Reinforcement Learning for Light Transport

Rwitaban Goswami

19 June 2020

Abstract

We use the similarities between temporal difference learning and the light transport equation, which was laid out in [1], to integrate reinforcement learning into an in house path tracer. This will help us to easily modify the path tracer code and insert the qtable updation and sampling, which in turn will make the path tracer converge faster, as the number of occluded sampled rays are reduced.

1 Introduction

There are two approaches to rendering scenes:

- Rasterization
- Ray/Path Tracing

Rasterization involves iterating over each polygon, and then over each fragment, and doing a *visibility* test (using the Z buffer) to determine whether that fragment will be written to the output buffer.

Ray Tracing involves iterating over each screen space fragment, and doing a a *ray intersection* test to find the polygon it intersects, and then use the rendering equation to find the color of the fragment

Rasterization is inherently fast and hardware accelerated by the GPU, because the *visibility* test takes $\mathcal{O}(1)$ time. Hence online rendering is possible with rasterization. But Ray tracing is neither hardware accelerated (commercially only **NVIDIA RTX** cards support ray tracing hardware acceleration as of now [6]), nor is *ray intersection* test fast. Naively it takes $\mathcal{O}(n)$ time and can be reduced to $\mathcal{O}(\log n)$ time if acceleration structures are used. Hence only offline rendering is possible with ray tracing.

Hence there is a need for making the rendering time of each frame faster. Introducing Reinforcement Learning to Ray Tracing is one such attempt at making the process online.

Traditionally Ray/Path tracers use *importance sampling* to calculate an approximation to the LTE (4). The paths are often sampled from the bs/rdf as the exitant radiance at each point is unknown (it is what we are trying to calculate in the first place!). Reinforcement learning will learn a discretizatin of the exitant radiance and will help the rendered converge faster.

2 Concepts

2.1 Rendering Equation

2.1.1 Light Transport Equation

The rendering equation (or light transport equation **LTE**) is a Fredholm equation of the second kind: [4]

$$L_o(\mathbf{p}, \omega_o, \lambda, t) = L_e(\mathbf{p}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{p}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{p}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, d\omega_i \tag{1}$$

We can assume the scene to be static and drop the time dependance, and drop the λ dependance because we will only need to calculate the intensity for the three channels *red, green, blue*. Thus we get the **LTE** [7]

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_\Omega f_r(\mathbf{p}, \omega_i, \omega_o)\, L_i(\mathbf{p}, \omega_i)\, (\omega_i \cdot \mathbf{n})\, d\omega_i \tag{2}$$

where

- $L_o(\mathbf{p}, \omega_o)$ is the outgoing radiance from point $\mathbf{p}$ in the direction ω_o
- $L_e(\mathbf{p}, \omega_o)$ is the emmitted radiance from point $\mathbf{p}$ in the direction ω_o
- $\int_\Omega \dots d\omega_i$ is an integral over the solid angle Ω subtended by the unit hemisphere centered around $\mathbf{n}$ containing all possible values of ω_i
- $\mathbf{n}$ is the surface normal at $\mathbf{p}$
- $f_r(\mathbf{p}, \omega_i, \omega_o)$ is the *bidirectional reflectance distribution function*, or brdf
- $L_i(\mathbf{p}, \omega_i)$ is the incident radiance at point $\mathbf{p}$ coming from the direction ω_i
- $\omega_i \cdot \mathbf{n}$ is the weaking factor of outgoing irradiance due to incident angle, the light flux is smeared across a surface whose area is larger than the projected area perpendicular to the ray. This is often written as $\cos\theta_i$

