

Rendering assignment two

Alexander Birke

study nr: 124044 email: alex@alexanderbirke.dk

December 19, 2012

Introduction

This is my handin for the third week for the course "Rendering".

Part 1 - Hard Shadows

I covered this in the last handin so I have chosen not to explain it again.

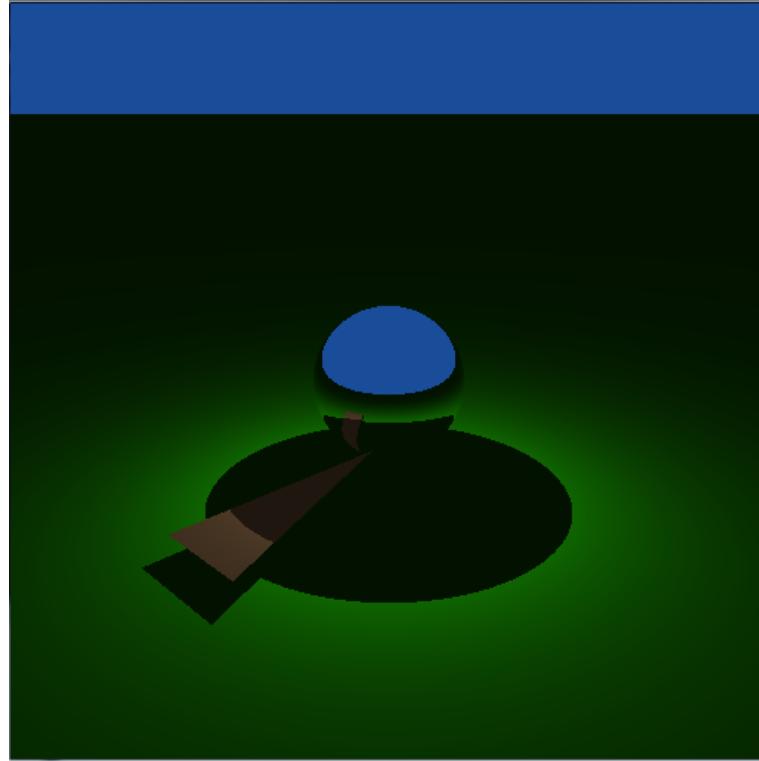
Part 2 - Mirror Reflection

The tracing of reflected rays is implemented in the following way:

```
1 bool RayTracer::trace_reflected(const Ray& in, const HitInfo& in_hit, Ray&
2   out, HitInfo& out_hit) const
3 {
4   float3 reflectedDir = optix::reflect(in.direction, in_hit.
5     geometric_normal);
6
7   out = Ray(in_hit.position, reflectedDir, 0, 0.01f);
8
9   if(this->trace_to_closest(out, out_hit))
10  {
11    out_hit.trace_depth = in_hit.trace_depth + 1;
12
13    out_hit.ray_ior = in_hit.ray_ior;
14
15    return true;
16  }
17  return false;
```

First the reflected vector is calculated with the use of the reflect function in the optix library. A new ray is then created with start point where the last ray hit and in the direction of the reflection. The ray is then traced. If it hits anything, the outgoing ray's trace depth is set to the incoming ray's plus one. This is done to make sure that the tracing will terminate when a certain trace depth has been reached. The outgoing ray's index of refraction is then set and since it is in the same medium as the incoming ray it

is set to have the same index of refraction as the first ray. The function then returns true if the outgoing ray hit anything. The end result look like this:



```
Building acceleration structure...(time: 0.001)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.314
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 2.15 secs
```

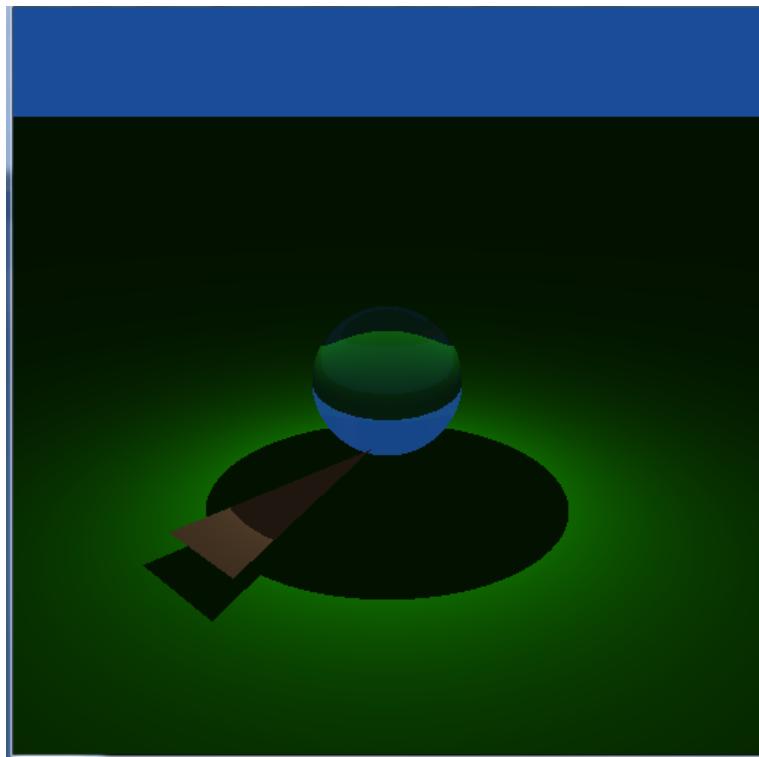
Part 3 - Refraction

```
1 bool RayTracer::trace_refracted(const Ray& in, const HitInfo& in_hit, Ray&
2   out, HitInfo& out_hit) const
3 {
4   float3 outDirection;
5   float3 outNormal;
6   float inTheta;
7   out_hit.ray_ior = get_ior_out(in, in_hit, outDirection, outNormal,
8     inTheta);
9   float3 refractedDirection;
10  if(optix::refract(refractedDirection, -outDirection, outNormal, out_hit.
11    ray_ior / in_hit.ray_ior))
```

```

12  {
13      out.ray_type = 0;
14      out.direction = refractedDirection;
15      out.origin = in_hit.position;
16      out.tmin = 0.001f;
17      out.tmax = 99999;
18
19      if(this->trace_to_closest(out, out_hit))
20      {
21          out_hit.trace_depth = in_hit.trace_depth + 1;
22
23          return true;
24      }
25  }
26  return false;
27 }
```

We first declare two variables of the type float3 to hold the direction and normal of the outgoing ray along with one to hold the theta value. The theta value is not used for anything in this piece of code but is needed in order to call the index of refraction helper function. This is the next thing to be called. This function checks if the ray will go out of the object it is currently in, if that is the case the normal of the incoming hit is reverted along with theta and an index of refraction of 1 is returned which means the ray is traveling in air. Otherwise the function finds the index of refraction based on the incoming ray's hit's material reference. Next we check if the outgoing ray will refract. This is done by calling the optix refract function that also calculates the refraction vector. optix functions in general assume that the ray direction they get in still is the original direction so we have to invert the direction that was inverted in the helper function. If no internal refraction occurs, we initialize the outgoing ray with direction of the refraction, with start in the incoming hit, and tmin and tmax set to sensible values. We then trace the ray and if we hit anything, add one to the trace depth of the outgoing hit and return true. The end result look like this:



```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.311
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 3.552 secs
```

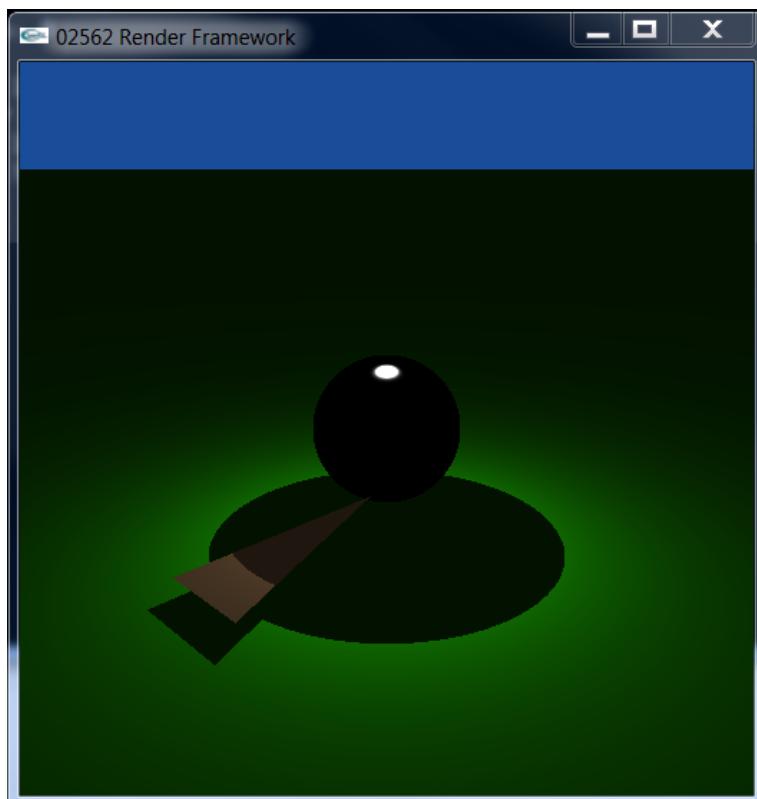
Part 4 - Phong Illumination

```
1 float3 Phong::shade(const Ray& r, HitInfo& hit, bool emit) const
2 {
3     float3 rho_d = get_diffuse(hit);
4     float3 rho_s = get_specular(hit);
5     float s = get_shininess(hit);
6
7     float3 result = make_float3(0);
8
9     for(int i = 0 ; i < lights.size() ; i++)
10    {
11        float3 L;
12        float3 dir;
13        if( lights[i]->sample(hit.position, dir, L) )
14        {
15            float3 R = optix::reflect(-dir, hit.shading_normal);
```

```

16     unsigned int sInt = (int)s;
17
18     result += rho_d*L* optix::fmaxf(optix::dot(hit.shading_normal, dir),
19                                     0) + rho_s*L* int_pow(optix::fmaxf(optix::dot(R, -r.direction),
20                                     0), sInt);
21 }
22 }
```

We start by getting the material specific parameters of the phong equation and declare a variable to store the result of the shading. Then we loop over the lights in the scene. If the light hits the surface we use the returned direction to the light source and its intensity in the phong equation. We also calculate the reflected light ray in order to implemented specular reflection. To make the calculation of the specular reflection faster we also convert the shininess of the material to an int so we can use the int pow function. The end result look like this:



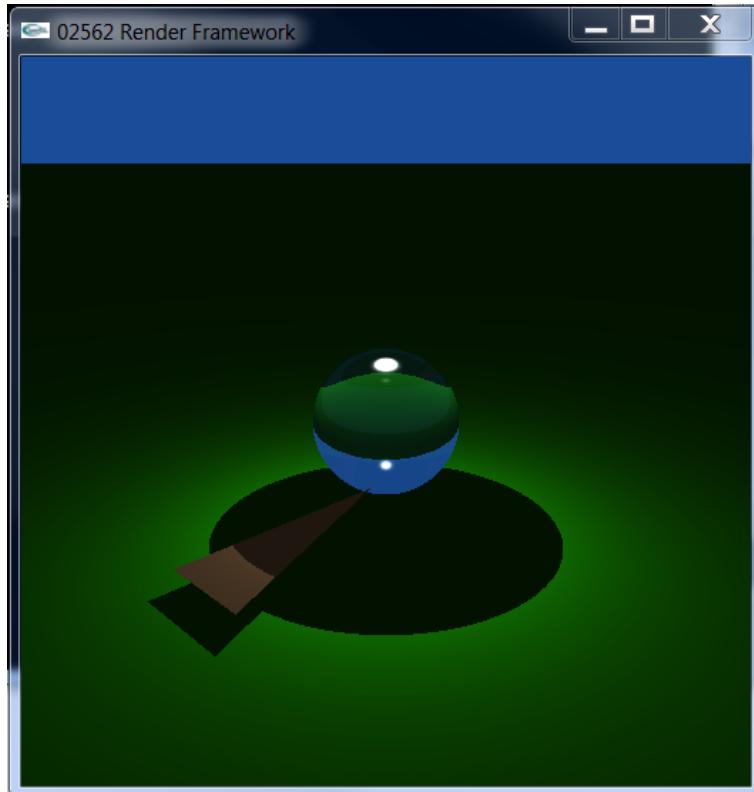
```

Building acceleration structure... (time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.341
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 2.19 secs
```

Part 5 - Combining it all

```
1 float3 Glossy::shade(const Ray& r, HitInfo& hit, bool emit) const
2 {
3     if(hit.trace_depth >= max_depth)
4         return make_float3(0.0f);
5
6     Ray reflected, refracted;
7     HitInfo hit_reflected, hit_refracted;
8     tracer->trace_reflected(r, hit, reflected, hit_reflected);
9     tracer->trace_refracted(r, hit, refracted, hit_refracted);
10
11    return 0.1f * Phong::shade(r, hit, emit) +
12        0.1f*shade_new_ray(reflected, hit_reflected) +
13        0.9f* shade_new_ray(refracted, hit_refracted);
14 }
```

The first part is to check if the hit given to the function has reached the maximum trace depth, if that is the case the shader returns black. Since the shader implements both refraction and reflection we need to declare a ray and hit info for each of these phenomena. Each of these rays is then calculated. And used to shade the surface with the shade new ray method. These can then again create a new call to the glossy shader as the light bounces around inside the sphere. It is therefore we have the trace depth. The end result is also added together with the phong reflection of the material. Since the ball is made of glass it only has a specular component which gives the highlights on the ball. This highlight is multiplied with the reflectance since it produces the reflection of the lightsource. The end result look like this:



```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.316
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 3.601 secs
```

As we can see, the more rays a method generates the longer it takes to calculate.

