GRP1 Report

Martin Stuurwold - 120104

6 November 2013

Contents

Implemented Features

Pathtracing

A pathtracer works by shooting a number of 'photons' from the camera's location, through a view plane in 3D space that represents a 2D screen's coordinates. Each photon starts with a vector of three floats representing its energy. Every time the photon intersects an object in the scene, its direction and energy are altered depending on the material of the surface hit, as well as the angle. When the photon hits a light source, its energy is returned and represents a colour of the pixel on the 2D screen of the application.

By turning the raytracer into a pathtracer, the images produced 'automatically' gain a number of visual improvements such as soft-shadowing and ambient occlusion due to the nature of how a pathtracer closely resembles light transport in real life. These 'automatic' improvements will be discussed and explained below.

Converging the Image

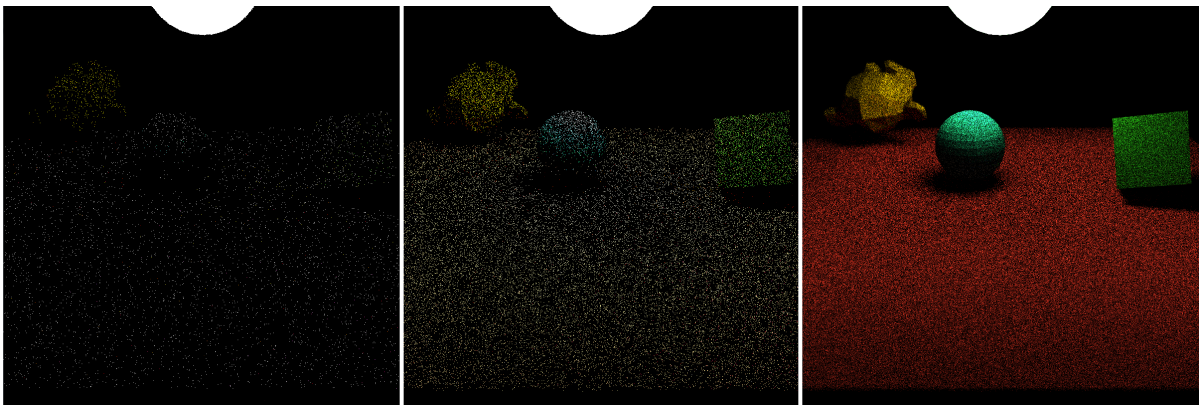
A pathtracer relies heavily on the Monte Carlo algorithm which states that by sampling enough random spots in the scene, given enough samples it will finally converge to the answer.

A pathtracer needs to take multiple samples of the same scene in order to create a realistic image. This works because each sample is slightly different than the previous one due to randomness determining a lot of factors about each individual photon, such as its starting location and direction. When all the samples are layered over each other, a proper scene begins to emerge.

To ensure that the colours are not clamped when the samples are merged together, the image is saved in a colour buffer that allows HDR representations of colour instead. This HDR colour buffer is then clamped to a value between zero and one to give to the 2D screen's pixel buffer.

The colour of the colour buffer has a larger weight than the returned colour as more samples are calculated. They are then added together and divided by the current number of samples to calculate the new colour, which is then placed back into the HDR colour buffer. A code sample for the converging of colours is shown below.

```
1 float3 color = Trace( photon );
2 color = (( color + m_colourBuffer[x + (yTimesWidth>>1)] * (float)a_iteration)) / ((float)
      a_iteration + 1);  // yTimesWidth>>1 is because the colour buffer is half the width of the
      screen
3 m_colourBuffer[x + (yTimesWidth>>1)] = color;
4 screen->GetBuffer()[x + yTimesWidth] = ConvertFPColor( color );
```



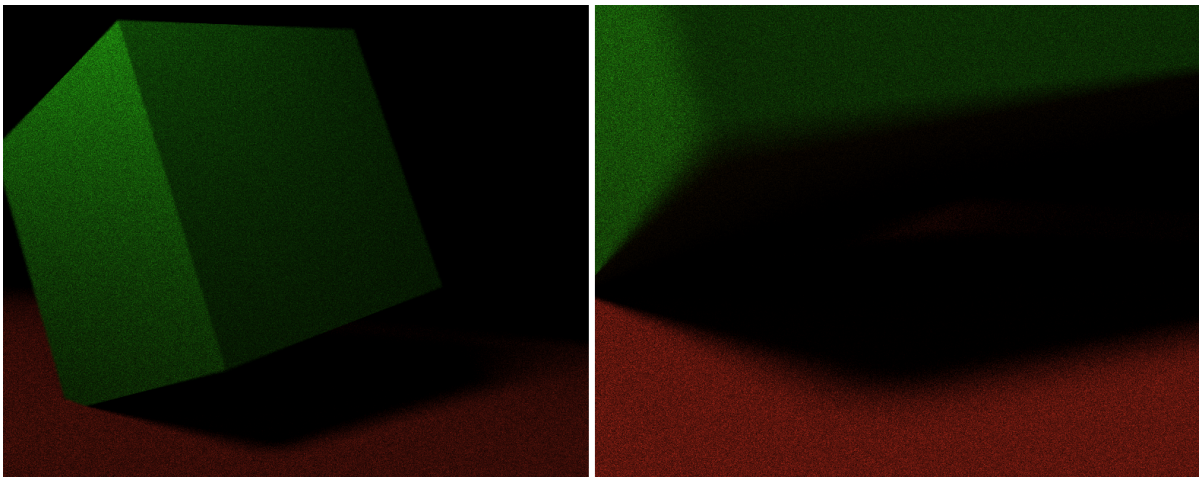
A scene with an increasing amount of samples per pixel

Soft Shadows

Soft shadows only work with area lights and are an example of what pathtracing 'automatically' adds to the final rendered image. Because in a pathtracer the path is only terminated when the photon hits a light source, photons that find themselves in a position where a direct path to the light source is obstructed will bounce around more before hitting the light source, which causes them to lose energy and return a darker pixel.