We assume no participating media, and hence the radiance is constant along rays through the scene. We can therefore relate the incident radiance at $\mathbf{p}$ to the outgoing radiance from another point $\mathbf{p}'$, as shown by Figure 1. If we define the ray-casting function $t(\mathbf{p}, \omega)$. as a function that computes the first surface point $\mathbf{p}'$ intersected by a ray from in the direction ω, we can write the incident radiance at $\mathbf{p}$ in terms of outgoing radiance at $\mathbf{p}'$: [7]

$$L_i(\mathbf{p}, \omega) = L_o(t(\mathbf{p}, \omega), -\omega) \tag{3}$$

Dropping the subscripts from L_o for brevity, this relationship allows us to write the LTE as

$$L(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_\Omega f_r(\mathbf{p}, \omega_i, \omega_o)\, L(t(\mathbf{p}, \omega_i), -\omega_i)\, (\omega_i \cdot \mathbf{n})\, d\omega_i \tag{4}$$

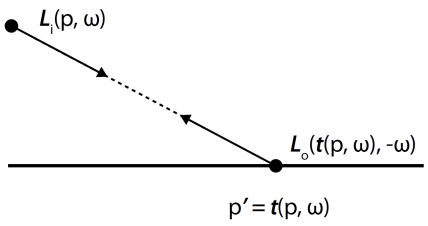


Figure 1: Radiance along a Ray through Free Space Is Unchanged

The key to the above representation is that there is only one quantity of interest, exitant radiance from points on surfaces.

2.1.2 Monte Carlo Integration

Now how do we calculate the solution to the LTE, i.e. $L(\mathbf{p}, \omega_o)$? We need to approximate it using an estimator. The popular estimator used is *Monte Carlo Estimator*

Suppose that we want to evaluate a 1D integral $\int_a^b f(x)\,dx$. Given a supply of uniform random variables $X_i \in [a, b]$, the Monte Carlo estimator says that the expected value of the estimator

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i) \tag{5}$$

$\mathbb{E}[F_N]$ is in fact equal to the integral. [7]

The restriction to uniform random variables can be relaxed with a small generalization. This is an extremely important step, since carefully choosing the PDF from which samples are drawn is an important technique for reducing variance in Monte Carlo. If the random variables are drawn from some arbitrary PDF $p(x)$, then the estimator

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{6}$$

can be used to estimate the integral instead. The only limitation on $p(x)$ is that it must be nonzero for all x where $|f(x)| > 0$. It is similarly not too hard to see that the expected value of this estimator is the desired integral of f [7]

This can easily be extended to multiple dimensions.

The thing to note here is that our goal here is to choose $p(x)$ in such a way such that the variance of F_N is minimal.

Thus, showing that the Monte Carlo estimator converges to the right answer is not enough to justify its use; a good rate of convergence is important too. It has been shown that error in the Monte Carlo estimator decreases at a rate of $\mathcal{O}(\sqrt{N})$

in the number of samples taken [7]. This is because with an appropriate sample distribution it is possible to exploit the fact that almost all higher-dimensional integrands are very localized and only small subspace notably contributes to the integral [5].

2.1.3 Path Integral Formulation of LTE

We can also write the LTE in a much more compact way which can help us compute it using Monte Carlo estimation [7]

$$L(\mathbf{p_1} \to \mathbf{p_0}) = \sum_{n=1}^{\infty} P(\bar{p}_n) \tag{7}$$

where

- $L(\mathbf{p_1} \to \mathbf{p_0})$ is the exitant radiance from a point $\mathbf{p_1}$ to a point $\mathbf{p_0}$, or $L(\mathbf{p_1}, \omega)$ where $\mathbf{p_1}$ and $\mathbf{p_0}$ are mutually visible and $\omega = \widehat{\mathbf{p_0} - \mathbf{p_1}}$
- $P(\bar{p}_n)$ is the contribution of path of length n,

$$= \underbrace{\int_A \int_A \cdots \int_A}_{n-1 \text{ times}} L_e(p_n \to p_{n-1}) T(\bar{p}_n)\, dA(p_2) \ldots\, dA(p_n) \tag{8}$$

- $T(\bar{p}_n)$ is the throughput,

$$= \prod_{i=1}^{n-1} f(p_{i+1} \to p_i \to p_{i-1}) G(p_{i+1} \leftrightarrow p_i) \tag{9}$$