Rendering assignment two

Alexander Birke

study nr: 124044 email: alex@alexanderbirke.dk

December 19, 2012

Introduction

This is my handin for the third week for the course "Rendering".

Part 1 - Hard Shadows

I covered this in the last handin so I have chosen not to explain it again.

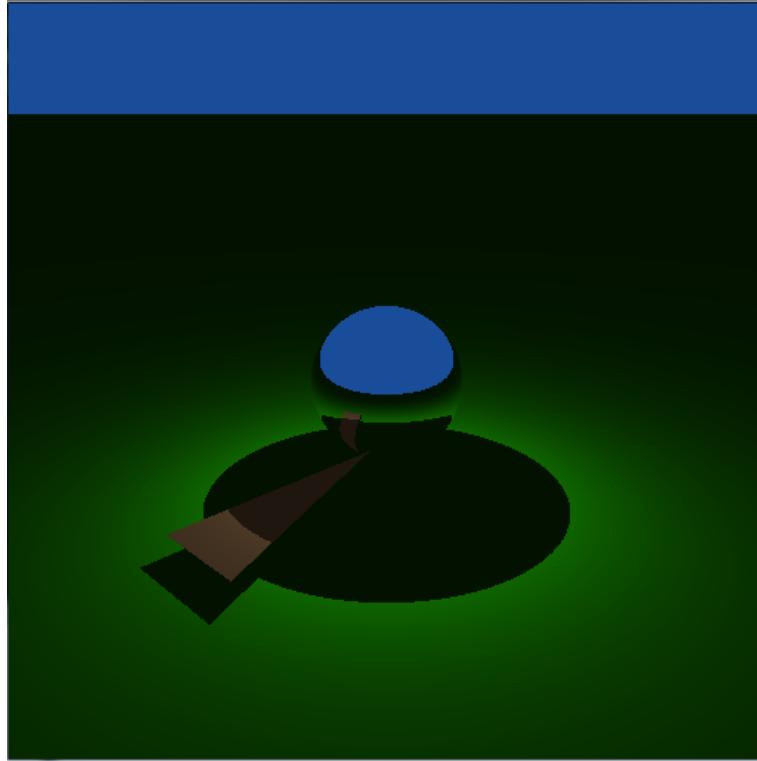
Part 2 - Mirror Reflection

The tracing of reflected rays is implemented in the following way:

```
1 bool RayTracer::trace_reflected(const Ray& in, const HitInfo& in_hit, Ray&
2   out, HitInfo& out_hit) const
3 {
4   float3 reflectedDir = optix::reflect(in.direction, in_hit.
5     geometric_normal);
6
7   out = Ray(in_hit.position, reflectedDir, 0, 0.01f);
8
9   if(this->trace_to_closest(out, out_hit))
10  {
11    out_hit.trace_depth = in_hit.trace_depth + 1;
12
13    out_hit.ray_ior = in_hit.ray_ior;
14
15    return true;
16  }
17  return false;
```

First the reflected vector is calculated with the use of the reflect function in the optix library. A new ray is then created with start point where the last ray hit and in the direction of the reflection. The ray is then traced. If it hits anything, the outgoing ray's trace depth is set to the incoming ray's plus one. This is done to make sure that the tracing will terminate when a certain trace depth has been reached. The outgoing ray's index of refraction is then set and since it is in the same medium as the incoming ray it

is set to have the same index of refraction as the first ray. The function then returns true if the outgoing ray hit anything. The end result look like this:



```
Building acceleration structure...(time: 0.001)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.314
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 2.15 secs
```

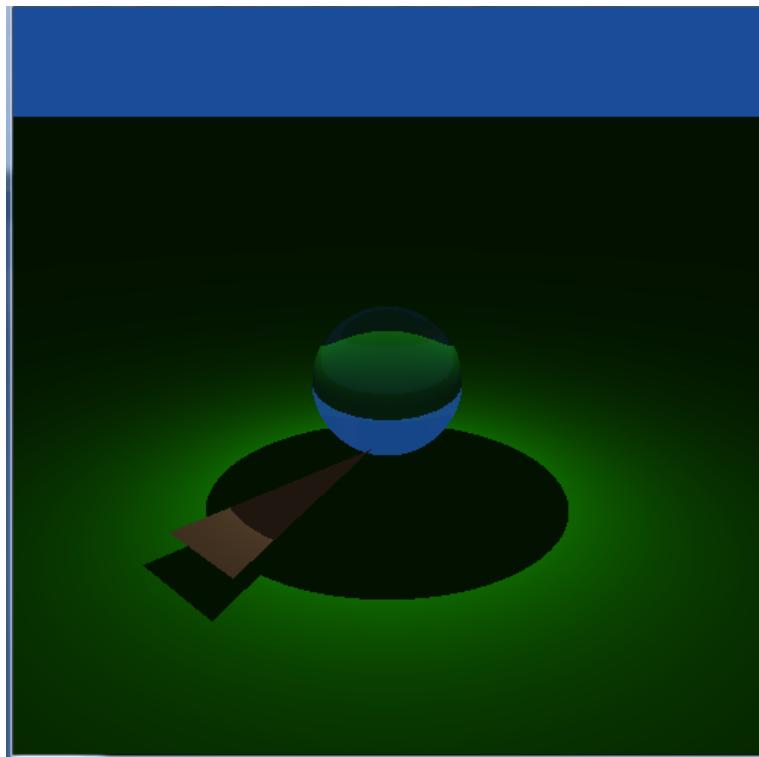
Part 3 - Refraction

```
1 bool RayTracer::trace_refracted(const Ray& in, const HitInfo& in_hit, Ray&
2   out, HitInfo& out_hit) const
3 {
4   float3 outDirection;
5   float3 outNormal;
6   float inTheta;
7   out_hit.ray_ior = get_ior_out(in, in_hit, outDirection, outNormal,
8     inTheta);
9   float3 refractedDirection;
10  if(optix::refract(refractedDirection, -outDirection, outNormal, out_hit.
11    ray_ior / in_hit.ray_ior))
```

```

12  {
13      out.ray_type = 0;
14      out.direction = refractedDirection;
15      out.origin = in_hit.position;
16      out.tmin = 0.001f;
17      out.tmax = 99999;
18
19      if(this->trace_to_closest(out, out_hit))
20      {
21          out_hit.trace_depth = in_hit.trace_depth + 1;
22
23          return true;
24      }
25  }
26  return false;
27 }
```

We first declare two variables of the type float3 to hold the direction and normal of the outgoing ray along with one to hold the theta value. The theta value is not used for anything in this piece of code but is needed in order to call the index of refraction helper function. This is the next thing to be called. This function checks if the ray will go out of the object it is currently in, if that is the case the normal of the incoming hit is reverted along with theta and an index of refraction of 1 is returned which means the ray is traveling in air. Otherwise the function finds the index of refraction based on the incoming ray's hit's material reference. Next we check if the outgoing ray will refract. This is done by calling the optix refract function that also calculates the refraction vector. optix functions in general assume that the ray direction they get in still is the original direction so we have to invert the direction that was inverted in the helper function. If no internal refraction occurs, we initialize the outgoing ray with direction of the refraction, with start in the incoming hit, and tmin and tmax set to sensible values. We then trace the ray and if we hit anything, add one to the trace depth of the outgoing hit and return true. The end result look like this:



```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.311
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 3.552 secs
```

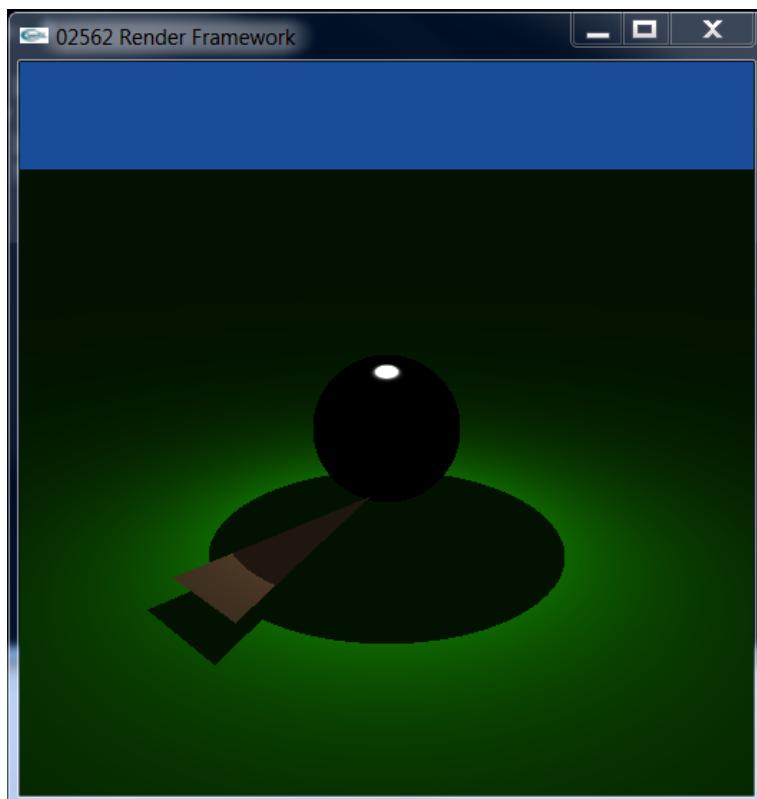
Part 4 - Phong Illumination

```
1 float3 Phong::shade(const Ray& r, HitInfo& hit, bool emit) const
2 {
3     float3 rho_d = get_diffuse(hit);
4     float3 rho_s = get_specular(hit);
5     float s = get_shininess(hit);
6
7     float3 result = make_float3(0);
8
9     for(int i = 0 ; i < lights.size() ; i++)
10    {
11        float3 L;
12        float3 dir;
13        if( lights[i]->sample(hit.position, dir, L) )
14        {
15            float3 R = optix::reflect(-dir, hit.shading_normal);
```

```

16     unsigned int sInt = (int)s;
17
18     result += rho_d*L* optix::fmaxf(optix::dot(hit.shading_normal, dir),
19                                     0) + rho_s*L* int_pow(optix::fmaxf(optix::dot(R, -r.direction),
20                                     0), sInt);
21 }
22 }
```

We start by getting the material specific parameters of the phong equation and declare a variable to store the result of the shading. Then we loop over the lights in the scene. If the light hits the surface we use the returned direction to the light source and its intensity in the phong equation. We also calculate the reflected light ray in order to implemented specular reflection. To make the calculation of the specular reflection faster we also convert the shininess of the material to an int so we can use the int pow function. The end result look like this:



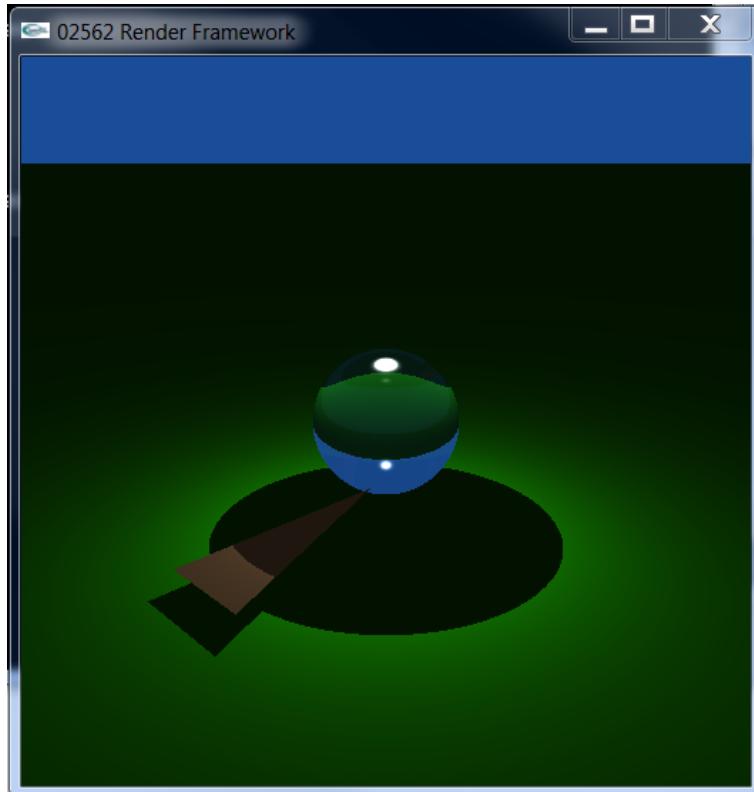
```

Building acceleration structure... (time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.341
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 2.19 secs
```

Part 5 - Combining it all

```
1 float3 Glossy::shade(const Ray& r, HitInfo& hit, bool emit) const
2 {
3     if(hit.trace_depth >= max_depth)
4         return make_float3(0.0f);
5
6     Ray reflected, refracted;
7     HitInfo hit_reflected, hit_refracted;
8     tracer->trace_reflected(r, hit, reflected, hit_reflected);
9     tracer->trace_refracted(r, hit, refracted, hit_refracted);
10
11    return 0.1f * Phong::shade(r, hit, emit) +
12        0.1f*shade_new_ray(reflected, hit_reflected) +
13        0.9f* shade_new_ray(refracted, hit_refracted);
14 }
```

The first part is to check if the hit given to the function has reached the maximum trace depth, if that is the case the shader returns black. Since the shader implements both refraction and reflection we need to declare a ray and hit info for each of these phenomena. Each of these rays is then calculated. And used to shade the surface with the shade new ray method. These can then again create a new call to the glossy shader as the light bounces around inside the sphere. It is therefore we have the trace depth. The end result is also added together with the phong reflection of the material. Since the ball is made of glass it only has a specular component which gives the highlights on the ball. This highlight is multiplied with the reflectance since it produces the reflection of the lightsource. The end result look like this:



```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.316
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing..... - 3.601 secs
```

As we can see, the more rays a method generates the longer it takes to calculate.

Rendering assignment four

Alexander Birke

study nr: 124044 email: alex@alexanderbirke.dk

December 19, 2012

Introduction

This is my answers to the fourth handin in the Rendering course. The formulation of the exercises is given in italic.

Part 1

small 25W light bulb has an efficiency of 20%. How many photons are approximately emitted per second? Assume in the calculations that we only use average photons of wavelength 500 nm.

$$f = \frac{c}{\lambda} = \frac{3 * 10^8 \frac{m}{s}}{500m * 10^{-9}} = 6 * 10^{14} s^{-1}$$

$$q[500nm] = hf = 6.626 * 10^{-34} Js * 6 * 10^{14} s^{-1} = 3.976 * 10^{-19} J$$

$$N_{photon}[25W * 0.2] = \frac{25W * 0.2}{3.976 * 10^{-19} J/photon} = 1.258 * 10^{19} photon/s$$

Part 2

A light bulb (2.4 V and 0.7 A), which is approximately sphere-shaped with a diameter of 1 cm, emits light equally in all directions. Find the following entities (ideal conditions assumed)

Radiant flux

This is equal to the effect of the bulb. This can be obtained from the voltage and amperage:

$$W = A * V = 2.4V * 0.7A = 1.68J/s$$

Radiant intensity

Radian intensity is how much power is emitted per solid angle. Since we are to take the whole sphere of the lightbulb into consideration the solid angle is equal to four pi:

$$I = W/sr = \frac{1.68J/s}{4\pi sr} = 0.42 \frac{J}{s * sr}$$

Radiant exitance

Radiant exitance is the power divided by the surface area. Since the bulb is a unit sphere the area of it is equal to 4 pi and the result is therefore the same as before but with different units:

$$M = W/m^2 = 0.42W/m^2$$

Emitted energy in 5 minutes

This is simply taking the flux effext of the bulb and multiply it with 5*60 seconds:

$$Q = W/s = 1.68J/s * 5 * 60s = 504J$$

Part 3

The light bulb from above is observed by an eye, which has an opening at the pupil of 6 mm and a distance of 1 m from the light bulb. Find the irradiance recieved in the eye
We use the inverse square law:

$$E = \frac{W}{4\pi r^2}$$

where E is irradiance, W is radiant flux and r is distance from light source. When we use the value of radiant flux calculated earlier and set r to 1m we get:

$$E = \frac{1.68J/s}{4\pi(1m)^2}$$

$$E = \frac{0.13J/s}{m^2}$$

Part 4

We start by calculating the flux of the bulb:

$$\Phi = P * \eta = 200W * 20$$

I assume the task is to calculate the irradiance at the closest point on the table. I can then use the inverse square law again:

$$E = \frac{W}{4\pi r^2}$$

$$E = \frac{40W}{4\pi(2m)^2}$$

$$E = \frac{40W}{4\pi4m^2}$$

The irradiance is therefore:

$$E = 0.80 \frac{W}{m^2}$$

To calculate the illuminance we use the relationship between radiometric and photometric quantities:

$$\text{Photometric} = \text{radiometric} * 685 * V(\lambda)$$

At 650 nm the luminous efficiency curve has the value 0.1. The illuminance at the table is therefore equal to:

$$\begin{aligned} Lux &= E * 685 * V(\lambda) \\ Lux &= 0.80 \frac{W}{m^2} * 685 * 0.1 L/W \\ Lux &= 54.8 L/m^2 \end{aligned}$$

Part 5

We have the values:

$$\begin{aligned} I_{known} &= 40 \frac{lm}{sr} \\ r_{known} &= 0.35m \\ r_{unknown} &= 0.65m \end{aligned}$$

If we assume that the light sources are isotropic and spherical we can use the inverse square law:

$$E = \frac{I}{r^2}$$

So we have that:

$$\begin{aligned} E_{known} &= \frac{I_{known}}{r_{known}^2} \\ E_{unknown} &= \frac{I_{unknown}}{r_{unknown}^2} \end{aligned}$$

since we know that:

$$E_{unknown} = E_{known}$$

We can derive the unknown irradiance:

$$\begin{aligned} \frac{I_{unknown}}{r_{unknown}^2} &= \frac{I_{known}}{r_{known}^2} \\ I_{unknown} &= I_{known} \frac{r_{unknown}^2}{r_{known}^2} \\ I_{unknown} &= 40 \frac{lm}{sr} \frac{0.65m^2}{0.35m^2} \\ I_{unknown} &= 138 \frac{lm}{sr} \end{aligned}$$

Part 6

The relationship between flux and radiance and radiosity and radiance for a diffuse emitter is:

$$\Phi = LA\pi$$

$$B = L\pi$$

The flux is therefore equal to:

$$\Phi = 5000 \frac{W}{m^2 sr} * (0.1m)^2 * \pi = 157W$$

and the radiosity:

$$B = 5000 \frac{W}{m^2 sr} \pi = 1.570 * 10^4 \frac{W}{m^2}$$

Part 7

The relationship between radiance and radiosity is equal to:

$$B = \int_{\Omega} L \cos \theta d\omega$$

We have that the projected solid angle is equal to:

$$\int_{\Omega} \cos \theta d\omega = \int_0^{2\pi} \int_0^{\pi} \cos \theta \sin \theta d\theta d\phi = \pi$$

The radiosity is therefore:

$$B = L\pi = 6000\pi \frac{W}{m^2}$$

To calculate the flux we use the relationship:

$$\Phi = B * A$$

$$\Phi = 6000\pi \frac{W}{m^2} (0.1m)^2 = 60\pi W$$

Rendering assignment week 5-6

Alexander Birke

study nr: 124044 email: alex@alexanderbirke.dk

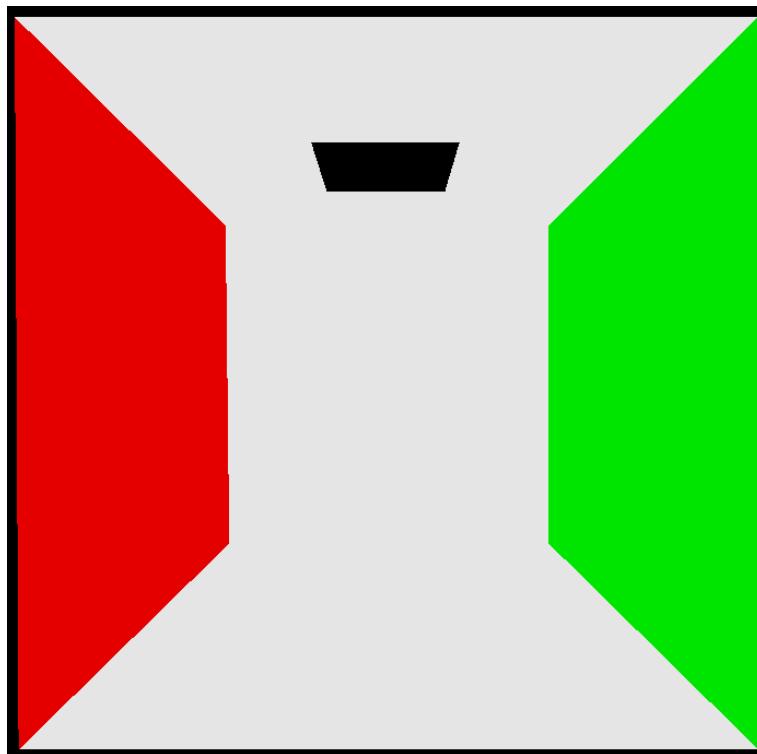
December 17, 2012

Introduction

This is my handin for week 5 and 6 of the Rendering course. The handin concerns the Radiosity method, which is specifically suited to do render LD*E paths.

Part 1 - Basic Setup & Meshing

When the radiosity framework is first compiled and run it produces the following flat shaded image:



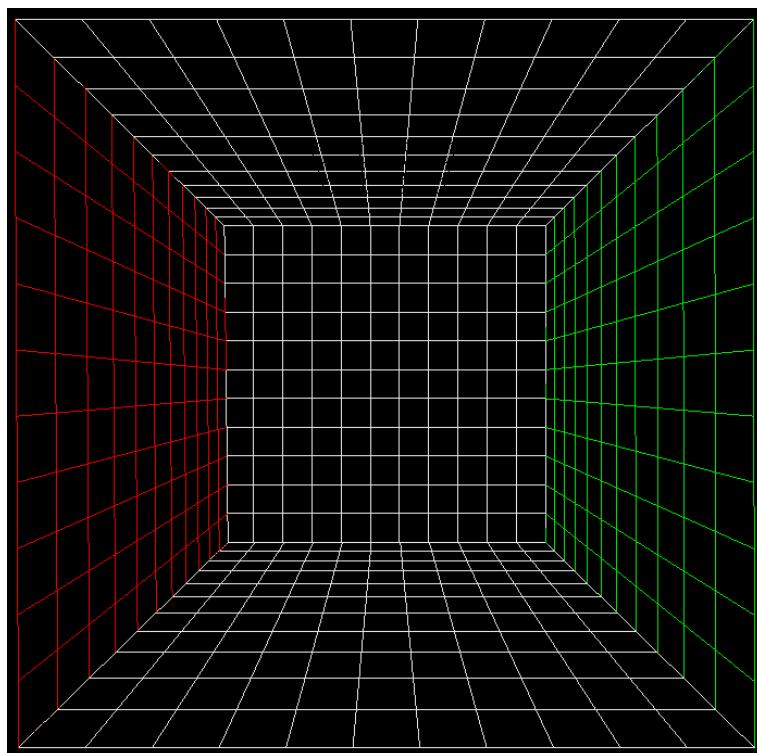
The next part is to implement tessellation of the scene geometry. This is done with the Mesher library and the code for doing this is the following:

```

1 //output variables
2 vector<MyPolygon*> tessellated_polygons;
3 vector<MyVertex*> tessellated_vertices;
4
5 //patch sizes
6 const float faceSize = 20;
7 const float lightSize = 20;
8
9 //tessellation
10 Mesher::mesh(loader_vertices, loader_polygons, tessellated_vertices,
11               tessellated_polygons, faceSize, lightSize);
12
13 polygons = tessellated_polygons;
14 vertices = tessellated_vertices;

```

And the result looks like this:



Part 2 - The Progressive Refinement Method

Progressive refinement is a method to do radiosity that is designed to over time produce a better and better result. This way the process can be cancelled if the artist finds that the result generated is not satisfactory. In the normal formulation of the radiosity method each patch gathers light from each other. In the progressive refinement method the radiosity is shot from the "brightest" patch, the patch with most unshot radiosity, to all the other patches in the scene. A patch thus has two fields, one that stores how much radiosity it has received overall, and another that stores how much radiosity it has received that it has not distributed out to other patches yet. The field for overall radiosity is used to

calculate the color of the patch.

The first part is to calculate the form factor between the patches based on the following analytical formula:

$$F_{dA_i dA_j} = \int_{A_j} \frac{\cos \Phi_i \cos \Phi_j}{\pi r^2} dA_j$$

If the patches are sufficiently small we do not need to take the integral over the patch since the form factor does not change that much when it is sufficiently small. The function can therefore be calculated in the following way:

```

1 MyPolygon* calcAnalyticalFF ()
2 {
3     // Reset all form factors to zero
4     for(unsigned int i = 0; i < polygons.size(); i++)
5         polygons[i]->formF = 0.0f;
6
7     // Find the patch with maximum energy
8     MyPolygon* maxEnergyPatch = maxenergy_patch();
9
10    //for all patches
11    for(unsigned int i = 0; i < polygons.size(); i++)
12    {
13        //find the vector between the two patches
14        Vec3f lineBetweenPatches = maxEnergyPatch->center - polygons[i]->center
15        ;
16        //and the normalized version of it
17        Vec3f normalizedLine = lineBetweenPatches;
18        normalizedLine.normalize();
19
20        //calculate phi for each patch
21
22        float phi1 = dot(normalizedLine, polygons[i]->normal);
23
24        if(phi1 < 0)
25            phi1 = 0;
26
27        float phi2 = dot(-normalizedLine, maxEnergyPatch->normal);
28
29        if(phi2 < 0)
30            phi2 = 0;
31
32        //finally calculate the form factor
33        polygons[i]->formF = polygons[i]->area * phi1*phi2 * M_1_PI / pow(
34            lineBetweenPatches.length(), 2 );
35
36    }
37    // Return the maximum patch
38    return maxEnergyPatch;
39 }
```

The next part is the function that distributes the unshot radiosity from the patch with most unshot radiosity to the other patches. This is based on the formula:

$$B_j = R_j B_i F_{ij} \frac{A_i}{A_j}$$

Where i refers to the patch being shot from and j refers to the patch being shot at. The code for doing this is shown below:

```

1 bool distributeEnergy (MyPolygon* maxP)
2 {
3     //if a null pointer is given, return false
4     if(maxP == 0)
5         return false;
6
7     // Distribute energy from the maximum patch to all other patches.
8     for(unsigned int i = 0; i < polygons.size(); i++)
9     {
10        //we do not distribute energy from the max energy patch onto itself
11        if(polygons[i] == maxP)
12            continue;
13
14        //calculate the radiosity received by patch i
15        Vec3f radiosity = polygons[i]->diffuse * polygons[i]->formF * ( maxP->
16            area / polygons[i]->area ) * maxP->unshot_rad ;
17
18        //add this to patch i's radiosity and unshot radiosity
19        polygons[i]->unshot_rad += radiosity;
20        polygons[i]->rad += radiosity;
21    }
22
23    // Set the unshot radiosity of the maximum patch to zero and return true
24    maxP->unshot_rad = Vec3f(0);
25
26    return true;
}

```

The function is quite simple. First it is tested if the function is passed a null pointer, if that is the case it returns false. Next it loops over all patches, skips the patch with max energy since a patch cannot distribute energy onto itself. Then the above formula is calculated and the result added to the current patch's radiosity and unshot radiosity.

Next, we need a function to calculate the color of each patch. The color is equal to the radiance of the patch. The relationship between radiosity and radiance for a diffuse reflector is:

$$B = L\pi$$

So this means the color is obtained by dividing the radiosity with phi.

```

1 void colorReconstruction ()
2 {
3     for(unsigned int i = 0; i < polygons.size(); i++)
4     {
5         polygons[i]->color = polygons[i]->rad * M_1_PI;
6     }
7 }

```

The last thing in this part is to call the function from inside the display function:

```

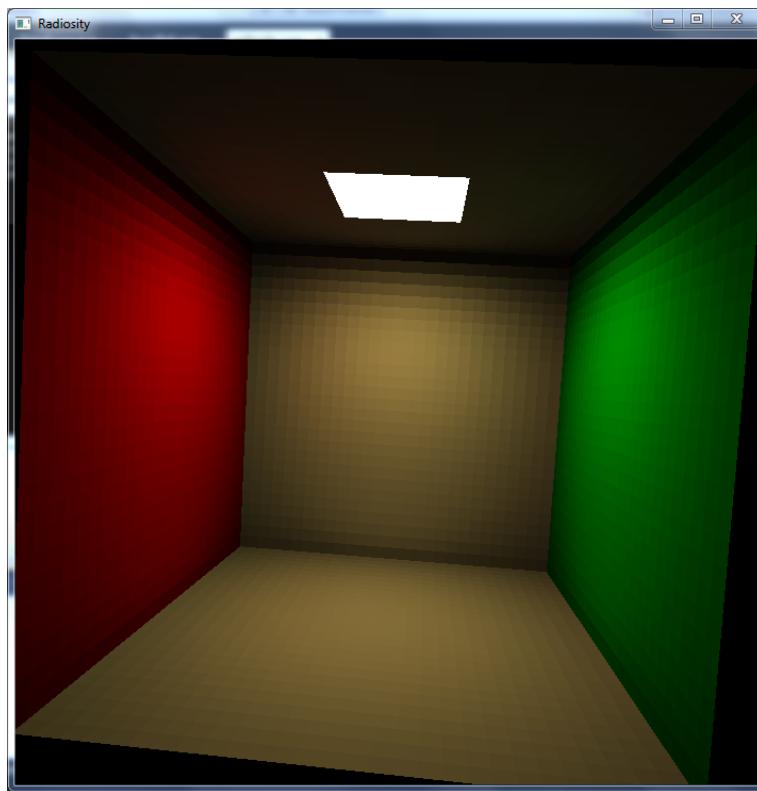
1 void display ()
2 {

```

```

3  MyPolygon* maxP = calcAnalyticalFF();
4
5  distributeEnergy(maxP);
6
7  colorReconstruction();
8
9  glViewport(0,0,screen_size,screen_size);
10 ball->do_spin();
11 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
12 glColor3f(1.0,0.0,0.0);
13 glLoadIdentity();
14 ball->set_gl_modelview();
15 displayMyPolygons();
16 glutSwapBuffers();
17 }
```

The result looks like this:



Part 3 - Smoothing via Nodal averaging

The main task here is to compute the color of each vertex based on the color of the patches that it is a part of. The strategy used is to first reset the colors of all the vertices, then go through all the patches and add the color of that patch to the vertices it is connected to. To obtain the color of a vertex we then divide with the number of patches it is connected to thus finding the mean color value. The function to calculate color is therefore changed to:

```
1 void colorReconstruction()
```

```

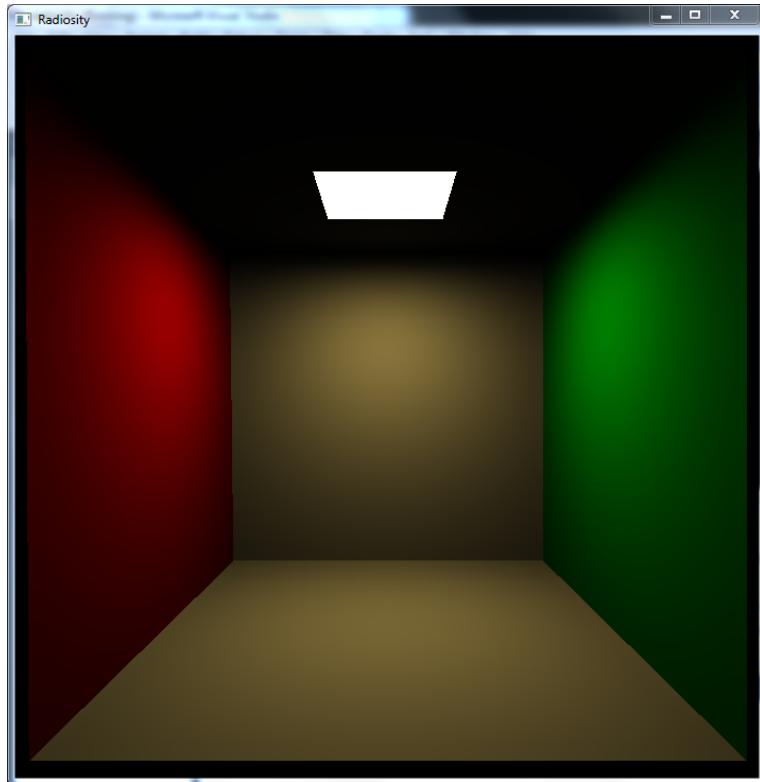
2 {
3     for(unsigned int i = 0 ; i < vertices.size() ; i++)
4     {
5         vertices[i]->color = Vec3f(0);
6         vertices[i]->colorcount = 0;
7     }
8
9     for(unsigned int i = 0; i < polygons.size(); i++)
10    {
11        polygons[i]->color = polygons[i]->rad * M_1_PI;
12
13        for(unsigned int k = 0 ; k < polygons[i]->vertices ; k++)
14        {
15            vertices[polygons[i]->vertex[k]]->color += polygons[i]->color;
16            vertices[polygons[i]->vertex[k]]->colorcount += 1;
17        }
18    }
19
20    for(unsigned int i = 0 ; i < vertices.size() ; i++)
21    {
22        vertices[i]->color = vertices[i]->color / vertices[i]->colorcount;
23    }
24 }
```

Next we need to change the `DisplayMyPolygons` function so it uses the vertex colors instead of the patch's color:

```

1
2 void displayMyPolygons()
3 {
4     for(int i=0;i<polygons.size();i++)
5     {
6         if (4==polygons[i]->vertices)
7             glBegin(GL_QUADS);
8         else if (3==polygons[i]->vertices)
9             glBegin(GL_TRIANGLES);
10        else
11            assert(false); // illegal number of vertices
12
13        for (int j=0;j<polygons[i]->vertices;j++)
14        {
15            glColor3f(vertices[ polygons[i]->vertex[j] ]->color[0], vertices[
16                polygons[i]->vertex[j]]->color[1], vertices[ polygons[i]->vertex[j]
17                ]->color[2]);
18
19            Vec3f position = vertices[polygons[i]->vertex[j]]->position;
20            glVertex3f(position[0], position[1], position[2]);
21        }
22    }
```

Since OpenGL's fixed function pipeline is based on Garroud shading and the shading is already set to be smooth we do not have to do more. The result looks like this:



Part 4 - Occlusion via the Hemicube Method

There are two functions that need to be implemented in order to use the hemicube method, the first is renderPatchIDs that prepares the scene to be rendered to the hemicube by setting the color of each patch to its index (plus one to avoid that areas where there are no patches are considered as the first patch in the vector). The main task is therefore to change what is done in the DisplayMyPolygons function to use the index of the patch instead. The index therefore needs to be changed into a RGB value. This is done with bit masks and bit shifting:

```

1 void renderPatchIDs()
2 {
3     // Render all polygons in the scene as in displayMyPolygons ,
4     // but set the colour to the patch index using glColor3ub .
5     // Look at the Hemicube::getIndex function to see how the
6     // indices are read back .
7
8     for (int i=0;i<polygons . size () ;i++)
9     {
10         if (4==polygons [ i ]->vertices )
11             //if(4==polygons [ i ]->vertices )
12             glBegin(GL_QUADS) ;
13         else if (3==polygons [ i ]->vertices )
14             glBegin(GL_TRIANGLES) ;
15         else
16             assert (false) ; // illegal number of vertices
17
18         unsigned int index = i + 1;

```

```

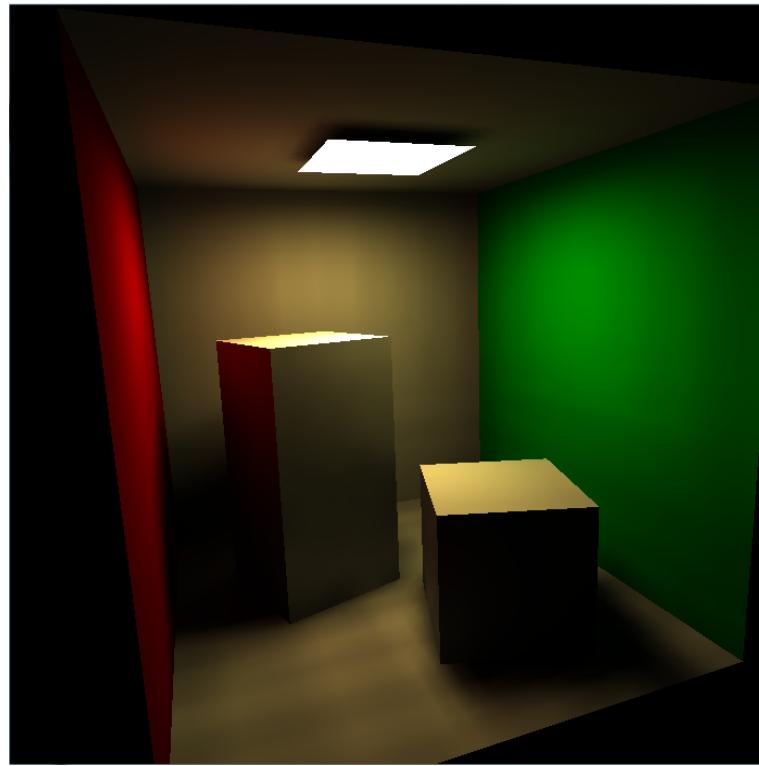
19
20     unsigned char r = ( index & 0xff0000 ) >> 16;
21     unsigned char g = ( index & 0x00ff00 ) >> 8;
22     unsigned char b = ( index & 0x0000ff );
23
24     glColor3ub(r, g, b);
25
26     for (int j=0;j<polygons[i]->vertices;j++)
27     {
28         Vec3f position = vertices[polygons[i]->vertex[j]]->position;
29         glVertex3f(position[0], position[1], position[2]);
30     }
31     glEnd();
32 }
33 }
```

Next, the form factors can be calculated. Here the hemicube class is used. This requires that a virtual camera is set up which looks in the direction of the max energy patch's surface normal, has its eye point in the plane of the patch and the up vector also lying in the same plane. When the scene has been rendered the indexes are read from the frame buffer and then used to calculate the form factor of each patch. The function looks like this:

```

1 MyPolygon* calcFF(Hemicube* hemicube)
2 {
3     for(unsigned int i = 0; i < polygons.size(); i++)
4     {
5         polygons[i]->formF = 0;
6     }
7
8     MyPolygon* maxP = maxenergy_patch();
9
10    Vec3f up = vertices[ maxP->vertex[0] ]->position - maxP->center;
11
12    hemicube->renderScene(maxP->center, up, maxP->normal, renderPatchIDs);
13
14    hemicube->readIndexBuffer();
15
16    for(unsigned int i = 0 ; i < hemicube->rendersize ; i++)
17    {
18        for(unsigned int k = 0 ; k < hemicube->rendersize ; k++)
19        {
20            unsigned int index = hemicube->getIndex(i,k);
21
22            if(index > 0)
23            {
24                float f = hemicube->getDeltaFormFactor(i,k);
25
26                polygons[index - 1]->formF += f;
27            }
28        }
29    }
30    // Return the maximum patch
31    return maxP;
32 }
```

With the hemicube the cornell box looks like this:



Part 5 - Answers to Questions

What is the effect of changing the subdivision level of the light source?

if the level is too low banding or aliasing artifacts appear. This is because the hemicube method is an approximation to calculating the form factor between two patches.

*What happens when you change the size of the hemicube? And why does it happen?
What can be done in order to avoid this problem?*

If the size of the hemicube is to small aliasing will mean that artefacts will be present in the final image. The cause of this is that the projection of the patches down on the hemicube cannot be represented properly if the hemicube's resolution is to low. The mean way to fix this problem is therefore to use the right resolution that produces an image without artefacts but which does not take to long time to evaluate.

What are the advantages and disadvantages of radiosity as compared to ray tracing?

The advantages are:

- viewpoint independent which means that for a static scene it only needs to be calculated once and the viewport can then be moved around where it is only the projection of the geometry that is needed to be calculated. This is probably also an advantage in architectural visualisations.
- Color bleeding. A surface with one color will project that color onto nearby surfaces.

- Soft shadows.

The disadvantages are:

- slower than for example basic ray tracing, and not much research has gone into how it can be optimized.
- Light and shadow leakage can occur if patch boundaries does not coresspond to boundaries of the geometry
- only LD*E paths can be simulated with the radiosity method.
- if only Garraud interpolation is used between patches, mach banding can occur if the patches are large enough.
- insufficient meshing strategies can cause discontinuities in the radiosity across a surface. This can for example be seen on my rendering of the cornell box where the box most in the front does not have shadow in its left lower corner where it should have.

Week 7 Report

Jakub Dutkowski, s121937

Alexander Birke, s124044

December 19, 2012

1 INTRODUCTION

The purpose of this week's set of exercises is to introduce anti-aliasing to the previously developed ray casting. This is done to make edges in the render look more smooth.

2 DESCRIPTON

- *Describe how the pixel subdivision level is changed while the program is running.*
The `subdivs` variable is incremented and `compute_jitters` function is called, which fills the `jitter` array.
- *Explain what the function `compute_jitters` stores in the vector array `jitter`.*
The `jitter` array stores coordinates of random points inside subpixels, which are calculated based on coordinates of the pixel and number of its subpixels.
- *Explain how many subpixels we get for each pixel when the pixel subdivision level is `subdivs = s`.*
Each dimension of a pixel is divided into s parts, therefore the number of subpixels equals s^2 .

3 CODE

Listing 1: Part of the displayMyPolygons function from radiosity.cpp file

```
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y)
    const
{
    float2 viewportCoords = optix::float2();
    5
    viewportCoords.x = lower_left.x + win_to_ip.x*x;
    viewportCoords.y = lower_left.y + win_to_ip.y*y;
    10
    float3 result = make_float3(0);

    // for each subpixel
    for(int i = 0; i < subdivs; i++)
        for(int j = 0; j < subdivs; j++)
    15
    {
        // create a ray and hit
        optix::Ray ray = scene->get_camera()->
            get_ray(viewportCoords + jitter[i*subdivs + j]);
    20
        HitInfo info;

        // and then trace it
        if(scene->closest_hit(ray, info))
    25
        {
            result += get_shader(info)->shade(ray, info);
        }
        else
    30
        {
            result += get_background(ray.direction);
        }

    }

    // when all rays have been traced, divide by the
    // number of subpixels to get the correct result
    35
    return result/(subdivs*subdivs);
}
```

4 RENDER RESULTS

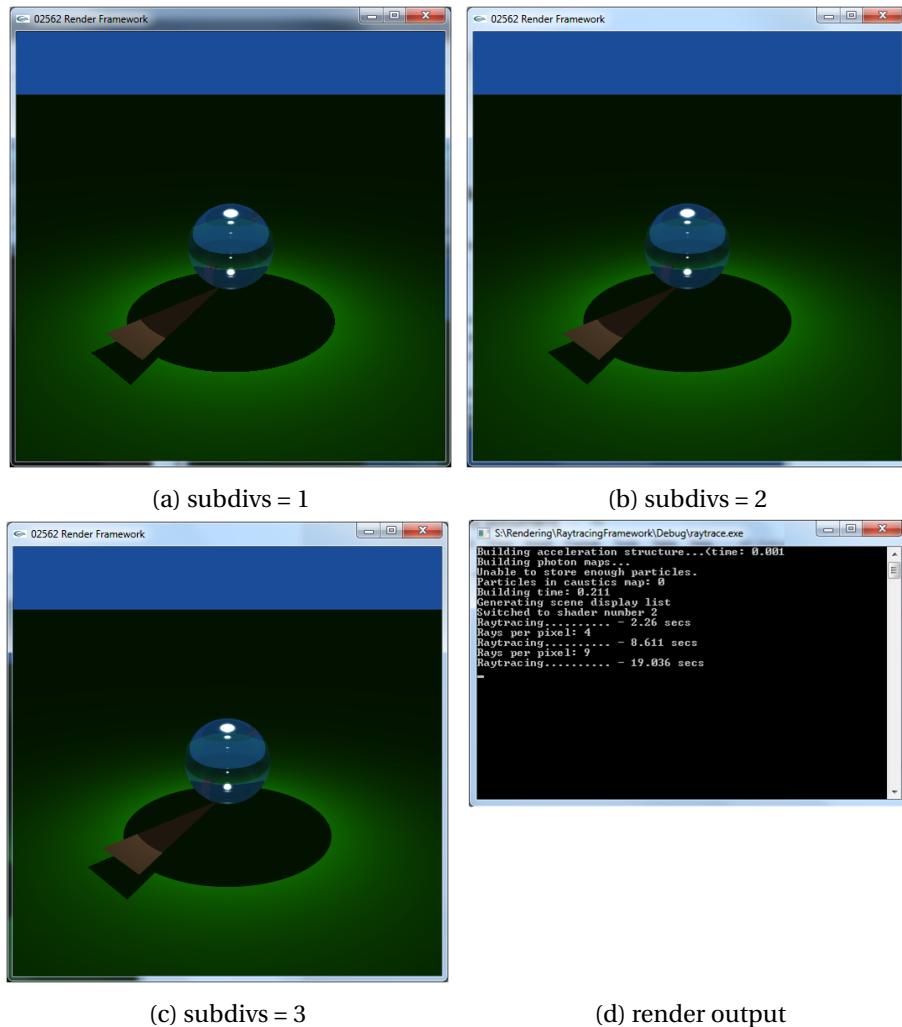


Figure 4.1: screenshots of render results

5 FINAL QUESTIONS

- *What is the relationship between the pixel subdivision level and the render time?*
The time is proportional to squared subdivision level.
- *What is the relationship between the pixel subdivision level and the aliasing error in the render result?*
Using a higher number of rays per pixel decreases the aliasing error. High number of rays shot through one pixel means that the resulting color of the pixel is calculated as an average of bigger number of values and therefore produces a more accurate result.
- *At what pixel subdivision level would you say that the improvement is no longer visible?*
We cannot see any improvement between subdivision level 2 and 3.
- *At what pixel subdivision level would you say that the improvement is no longer worth the increase in render time?*
It depends on the situation, for a test render we can say that there is no point using subdivision, for the final render it's worth waiting for render with subdivision level 2, but not more as there is no further improvement visible.

Weeks 10 Report - Texture Mapping

Jakub Dutkowski, s121937
Alexander Birke, s124044

December 19, 2012

1 INTRODUCTION

The purpose of these exercises was to add texture mapping to the raytracing framework.

2 LOADING OF IMAGE FILES

The loading of textures is first started when the plane is loaded with the following line in RenderEngine.cpp:

```
1 scene.add_plane(make_float3(0.0f, 0.0f, 0.0f), make_float3(0.0f, 1.0f, 0.0f), ".../models  
/default_scene.mtl", 1, 0.02f);
```

The add_plane function in In Scene.cpp is passed the path to the default scene material description file. The add_plane function looks like this.

```
1 void Scene::add_plane(const float3& position, const float3& normal, const string&  
    mtl_file, unsigned int idx, float tex_scale)  
2 {  
3     vector<ObjMaterial> m;  
4     if (!mtl_file.empty())  
5         mtl_load(mtl_file, m);  
6     if (m.size() == 0)  
7         m.push_back(ObjMaterial());  
8     Plane* plane = new Plane(position, normal, idx < m.size() ? m[idx] : m.back(),  
        tex_scale);  
9     planes.push_back(plane);  
10}
```

First a vector is created to store all the materials in the file. The file is then passed with the call to mtl_load and all the materials in the file is put into the vector. A plane is then created and assigned a copy of the created material that corresponds to its index. An improvement to the parsing would be to only parse the mtf file once before the scene is created. As it is right now the mtf file is parsed each time a primitive is loaded. For each material the following three variables are defined: In ObjMaterial.h:

```

1 bool has_texture;
2 std::string tex_path, tex_name;
3 int tex_id;
```

They are set by the material's constructor. After the path to the texture has been stored in the material by the call to load_files the init_textures in the render_engine is called afterwards. This in turn calls the function load_texture for all the planes, spheres and other meshes in the scene.

```

1 void Scene::load_texture(const ObjMaterial& mat, bool is_sphere)
2 {
3     if(mat.has_texture && textures.find(mat.tex_name) == textures.end())
4     {
5         Texture*& tex = textures[mat.tex_name];
6         tex = is_sphere ? new InvSphereMap : new Texture;
7         string path_and_name = mat.tex_path + mat.tex_name;
8         tex->load(path_and_name.c_str());
9     }
10 }
```

If the material has a texture the Texture::load function gets called:

```

1 void Texture::load(const char* filename)
2 {
3     SOIL_free_image_data(data);
4     data = SOIL_load_image(filename, &width, &height, &channels, SOIL_LOAD_AUTO);
5     if (!data)
6     {
7         cerr << "Error:_Could_not_load_texture_image_file ." << endl;
8         return;
9     }
10    int img_size = width*height;
11    delete [] fdata;
12    fdata = new float4 [img_size];
13    for (int i = 0; i < img_size; ++i)
14        fdata[i] = look_up(i);
15    tex_handle = SOIL_create_OGL_texture(data, width, height, channels, tex_handle,
16                                         SOIL_FLAG_INVERT_Y);
17    tex_target = GL_TEXTURE_2D;
```

This function utilizes the Simple OpenGL Image Library that does all the more low level work like reading the compressed data in the texture into the data variable that can be accessed by the rest of the program. In order to use the data of the texture in the rest of the framework we also need to convert it into floating point format. This is done in the next section.

3 TEXTURE COLOUR'S USE IN RENDERING

In the load function an array called fdata is declared and filled with normalized floating point values of each pixels. This is done with the functions look_up and convert:

```
1 float4 Texture::look_up(unsigned int idx) const
2 {
3     idx *= channels;
4     switch(channels)
5     {
6         case 1:
7         {
8             float v = convert(data[idx]);
9             return make_float4(v, v, v, 1.0f);
10        }
11        case 2:
12        {
13            return make_float4(convert(data[idx]), convert(data[idx]), convert(data[idx]),
14                               convert(data[idx + 1]));
15        }
16        case 3:
17        {
18            return make_float4(convert(data[idx]), convert(data[idx + 1]), convert(data[idx +
19                           2]), 1.0f);
20        }
21        case 4:
22        {
23            return make_float4(convert(data[idx]), convert(data[idx + 1]), convert(data[idx +
24                           2]), convert(data[idx + 3]));
25        }
26    }
27    return make_float4(0.0f);
28 }
```

In look_up the index it is passed is first divided with the number of channels in the texture. This is done since the texture's data is stored in a flat array and the number of channels thus specifies how many bytes each pixel fills. The channel variable is also used in a switch case to determine how the floating data returned should be formatted. If it is a monocromatic texture then a greyscale color is returned. If the texture also contains an alpha channel this is also used in the calculation as well. If the texture has the three colour channels but no alpha channel then the alpha value returned is set to the default opaque value of 1.0f. The last case is if the texture contains an alpha channel, then this is also converted and returned as well.

The convert function takes an char value in the range of 0 to 255 and remaps it to the range 0 to 1 in floating point format. What is weird is that the maximum value that can be mapped to is $255 + 0.5f / 256 = 0.998046875$. A more correct way would be to add 1.0f instead of 0.5f.

4 COMPUTATION OF TEXTURE COORDINATES & TEXTURE LOOKUP

We add the following lines to the plane's intersect function:

```

1  if(material.has_texture)
2  {
3      float3 vector = hit.position - position;
4
5      // to get a basis change matrix we need an inverse matrix of [tangent, binormal,
6      // normal]
6      // as the vectors in the basis are orthogonal, we can use transposed matrix
7      instead
7
8      float3 transposed1 = make_float3(onb.m_tangent.x, onb.m_binormal.x, onb.m_normal.x
9          );
9      float3 transposed2 = make_float3(onb.m_tangent.y, onb.m_binormal.y, onb.m_normal.y
10         );
10     float3 transposed3 = make_float3(onb.m_tangent.z, onb.m_binormal.z, onb.m_normal.z
11         );
11
12     hit.texcoord = (vector.x * transposed1 + vector.y * transposed2 + vector.z *
13         transposed3) * tex_scale;
13 }
14
15 return true;

```

What we do here is to remap the hit position in world coordinates into coordinates relative to the plane's center. This is done by first finding the vector that goes from the positon of the plane in world coordinates and to the hit also in world coordinates. We then use the ortonormal basis of the plane to transform this vector into the coordinate system of the plane. We then scale it with `tex_scale` in order to be able to control the tiling of the texture.

The next step is to implement basic point sampling. This is done via the function `sample_nearest`:

```

1 float4 Texture::sample_nearest(const float3& texcoord) const
2 {
3     if (!fdata)
4         return make_float4(0.0f);
5
6     int a = floor((texcoord.x - floor(texcoord.x)) * (width - 1) + .5);
7     int b = floor((1 - (texcoord.y - floor(texcoord.y))) * (height - 1) + .5);
8
9     return fdata[a + b * width];
10 }

```

The texel coordinates is found by first remapping the `texcoord` into a value between 0 and 1. This is done by substracting the floor'ed value. This normalized value is then multiplied with the width (or height respectively) minus one to make sure we do not go over the length of the pixel array. at the end we add 0.5 in order to take the pixel whose center is closest to the specified uv coordinates. At the end the texel coordinates is used to look into the flat array and return the correct pixel value.

5 THE ROLE OF THE PIXEL SUBDIVISION LEVEL

The next task was to render the image with different subdivisions where nearest neighbour sampling was used. The result can be seen on figure 5.1. As it can be seen the quality of the texture lookup goes up when more samples are introduced. However because the samples originate from the image plane and is not performed in the texture lookup the quality only goes up when the jittered rays has had some distance to scatter or said in another way when the rays have traveled farther. In order to improve the quality of the texture no matter how close it is we need to introduce more samples directly in the texture lookup.

6 BILINEAR SAMPLING

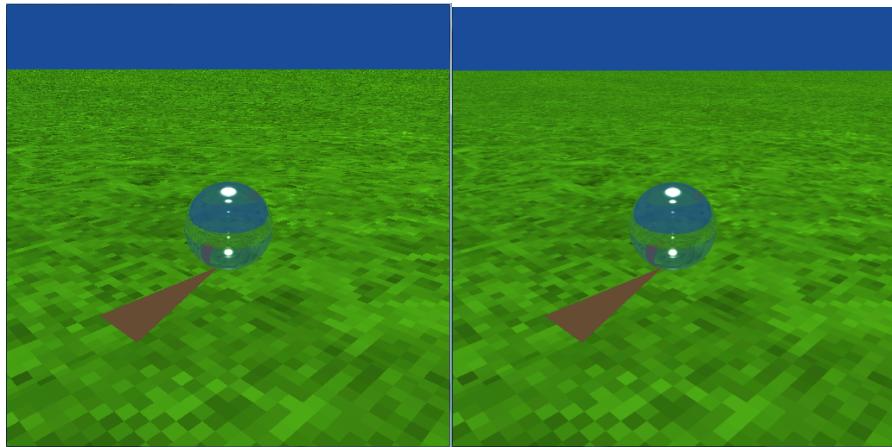
The next step is to implement bilinear sampling:

```
1 float4 Texture::sample_linear(const float3& texcoord) const
2 {
3     if (!fdata)
4         return make_float4(0.0f);
5
6     float a = (texcoord.x - floor(texcoord.x)) * width;
7     float b = (texcoord.y - floor(texcoord.y)) * height;
8
9     float2 alpha, beta, gamma, delta;
10
11    alpha = make_float2(floor(a), floor(b));
12    beta = make_float2(fmodf(ceil(a), width), floor(b));
13    delta = make_float2(fmodf(ceil(a), width), fmodf(ceil(b), height));
14    gamma = make_float2(floor(a), fmodf(ceil(b), height));
15
16    float4 c = lerp( fdata[ (int)alpha.x + (int)alpha.y * width ] , fdata[ (int)beta.
17        x + (int)beta.y * width ], a - floor(a) );
18    float4 d = lerp( fdata[ (int)gamma.x + (int)gamma.y * width ] , fdata[ (int)delta.x +
19        (int)delta.y * width ], a - floor(a) );
20
return lerp(c, d, b - floor(b));
```

We first start by finding the texel coordinates of the sample. We then find the four pixels that are closest to this sample. After this is the bilinear interpolation performed. This is done by first doing a linear interpolation between the two sets of pixels that lies on top of each other. At the end another interpolation is done between the two points to produce the final result. The bilinear sampling produces the result seen on figure ?? As it can be seen this produces a much better result than point sampling. This is especially noticeable when the texture gets magnified - is closer to the viewer. There is still aliasing farther ahead but it comes at a greater distance. When this happens the only way to improve the quality is to introduce more samples through stochastic sampling. Still the bilinear sampling does give a better result at a lower amount of jittered rays but

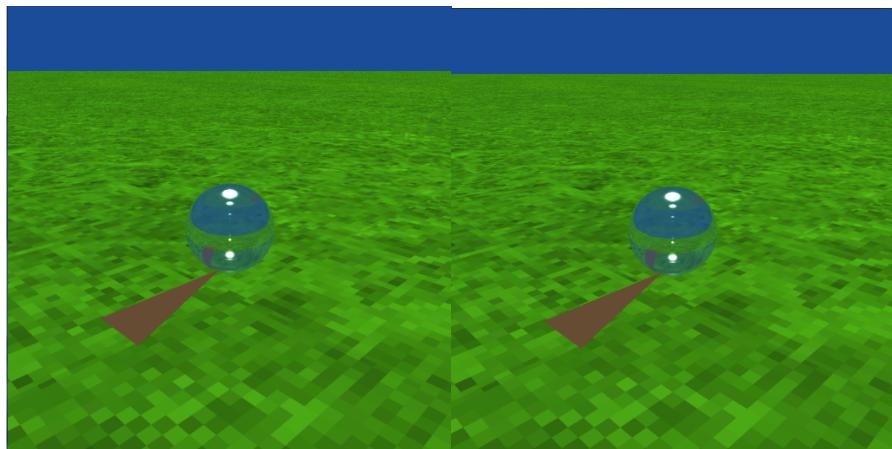
7 SCALING OF TEXTURE COORDINATES

When the texture scaling is increased to 10 it means the texture will repeat 10 times as often since the uv coordinates changes 10 times faster than if the scaling was set to 1. This results in figure 7.1



(a) Subdivision 0

(b) Subdivision 1



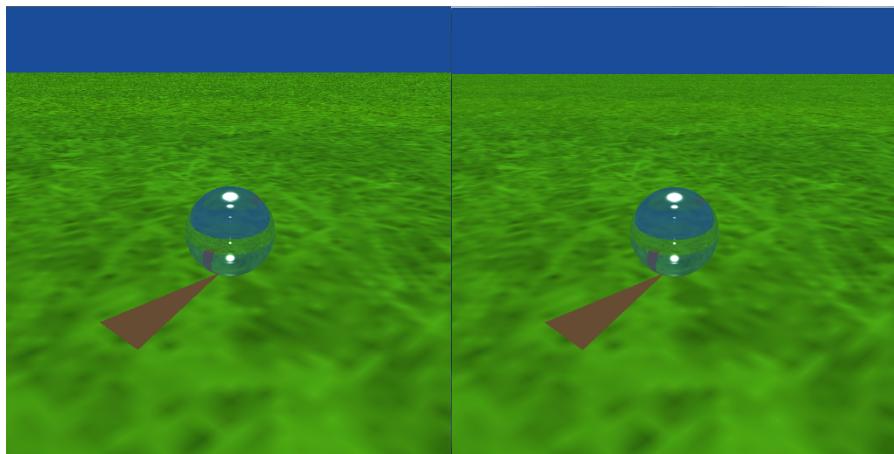
(c) Subdivision 2

(d) Subdivision 3

```
C:\Users\alexander\Downloads\RaytracingFramework>
Building acceleration structure...(time: 0.000)
Building photon maps...
Particles in caustics map: 20007
Building time: 1.769
Generating scene display list
Toggled textures on.
Raytracing..... - 4.322 secs
Rays per pixel: 4
Raytracing..... - 19.247 secs
Rays per pixel: 9
Raytracing..... - 39.187 secs
Rays per pixel: 16
Raytracing..... - 67.865 secs
```

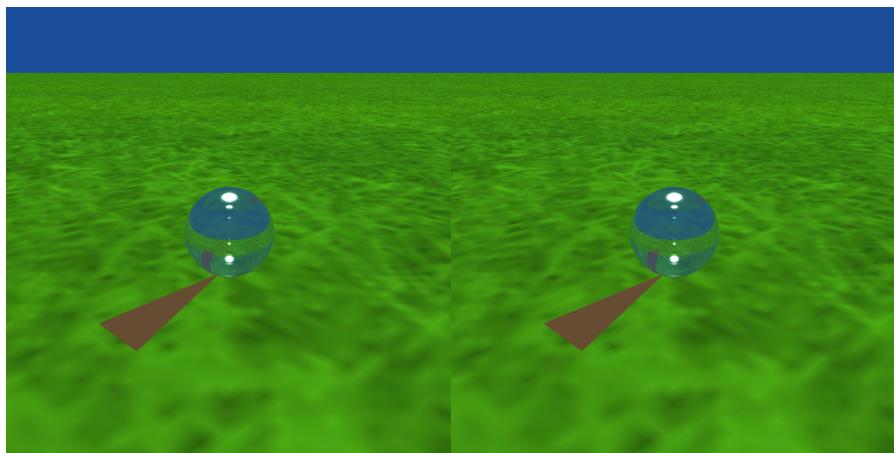
(e) Console output of subdivision

Figure 5.1: Renderings of default scene with nearest neighbour sampling



(a) Subdivision 0

(b) Subdivision 1



(c) Subdivision 2

(d) Subdivision 3

```
C:\Users\alexander\Downloads\RaytracingFramework
Building acceleration structure...(ti
Building photon maps...
Particles in caustics map: 20007
Building time: 2.047
Generating scene display list
Toggled textures on.
Raytracing..... - 6.149 secs
Rays per pixel: 4
Raytracing..... - 24.936 secs
Rays per pixel: 9
Raytracing..... - 54.275 secs
Rays per pixel: 16
Raytracing..... - 95.801 secs
```

(e) Console output of subdivision

Figure 6.1: Renderings of default scene with linear interpolation sampling

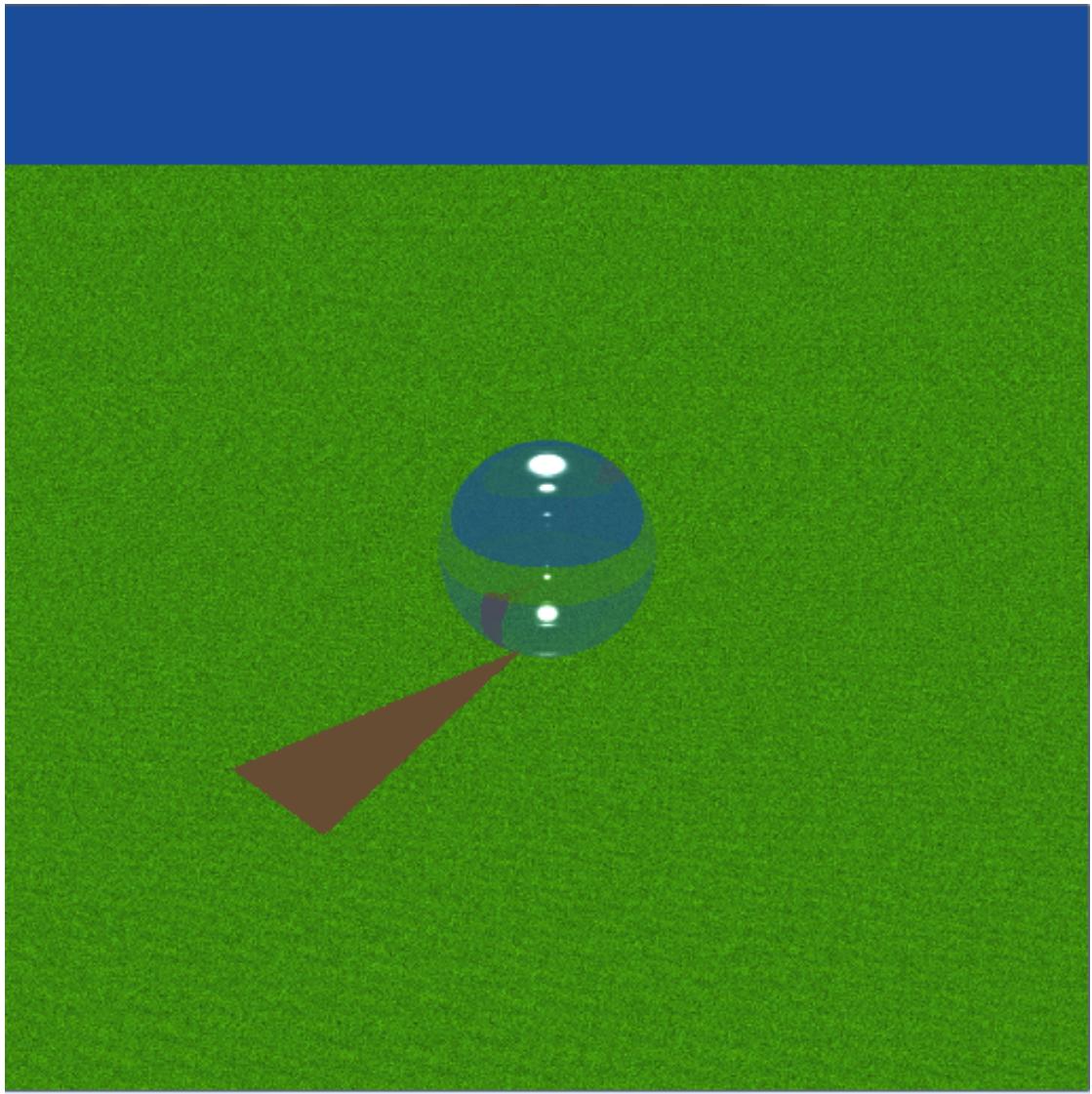


Figure 7.1: Image where texture is scaled with a factor of 10

TECHNICAL UNIVERSITY OF DENMARK

02562 - INTRODUCTION TO RENDERING

Ambient Occlusion & Environment Mapping

Authors:

Alexander BIRKE

Jakub DUTKOWSKI

December 19, 2012

Contents

1 Motivation	3
2 Theory	4
2.1 The Basic Concept	4
2.2 Mathematical Foundation	5
2.3 Environment maps & High Dynamic Range Imaging	6
2.4 Lightprobe lookup	7
2.5 Disadvantages of Ambient Occlusion	8
3 Implementation	9
3.1 Ambient Occlusion Shader	9
3.1.1 Rejection Sampling	9
3.1.2 Cosine Weighted Hemisphere	10
3.2 Environment Sampling	11
3.2.1 HDR Image Conversion	11
3.2.2 Spherical Texture Lookup	12
4 Results & Discussion	14
4.1 Comparison of ambient occlusion results	14
5 Conclusion	16

Chapter 1

Motivation

Even though global illumination techniques have come a long way they still have many limitations. One of them is the often long render times they require. Ambient occlusion is a rather simple technique that allows indirect lighting from an environment to be simulated cheaply. The technique was originally developed by Hayden Landis and colleagues at Industrial light and Magic to accommodate the needs they had for production ready global illumination [5]. Ambient occlusion together with environment mapping has since then often been used in the industry. The reason for this is that ambient occlusion besides being faster also allows for easy compositing of synthetic 3D images into live video footage[5]. The technique is also often sought implemented in a realtime context since its cheap evaluation lends itself well to realtime graphics[4, 9, 8].

We will in this report present an implementation of Ambient Occlusion into the Ray-tracing framework used for the course "02562 - Rendering Introduction" at Technical University of Denmark.

Chapter 2

Theory

This section will explain the basic theory and concept behind Ambient Occlusion. Besides covering the basic concept and math behind ambient occlusion it will also be covered how texture lookup into light probe images are performed. The chapter will conclude with a discussion of the disadvantages of ambient occlusion.

2.1 The Basic Concept

Ambient Occlusion works by seeing how much of an external environment can be seen from each surface point on the object or scene to be rendered. The more of the external environment that can be seen, the more ambient lighting that point will receive. This type of illumination is therefore often called "sky light" in production, because it simulates how a model would be lighted on an overcast day, since the light from ambient occlusion is incident on the point from all directions due to the scattering of the clouds.

When ambient occlusion is calculated two factors are obtained. The first is the accessibility term. This is the measure of how much of the environment that is accessible from the surface point. The other is the "bent normal", which is the average direction of the incident light from the environment. The bent normal can then be used to look up into a light probe, which is a special type of image, in order to give more realistic lighting without the added cost of a global illumination algorithm. These two concepts are illustrated on figure 2.1 that illustrates how these two variables are calculated in a ray tracer. The accessibility at point P is determined by shooting rays out from the hemisphere

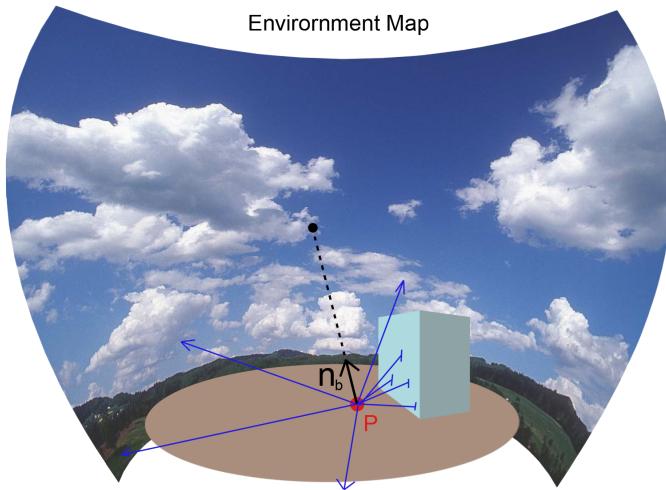


Figure 2.1: Overview of ambient occlusion.

surrounding the surface around P. The number of rays that does not hit other objects determines the accessibility. The average of the unoccluded rays is the bent normal that can be used to make a lookup into an environment map such as a light probe.

2.2 Mathematical Foundation

Ambient Occlusion as all other realistic rendering methods tries to approximate the rendering equation:

$$L_o(\mathbf{x}, \vec{w}) = L_e(\mathbf{x}, \vec{w}) + \int_{2\pi} f_r(\mathbf{x}, \vec{w}', \vec{w}) L_i(\mathbf{x}, \vec{w}') \cos \theta dw'$$

We will approximate this with the Monte Carlo estimator[2]:

$$L_o(\mathbf{x}, \vec{w}) = L_e(\mathbf{x}, \vec{w}) + \frac{1}{N} \sum_{i=1}^N \frac{f_r(\mathbf{x}, \vec{w}', \vec{w}_i) L_i(\mathbf{x}, \vec{w}') \cos \theta}{pdf(\vec{w}'_i)}$$

if we only consider diffuse reflectance then the BRDF is equal to the BRDF for Lambertian reflectance:

$$f_r(\mathbf{x}, \vec{w}', \vec{w}) = \frac{R_d}{\pi}$$

Then the expression becomes:

$$L_o(\mathbf{x}, \vec{w}) = L_e(\mathbf{x}, \vec{w}) + \frac{1}{N} \sum_{i=1}^N \frac{\frac{R_d}{\pi} L_i(\mathbf{x}, \vec{w}') \cos \theta}{pdf(\vec{w}'_i)}$$

This expression can be evaluated directly by rejection sampling the hemisphere. Since the probability density function for this sampling method is[2]:

$$PDF : p(\Theta) = \frac{1}{2\pi}$$

This means the above Monte Carlo Estimator end up being:

$$L_o(\mathbf{x}, \vec{w}) = L_e(\mathbf{x}, \vec{w}) + \frac{1}{N} \sum_{i=1}^N \frac{\frac{R_d}{\pi} L_i(\mathbf{x}, \vec{w}') \cos \theta}{pdf(\frac{1}{2\pi})}$$

$$L_o(\mathbf{x}, \vec{w}) = L_e(\mathbf{x}, \vec{w}) + \frac{2}{N} R_d \sum_{i=1}^N L_i(\mathbf{x}, \vec{w}') \cos \theta$$

However a better approach would be to simplify the expression even more. A good probability density function would therefore be[2]:

$$pdf((\vec{w}'_i)) = \cos \theta / \pi$$

The goal is therefore to find a sampling method that has this probability density function. One is[2]:

$$\vec{w}'_i = (\theta, \phi) = (\cos^{-1} \sqrt{r_1}, 2\pi r_2)$$

Where r1 and r2 are random numbers between 0 and 1. When this sampling method and a visibility function denoted V is used to describe incident illumination the equation finally becomes:

$$L_o(\mathbf{x}, \vec{w}) = \frac{1}{N} \sum_{i=1}^N \frac{\frac{R_d}{\pi} V(\mathbf{x}, \vec{w}') \cos \theta}{\cos \theta / \pi}$$

$$L_o(\mathbf{x}, \vec{w}) = \frac{1}{N} \sum_{i=1}^N R_d V(\mathbf{x}, \vec{w}')$$

$$L_o(\mathbf{x}, \vec{w}) = \frac{R_d}{N} \sum_{i=1}^N V(\mathbf{x}, \vec{w}')$$

Where V is equal to 1 if the direction is unoccluded and 0 otherwise. In a raytracer this is normally done by casting N rays with origin in the surface point being sampled with a cosine weighted random direction in the point's hemisphere, and divide the result by N. A similar approach is used to calculate the bent normal. This is the sampling method used in Landis' paper[5] but other papers choose not to include the cosine term in the ambient occlusion integral to start with [3]. Some also choose to let V return a value between 0 and 1 depending on the distance to the occluder where V returns 1 if the occluder being hit is farther away than a certain threshold [6].

2.3 Envirornment maps & High Dynamic Range Imaging

As mentioned earlier the bent normal can be used to do a lookup into an image. Most often this image comes in the form of a light probe, which is an omnidirectional image that stores incident light at a certain point in space. Since the human visual perception is able to detect a high range of luminosity values, a high dynamic range format such as RGBE is often used. In RGBE the fourth byte stores an exponential value used to modify the RGB value. This allows the format to have the same range as floating point values. This way it can handle both very bright and dark pixels. The formula for converting RGBE value is simply:

$$R_w = \frac{R_m + 0.5}{256} 2^{E-128}$$

$$G_w = \frac{G_m + 0.5}{256} 2^{E-128}$$

$$B_w = \frac{B_m + 0.5}{256} 2^{E-128}$$

When the light probe is used to illuminate a diffuse surface the most used method is to sample the image in a region around the point sample from the bent normal. This way the lookup becoems blurred which gives a more realistic look when used to color a diffuse surface[5].

2.4 Lightprobe lookup

A lightprobe is stored as a latitude longitude map as shown on figure 2.2.



Figure 2.2: Example of a light probe from St Peter's Basilica, Rome. Source[1]

When sampling a light probe it is therefore necessary to convert the 3 dimensional direction of the bent normal into a set of uv coordinates. We will now derive how. Since it is custom to have the center of the lightmap be the upwards direction we get that the base of the uv coordinates should be:

$$(u, v) = \left(\frac{1}{2}, \frac{1}{2}\right)$$

the next part is to determine at what distance and in what direction from the center of the texture we should sample. Since we start from the middle of the texture the distance should go from 0 to 0.5. We can obtain this range by taking the arccos of the z value and divide by two pi:

$$dist = \frac{\arccos(D_z)}{2\pi}$$

The x and y values tells us in what direction from the center we should go out to get to the right position. We therefore have to get the normalized xy vector:

$$direction = \frac{1}{\sqrt{D_x^2 + D_y^2}}(D_x, D_y)$$

At the end we can rearrange to end up with the following:

$$r = \frac{\arccos(-D_z)}{2\pi\sqrt{D_x^2 + D_y^2}}$$

$$(u, v) = \left(\frac{1}{2} + rD_x, \frac{1}{2} + rD_y\right)$$

Where D_x , D_y and D_z is the lookup direction. Here it is assumed that the z-axis is upwards. One has also to take care of the event where D_x and D_y is close or equal to zero in which case the function should return $(0.5, 0.5)$.

2.5 Disadvantages of Ambient Occlusion

The main problem with Ambient Occlusion is from the fact that it reduces the problem of finding incident light, to using the calculated bent normal to do a texture lookup in an environment map. Since the bent normal is the average direction of incident light, it can in certain cases point in a direction that is actually occluded as illustrated by figure 2.3.

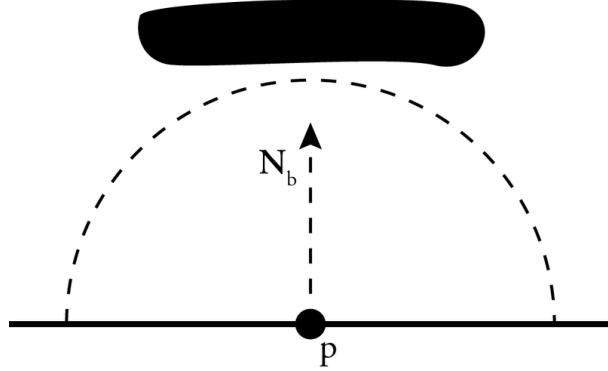


Figure 2.3: The problem of using a bent normal N_b to find the irradiance at point P. However since ambient occlusion is used to calculate the diffuse term the lookup into the environment map is usually blurred which means this error is not so noticeable.

The next section will cover our implementation of Ambient Occlusion.

Chapter 3

Implementation

3.1 Ambient Occlusion Shader

To implement ambient occlusion in the ray tracing framework, it was necessary to create a new shader which can be found in `Shaders\DiffuseAmbientOccluded.cpp`. Later we implemented the functions that were needed to add support of HDR environment maps to our solution. This chapter will explain how different methods needed to achieve it were implemented.

3.1.1 Rejection Sampling

First, we implemented a basic ambient occlusion shader which uses simple rejection sampling to generate rays to be traced from each point. The function called in `DiffuseAmbientOccluded::doRejectionSampling` can be seen in Listing 3.1 while Listing 3.2 shows the function `sampleHemisphere` responsible for the rejection sampling itself.

```
1 optix::float3 DiffuseAmbientOccluded::doRejectionSampling(const optix::Ray& r, HitInfo
2   & hit, bool emit) const
3 {
4   float3 rho_d = get_diffuse(hit);
5
6   float3 avgUnoccluded = make_float3(0,0,0);
7   int num0fUnoccluded = 0;
8
9   // define number of rays that are sampled
10  const int num0fRays = 20;
11  for(int i = 0 ; i < num0fRays ; i++)
12  {
13    Ray hemiRay = r;
14    HitInfo hemiHit;
15    // for each ray perform rejection sampling
16    if(sampleHemisphere(hemiRay, hemiHit, hit))
17    {
18      // add direction of each ray that survived rejection sampling
19      // and count them to compute the average direction
20      avgUnoccluded += hemiRay.direction;
21      num0fUnoccluded++;
22    }
23    //compute the average direction of unoccluded rays and accesability value at given
24    // point
25    float3 averageNormal = normalize(avgUnoccluded / num0fUnoccluded);
26    float accessability = ((float)num0fUnoccluded) / num0fRays;
27    // return diffuse
```

```

28     return M_1_PI * accessability * rho_d; // 1/PI = BRDF value
29 }
```

Listing 3.1: Ambient occlusion shader with rejection sampling.

The function in Listing 3.1 shoots a number of rays over the hemisphere as described in [7]. It calls the function from Listing 3.2 to sample directions on the hemisphere.

```

1 bool DiffuseAmbientOccluded::sampleHemisphere(Ray& r, HitInfo& hemiHit, HitInfo&
2   surfaceHit) const
3 {
4   float x,y,z;
5   while(true)
6   {
7     // get a random point in a unit cube around the point
8     x = randomizer.mt_random() * 2 - 1;
9     y = randomizer.mt_random() * 2 - 1;
10    z = randomizer.mt_random() * 2 - 1;
11
12    // reject those outside of a unit sphere
13    if(x*x + y*y + z*z > 1)
14      continue;
15    // reject the lower hemisphere
16    if(dot(make_float3(x,y,z), surfaceHit.shading_normal) < 0)
17      continue;
18
19    break;
20
21  // create a ray in the sampled direction
22  float3 dir = normalize(make_float3(x, y, z));
23
24  r.direction = dir;
25  r.origin = surfaceHit.position;
26  r.tmin = 0.001f;
27  r.tmax = 99999;
28
29  // return false if it hit anything
30  if (tracer -> trace_to_any(r, hemiHit))
31    return false;
32
33  return true;
34 }
```

Listing 3.2: Rejection sampling.

3.1.2 Cosine Weighted Hemisphere

In order to improve performance and result of our solution we introduced the method of sampling ray directions from a cosine weighted hemisphere instead of the rejection sampling method used in the previous subsection. This provides a much more efficient way of generating rays distributed over a hemisphere and allows us to save computations. The relevant code is shown in Listing 3.3.

```

1 bool DiffuseAmbientOccluded::sampleCosineWeightedHemisphere(Ray& ray, HitInfo& hemiHit
    , HitInfo& surfaceHit) const
2 {
3     // get an orthonormal basis at the given point
4     optix::Onb onb = optix::Onb(surfaceHit.shading_normal);
5
6     // get a random point from a unit circle around the point using polar coordinates
7     float phi, r;
8     phi = randomizer.mt_random() * M_PI / 2;
9     r = randomizer.mt_random();
10
11    // "raise" the point from the circle to the hemisphere above it
12    float x = r*cos(phi);
13    float y = r*sin(phi);
14    float z = sqrt(1-r*r);
15
16    // transform the coordinates to the onb coordinate system
17    float3 dir = x*onb.m_tangent + y*onb.m_binormal + z*onb.m_normal;
18
19    dir = normalize(dir);
20
21    ray.direction = dir;
22    ray.origin = surfaceHit.position;
23    ray.tmin = 0.001f;
24    ray.tmax = 99999;
25
26    // trace the ray in the sampled direction, return false if it hit anything
27    if (tracer -> trace_to_any(ray, hemiHit))
28        return false;
29
30    return true;
31 }

```

Listing 3.3: Cosine weighted hemisphere sampling

3.2 Environment Sampling

The next part of our project is to add sampling of light from the environment in form of HDR light probes. To do this we had to extend the framework to handle conversion from RGBE to floating point RGB as well as implement the projection of direction to texture coordinates. Then we could use them in our ambient occlusion solution.

3.2.1 HDR Image Conversion

The first part focuses on conversion of RGBE format used in HDR images to floating point RGB. This is done with the function from Listing 3.6.

```

1 float HDRTexture::convert(unsigned char c, int e)
2 {
3     return ((float)c + .5) * pow(2.0f, e - 8);
4 }

```

Listing 3.4: RGBE to RGBA conversion.

3.2.2 Spherical Texture Lookup

Furthermore we needed to implement a function which returns coordinates in a spherical texture by converting direction from which the environment should be sampled to uv coordinates. This is done in Listing 3.5.

```

1 void SphereTexture::project_direction(const float3& d, float& u, float& v) const
2 {
3     // handling denominator = 0
4     float dxdy = sqrt(d.x * d.x + d.y * d.y);
5     if (dxdy < .001f)
6     {
7         u = v = .5;
8         return;
9     }
10
11    float r = .5 * M_1_PI * acos(-d.z)/dxdy;
12
13    u = .5 + r * d.x;
14    v = .5 + r * d.y;
15 }
```

Listing 3.5: Conversion to texture coordinates

In the end we could modify our shader code to include the contribution coming from the lookup in the spherical texture as shown in Listing 3.6.

```

1 optix::float3 DiffuseAmbientOccluded::doWeightedCosineSamplingWithEnvironmentSampling(
2     const optix::Ray& r, HitInfo& hit, bool emit) const
3 {
4     float3 rho_a = get_emission(hit);
5     float3 rho_d = get_diffuse(hit);
6
7     float3 avgUnoccluded = make_float3(0,0,0);
8     int numOfUnoccluded = 0;
9
10    const int numOfRays = 200;
11    for(int i = 0 ; i < numOfRays ; i++)
12    {
13        Ray hemiRay = r;
14        HitInfo hemiHit;
15        if(sampleCosineWeightedHemisphere(hemiRay, hemiHit, hit))
16        {
17            avgUnoccluded += hemiRay.direction;
18            numOfUnoccluded++;
19        }
20    }
21
22    float3 averageNormal = normalize(avgUnoccluded / numOfUnoccluded);
23    float accessibility = ((float)numOfUnoccluded) / numOfRays;
24
25    // sample the environment in the average unoccluded direction
26    float3 hdrValue = tracer -> get_background(averageNormal);
27
28    return M_1_PI * accessibility * rho_d * hdrValue;
}
```

Listing 3.6: RGBE to RGBA conversion.

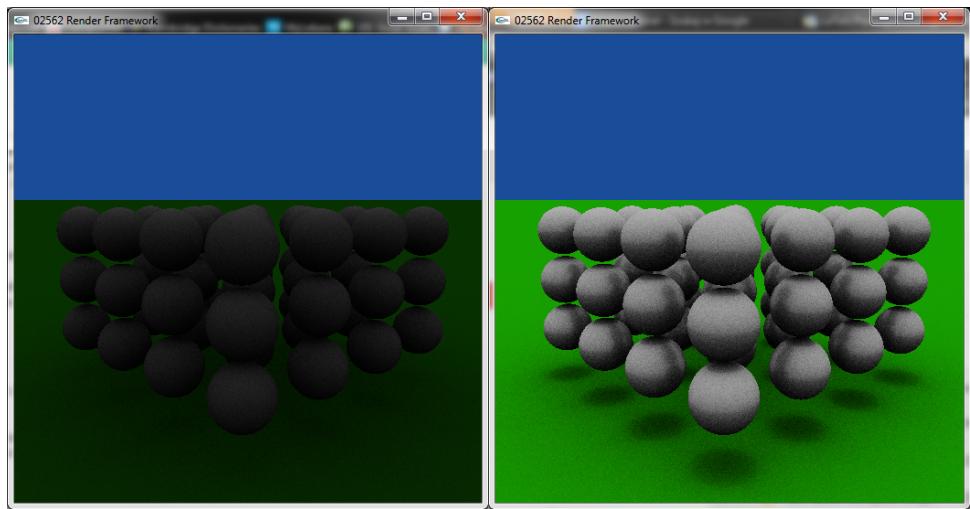
Chapter 4

Results & Discussion

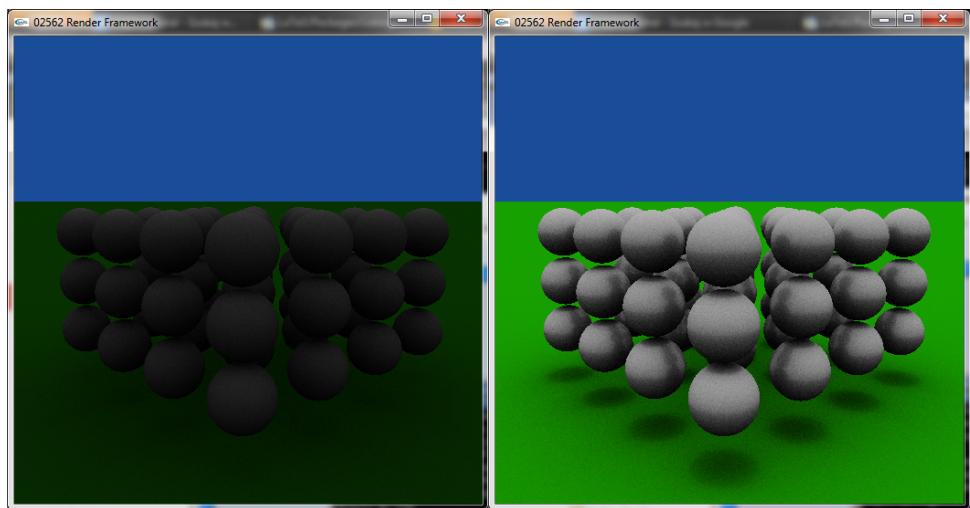
In the following chapter we will try to summarise the results we got from our ambient occlusion solution as well as compare the two implemented sampling methods and their efficiency with regard to the number of used samples.

4.1 Comparison of ambient occlusion results

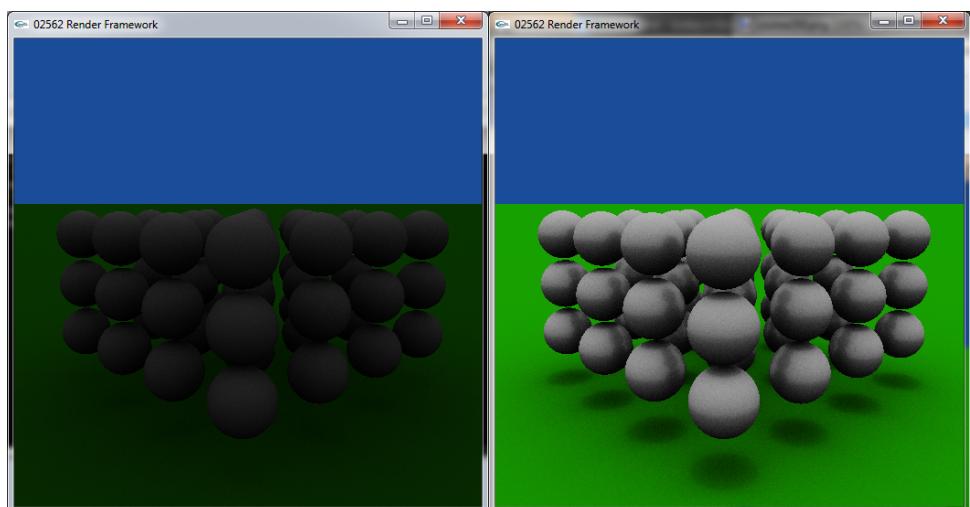
This section contains comparison of results using rejection sampling (described in subsection 3.1.1) to the use of cosine weighted hemisphere sampling (subsection 3.1.2). The renders were run with 100, 200 and 300 sample rays and the results can be seen in Figure 4.1.



(a) 100 samples, rejection sampling (b) 100 samples, cosine weighted sampling



(c) 200 samples, rejection sampling (d) 200 samples, cosine weighted sampling



(e) 300 samples, rejection sampling (f) 300 samples, cosine weighted sampling

Figure 4.1: Comparison of sampling methods if different number of samples

Chapter 5

Conclusion

We have demonstrated two different ways of performing the sampling for ambient occlusion that produced two different visual results. From the litterature it seems that both methods are acceptable so it is a matter of what visual style that fits the desired result the best. That the two different methods produce two overall different levels of brightness. A likely candidate is the lack of a modification with the probability density function in the rejection sampling method but it does not seem to be something that other papers take into consideration [7]. The lookup into the light probe image and conversion from the RGBE format to floating point RGB seems to work in a satisfactory way.

The two most obvious improvements to our method would be to change the weight of each occluded ray based on how far away the hit on the occluder are. Another improvement would be to implemnt a blurred lookup into the envirornment map.

Bibliography

- [1] P. DEBEVEC, *Light probe image gallery*, December 2012.
- [2] P. DUTRÉ, *Global Illumination Compendium*, Program of Computer Graphics, Cornell University, Sept. 2003.
- [3] O. KLEHM, T. RITSCHEL, E. EISEMANN, AND H.-P. SEIDEL, *Bent normals and cones in screen-space*, in Vision, Modeling and Visualization Workshop, 2011.
- [4] J. KONTKANEN AND S. LAINE, *Ambient occlusion fields*, in Proceedings of the 2005 symposium on Interactive 3D graphics and games, I3D '05, New York, NY, USA, 2005, ACM, pp. 41–48.
- [5] H. LANDIS, *Production-Ready Global Illumination*, in Siggraph Course Notes, vol. 16, 2002.
- [6] M. MC GUIRE, *Ambient occlusion volumes*, in Proceedings of the Conference on High Performance Graphics, HPG '10, Aire-la-Ville, Switzerland, Switzerland, 2010, Eurographics Association, pp. 47–56.
- [7] M. PHARR AND S. GREEN, *Chapter 17. ambient occlusion*, in GPU Gems, Pearson Education Inc., 2004.
- [8] P. SHANMUGAM AND O. ARIKAN, *Hardware accelerated ambient occlusion techniques on gpus*, in Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07, New York, NY, USA, 2007, ACM, pp. 73–80.
- [9] T. UMENHOFFER, B. TÓTH, AND L. SZIRMAY-KALOS, *Efficient methods for ambient lighting*, in Proceedings of the 25th Spring Conference on Computer Graphics, SCCG '09, New York, NY, USA, 2009, ACM, pp. 87–94.