Soft shadows in the pathtracer can be seen when the object casting the shadow distances itself from the surface it is casting the shadow to. Shadows which are closer to the object casting it appear to have a much sharper edge.

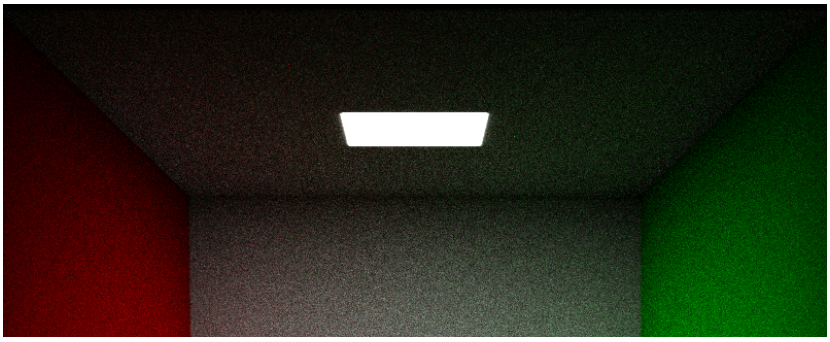
This effect happens in a pathtracer because photons that hit the surface close to a shadow edge will see the light source, but it would be partially obscured (because the light has an area), this means that a percentage of photons would be able to directly hit the light source, while the rest will need to bounce at least one more time and lose energy, causing the softness of the shadow. The closer the object casting the shadow is to its own shadow, the more it will obscure the light at the edge, causing the edge of the shadow to be harder the closer the shadow is to the object.



Soft shadows showing how the shadow edge is sharper near the cube

Ambient Occlusion

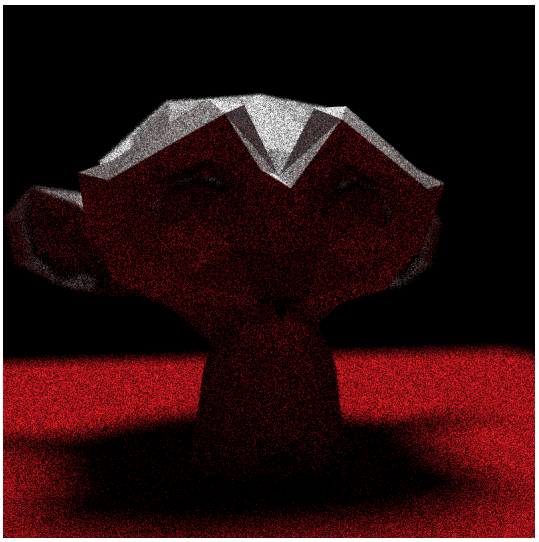
Ambient occlusion is yet another example of an effect that happens 'automatically' in a pathtracer. It is apparent on edges and corners of rooms, where photons have a lower chance of escaping without losing much energy. When a photon hits a corner, a large percentage of rays will bounce and immediately hit a wall again and lose energy, while a small percentage will make it out of the corner on the first bounce and eventually hit a light source.



The inside of a room showing how much darker the area is where the wall meets the ceiling

Colour Bleeding

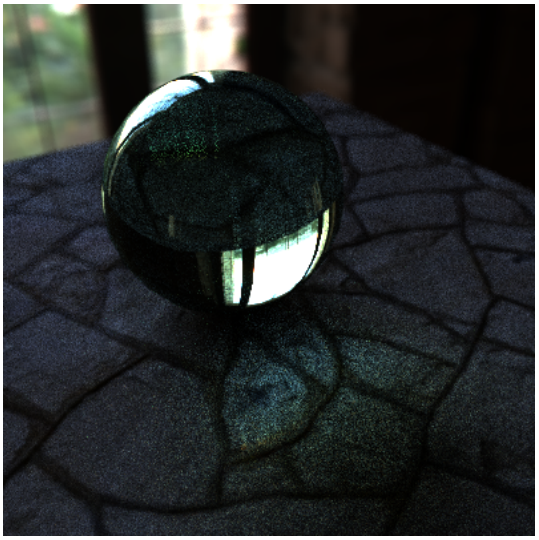
Colour bleeding happens in a pathtracer when an object hits a coloured diffuse surface and bounces randomly to another surface, the colour information it had previously from the first surface is altered by the second surface. The photon will then eventually return a pixel colour on the first surface which has had another surface's influence. In a raytracer this would not happen because as soon as a ray hits the first surface, a shadow ray is cast and that determines the colour.



A white model with the colour of the floor bleeding into it's surface

Caustics

Caustics happen when a photon shot from the camera hits a plane, goes through glass and hits a light. In a typical raytracer, once a ray would hit the plane, a shadow ray would be cast which would then find an obstruction because of the glass sphere and return a shadow. In pathtracing, the path of the photon is followed all the way through until it hits a light source, which allows the photon to 'carry' the colour of the glass until it hits the light.