- $G(p_{i+1} \leftrightarrow p_i)$ is the geometric coupling term

$$= V(p_{i+1} \leftrightarrow p_i) \frac{|\cos\theta||\cos\theta'|}{||\mathbf{p_{i+1}} - \mathbf{p_i}||^2} \tag{10}$$

- $V(p_{i+1} \leftrightarrow p_i)$ is the visibility function between points $\mathbf{p_{i+1}}$ and $\mathbf{p_i}$
- $f(p_{i+1} \to p_i \to p_{i-1})$ is the bs/rdf $f(p_i, \omega_o, \omega_i)$

It is this integral which we are going to estimate using Path tracing.

2.2 Reinforcement Learning

2.2.1 Temporal Difference Learning

The following material is sourced from [8]

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model

of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

TD uses experience to solve the prediction problem. Given some experience following a policy π, it updates its estimate V of v_π for the nonterminal states S_t occurring in that experience.

TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{11}$$

immediately on transition to S_{t+1} and receiving R_{t+1}. In effect, the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$

The most obvious advantage of TD methods is that they are naturally implemented in an on-line, fully incremental fashion. This ties in neatly with our goal of implementation of an online renderer.

2.2.2 SARSA

The first step is to learn an action-value function rather than a state-value function. In particular, we must estimate $q_pi(s, a)$ for the current behavior policy π and for all states s and actions a. In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state–action pair to state–action pair, and learn the values of state–action pairs. [8]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \tag{12}$$

This update is done after every transition from a nonterminal state St. If St+1 is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm. [8]

2.2.3 Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [9], defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{13}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. [8]

2.2.4 Expected SARSA

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}\left[Q(S_{t+1}, A_{t+1}) | S_{t+1}\right] - Q(S_t, A_t)\right] \quad (14)$$

$$\leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (15)$$

but that otherwise follows the schema of Q-learning. Given the next state, St+1, this algorithm moves deterministically in the same direction as Sarsa moves in expectation, and accordingly it is called Expected Sarsa. [8]

3 Background Literation

Now we come to the paper on integrating reinforcement learning and light transport [see 1]. Traditionally the $p(x)$ in (6) is taken to be the bs/rdf in (7). But this can lead to slow convergence, especially when the light sources are occluded (see Figure 2).

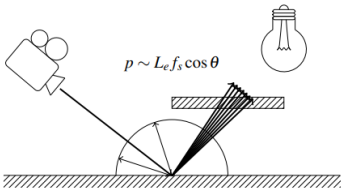


Figure 2: Most of the importance sampled rays are occluded hence have zero contributions

Hence there is a need to *learn* the incident radiance on each point so that it can be considered for importance sampling.

If we consider continuous instead of discrete space and apply the integral formulation of Expected SARSA (15), we get

$$Q(s, a) \leftarrow (1 - \alpha) Q(s, a) + \alpha [R(s, a) + \gamma \int_A \pi(s', a') Q(s', a')\, da'] \quad (16)$$

A comparison of (16) with (4) for $\alpha = 1$ reveals structural similarities of the formulation of reinforcement learning and the light transport integral equation, respectively, which lend themselves to matching terms: Interpreting the state s as

a location $x \in \mathcal{R}^3$ and an action a as tracing a ray from location x into direction ω resulting in the point $y := t(x, \omega)$ corresponding to the state s', the reward term $r(s, a)$ can be linked to the emitted radiance $L_e(y, -\omega) = L_e(htx, \omega), -\omega)$ as observed from x. Similarly, the integral operator can be applied to the value Q, yielding

$$Q(x, \omega) = L_e(x, \omega) + \int_{\mathscr{S}+} Q(y, \omega_i) f(y, \omega_o, \omega_i) \cos\theta_i \, d\omega_i \tag{17}$$

where we identified the discount factor γ multiplied by the policy π and the bidirectional scattering distribution function f. Taking a look at the geometry and the physical meaning of the terms, it becomes obvious that Q in fact must be the radiance $L(x, \omega)$ incident in x from direction ω and in fact is described by a Fredholm integral equation of the second kind - like the light transport equation. [1]

