Rendering assignment one

Alexander Birke
study nr: 124044 email: alex@alexanderbirke.dk

December 19, 2012

Introduction

This is my handin for the first three weeks for the course "Rendering". The last part contains the screenshots required for this handin.

Part 1 - the image loop

The task was to implement the loop over all pixels. This was simply done as a double for loop:

```
1  void RenderEngine::render()
2  {
3    cout << "Raytracing";
4    Timer timer;
5    timer.start();
6
7    #pragma omp parallel for private(randomizer)
8
9    for(int j = 0; j < static_cast<int>(res.y); ++j)
10   {
11     for(int k = 0 ; k <static_cast<int>(res.x) ; k++)
12     {
13       image[j*res.x + k] = tracer.compute_pixel(k,j);
14     }
15     if(((j + 1) % 50) == 0)
16       cerr << ".";
17   }
18   timer.stop();
19   cout << "  " << timer.get_time() << " secs " << endl;
20
21   init_texture();
22   done = true;
23 }
```

We start by writing to the console and initialising a timer which tells how long it took to render the frame. Line 7 instructs the compiller to distribute the loop iterations encountered in the next for loop. I think this is done to allow multicore support. The loop itself is just a simple double for loop that loops over all the pixels in the image where

the limits for the loop is the width and height of the image. for each pixel is the compute pixel function called with the coordinates of the current pixel. For each 50th horizontal line that has been rendered a dot is sent to the console. At last the timer is stopped and the time is shown. The texture is then shown and the done flag set to true.

Part 2 - generating camera rays

The goal of this part was to set up the cameras and generate rays from it. First we needed to set up the `set` function of the camera. This was implemented in the following way:

```
1  void set(const optix::float3& eye_point,
2           const optix::float3& view_point,
3           const optix::float3& up_vector,
4           float focal_distance)
5  {
6    eye = eye_point;
7    lookat = view_point;
8    up = up_vector;
9    focal_dist = focal_distance;
10
11 ip_normal = optix::normalize(lookat - eye);
12 ip_xaxis = optix::normalize(optix::cross(ip_normal, up));
13 ip_yaxis = optix::cross(ip_xaxis, ip_normal);
14
15 float fovInRadians = 2*atan(1.0f/2.0f*focal_dist);
16
17   fov = fovInRadians * 180 * M_1_PIf;
18
19 }
```

First we set up the camera basis. This is first done by creating the normal vector to the image plane by substracting the lookat point's coordinates from the eye point's coordinates Next the x-axis of the camera basis is created by taking the cross product between the normal vector and the up vector. The y-axis is then created by crossing the x-axis and the normal vector(z-axis). All of the resulting vectors are also normalized so they form a set of proper orthonormal basis vectors. The field of view is implemented by rearranging the equation:

$$h = 2d\tan(\frac{fov}{2})$$

to:

$$fov = 2\arctan(\frac{1}{2}d) * blah$$

Since this is in radians we need to convert it to degrees so the final equation becomes:

$$fov = 2\arctan(\frac{1}{2}d) * 180 * frac1\pi$$

The function `get_ray_dir` is implemented next:

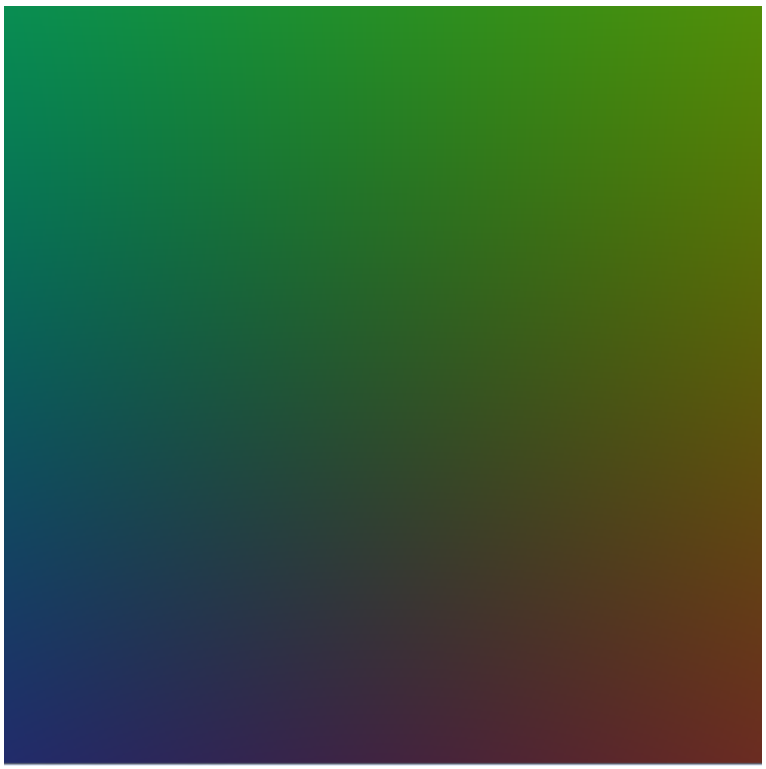
```
1  optix::float3 get_ray_dir(const optix::float2& coords) const
2  {
```

```
3       optix::float3 dir = ip_xaxis*coords.x + ip_yaxis*coords.y + ip_normal*
            focal_dist;
4
5       dir = optix::normalize(dir);
6
7       return dir;
8     }
```