A glass ball on a plane showing the caustics on the plane

Random Number Generation

Due to random numbers being needed quite frequently in a pathtracer, using *rand*() was not good enough. The numbers returned by *rand*() were not 'random enough' over long sequences and would often lead to artifacts such as banding in the pathtracer. This is why another random number generator was used, which is an implementation of George Marsaglia's Mother Random algorithm by Agner Fog. This implemenation generates 'random' numbers somewhat evenly and with no noticible pattern.

Russian Roulette and Bounce Limits

Russian Roulette is an optimisation technique for pathtracing. Every bounce there is a 50% chance that the photon will be terminated, while the remaining photons have their energy doubled. By doing this, the overall energy in the system remains the same, but the chance of having a photon that bounces forever becomes incredibly small and its energy contribution will become more significant the more bounces it has, which is important because very few photons would make it that far.

```
// g_randomNumberGen->IRandon(min, max) generates an integer between min and max inclusive.
if( (g_randomNumberGen->IRandom(0,1) == 0 && a_bounces != 0) ) {
   return float3(0.0f,0.0f,0.0f);
}
a_photon.m_energy *= 2.0f;
```

Furthermore, there is a bounce limit which ensures that photons have a finite maximum number of bounces before they get forcibly terminated. This prevents the incredibly unlikely but still theoretically possible case of a photon that will bounce indefinitely and never hit a light source.

```
if( a_bounces > BOUNCELIMIT ) {
   return float3(0.0f,0.0f,0.0f);
}
```

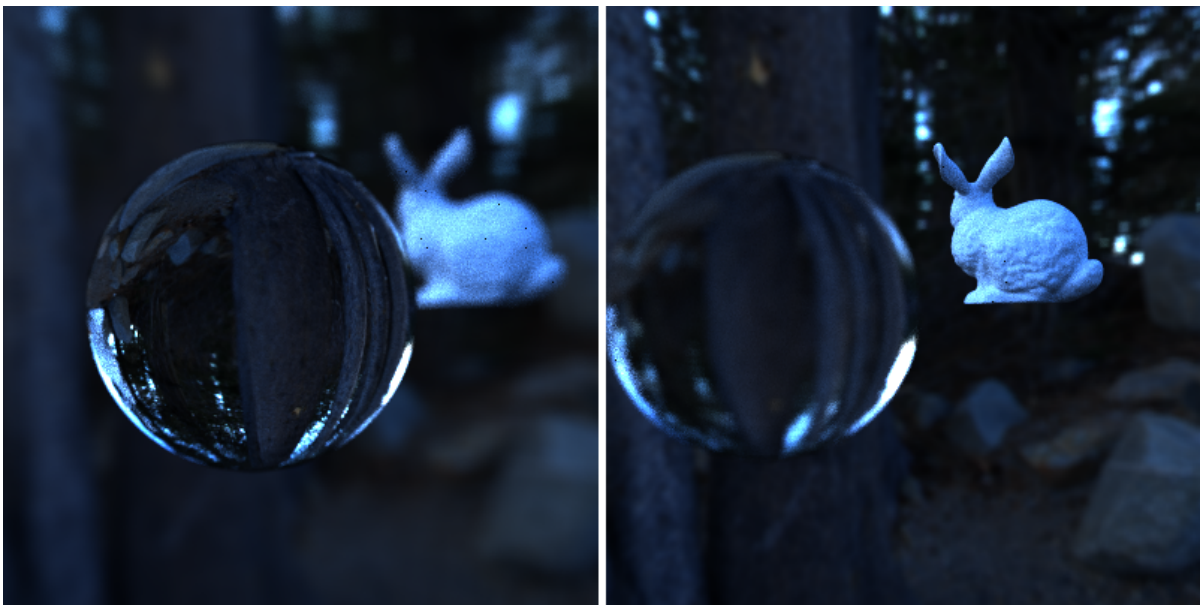
Depth of Field

Depth of field is the effect of an object appearing sharp and in focus while the rest of the image is blurry and out of focus. It is easily implemented in the pathtracer by giving a random offset to the origin position of a photon when it is created. Every pixel sample will have a different offset which will smoothen out the depth of field effect as the image converges.

The code below shows the origin position of a new photon being set to the camera's current position with a random offset applied to it. However, the very first sample of an image does not use depth of field or anti-aliasing (covered below) in order to preserve the original object edges. The boolean value a_AA will zero the offset if depth of field is not wanted (for example in the first sample).

```
a_photon.m_origin =
m_position + float3( (g_randomNumberGen->IRandom(-500,500)/10000.0f)*a_AA, (g_randomNumberGen->
    IRandom(-500,500)/10000.0f)*a_AA, (g_randomNumberGen->IRandom(-500,500)/10000.0f)*a_AA );
    // m_position is the camera position
```

The photon direction would then still be through the viewplane, but the focal point of the image would then be the position of the viewplane in 3D space (because the origin position is random and we know that every photon will go through the viewplane at a specific point). This means that if focus is changed, the view plane needs to be scaled and translated so that it is in line with where focus should be.



Changing focus between two objects

Anti-aliasing

Anti-aliasing reduces jagged edges in renders by smoothing the edges, and is implemented in a way very similar to depth of field. However, instead of applying a random offset to the origin of the photon position, a random offset is applied to the direction of the photon. Every pixel sample will have a different offset which will reduce aliasing as the image converges.