We therefore progressively approximate (17) using reinforcement learning: Once a direction has been selected and a ray has been traced by the path tracer,

$$Q'(x, \omega) = (1 - \alpha) Q(x, \omega) + \alpha \left(L_e(x, \omega) + \int_{\mathscr{S}+} Q(y, \omega_i) f(y, \omega_o, \omega_i) \cos\theta_i \, d\omega_i \right) \tag{18}$$

is updated using a learning rate α. The probability density function resulting from normalizing Q in turn is used for importance sampling a direction to continue the path. As a consequence more and more light transport paths are sampled that contribute to the image.

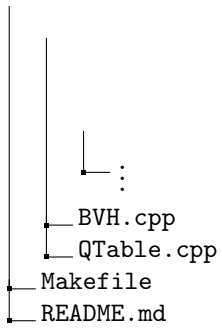
4 Implementation

Since open source production renderers like *Mitsuba* has it's own setup of build and class structure, I decided to implement this in my own in house path tracer which uses C++ *OpenGL* and its compute shader to do the task.

The entire codebase is hosted on `https://github.com/dropTableUsers42/raytracer`

The directory structure is

```
root/
├── bin/
├── include/
├── lib/
├── obj/
├── res/
└── src/
    └── Shaders/
        ├── geometry.glsl
        ├── qtablehelper.glsl
        └── trace3.cs
```



The main shader code implementing the path tracer is present in *trace3.cs*, *geometry.glsl* and *qtablehelper.glsl*

4.1 Bounding Volume Heirarchy

As mentioned earlier, each ray intersection test naively takes $\mathcal{O}(n)$. But that is simply not affordable for an online renderer. So I opted to use an acceleration structure called the *Bounding Volume Heirarchy* using stackless traversal [see 3]. This brings it down to $\mathcal{O}(\log n)$

The bvh and vertex data is passed into the shader via a *shader storage buffer object*

The detail of the contruction of this BVH can be seen in the recursive `buildBVH()` function in BVH.cpp. This function builds the BVH by splitting each node along the median of triangle positions of the longest axis of the node.

The stackless traversal is enabled using the function `buildNextLinks()`. It basically stores a `hitnext` and `missnext` pointer which are to be taken in case the ray misses/hits the node. These pointers are passed into the shader in the form of the index of the node they represent.

```
void BVHContainer::buildNextLinks(BVH *root)
{
    if(root->left != NULL)
    {
        buildNextLinks(root->left);
    }
    if(root->right != NULL)
    {
        buildNextLinks(root->right);
    }

    if(root->parent == NULL)
    {
        //root
        root->hitNext = root->left;
        root->missNext = NULL;
    }
    else if(root->isLeft && root->left == NULL)
    {
        //left leaf
        root->hitNext = root->missNext = root->parent
            ->right;
```

```
22      }
        else if(root->isLeft && root->left != NULL)
24      {
            //left non-leaf
26          root->hitNext = root->left;
            root->missNext = root->parent->right;
28      }
        else if(!root->isLeft && root->left != NULL)
30      {
            //right non-leaf
32          root->hitNext = root->left;
            BVH *next = root->parent;
34          while(!next->isLeft && next->parent != NULL)
            {
36              next = next->parent;
            }
38          if(next->parent != NULL)
            {
40              root->missNext = next->parent->right;
            } else
42          {
                root->missNext = NULL;
44          }

46      }
        else if(!root->isLeft && root->left == NULL)
48      {
            BVH *next = root->parent;
50          while(!next->isLeft && next->parent != NULL)
            {
52              next = next->parent;
            }
54          if(next->parent != NULL)
            {
56              next = next->parent->right;
            }
58          else
            {
60              next = NULL;
            }
62          root->missNext = root->hitNext = next;

64      }
    }
```

The intersection test is done in trace3.cpp in `intersectGeometry()`, by

basically unpacking the bvh which was done in `buildNextLinks()`

```
    void intersectGeometry(vec3 origin, vec3 dir, vec3
        invdir, out surfaceInteraction SI)
{
  int nextIdx = 0; // <- start with the root node
  float nearestTri = 1.0/0.0; // <- +inf
  vec3 normal = vec3(0.0);
  vec3 color = vec3(0.0);
  vec2 uv = vec2(0.0);
  int iterations = 0;
  int rettridx = -1;
  vec3 point = vec3(0.0);
  while (nextIdx != NO_NODE) {
    iterations++;
    if (iterations > MAX_FOLLOWS)
    {
      SI.tridx = -1;
      SI.color = ERROR_COLOR;
      SI.normal = ERROR_COLOR;
      return;
    }

    const node next = nodes[nextIdx];

    float t;
    if (!intersectBox(origin, invdir, next.min, next.
       max, t))
    {
      nextIdx = next.missNext;
    } else
    {

      if (t > nearestTri)
      {

        nextIdx = next.missNext;
        continue;
      }
      if (next.numTris > 0)
      {

        trianglehitinfo thinfo;
        int tridx = intersectTriangles(origin, dir,
           next, nearestTri, thinfo);
        if (tridx != NO_INTERSECTION)
```

```
42          {

44            normal = normalForTriangle(thinfo, tridx);
              uv = uvForTriangle(thinfo, tridx);
46            color = mtls[triangles[tridx].mtlIndex].Kd;
              nearestTri = thinfo.t;
48            rettridx = tridx;
              point = origin + nearestTri * dir;
50          }
        }
52      nextIdx = next.hitNext;
    }
54  }

56  SI.tridx = rettridx;
    SI.matidx = triangles[SI.tridx].mtlIndex;
58  SI.normal = normal;
    SI.color = color;
60  SI.point = point;
    SI.uv = uv;
62  return;
}
```

4.2 Path Tracing

The path tracing is implemented in trace3.cs, in the function `traceRL()`

First we setup the variables before shooting the ray. The variable `beta` represents the throughput in (9)

```
vec3 traceRL(vec3 origin, vec3 dir, vec3 invdir)
2 {
  vec3 L = vec3(0.0);
4 vec3 beta = vec3(1.0);
  int bounces;
6 vec3 prevN, prevPos;
  int prevObjId;
8 int prevQIdx = -1;
  surfaceInteraction prevSI;
```

We iterate infinitely

```
1 for(bounces = 0;;++bounces)
  {
```

Then we do an intersection test. If it fails, we have hit the environment and we set the color accordingly. If it succeeds, we check if we have hit a light source. If we have, we again set the color accordingly

```
surfaceInteraction SI;
intersectGeometry(origin, dir, invdir, SI);
bool foundIntersection = (SI.tridx != -1);
bool isAreaLight = false;

if(foundIntersection)
{
  if(bounces == 0)
    write(SI.point, SI.normal);

  if(mtls[SI.matidx].emitter)
  {
    L += beta * mtls[SI.matidx].Ke * 30;
    isAreaLight = true;
  }
}
else
{
  L += beta * SKY_COLOR;
  if(bounces == 0)
    write(vec3(0.0), vec3(0.0));
}
```

Then comes the crux of the paper [1], the QTable update function

```
if(bounces > 0)
{
  int temp;
  int status = updateQtable(prevSI, SI, dir, max(
      beta.r,max(beta.g,beta.b)), prevQIdx, temp);
  prevQIdx = temp;
}
```

Then we return the color if we had hit an area light, or if the bounces has exceeded the max bounces allowed (typically set to 5). We also update the point and the normal based on our intersection

```
if(!foundIntersection || bounces >= maxBounces ||
   isAreaLight) break;

origin = SI.point + SI.normal * EPSILON;
prevN = SI.normal; prevPos = origin; prevObjId =
   triangles[SI.tridx].objId;
```

