

# Path Tracing

## CS500 – Project2

Rahil Momin

### Contents:

Anti-aliasing	2
Numerical Precision	2
Path Tracing Algorithm	3
Added functionality	4
Implicit light connection	4
Explicit light connection	5
Convergence	6

## Anti-aliasing:

Since the rays were always casted through the center of the pixel the generated image contained jagged edges. Therefore, to mitigate that an anti-aliasing method was added in the calculations. Instead of always casting the ray through the center of the pixel it is now casted through anywhere within the pixel by replacing  $x + 0.5f$  and  $y + 0.5f$  to  $x + \text{random}$  and  $y + \text{random}$ , where the random value is between  $0.0f$  and  $1.0f$ .

```
float dx = (2.0f * (x + myrandom(RNGen)) / width) - 1.0f;
float dy = (2.0f * (y + myrandom(RNGen)) / height) - 1.0f;
```

## Numerical Precision:

The algorithm suffers from numerical precision problem. This problem can cause the ray to change paths from within an object, causing it to intersect with the same object and repeat the behavior, this will in turn result that path to never find the light in the scene. To avoid that all intersection calculations were mended to return the first ray intersect of which the length along the ray is lesser than epsilon. In this project the epsilon value is

```
#define MINPEN 0.0001f
```

Sphere:

```
if (t1 < MINPEN && t2 < MINPEN)
{
    return false;
}

intersect.t = std::fmin(t1, t2);

if (intersect.t < MINPEN)
{
    intersect.t = std::fmax(t1, t2);
}
```

Box:

```
if (interval.t0 < MINPEN && interval.t1 < MINPEN)
{
    return false;
}

intersect.t = std::fmin(interval.t0, interval.t1);

if (intersect.t < MINPEN)
{
    intersect.t = std::fmax(interval.t0, interval.t1);
}
```

Triangle:

```

if (t < MINPEN)
{
    return false;
}

```

Cylinder:

```

if ((t0 == FLT_MAX && t1 == FLT_MAX) || (t0 < MINPEN && t1 < MINPEN))
{
    return false;
}

if (t0 > MINPEN)
{
    intersect.t = t0;
    intersect.N = interval1.n0;
}
else
{
    intersect.t = t1;
    intersect.N = interval1.n1;
}

```

### Path Tracing Algorithm:

The path tracing algorithm starts by initializing the color to (0,0,0) and weights to (1,1,1), then a ray casted through the pixel on the screen is checked for intersection if this intersection is a light then light's radiance is returned. If the object is a simple object the ray is changed to start from the point of intersection on that object into the world at a random direction. This process is repeated every iteration until the ray intersects with a light or until the current iteration is beyond the random limit.

To set the random limit a 'RussianRoulette' variable is set to 0.8f. every iteration a random number [0,1] is generated if this random number is less than RussianRoulette then the iteration is executed. As the ray bounces off many objects, their diffuse colour is multiplied with the weights divided by the probability of hitting that object. The probability also has check to avoid division by zero or near zero.

```

if (p < 0.000001f)
{
    break;
}

W *= (f / p);

```

At the end if a light is found. Weight is added to the colour as a product of light's colour, if not then the initial (0,0,0) is returned. For the purposes of better visuality the path trace result was enhanced by 2.0f.

```

return C * 2.0f

```

## Added functionality:

During the passes from 1 to infinity, the algorithm is set to produce the current output in between at certain passes milestones.

```
if (pass == 64 || pass == 128 || pass == 256 || etc...)
{
    // Write the image
    std::string outfile = inName;
    outfile.replace(outfile.size() - 4, outfile.size(), std::to_string(pass));
    outfile += ".hdr";

    WriteHdrImage(outfile, scene->width, scene->height, image, float(pass));
}
```

From the previous implementation, 9 more functions were introduced in order to supplement functionality of the path trace algorithm.

```
float GeometryFactor(Intersection& A, Intersection& B);

Vector3f SampleLobe(Vector3f N, float c, float phi);

Intersection SampleSphere(Vector3f C, float R, Shape* sphere);

Vector3f Material::SampleBrdf(Vector3f N);

float Material::PdfBrdf(Vector3f N, Vector3f Wi);

Vector3f Material::EvalScattering(Vector3f N, Vector3f Wi);

Intersection RayTrace::SampleLight();

float RayTrace::PdfLight(Intersection& Q);

Color RayTrace::EvalRadiance(Intersection& Q);
```

## Implicit light connection:

Implicit light connection is the base implementation of the path tracing algorithm, it is the connection that is only made if and only if the path reaches a light in the scene.

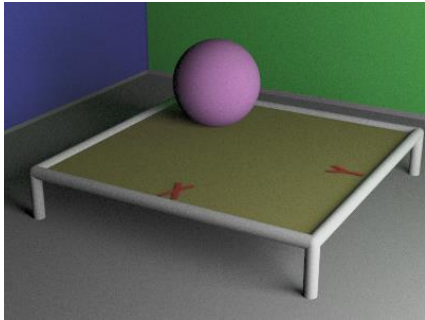
```
Color f = P.object->mat->EvalScattering(N, Wi);
p = Q.object->mat->PdfBrdf(N, Wi) * RussianRoulette;

if (p < 0.000001f)
{
    break;
}

W *= (f / p);

if (Q.object->mat->isLight())
{
    C += W * EvalRadiance(Q);
    break;
}
```

The following is the render result after 4096 passes (12.8 passes per second).



### Explicit light connection:

Explicit light connection is when at each bounce a shadow ray is casted from the point of intersection to a random light, if the ray's intersection returns the same light object then the light results are added to result.

```
Intersection L = SampleLight();
p = PdfLight(L) / GeometryFactor(P, L);
Wi = L.P - P.P;
Wi.normalize();

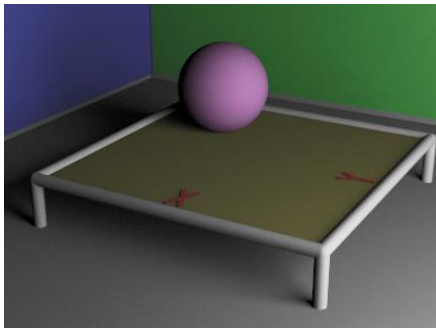
mini.Reset(P.P, Wi);
BVMinimize(tree, mini);
Intersection I = mini.record;

if (p > 0.000001f && I.object && I.object == L.object)
{
    Color f = P.object->mat->EvalScattering(N, Wi);
    C += 0.5f * W * (f / p) * EvalRadiance(L);
}
```

And a part of the implicit connection is changed to

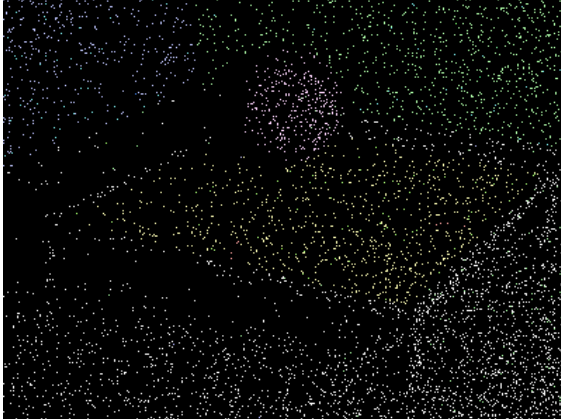
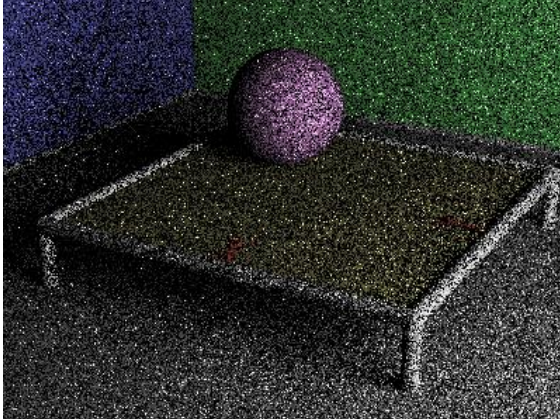
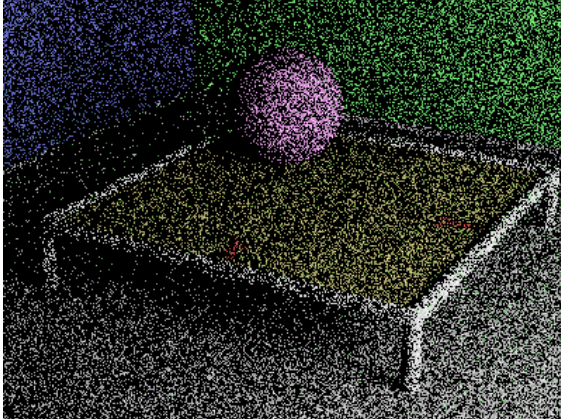
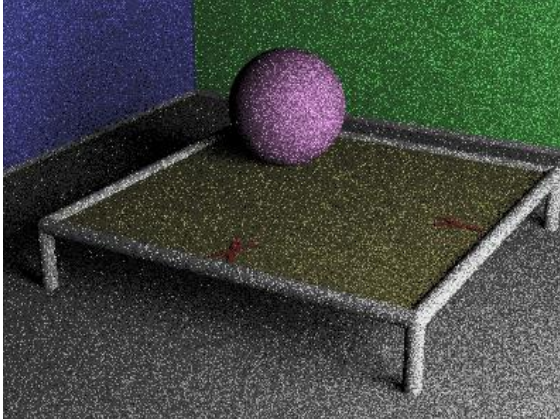
```
C += 0.5f * W * EvalRadiance(Q);
```

The following is the render result after 4096 passes (7.11 passes per second).



Which results is the same picture but is much more detailed due to faster rate of convergence.

## Convergence:

Pass	Implicit light connection (12.8 passes per second)	Implicit + Explicit light connection (7.11 passes per second)
1		
8		
64	