The code below shows the variables x and y which are values between zero and one representing the fraction of the up and right vectors of the view plane. The photon direction is then the normalised vector from the photon origin position to the point on the view plane, which is a 2D pixel coordinate of the screen, converted a position on the 3D view plane with a random offset. The boolean value a_AA will zero the offset if anti-aliasing is not wanted.

```
const float x = (float)(a_x + (g_randomNumberGen->IRandom(-500,500)/1000.0f)*a_AA ) / (SCRWIDTH
    / 2);
const float y = (float)(a_y + (g_randomNumberGen->IRandom(-500,500)/1000.0f)*a_AA ) / (SCRHEIGHT
    );
// m_px are the points of the view plane
const float3 P = m_p1 + (m_p2 - m_p1) * x + (m_p4 - m_p1) * y;
a_photon.m_direction = Normalize( P - a_photon.m_origin );
```

.obj Importing

Since triangles are the only primitive supported by the pathtracer, .obj importing was a way to test scenes with a large amount of triangles without needing to create triangles by hand. An .obj is imported by reading a file in plain text line by line and storing vertex coordinates, vertex normals, and vertex UV coordinates per triangle.

A sample .obj file is shown below.

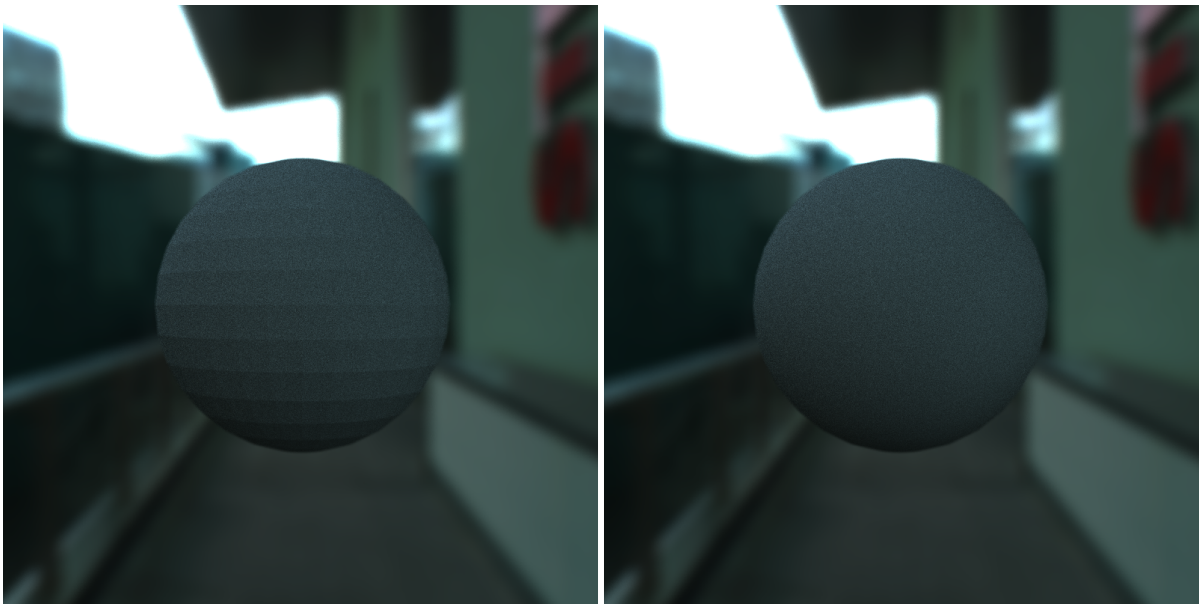
```
v 2.348512 -1.479918 -1.660182                // vertex coordinate
vt 0.023822 0.138314                          // vertex UV coordinate
vn -0.605327 0.259130 0.752616                // vertex normal
f 5139/2118/1630 5133/2084/1606 33/2087/1422 // constructing the face with vertex coordinate
    index/vertex UV coordinate index/vertex normal index
```

Interpolated Normals

Interpolated normals make an object look like it is smooth by the way photons react to the surface, even though geometrically it is still made out of distinct triangles.

Interpolating the normal is not overly complex as long as the barycentric coordinate of the triangle is known. With this barycentric coordinate, interpolating between the three normal vectors on each triangle vertex is quite simple, shown in the code sample below.

```
const float b0 = 1.0f - s - t;                // s and t are the barycentric coordinates
const float3 normal = Normalize( (b0 * n0) + (s * n1) + (t * n2) );  // n0 n1 and n2 are the
    vertex normals
```



A sphere without and with interpolated normals

Material Loading

Material loading is done in a manner similar to .obj loading. Whenever the .obj file declares that it uses a material, an additional file (.mtl) is read. A snippet of an .mtl file is shown below.

```
newmtl initialShadingGroup              // name of the material
illum 4                                 // illumination model to be used
Kd 0.00 0.00 0.00                       // diffuse colour
map_Kd football.jpg                     // image file to be used as the texture
```