Now we importance sample the outgoing ray. Normally we would be sampling the bs/rdf, but here we use sample the QTable as in [1].

```
vec3 sample_rand = randvec3(bounces);
//Sample BRDF for new ray
```

```
    vec3 wi; float pdf; vec3 f;
    wi = sampleOutDirRL(dir, SI, sample_rand, pdf, f,
        prevQIdx);
    beta *= f * dot(normalize(SI.normal), normalize(wi
        )) / pdf;

    dir = wi;
    invdir = 1.0 / wi;
```

Now we implement Russian Roulette to randomly stop the ray and weigh the result accordingly [7]. This ensures short path lengths

```
    float rr_rand = randfloat(bounces);
    float rrbetamax = max(beta.r, max(beta.g, beta.b))
        ;
    if(bounces > 3)
    {
      float q = max(float(0.05), 1- rrbetamax);
      if(rr_rand < q) break;
      beta /= (1.0 - q);
    }

    prevSI = SI;
```

with this we can stop the iteration and return the color

```
    }
  return L;
}
```

4.3 Reinforcement Learning

The reinforcement learning functions have been implemented in qtablehelper.glsl

4.3.1 Voronoi Division

As suggested in [1], I divided the scene surfaces into Voronoi cells, with the center of each cell representing each "state" in the QTable.

The voronoi division is done by first generating 2D Hammersley points and mapping it into 3D by using the uv texture mapping for each object. This data is passed into the compute shader using a *shader storage buffer object*

Then in qtablehelper.glsl, the voronoi cell for each point is calculated in the function `nearestWithNormalSpace()`

```
uint nearest_with_normal_space(vec3 pos, int objId,
    vec3 normal)
{
    float nearest = MAX_LENGTH;
```

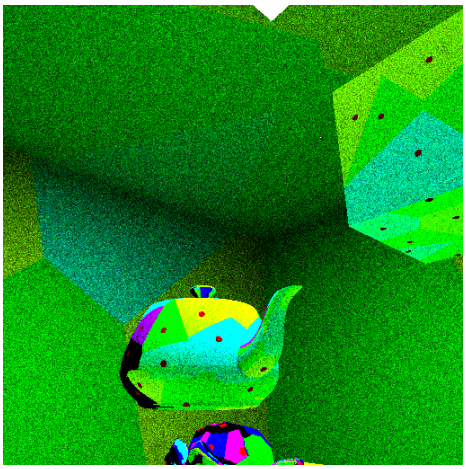


Figure 3: Division of scene into voronoi cells. Each voronoi centre is shown in red. Notice that the voronoi cell division also takes into account the nearest normal along with the nearest neighbor

```
    uint nearestIdx = 0;
    for(uint i = 0; i < numPoints; i++)
    {
        if(normals_per_voronoi[objId * numPoints + i]
           != vec3(0.0))
        {
            if(dot(normalize(normals_per_voronoi[objId
                * numPoints + i]), normalize(normal))
               > 0.866)
            {
                float len = length(voronoi_centres[
                    objId * numPoints + i] - pos);
                if(len < nearest)
                {
                    nearestIdx = i;
                    nearest = len;
                }
            }
        }
    }
    return nearestIdx;
}
```

4.3.2 Updating QTable

This is implemented in qtablehelper.glsl in `updateQtable()` and is fairly straightforward. The only thing of note here is that α is updated based on the formula $\alpha(s,a) = \frac{1}{1+\text{visits}(s,a)}$. This is to ensure consistency [suggested in 1].

```
int updateQtable(surfaceInteraction oSI,
    surfaceInteraction hSI, vec3 dir, float
    lastAttenuation, int prevIdx, out int nextIdx)
{
    uint idxPrev;
    if(prevIdx >= 0)
    {
        idxPrev = prevIdx;
    }
    else
    {
        idxPrev = findCell(oSI.point, triangles[oSI.
            tridx].objId, oSI.normal);
    }
    uint idxCurr = findCell(hSI.point, triangles[hSI.
        tridx].objId, hSI.normal);
    nextIdx = int(idxCurr);
    float update = 0;

    if(mtls[hSI.matidx].emitter)
    {
        update = clamp(length(mtls[hSI.matidx].Ke),
            0.0, 1.0);
    }
    else
        update = lastAttenuation * maxQ(idxCurr,
            triangles[hSI.tridx].objId);

    int idxQ = findIndex(idxPrev, triangles[oSI.tridx
        ].objId, -dir, oSI.normal);

    float alpha = 1.0 / (1.0 + visits[numPoints *
        triangles[oSI.tridx].objId + int(idxPrev)]);
    visits[numPoints * triangles[oSI.tridx].objId +
        int(idxPrev)]++;
    qt[idxQ] = (1- alpha) * qt[idxQ] + alpha * update;

    if(isnan(lastAttenuation))
        return -1;
```

```
    return 0;
}
```

4.3.3 Sampling the QTable

This is implemented in qtablehelper.glsl in `sampleScatteringDir()` and is fairly straightforward.

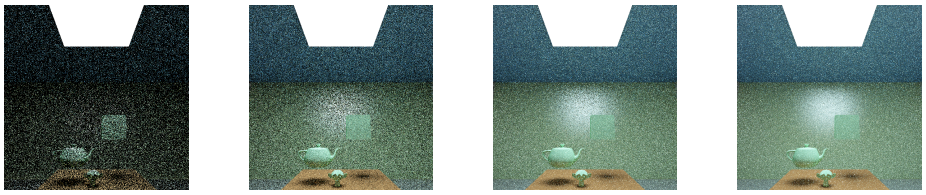
```
vec3 sampleScatteringDir(surfaceInteraction SI, int
    QIdx, vec3 rand, out float outpdf)
{
    uint idx;
    if(QIdx >= 0)
    {
        idx = QIdx;
    }
    else
    {
        idx = findCell(SI.point, triangles[SI.tridx].
            objId, SI.normal);
    }

    float pdf;
    ivec2 idxPatch = sampleCdf(int(idx), triangles[SI.
        tridx].objId, rand.x, pdf);

    vec3 dir = uniformSamplePatch(SI, idxPatch, rand.
        yz);

    outpdf = pdf * maxPhi * maxTheta / (2 * PI);

    return dir;
}
```

5 Results

Now we can look at some results of rendering using RL. The results are compared with rendering without RL, using importance sampling.

In each of the given figures, each spp takes 1 frame. 1000 spp means 1000 frames have been averaged to give the output. Hence for equal spp, higher framerates mean faster convergence.

As we can see, the rate of convergence in both RL and non RL versions are pretty similar. The RL rendering does seem to converge a bit faster than the non RL one. The RL rendering performs better in the scene where the light source is occluded, as is expected.

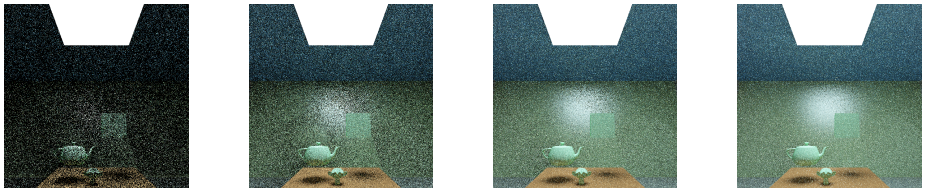


(a) 10 spp (b) 100 spp (c) 400 spp (d) 1000 spp

Figure 4: Rendering of a simple scene without RL

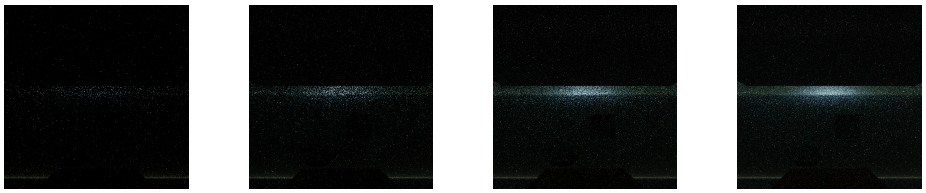
The RL rendering suffers from more fireflies than the non RL version in both scenes.