This is simply a mater of transforming the normalized device coordinates into the corresponding point in the image plane world units. This is done by a change of basis. The `get_ray` is the last function in the camera to implement:

```
1   optix::Ray get_ray(const optix::float2& coords) const
2   {
3     return optix::Ray(eye, this->get_ray_dir(coords), 0, 0.1f);
4   }
```

The origin is simply the eye position, while the direction is determined by the normalized device coordinates that is given as input to the function. We then use the `compute_pixel` to get the result:



Which is what we expected.

Part 4 and 5 - Conditional Acceptance of Collision, Shading, and Hard Shadows

The task was here to implement ray-plane-, ray-triangle- as well as ray-sphere intersection and afterwards implement the surfaces as lambertian reflectors. This is done by implementing the `intersect` function in each of the three primitives. For a ray:

$$r(t) = \mathbf{o} + t\mathbf{w}$$

and plane:

$$ax + by + cz + d = 0 \Leftrightarrow \mathbf{p} * \mathbf{n} + d = 0$$

By putting the ray definition into the equation for the plane and solving for t we find at what distance along the ray we intersect the plane.

$$(\mathbf{o} + t\mathbf{w}) * \mathbf{n} + d = 0 \Leftrightarrow t = \frac{\mathbf{o} * \mathbf{n} + d}{\mathbf{w} * \mathbf{n}}$$

We then check if the found distance is between the minimum and maximum values we have now. If it is we record the hit. In the code this is done like this:

```
1  bool Plane::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
       const
2  {
3    float t = -( optix::dot(r.origin, get_normal()) + d ) / ( optix::dot(r.
       direction, get_normal()) );
4
5    if(t >= r.tmin && t <= r.tmax)
6    {
7      hit.has_hit = true;
8      hit.position = r.origin + r.direction*t;
9      hit.dist = t;
10         hit.geometric_normal = get_normal(); // (the normalized normal of
               the plane)
11         hit.shading_normal = get_normal();   //   (the normalized normal of
                the plane)
12         hit.material = &material;             //      (pointer to the
               material of the plane)
13
14     return true;
15   }
16   else
17   {
18      return false;
19   }
20 }
```

The sphere is implemented in a similar way:

```
1  bool Sphere::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
       const
2  {
3    float bDiv2 = optix::dot( (r.origin - center) , r.direction );
4    float c = optix::dot( (r.origin - center), (r.origin - center));
5    c += - radius*radius;
6
7    if(bDiv2*bDiv2 - c >= 0)
8    {
9      //std::cout << "in plane " << hit.has_hit << "\n";
10     float t = - bDiv2 - sqrt(bDiv2*bDiv2 - c);
11
12     if(t > r.tmin && t < r.tmax)
```

```
13      {
14        hit.has_hit = true;
15        hit.position = r.origin + r.direction*t;
16        hit.geometric_normal = optix::normalize(hit.position - center); // (
              the normalized normal of the plane)
17        hit.shading_normal = hit.geometric_normal;    //    (the normalized
              normal of the plane)
18        hit.material = &material;                //       (pointer to the
              material of the plane)
19
20        return true;
21      }
22    }
23    return false;
24  }
```

Here we insert the ray equation into the equation for the sphere:

$$(p-c)*(p-c) = r^2 \Leftrightarrow t = (\mathbf{o}-\mathbf{c}+t\mathbf{w})*(\mathbf{o}-\mathbf{c}+t\mathbf{w}) = r^2$$

This yields a second degree polynomial with the a,b and c coeficients equal to:

$$a = \mathbf{w}*\mathbf{w} = 1b/2 = (\mathbf{o}-\mathbf{c}*\mathbf{w}c = (\mathbf{o}-\mathbf{c})*(\mathbf{o}-\mathbf{c}) - r^2$$

This is solved by finding the first root:

$$t_1 = -b/2 - \sqrt[2]{(b/2)^2 - c}$$

Of course if the determinant $(b/2)^2$ is less than zero then there is no intersection. Due to time constraints I have not implemented the ray-triangle intersection myself but have used the one in the optix library.