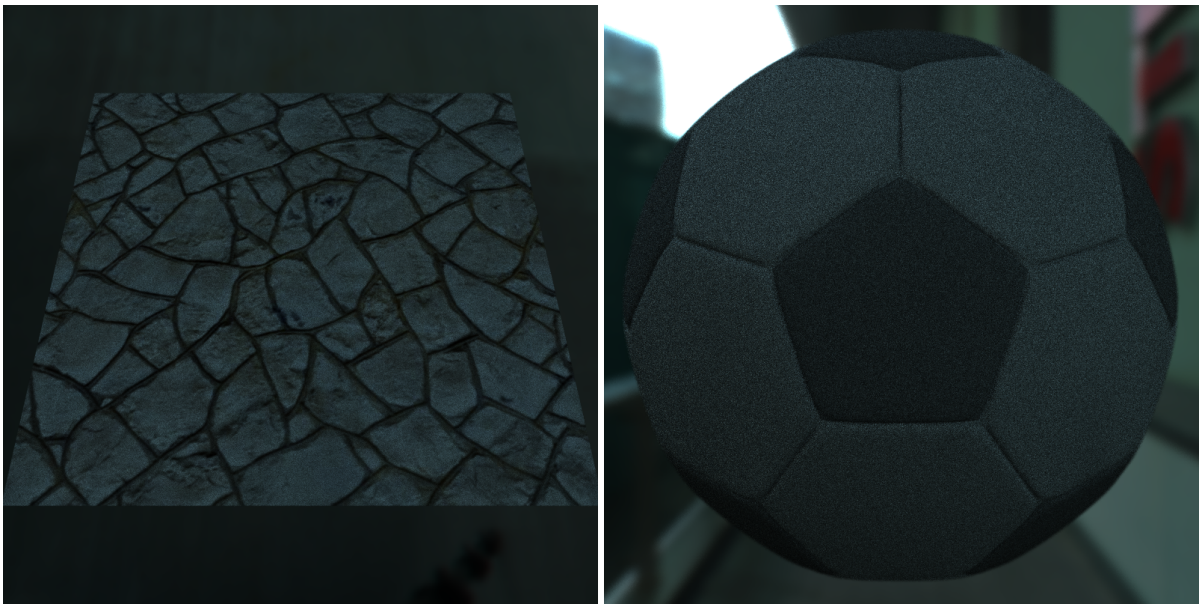
Then afterwards while the .obj file is being read, a material may be specified before creating a triangle, which is the material that the triangle will use.

```
usemtl initialShadingGroup              // material to be used for the faces below
f 4687/1/1 2343/2/2 61/4/3
```

UV Mapping

UV mapping is done very similarly to normal interpolation. However, instead of looking at the normals of each vertex and then interpolating, the UV coordinate of each vertex is looked at, as shown below.

```
const float b0 = 1.0f - s - t;          // s and t are the barycentric coordinates
// uvx[0] are the u coordinates, uvx[1] are the v coordinates
const float u = b0 * uv1[0] + s * uv2[0] + t * uv3[0];
const float v = b0 * uv1[1] + s * uv2[1] + t * uv3[1];
```



A textured plane and football

Bilinear Filtering

Normally when a texture needs to be read, the UV coordinates are used to lookup the appropriate pixel on the texture, however this causes artifacts when the object is close to the camera and the individual pixels become clear. This is solved with bilinear filtering.

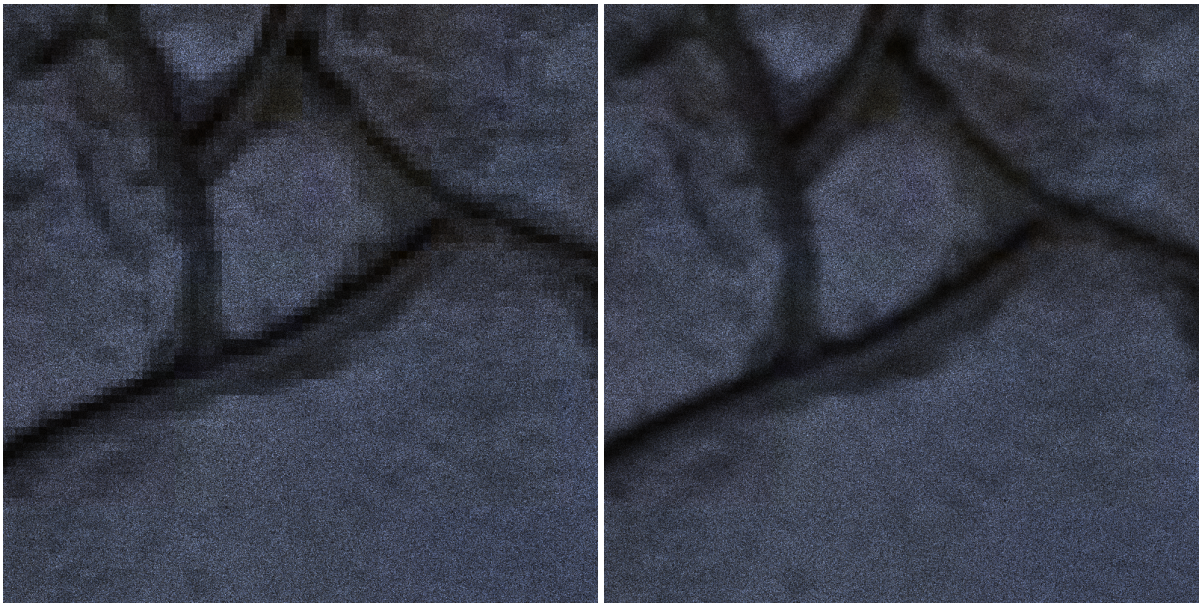
Bilinear filtering is when not just an individual pixel is returned as the colour of the point on the object, but an area is sampled. The returned colour is a weighted average of the surrounding pixels of the current UV coordinate. Below is a snippet of bilinear filtering in the path tracer.

```
// pixel 1 is the pixel that is half a pixel to the left of the UV coordinate and half a pixel above
const float2 p1 = float2( CLAMP( (a_UV[0]*width) - 0.5f, 0, width-1 ), CLAMP( ( (1.0f - a_UV[1]) *height) - 0.5f, 0, height-1 ) );
// pixel 2 is the pixel that is half a pixel to the right of the UV coordinate and half a pixel above
const float2 p2 = float2( CLAMP( (a_UV[0]*width) + 0.5f, 0, width-1 ), CLAMP( ( (1.0f - a_UV[1]) *height) - 0.5f, 0, height-1 ) );
// pixel 3 is the pixel that is half a pixel to the left of the UV coordinate and half a pixel below
const float2 p3 = float2( CLAMP( (a_UV[0]*width) - 0.5f, 0, width-1 ), CLAMP( ( (1.0f - a_UV[1]) *height) + 0.5f, 0, height-1 ) );
// pixel 4 is the pixel that is half a pixel to the right of the UV coordinate and half a pixel below
const float2 p4 = float2( CLAMP( (a_UV[0]*width) + 0.5f, 0, width-1 ), CLAMP( ( (1.0f - a_UV[1]) *height) + 0.5f, 0, height-1 ) );

// to be used when calculating the area of each pixel (their weight)
const float2 uvFrac = float2( (p1[0] - (int)p1[0]), (p1[1] - (int)p1[1]) );

// pixel weights
const float w1 = (1 - uvFrac[0]) * (1 - uvFrac[1]);
const float w2 = (uvFrac[0]) * (1 - uvFrac[1]);
const float w3 = (1 - uvFrac[0]) * (uvFrac[1]);
const float w4 = 1-(w1+w2+w3);

return (colour[0] * w1) + (colour[1] * w2) + (colour[2] * w3) + (colour[3] * w4);
```



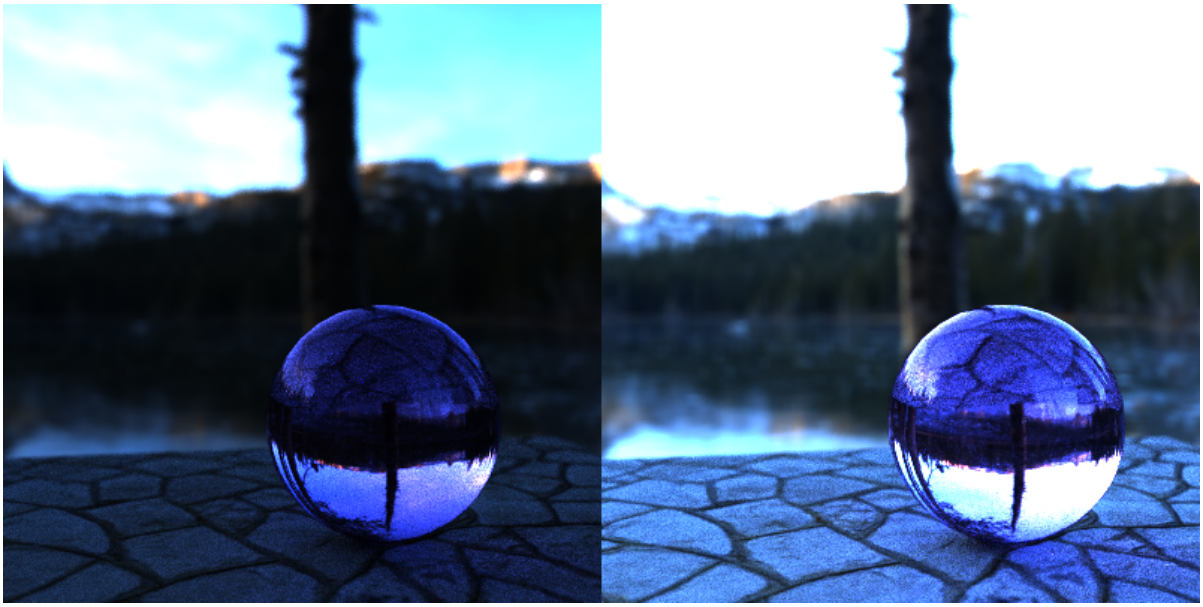
A textured plane without and with bilinear filtering

High Dynamic Range (HDR) Lightbox

High dynamic range is when colour is stored as a number higher than the value of one in each RGB component. This allows for the intensity of a colour to be stored as well, and retains a lot more colour information than if the colours were clamped at one.

By using a lightbox, instead of returning black whenever a photon does not hit anything, a colour of the HDR image can be returned. Furthermore, because the colours of the HDR lightbox can go above one, it

acts as if the lightbox emits light - removing the need for any explicit lights, and giving the impression that the objects being rendered are a part of the scene.



A scene with the HDR values multiplied by 2.5 and 8.5. Even though the second image's colours look a bit blown out, the colour information remains as is demonstrated in the first image.

Bounding Volume Hierarchy (BVH)

A bounding volume hierarchy does a lot to reduce the number of unnecessary triangle intersection tests that need to be done in the scene. A BVH encloses the triangles of the scene inside axis-alligned bounding boxes, separating groups of triangles. A benefit of this is that now instead of doing triangle intersection tests against every triangle in the scene, a couple of AABB intersection tests are performed to find out what group of triangles to test. This yields an extreme improvement to performance, since now for each AABB intersection test performed, a large number of triangle intersection tests can be skipped.