Though the rate of convergence in number of frames is similar, the RL version without next event estimation performs similar to the non RL version with, hence



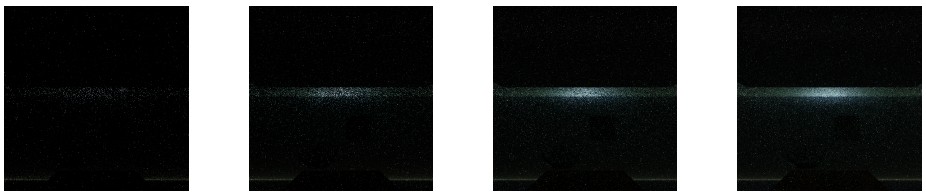
(a) 10 spp (b) 100 spp (c) 400 spp (d) 1000 spp

Figure 5: Rendering of a simple scene with RL



(a) 10 spp (b) 100 spp (c) 400 spp (d) 1000 spp

Figure 6: Rendering of a scene with occluded light source without RL



(a) 10 spp (b) 100 spp (c) 400 spp (d) 1000 spp

Figure 7: Rendering of a scene with occluded light source with RL

the RL version has faster fps due to the extra intersection tests in the next event estimation.

Non RL with NEE averages at about 76 milliseconds per frame in the simple scene, and 83 milliseconds per frame in the occluded light scene RL without NEE averages at 33 miliseconds per frame in the simple scene, and 37 milliseconds per frame in the occluded light scene

Hence RL achieves similar results with faster convergence, since the fps is faster, so the same spp with similar quality is reached faster for RL

6 Conclusion & Future Work

With these results we can conclude that integrating RL into light transport does make the rendering faster, but not by a huge margin. The rate of convergence is almost similar to the non RL rendering, even if the light source is fairly occluded. Despite these benefits, the RL rendering suffers more fireflies.

Future work that is possible in this area is to integrate RL into next event estimation itself. This can be done by learning the contribution of each *light source* to the point. In this case the action space will be the light sources instead of the direction solid angle. The light will be sampled from the QTable while estimating direct lighting.

Another possible area of exploration is to store the action and state space in some heirarchical data structure, maybe a BVH. This will speedup the updation and sampling quite a bit.

One more thing that can be done is to implement RL into a bidirectional path tracer.

One thing that has to be explored in this is to reduce the fireflies in the RL rendering. Some kind of filtering (e.g. Á-trous filtering [see 2]) can be done to avoid fireflies.

References

[1] Ken Dahm and Alexander Keller. "Learning Light Transport the Reinforced Way." In: *arXiv e-prints*, arXiv:1701.07403 (Jan. 2017), arXiv:1701.07403. arXiv: 1701.07403 [cs.LG].

[2] Holger Dammertz et al. "Edge-avoiding À-Trous wavelet transform for fast global illumination filtering." In: *HPG '10: Proceedings of the Conference on High Performance Graphics* (June 2010), pp. 67–75.

[3] Sebastian Dorn. *Stackless BVH traversal*. Mar. 2015. URL: https://sebadorn.de/2015/03/13/stackless-bvh-traversal (visited on 06/19/2020).

[4] James Kajiya. "The Rendering Equation." In: *ACM Siggraph* (1986), pp. 143–150. DOI: 10.1145/15922.15902.

[5] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2005. ISBN: 978-0-521-64298-9.

[6] Nate Oh. *NVIDIA Announces RTX Technology: Real Time Ray Tracing Acceleration for Volta GPUs and Later*. Mar. 2018. URL: https://www.anandtech.com/show/12546/nvidia-unveils-rtx-technology-real-time-ray-tracing-acceleration-for-volta-gpus-and-later (visited on 06/19/2020).

[7] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. 2004-2019. URL: http://www.pbr-book.org.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. A Bradford Book, Nov. 2018. ISBN: 0262039249.

[9] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards." PhD thesis. King's College, 1989.