1\right\}$$

Generate random number r in $[0, 1]$.
if $r < a(\mathbf{z}_i \rightarrow \mathbf{z}_t)$ **then** $\mathbf{z}_{i+1} = \mathbf{z}_t$
else $\mathbf{z}_{i+1} = \mathbf{z}_i$

If the sample is accepted then, we will keep the new path for the next iteration otherwise we discard it.

3.2.2 Store the contribution

Even if we are discarding a path, we are going to store the information obtained from this path in our final image. It makes sense to keep this information because this path was created from a path that was already accepted. We will compute the weights of both paths and finally add them to final image.

```
float a = std::min(1.0f, luminance(new_path.F) / luminance(old_path.F));

const float new_path_weight = (a + mlt.large_step) / (luminance(new_path.Color) / b + p_large) / M;
const float old_path_weight = (1.0f - a) / (luminance(old_path.Color) / b + p_large) / M;

tmp_image[newPos] = tmp_image[newPos] + new_path.weight * new_path_weight * new_path.Color;
tmp_image[oldPos] = tmp_image[oldPos] + old_path.weight * old_path_weight * old_path.Color;
```

Figure 7: Compute Weights and Store

4 Examples Demo

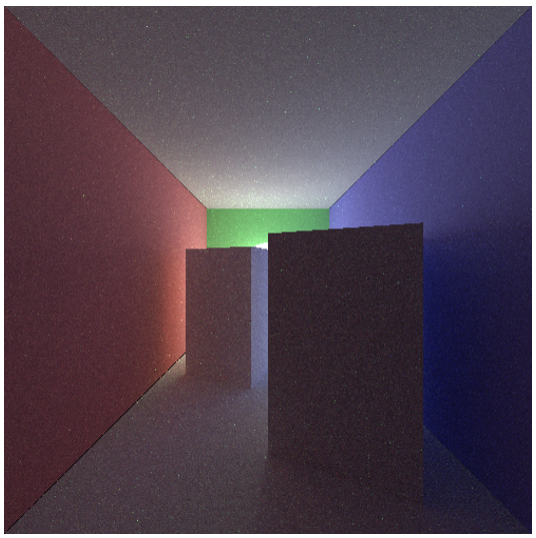
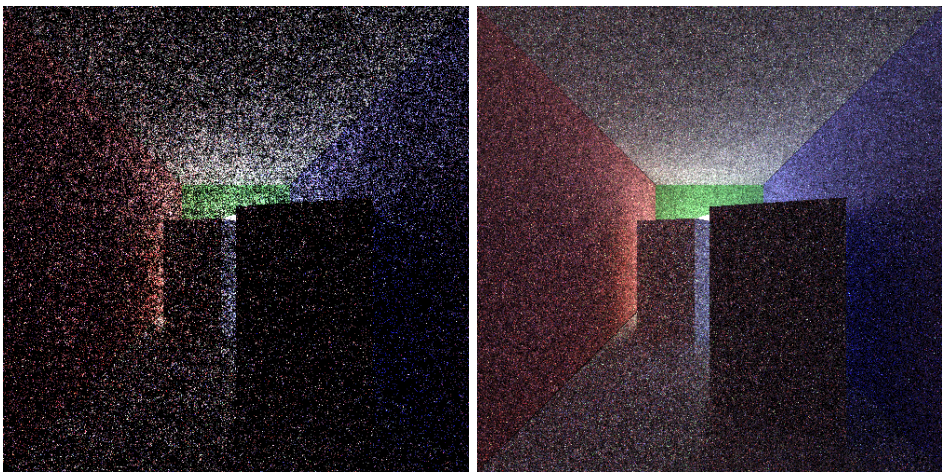