How a BVH determines what triangles to enclose in a box in this pathtracer is as follows: first a large AABB encloses every single triangle in the scene, then the box is split through the middle along the x, y, and z axes. For each axis, the number of triangles on each side is stored, as well as the area of the bounding box. The final axis to split by is the one with the lowest cost, which is determined by

$$cost = Triangles_{LeftSide} * Area_{LeftAABB} + Triangles_{RightSide} * Area_{RightAABB}$$

The BVH will then keep splitting until either all axes have all triangles only on one side, or the number of triangles inside the current box is < 5.

Multithreading

A very barebones job system was implemented in order to multithread the pathtracer. A render job is a range of x and y coordinates (a tile) that should be rendered. A number of threads then check the list of available jobs and take a render job until the screen has been fully rendered before allowing the main thread to copy the image to the screen.

Overview of Light Transport

Diffuse

A diffuse material is a material that scatters a photon in a random direction whenever a photon hits it. The photon's energy then gets multiplied by the colour of the material.

A model with a diffuse material

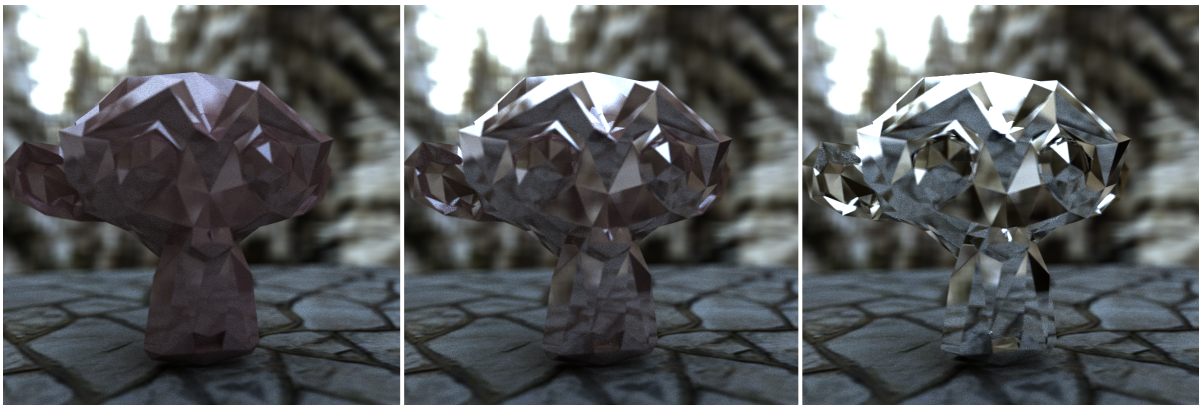
Random Bounce

A random bounce occurs whenever a photon hits a diffuse surface. However, since the photon needs to bounce away from the surface it just hit, the normal of the surface is used to calculate a new random direction, as shown below.

```
1  void Photon::RandomBounce( float3 a_normal ) {
2     float dotDN;
3     do {
4        // generating completely random direction
5        m_direction = float3((g_randomNumberGen->IRandom(-50,50)), (g_randomNumberGen->IRandom
       (-50,50)), (g_randomNumberGen->IRandom(-50,50)));
6        dotDN = Dot(m_direction, a_normal);
7     } while( dotDN == 0 );  // if the dot product is zero, retry
8
9     // if the dot product is negative (which we don't want) the direction is flipped anyway!
10    m_direction = Normalize( dotDN * m_direction);
11 }
```

Specular

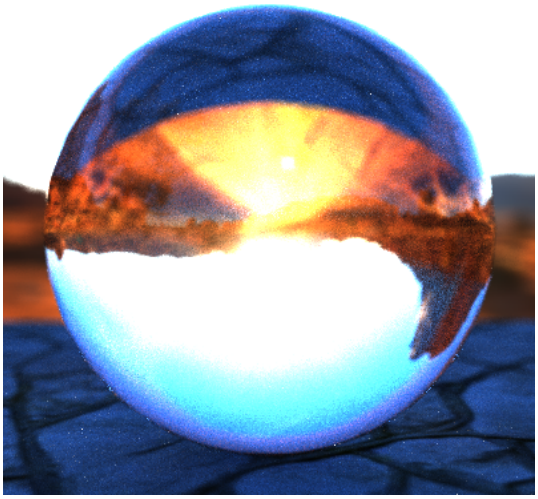
A fully specular material is a perfect mirror - each photon that hits the surface is reflected in a very predictable manner. If the material is only partically specular, a percentage of photons will perform a random bounce upon hitting the surface while the rest reflect off the surface of the material.



A model with 20%, 50%, and 100% specularity respectively

Glass

When a photon hits a glass surface, depending on Schlick's approximation of Fresnel's equations it will either refract and change direction based on the relative refractive indicies of the two surfaces it is going through, or reflect. The energy of the photon is also reduced based on beer's law, due to the glass absorbing the energy of the photon as it passes through.



A glass sphere

Schlick's Approximation

Schlick's approximation of Fresnel's equations determines the probability of a photon either refracting through the glass or reflecting off of it.

```
// shlick's approximation of fresnel's equations
float dirDotN = Dot( a_photon.m_direction, normal ); // normal is inverted when inside the
    object
const float v = ( ( nt - 1 ) * ( nt - 1 ) ) / ( ( nt + 1 ) * ( nt + 1 ) );
const float oneDirDotN = 1 + dirDotN;
const float wr = v + ( 1.0f - v ) * (oneDirDotN*oneDirDotN*oneDirDotN*oneDirDotN*oneDirDotN);
    // weight of reflection
const float wt = 1.0f - wr;   // weight of transmission (refraction)
```

Refraction

How the photon refracts depends on the relative refractive indicies of both materials, as well as the angle of incidence.

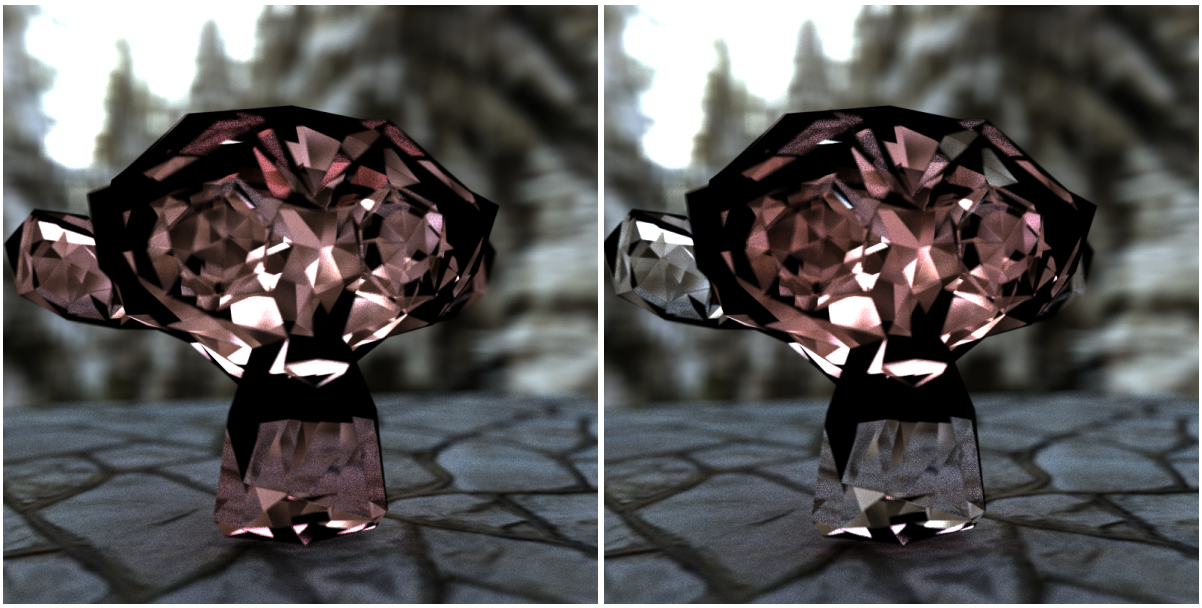
```
// if the weight of transmission is high enough then we can refract
if( g_randomNumberGen->IRandom(0, 100) <= wt*100.0f ) {
   dirDotN *= -1.0f;       // we now need the dot of the negative direction and the normal so we
     can just *-1 the previously calculated dot
   const float ratio = n1/n2; // ratio of the two refractive indicies
   const float cos2t = 1.0f - ratio*ratio * ( 1.0f - (dirDotN*dirDotN) );  // used to determine
    TIR

   if ( cos2t >= 0 ) { // if this is false, total internal reflection (TIR) happens so there's
    no refraction
      // new direction
      a_photon.m_direction = Normalize( ratio*a_photon.m_direction + ((ratio*dirDotN - sqrt(
    cos2t)) * normal) );
   }
   else {
      return float3(0.0f, 0.0f, 0.0f); // no refraction
   }
}
```

Beer's Law

Beer's law is the phenomenon that happens when a photon travels through glass. The longer the photon's path through glass, the more energy is absorbed by it. The equation for Beer's law is $energy_{out} = energy_{in} * e^{-(d*length)}$. Where d is equal to the absorptivity of the glass object. Below is a code sample from the pathtracer.

```
// this code happens inside the if( cos2t >= 0 ) statement in the previous code snippet
if( out ) {
   // the material colour is actually the absorptivity (1.0 - visible colour) but the colour is
    converted to the absorptivity when the glass is being created in code
   const float3 colourTimesDistance = a_photon.m_intersection.m_pMaterial->GetColour(&a_photon)
    * a_photon.m_distance;
   // beer's law
   a_photon.m_energy *= float3( pow( E, (-colourTimesDistance.x) ), pow( E, (-
    colourTimesDistance.y) ), pow( E, (-colourTimesDistance.z) ) );
}
```



A glass model without and with Beer's law

Emissive

When a photon hits an emissive surface, it returns from the trace and the photon's energy is multiplied by the colour of the light. If the light's colour is stored in HDR values, the intensity of the light can be altered by multiplying the light colour by the factor that the light should be increased by.

An emissive model with an increasing intensity of colour

Manual

Controls

WASD	Move the camera forwards/backwards/left/right
ZX	Move camera straight up and down
QE	Rotate camera left and right
Up/Down Arrow Keys	Tilt the camera up and down)
IK	Increase/Decrease the focal plane distance

Each keypress moves the camera a fixed amount, keep tapping to keep moving the camera. The number of iterations/runtime timer will automatically reset when the camera status is changed.

The camera automatically tries to focus on whatever object is at the centre of the screen every time its position is changed. The depth of field is still able to be tweaked after the camera is in position.

Defines/Constants

raytracer.cpp	line 12
BVHVIS	Comment/Uncomment to change the rendering mode to either BVH visualisation or pathtracing
game.cpp	line 19
g_THREADS	Number of threads to be created by the job manager

Additional Screenshots