Triangle intersection is based on finding the baryocentric coordinates of the hit between the triangle and the ray. If we dfine the triangle to have the sides:

$$e_1 = V_1 - V_0$$

$$e_2 = V_2 - V_0$$

We can use it together with point V0 to create an affine combination that spans the plane of the triangle with the baryocentric coordinates (u,v). A formula for a point on the plane of the triangle is therefore:

$$T(u,v) = V_0 + ue_1 + ve_2$$
$$T(u,v) = V_0 + u(V_1 - V_0) + v(V_2 - V_0)$$

This can be rearanged to:

$$T(u,v) = (1-u-v)V_0 + uV_1 + vV_2$$

Since we are only interested in points inside the triangle we have that:

$$u \geq 0$$

$$v \geq 0$$

$$u + v \leq 1$$

To compute the uv coordiantes we set the line equation of the ray equal to the equation of the plane in baryocentric coordinates:

$$P = td = (1 - u - v)V_0 + uV_1 + vV_2$$

by rearanging the equation we can express it as a matrix product:

$$\begin{bmatrix} -d & V_2 - V_0 & V_1 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = P - V_0$$

By using either Cramer's rule or row reduction we can find that:

$$t = \frac{q * e_2}{p * e1}$$

$$u = \frac{p * s}{p * e_1}$$

$$v = \frac{q * d}{p * e_1}$$

Where:

$$s = P - V_0 \quad p = d \times e_2 \quad q = s \times e_1$$

Below is my implementation of this approach which also checks if the ray and triangle are parallel before it proceeds to calculate the intersection.

```
1  bool intersect_triangle(const Ray& ray,
2                          const float3& v0,
3                          const float3& v1,
4                          const float3& v2,
5                          float3& n,
6                          float& t,
7                          float& v,
8                          float& w)
9  {
10   //find sides of triangle
11   float3 e1 = v1 - v0;
12   float3 e2 = v2 - v0;
13
14   //find perpendicular vector between ray direction and one of the sides
15   float3 p = optix::cross(ray.direction, e2);
16
17   //take the dot product between this vector and the other side of the
         triangle
18   float a = dot(e1, p);
19
20   //if a is equal to zero then the ray is parallel with the triangle and no
         intersection ocurs
21   if(a == 0 || (a < 0.001f && a > -0.001f))
22   {
```

```
23      return false;
24    }
25    //compute denominator
26    float f = 1.0f/a;
27
28    //compute barycentric coordinates and check if they are within the
          accepted boundaries
29    float3 s = ray.origin - v0;
30    v = f * dot(s,p);
31    if(v < 0.0f || v > 1.0f)
32      return false;
33
34    float3 q = cross(s, e1);
35    w = f * dot(ray.direction, q);
36    if(w < 0.0f || w + v > 1.0f)
37      return false;
38
39    t = f * dot(e2,q);
40
41    //calculate normal
42    n = cross(e1, e2);
43
44    return true;
45  }
```

A last note, in the Accelerator's render function the max distance of the ray is set to the t value returned from a successful hit. That way we make sure that closer hits overrides hits farther away. This means we in the end shades the pixel according to the closest hit.

Point light implementation & Lambertian shading

The next part is to implement Kepler's inverse square law of radiation that governs how light from a point light spreads. Since I have also implemented a simple version of shadow rays the function look like this:

```
1   bool PointLight::sample(const float3& pos, float3& dir, float3& L) const
2   {
3     if(!shadows)
4     {
5       dir = optix::normalize(light_pos - pos);
6
7       L = intensity / ( optix::length(light_pos - pos) * optix::length(
            light_pos - pos) ) ;
8
9       return true;
10    }
11    else
12    {
13      optix::float3 shadowDir = optix::normalize( light_pos - pos );
14      HitInfo hit;
15
16      optix::Ray shadowRay = Ray(pos, shadowDir, 0, 0.0001f);
17
18      if(!tracer->trace_to_any(shadowRay, hit))
```

```
19      {
20
21        dir = optix::normalize(light_pos - pos);
22
23        L = intensity / ( optix::length(light_pos - pos) * optix::length(
              light_pos - pos) ) ;
24
25        return true;
26      }
27      else
28      {
29        L = make_float3(0,0,0);
30
31        return false;
32      }
33    }
34  }
```

First we check if we should use shadows. If no then we just compute the radiance and return true. If we should use shadows we create a shadow ray and see if it intersects any objects on its way from the surface to the light source. The first part of this is simply to declare variables used to check for shadows. First we find the direction of the shadow ray by finding the vector from the location on the surface to the light source and normalizing it. A HitInfo variable is then declared so we can call the `tracer`'s `trace_to_any` function. This function is used since we only wants to know if the ray hits something. This ray is declared with the direction we made earlier and the position equal to the location on the surface. If the function return false then the light ray hits the surface and we compute the radiance based on the inverse square law and return true. If not then then we set the returned radiance to 0 and return false. When the radiance is calculated we also return the direction of the light ray which we need in order to evaluate Lambert's cosine law.