Figure 8: 50 mutations, 15 threads, 26 minutes

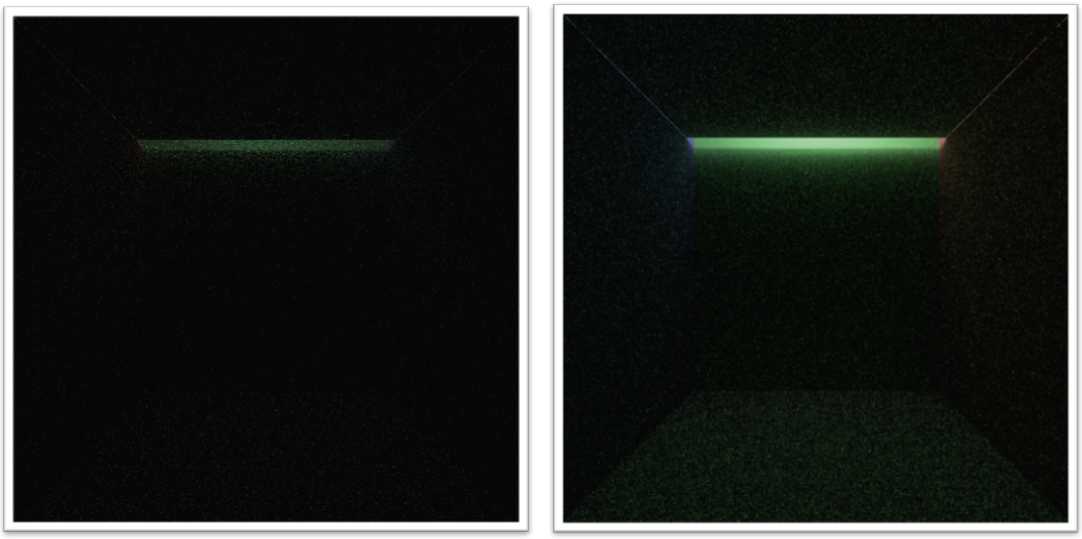
This Cornel Box scene has a light behind and two boxes hiding it. The rays of the camera will not reach the light easily so this scene is a nice example for MLT. As explained before the MLT will focus on sampling paths that have more probabilities on reaching the light. In this image, we can appreciate how the part where the light is has less noise because it has been sample more times that corners of the image. Now I am going to show different results varying the mutation number.



(a) 1 Mut, 1 Thread, 8 seconds (b) 10 Mut, 1 Thread, 1 min

Figure 9: Different Mutations

The more mutations we add the more time the image takes but the clearer it looks. Now, the idea is to see where this MLT is best at, so we are going to set a hardly illuminated scene and compare it with a regular path tracer.



(a) Monte Carlo (b) MLT

Figure 10: Cornel Box with Light hidden

Even if the images are pretty dark we can appreciate the differences. In the Monte Carlo result, we can hardly see the light while the Metropolis has almost no noise in the light shaft.

4.1 How to use the Demo

The demo is a Metropolis Light Transport path tracer that includes some scenes to test it with. To run the program we just need to use a similar command line to this.

```
[executable] [width] [height] [mutations] [threads] [depth]          [fileName]

./MLT.exe        500      500        1            1         5      ../scenes/CornellBox2.txt
```

Figure 11: Command Line

We can modify the size of the final image, the number of mutations per pixel that we want to perform, the number of threads that we want to use and the Maximum Depth. Depending on the parameters the image will take a longer time to render and once it is finished the result will show up on the window at it will also produce a png.

5 Other Approaches

5.1 Bi-directional Path tracing

The bi-directional path tracer is another approach to solve this solution. It works but generating the light from both sources, the light, and the camera and creating connections between them. It is not one or the other, the Metropolis Light Tracer can be built up on top of a bi-directional path tracer but it implies adding an overhead that maybe is not worth it.

6 Problems during development

It is very hard to debug and work with random values. Sometimes I didn't know if the program was not working or simply the image was too dark. Problems with threads that took me a while to understand.

7 Conclusion

I think that Metropolis Light Transport is a good approach to solve the rendering of hardly illuminated scenes. There are different modifications to the original one, like PSSMLT, these different approaches try to be equally robust but easier to implement. However, I think that the line between when an MLT is helping and when is not is very thin and needs to be chosen with caution not to add unnecessary overhead to the path tracer.

References

[1] Original MLT,
Link

[2] Primary Sample Space Metropolis,
Link

[3] Anis Benyoub Presentation MLT,
Link

[4] Light Transport III,
Link