The last step is to shade the surface according to Lambert's cosine law. This is done in the `Lambertian` shader's `shade` function:

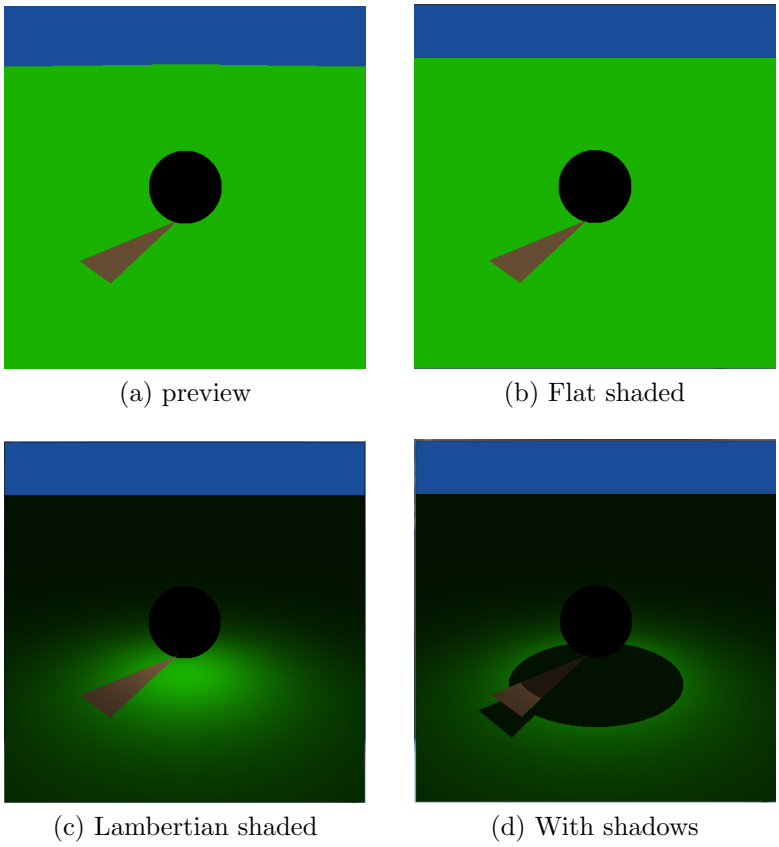
```
1  float3 Lambertian::shade(const Ray& r, HitInfo& hit, bool emit) const
2  {
3    float3 rho_d = get_diffuse(hit);
4    float3 result = make_float3(0);
5
6    // Implement Lambertian reflection here.
7    //
8    // Input:  r          (the ray that hit the material)
9    //         hit        (info about the ray-surface intersection)
10   //         emit       (passed on to Emission::shade)
11   //
12   // Return: radiance reflected to where the ray was coming from
13   //
14   // Relevant data fields that are available (see Lambertian.h, HitInfo.h,
         and above):
15   // lights             (vector of pointers to the lights in the scene)
16   // hit.position       (position where the ray hit the material)
17   // hit.shading_normal (surface normal where the ray hit the material)
18   // rho_d              (difuse reflectance of the material)
```

```
19    //
20    // Hint: Call the sample function associated with each light in the scene
          .
21
22    float3 lambertianBRDF = rho_d * M_1_PIf;
23
24    for(int i = 0 ; i < lights.size() ; i++)
25    {
26      float3 Li;
27      float3 dir;
28
29      if(lights[i]->sample(hit.position, dir, Li))
30      {
31        result += lambertianBRDF * Li * optix::dot(dir, hit.shading_normal);
32      }
33    }
34
35    return result + Emission::shade(r, hit, emit);
36  }
```

We loop over all the lights in the scene and sample each of them. If the point being shaded is hit by the light we add the diffuse reflectance divided by pi, times the radiance returned by the light times the dot product between the direction of the light and the ray.

Part 6 - deliverables

Here is the finished results:



(a) preview (b) Flat shaded

(c) Lambertian shaded (d) With shadows

Figure 1: The different renderings

In the console we can see the rendering time goes up for each extra feature added: