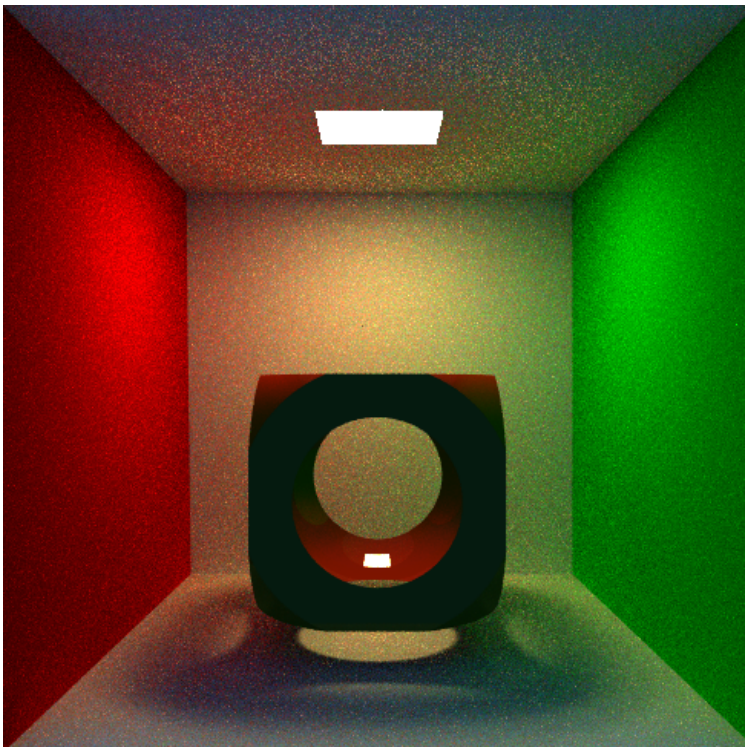
Signed Distance Field and Constructive Solid Geometry Raytracer

Lab Journal for " 02562 Rendering - Introduction E20"

Tim Kaiser s201792



20.12.2020

Contents

1 Project

1.1 Introduction

Computer Graphics has come far in recent years, inching ever closer to being indistinguishable from reality. This however, is in the vast majority of applications not based on a simulation of actual physics but rather an approximation of it in form of a rasterization pipeline. And while this is fast and easy to support via hardware, it does require a whole lot of tricks and workarounds to give realistic results. One limitation of such a rasterization based approach is that it can only render flat surfaces. Thanks to the enormous compute power modern machines offer we can approximate curved surfaces using millions of tiny polygons, but the fact remains that we cannot render actual curves that way. Ray tracer however do not have these limitations. There exist many mathematical and algorithmic approaches to representing and visualizing three-dimensional, curved surfaces. One of them are Signed Distance Fields (SDF). A SDF is basically a function that returns the distance to the closest surface. This gives an implicit representation of surface.
In this project I extend the raytracer implemented in the "Rendering - Introduction" course at DTU, the develoment of wich is documented in section 2, to include the ability of rendering SDFs.

1.2 Signed Distance Fields

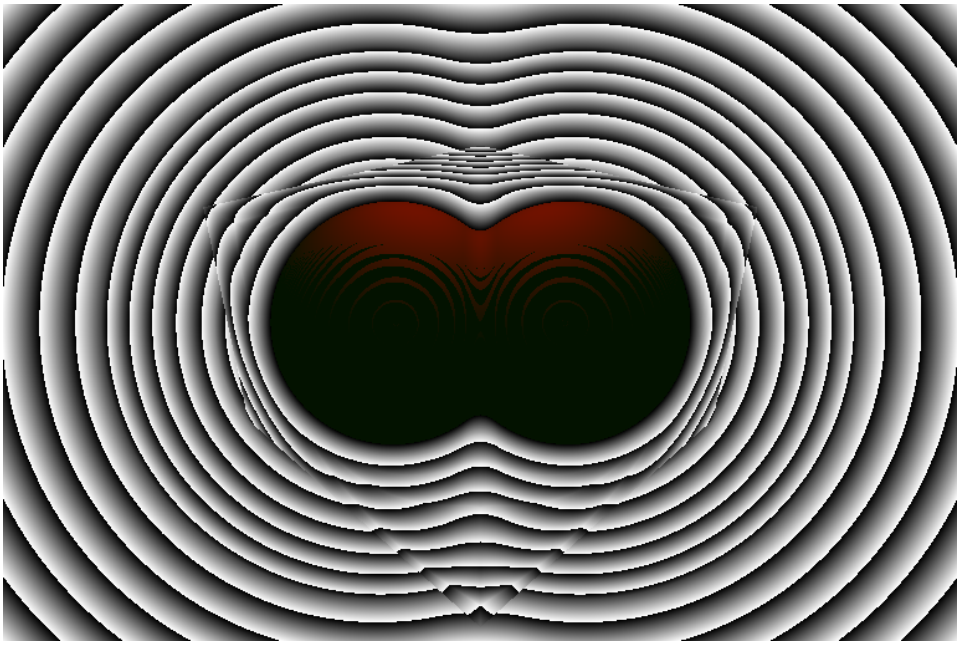


Figure 1: A visualization of the Signed Distance Field around a simple shape. The black and white lines are created by taking the modulo of the closets distance between the ray and the object.

Signed Distance Fields are an implicit way of representing surfaces. This means instead of explicitly describing where a surface lies or giving a set of points on that surface we have a function f that returns a value for every point in space. In case of distant fields, that value describes the distance to the nearest surface. And obviously surfaces lie where that value is zero. It is a signed distance field, because inside objects the value is negative. This can be useful for example for calculation the normals[1] . Normals are calculated by taking the derivative of the function f. The easiest way for us to do that is by taking a tiny step ε along each axis and calculate the distance. This gives us the normal vector. The formula looks like that[2]:

$$\nabla f = (\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}) \tag{1}$$

$$\vec{n} = [f(x+\varepsilon, y, z) - f(x, y, z), f(x, y+\varepsilon, z) - f(x, y, z), f(x, y, z+\varepsilon) - f(x, y, z)] \tag{2}$$

In code that looks like this:

```
//SDF.cpp
float3 SDF::calcNormal(const float3& pos) const {
   const float h = 1e-4; //epsilon
   float dist = distance(pos);  //f(x,y,z) := distance(float3 position)
   float3 normal = make_float3(
         distance(make_float3(pos.x + h, pos.y, pos.z)) - dist,
         distance(make_float3(pos.x, pos.y + h, pos.z)) - dist,
         distance(make_float3(pos.x, pos.y, pos.z + h)) - dist
      );
   return normalize(normal);
}
```

[1] Normal rendering mode can be turned on by pressing "n" before raytracing. SDF visualization mode can be turned on with "v"

[2] http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions

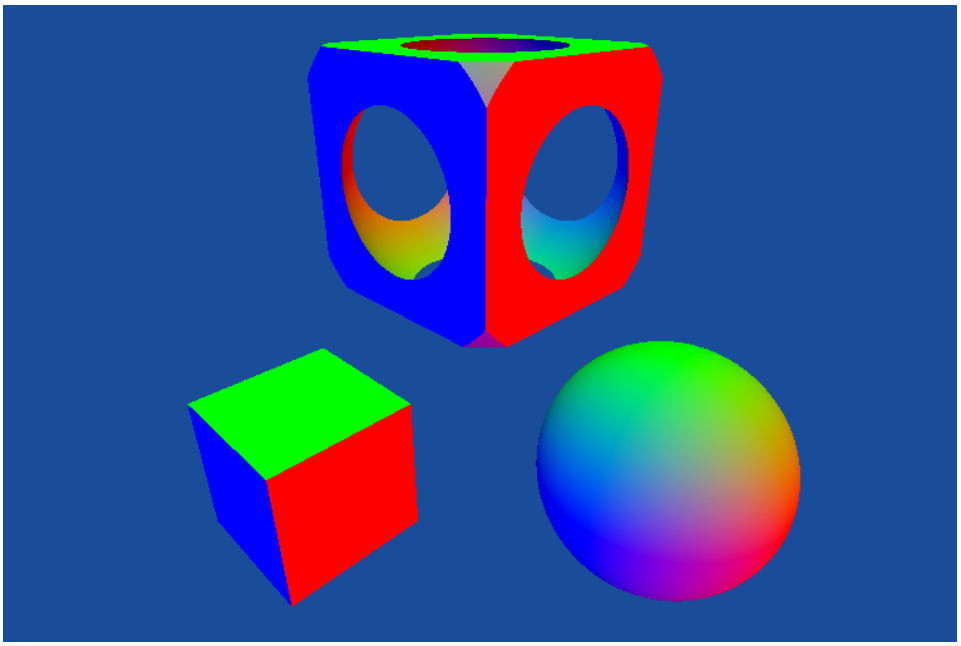


Figure 2: Some examples of normals calculated in the SDF.

1.3 Ray Marching

Since SDFs are an implicit representation of surfaces we cannot easily calculate the intersection of the surface and the ray analytically like we did before. Instead we have to do an approach called ray marching. The general principle is quiet simple. We know at every point in space how far the closest surface is. This means that we know how far we can at least go in ray direction without hitting an object. So we take the distance at the starting point, take one step of that distance in direction of the ray, take the distance again, take another step and repeat until we find a surface. There is only one problem with that approach. In 99.9% of all cases we will never reach a surface, because the surface is at distance 0 and we only slowly approach the surface, never actually reaching zero. The problem is sketched in figure 3. So we have to define a threshold value. Once the distance is below the threshold we register it as a hit. This, however leads to a tiny error in the distance measured. The error is always below the threshold. The smaller the threshold the smaller the error, but the larger the computational cost. A visualization of the error can be seen in figure 4.

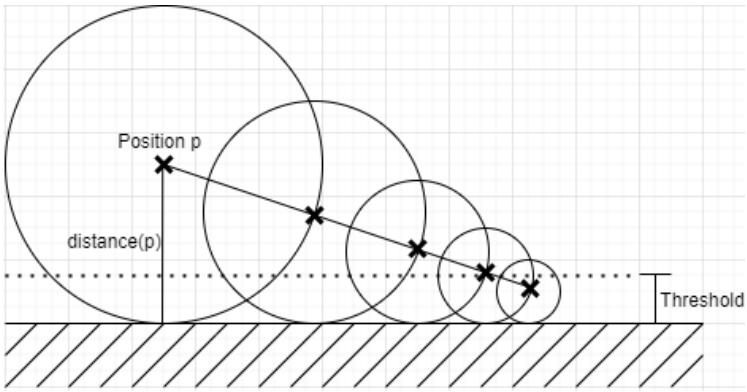


Figure 3: Sketch of the ray marching process.

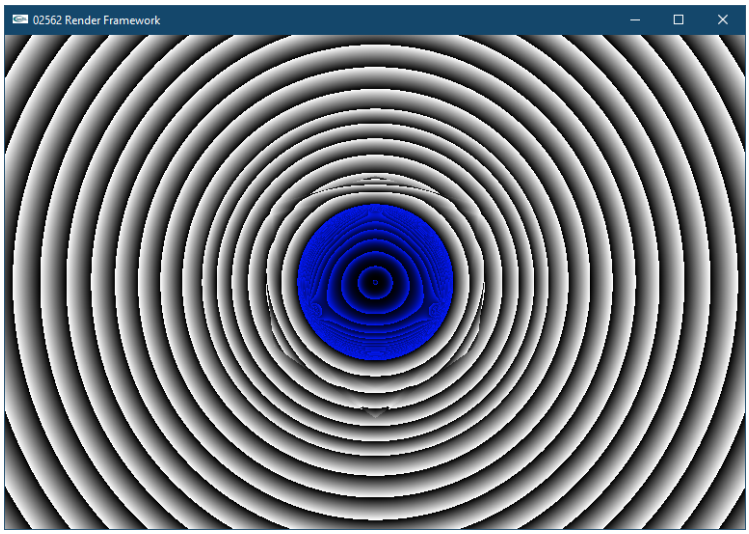


Figure 4: A visualization of the error in the Ray Marching algorithm caused by a large threshold value. The brighter the blue on the object is the further away the calculated point from the actual surface. The lines outside the object are a visualization of how close the ray got to the object.

The code for the algorithm is fairly simple in principle, though it required some fine-tuning to minimize the error. In the end the only real solution was to decrease the threshold at the expense of rendering time.

```
//SDF.cpp
bool SDF::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
    //How close we have to get to the surface until it counts as a hit
   float threshold = 0.00001;
   //After what distance we give up looking for surfaces
   float maxDist = 50;
```

```
    //normalizing the direction is important
    float3 dir = normalize(r.direction);
    //making sure that we take the minimum distance into account. important
        for reflecting
    float3 pos = r.origin + dir * r.tmin;

     //abs(), because we could be inside the object
    float dist = abs(distance(pos));
    //to measure the distance to the object
    float distTraveled = 0;

    //only needed for sdf visualization
    hit.closest_dist = dist;

    //Ray Marching
    while (distTraveled < maxDist && dist > threshold) {
       pos = pos + dir * dist;
       distTraveled += dist;
       dist = abs(distance(pos));
       hit.closest_dist = fminf(hit.closest_dist, dist);
    }

    //no hit
    if (dist > threshold) {
       return false;
    }

    //trying to minimize the error. It worked. Kind of.
    pos = pos + dir * dist;

    //register hit
    hit.closest_dist = abs(distance(pos));
    hit.has_hit = true;
    hit.position = pos;
    float3 normal = calcNormal(pos);
    hit.dist = distTraveled;
    hit.geometric_normal = normal;
    hit.shading_normal = normal;
    hit.material = &material;

    return true;
}
```

```
//distance function of the SDF. sdfObjects is a list of all objects in the
    sdf
float SDF::distance(const float3& pos) const {
   float dist = 100000;
   for each (SDFObject* obj in sdfObjects) {
      float distObj = obj->distance(pos);
      //we cannot just use min() here since the distance can be negative
      dist = abs(distObj) < abs(dist) ? distObj : dist;
   }
   return dist;
}
```

1.4 Constructive Solid Geometry

1.4.1 CSG

The power of a SDF tracer strongly depends on what distance functions we can define. Mathematically defining surfaces can be very complex and unintuitive. This makes 3d modelling for SDFs rather challenging. An easier and more accessible approach is to define simple shapes and combine them to more complex ones. This is exactly the idea behind Constructive Solid Geometry (CSG) Trees. In this tree, here a binary tree, though other varients are theoretically possible, every node except leaf nodes consists of two child nodes and a operator that combines them. Leaf nodes are simple geometric shapes, which are explained in section 1.4.2. Operators are described in detail in section 1.4.3. An example of a CSG Tree can be seen in figure 5.

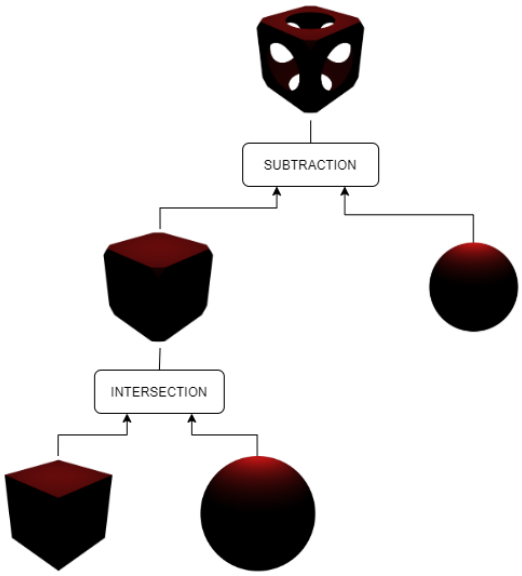


Figure 5: A simple example of a CSG Tree using the INTERSECTION and SUBTRACTION operators.

The distance function of a CSG Tree is rather simple. You simply take call the distance function of the root node which then recursively calls the distance functions of its child nodes and blends them together using the blend function which implements the the blend operators. Some of the blendfunctions include a smoothing parameter which leads to nicer curves than just the ordinary parameter. The code for that looks like this:

```
//SDFCSGTree.cpp
//distance function of a CSG node
float SDFCSGTree::distance(const float3& pos) const{
   return blend(blendfunction, left->distance(pos), right->distance(pos),
       blendparameter);
}
```

1.4.2 Basic Shapes

For creating the CSG Tree we first need to define our leaf nodes. This means we need structure for a none-leaf node and some basic shapes. This sections gives a quick overview over all shapes. Every one of them inherits from SDFObject and can therefore easily be plugged into the SDF as seperate object.

Sphere

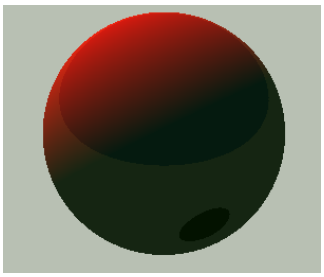


Figure 6: A simple sphere rendered without complex shading

The probably easiest to define shape is the sphere. It only needs a center position and a radius as input. The distance function is simple as well:

```
//SDFSphere.cpp
float SDFSphere::distance(const float3& pos) const{
   return length(pos - center) - radius;
}
```

Figure 6 shows a simple rendering of a sphere using the SDF distance function.

Box

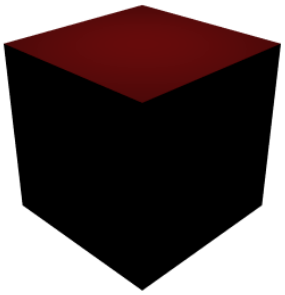


Figure 7: A simple box rendered without complex shading

A bit more complex but still rather simple is the box. It is define by a center vector and a bounds vector. Doing it this way allows it to calculate the distance using vector math and gets rid of costly comparisons. The distance function looks like this[3]:

```
//SDFBox.cpp
float SDFBox::distance(const float3& pos) const {
    float3 q = make_float3(
        fmaxf(0, abs(pos.x - center.x) - bounds.x),
        fmaxf(0, abs(pos.y - center.y) - bounds.y),
        fmaxf(0, abs(pos.z - center.z) - bounds.z));

    return length(q) + min(max(q.x, max(q.y, q.z)), 0.0);
}
```

Figure 8 shows a simple rendering of a box.

Cylinder

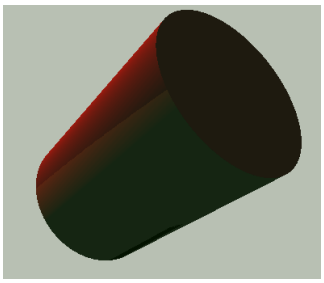


Figure 8: A simple cylinder rendered without complex shading

And finally, probably the most complex of the basic shapes implemented in this project, the cylinder. The cylinder is defined by three parameters. The center point that defines its

[3] https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

position, the axis vector that defines both its orientation and height and the radius. Its distance function is a bit more complex [4]:

```
//SDFCylinder.cpp
float SDFCylinder::distance(const float3& pos) const {
    float3 ba = axis - center;
    float3 pa = pos - center;
    float baba = dot(ba, ba);
    float paba = dot(pa, ba);
    float x = length(pa * baba - ba * paba) - radius * baba;
    float y = abs(paba - baba * 0.5) - baba * 0.5;
    float x2 = x * x;
    float y2 = y * y * baba;
    float d = (max(x, y) < 0.0) ? -min(x2, y2) : (((x > 0.0) ? x2 : 0.0) +
        ((y > 0.0) ? y2 : 0.0));
    return d<0? -1 : d==0 ? 0 : 1 * sqrt(abs(d)) / baba;
}
```

Figure 8 shows a simple rendering of a cylinder.

1.4.3 Blendfunctions

To combine these simple shapes into more complex ones we have to define blend functions. In total there are there available in this project. union, intersection and subtraction. All of these functions are implemented according as they are described by Inigo Quilez [5].

Union

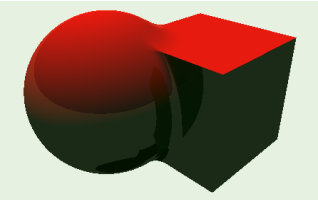


Figure 9: Union between a sphere and a box with a smoothing factor of 0.1

The union operator is probably the simplest function. It just takes the minimum of both distances, effectively just combining both shapes. It gets however a bit more difficult once we include the smoothing parameter.

```
//Blending.cpp
float smoothUnion(float a, float b, float k) {
   float h = fmaxf(k - abs(a - b), 0.0);
   return fminf(a, b) - h * h * 0.25 / k;
```

[4] https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
[5] https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

```
}
```

Figure 9 shows an example.

Intersection

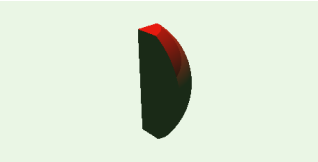


Figure 10: Intersection between a sphere and a box

The intersection operator is essentially taking the maximum of both distances. Only when the ray intersects both shapes, meaning the distance to both is smaller than the threshold, a hit is registered. Again, the smoothing makes things a bit more difficult:

```
//Blending.cpp
float smoothIntersection(float a, float b, float k) {
   float h = fmaxf(k - abs(a - b), 0.0);
   return fmaxf(a, b) + h * h * 0.25 / k;
}
```

Figure 10 shows an example.

Subtraction

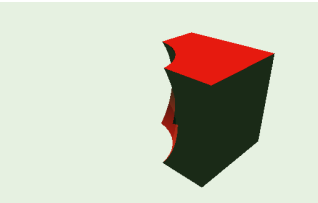


Figure 11: A sphere subtracted from a box

And lastly the subtraction operator. This is the only one where the order of the input parameters matters. It is an intersection between one shape and the inverse of the other shape. With smoothing the code looks like this:

```
//Blending.cpp
float smoothSubtraction(float a, float b, float k) {
   float h = fmaxf(k - abs(a + b), 0.0);
   return fmaxf(a, -b) + h * h * 0.25 / k;
}
```

Figure 11 shows an example.

1.5 Changes in the Framework and new Code

The actual code changes from the framework to the project are mostly limited to four places, most of which are new classes. There are minor changes to RenderEngine.cpp and RayCaster.cpp to initialize the SDF, a new class SDF.cpp/SDF.h that inherits from Object3D and handles the intersection tests, SDFObjects.cpp/SDFObjects.h and the classes that inherit from it (SDFBox, SDFShpere, SDFCylinder, SDFCSGTree) and Blending.cpp/Blending.h, which handles the blending functions for the CSG tree.

1.5.1 RenderEngine and Rest of Framework

RenderEngine.cpp is where all the new classes are integrated into the existing framework. New objects are added to the scene here. In *RenderEngine::load_files(...)* the default scene has been overwritten to create new SDF objects. After their creation, objects can be added by calling *Scene::add_SDF(std::vector<SDFObject*>, string, int)* on the scene object in RenderEngine.cpp. Loading the Cornell Box using the commandline parameter also automatically adds a SDF object.
RenderEngine::keyboard(...) has also been modified to allow two new rendermodes. When pressing 'v' SDF visualition mode is enabled. In this mode the renderer draws SDF distance line instead of the background and visualizes the error caused by the threshold instead of shading the object. 'n' activates the normals rendering mode in which every object is rendered with normals instead of their usual shading. Pressing either key returns the renderer to shading mode. The rendermode is passed on as a additional parameter to the raycaster through a modified *RayCaster::compute_pixel(...)*.
Changes to *Glossy::shade(...)* where necessary since the raytracer unfortunately has problems with refracted rays in SDFs. Problems that I did not manage to fix.

1.5.2 SDF

SDF.h/SDF.cpp implement the actual important parts of the SDF. As a class inheriting from Object3D it also implements *transform(...)*, *compute_bbox(...)*, *get_material(...)* and *intersect(...)*. *transform* and *compute_bbox* just pass on the command to all objects contained in *vector<SDFObject*> sdfObjects*. *Intersect* has already been explained in detail in section 1.3. SDF.cpp contains two more methods. *distance(...)* calculates the distance function of the SDF by taking the minimum of all distance functions from objects in *sdfObjects* and *calcNormal(...)* calculates the normal at that position.

1.5.3 SDFObjects

The class SDFObject and all its children are mostly datsstructures to contain the position, dimension and distance function for objects in the SDF. Each subclass describes a specific shape, all of which have been explained in section 1.4.2. SDFCSGTree is special, in that

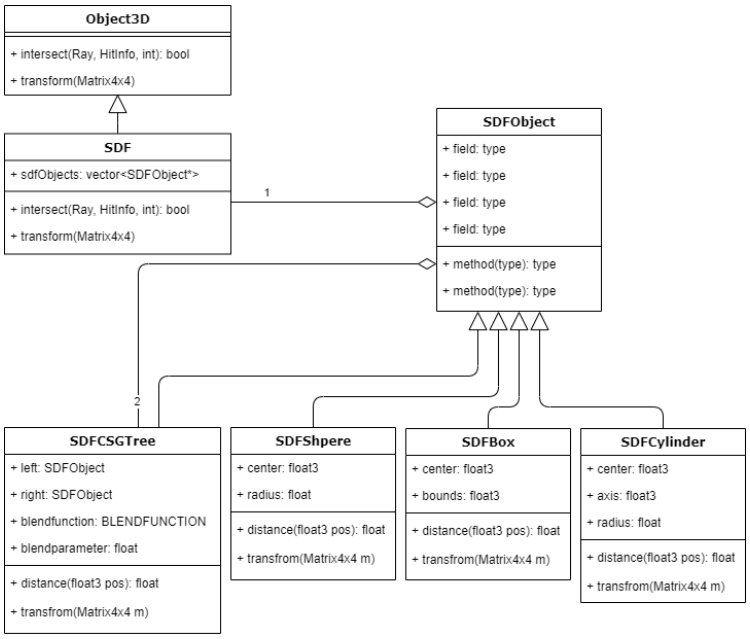


Figure 12: A class diagram for the newly implemented classes.

it isn't a shape in itself but rather combines two shapes into one (or more if used nested). That too has already been explaine in section 1.4. All SDFObjects also contain the functions *transform(...)* and *compute_bbox(...)* which are called from the SDF. While compute_bbox works as intended, transform has not been tested. Transforming the position should work, but both rotation and scaling are not properly implemented. Figure 12 gives an overview over the class relations of the newly implemented code.

1.5.4 Blending

All the blending functions used in the CSG Tree have been outsourced to Blending.cpp. Almost all functions in this file have already been described in section 1.4.3. Only *blend(float a, float b, Blendfunction f, float s)* is missing. It simply uses a switch case to decide which blend function to call. The datatype 'Blendfunction' is an enum defined in Blending.h.

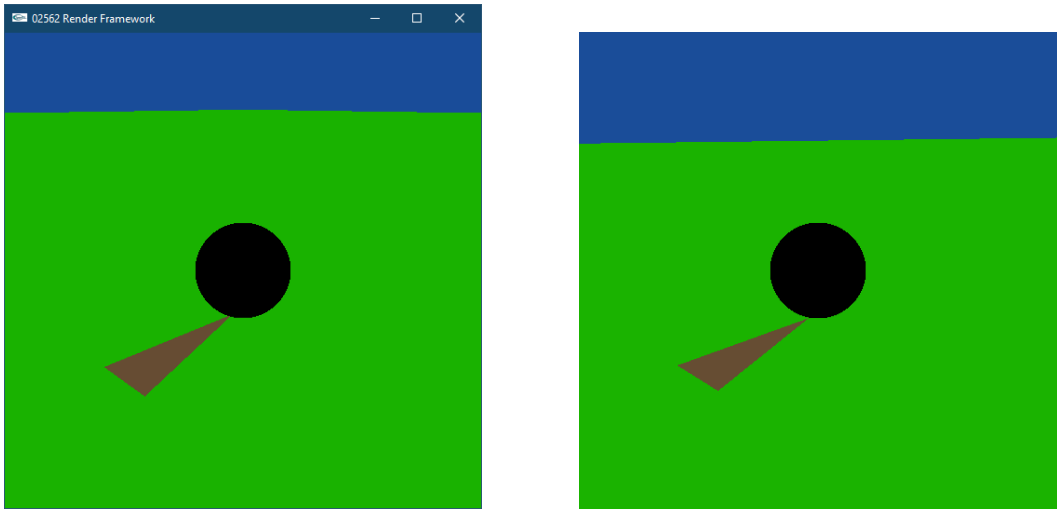
1.6 Conclusion

Overall I am satisfied with the results. The CSG Tree gives nice, complex structures and visualizations of the SDF can be quite beautiful. It is an interesting approach to modelling and rendering 3d objects, considering that there direct representation of the surface. It would be interesting for future projects to maybe store the distaces along a grind in an array and only interpolate between them during runtime. This could speed up the algorthim because there is no need to calculate the distance to every object and could maybe even be hardwae accerlerated, since interpolation is a strength of graphics hardware. In general the implementation in this project could use some optimization to speed up the rendering.

2 Lab Journal

Disclaimer: Most of the worksheets were solved with the help of and in cooperation with Nynne Kajs (s193156).

2.1 Worksheet 1



(a) Preview (b) Ray Tracer

Figure 13: The default scene in (a) preview mode and (b) rendered with a very basic ray tracer. The images are nearly identical as expected.

In this the very first worksheet, we extend the existing framework by implementing the very basic steps of a raytracer. this starts by implmenting the render methode:

```
//RenderEngine.cpp
void RenderEngine::render()
{
  cout << "Raytracing";
  Timer timer;
  timer.start();
  #pragma omp parallel for private(randomizer)
  for(int y = 0; y < static_cast<int>(res.y); ++y)
  {
     for (int x = 0; x < static_cast<int>(res.x); ++x)
     {
        image[x + y * res.x] = tracer.compute_pixel(x, y);
     }
     if(((y + 1) % 50) == 0)
      cerr << ".";
  }
  timer.stop();
```

```
  cout << " - " << timer.get_time() << " secs " << endl;

  init_texture();
  done = true;
}
```

Next, we complete the implementation of the camera:

```
//Camerea.cpp
/// Get direction of viewing ray from image coords.
float3 Camera::get_ray_dir(const float2& coords) const
{
   return normalize(ip_xaxis * coords.x + ip_yaxis * coords.y + ip_normal *
       cam_const);
}

/// Return the ray corresponding to a set of image coords
Ray Camera::get_ray(const float2& coords) const
{
    return Ray(eye, get_ray_dir(coords), 0, 0);
}
```

Then we can implement the function that renders a single pixel:

```
//RayCaster.cpp
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y) const
{
    float2 ip_coords = make_float2(x, y) * win_to_ip + lower_left;

   Camera cam = *scene->get_camera();
   HitInfo* hitInfo = new HitInfo();
   Ray ray = cam.get_ray(make_float2(x, y));
   scene->closest_hit(ray,*hitInfo);

    if (hitInfo->has_hit) {
      return get_shader(*hitInfo)->shade(ray,*hitInfo);
   }
   else {
      return get_background();
   }
}
```

We continue with the Accelerator:

```
//Accelerator.cpp
bool Accelerator::closest_hit(optix::Ray& r, HitInfo& hit) const
{
```

```
   closest_plane(r, hit);

   for(unsigned int i = 0; i < primitives.size(); i++)
   {
      HitInfo* hitTemp = new HitInfo();

      AccObj* obj = primitives[i];
      obj->geometry->intersect(r, hit, obj->prim_idx);
      if (hitTemp->dist < hit.dist) {
         //hit = *hitTemp;
      }
   }
   return hit.has_hit;
}

bool Accelerator::any_hit(optix::Ray& r, HitInfo& hit) const
{
  if(!any_plane(r, hit))
  {
    unsigned int i = 0;
    while(i < primitives.size() && !hit.has_hit)
    {
      AccObj* obj = primitives[i++];
      obj->geometry->intersect(r, hit, obj->prim_idx);
    }
  }
  return hit.has_hit;
}

void Accelerator::closest_plane(Ray& r, HitInfo& hit) const
{
  for(unsigned int i = 0; i < planes.size(); ++i)
    if(planes[i]->intersect(r, hit, 0))
      r.tmax = hit.dist;
}

bool Accelerator::any_plane(Ray& r, HitInfo& hit) const
{
  for(unsigned int i = 0; i < planes.size(); ++i)
    if(planes[i]->intersect(r, hit, 0))
      return true;
  return false;
}
```

No we need to implement a few more very basic and very important functions. The intersect functions for spheres, planes and triangles. Ray intersection tests are the fundamentals of raytracing. The implementation looks like this:

```
//Plane.cpp
bool Plane::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
    //test if plane and ray are parallel
   float wn = dot(r.direction, get_normal());
   if ( wn == 0.0000000f) {
      return false;
   }

   //check if intersection is btw min and max
   float dist = - (dot(r.origin, get_normal()) + d) / wn;
   if (dist < r.tmin || dist > r.tmax) { return false; }

   //set hit parameter
   hit.has_hit = true;
   hit.dist = dist;
   hit.geometric_normal = get_normal();
   hit.shading_normal = get_normal();
   hit.position = r.origin + dist * r.direction;
   hit.material = &material;

   return true;
}
```

```
//Sphere.cpp
bool Sphere::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
   float3 oc = r.origin - center;
   float bhalf = dot(oc, r.direction);
   float c = dot(oc, oc) - radius * radius;
   float b2c = bhalf * bhalf - c;
   if (b2c < 0) { return false; }

   float dist = -bhalf - sqrt(b2c);
   if (dist < r.tmin || dist > r.tmax) {
      dist = -bhalf + sqrt(b2c);
      if (dist < r.tmin || dist > r.tmax) {
         return false;
      }
```

```
    }

    hit.has_hit = true;
    hit.position = r.origin + dist * r.direction;
    float3 normal = normalize(hit.position - center);
    hit.dist = dist;
    hit.geometric_normal = normal;
    hit.shading_normal = normal;
    hit.material = &material;

    return true;
}
```

```
//Triangle.cpp
bool Triangle::intersect(const Ray& r, HitInfo& hit, unsigned int
    prim_idx) const
{
    float dist, v, w;
    if (!optix::intersect_triangle(r, v0, v1, v2, normal, dist, v, w)) {
        return false; }

    //set hit parameter
    hit.has_hit = true;
    hit.dist = dist;
    hit.geometric_normal = normalize(normal);
    hit.shading_normal = hit.geometric_normal;
    hit.position = r.origin + dist * r.direction;
    hit.material = &material;

    return true;
```

With the geometry in place we now have to implement a simple light:

```
//PointLight.cpp
bool PointLight::sample(const float3& pos, float3& dir, float3& L) const
{
    dir = light_pos - pos;
    float dist = length(dir);
    dir = normalize(dir);

    //lighting
    L = intensity / pow(dist, 2);

    return true;
}
```

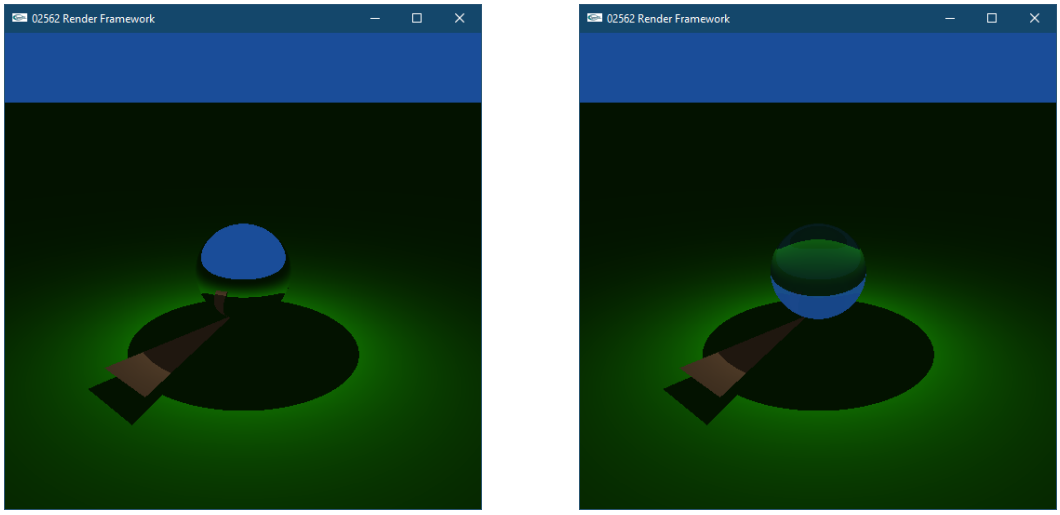
We also need a shade function:

```
//Lambertian.cpp
float3 Lambertian::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  const float3 rho_d = get_diffuse(hit);
  float3 result = make_float3(0.0f,0.0f,0.0f);

  float3 dir, L;
  for (Light* light : lights) {
      float3 resultL = make_float3(0);
     for (int i = 0; i < light->get_no_of_samples(); i++) {
        if (light->sample(hit.position, dir, L)) {
           resultL += L * fmax(0.0f, dot(hit.shading_normal, dir));
        }
     }

     result += resultL / light->get_no_of_samples();
  }


  result = rho_d / M_PIf * result;

  return result + Emission::shade(r, hit, emit);
}
```

The result of all of this can bee seen in figure 29

2.2 Worksheet 2

2.2.1 Ray Tracing



(a) Mirror (b) Transparent

Figure 14: The default scene rendered with shadows and (a) a completely reflection sphere or (b) a 90% refracting sphere.

In this exercise we implement a few basic ray tracing effects, Those are shadows, reflection and refraction. The shadows are implemented in the light the following way:

```
//PointLight.cpp
bool PointLight::sample(const float3& pos, float3& dir, float3& L) const
{
    dir = light_pos - pos;
   float dist = length(dir);
   dir = normalize(dir);

   //shadows
   if (shadows) {
      Ray shadowray = Ray(pos, dir, 0, 0.0001f , dist);
      HitInfo hit;
      tracer->trace_to_any(shadowray, hit);
      if (hit.has_hit) { return false; }
   }

   //lighting
   L = intensity / pow(dist, 2);

   return true;
}
```

Reflection and Refraction are taken care of here:

```
//RayTracer.spp
bool RayTracer::trace_reflected(const Ray& in, const HitInfo& in_hit, Ray&
    out, HitInfo& out_hit) const
{
    out = Ray(in_hit.position, optix::reflect(in.direction,
        in_hit.shading_normal), 0, 0.0001, RT_DEFAULT_MAX);
    out_hit.ray_ior = in_hit.ray_ior;
    out_hit.trace_depth = in_hit.trace_depth + 1;
    return this->trace_to_closest(out, out_hit);
}

bool RayTracer::trace_refracted(const Ray& in, const HitInfo& in_hit, Ray&
    out, HitInfo& out_hit) const
{
   float3 normal = in_hit.shading_normal;
   float ior = get_ior_out(in, in_hit, normal);
   float3 refDir;
   if (optix::refract(refDir, in.direction, normal, ior / in_hit.ray_ior)) {
      out = Ray(in_hit.position, refDir, 0, 0.0001, RT_DEFAULT_MAX);
      out_hit.ray_ior = ior;
      out_hit.trace_depth = in_hit.trace_depth + 1;
      return trace_to_closest(out, out_hit);
    }else{
      out = Ray(in_hit.position, refDir, 0, 0.0001, RT_DEFAULT_MAX);
        return false;
    }

}
```

Those functions are applied in the shaders:

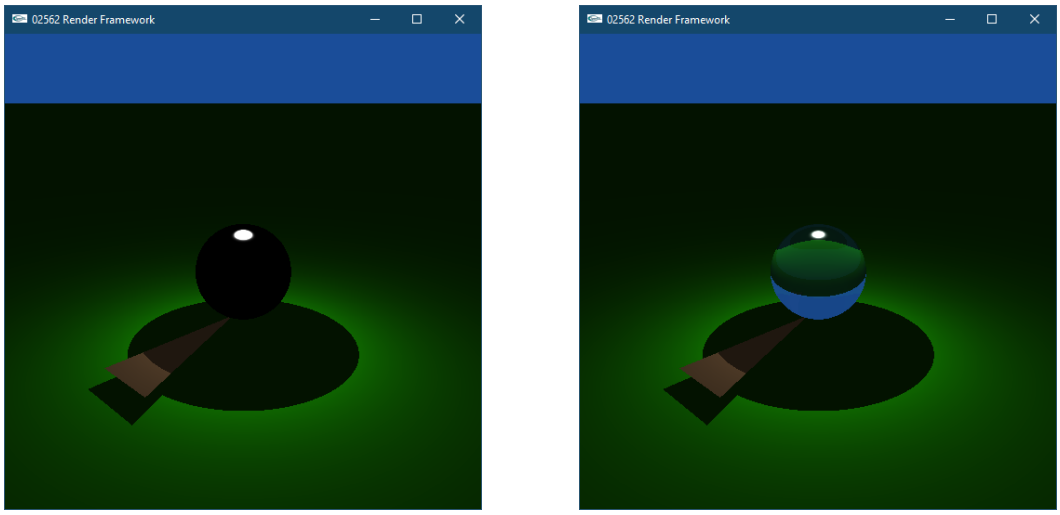
```
//Mirror.cpp
float3 Mirror::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  if(hit.trace_depth >= max_depth)
    return make_float3(0.0f);

  Ray reflected;
  HitInfo reflected_hit;
  tracer->trace_reflected(r, hit, reflected, reflected_hit);
  return shade_new_ray(reflected, reflected_hit);
}
```

```
//Transparent.cpp
```

```
float3 Transparent::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  if(hit.trace_depth >= max_depth)
    return make_float3(0.0f);

  float R;
  Ray reflected, refracted;
  HitInfo hit_reflected, hit_refracted;
  tracer->trace_reflected(r, hit, reflected, hit_reflected);
  tracer->trace_refracted(r, hit, refracted, hit_refracted, R);
  return R*shade_new_ray(reflected, hit_reflected) + (1.0f -
      R)*shade_new_ray(refracted, hit_refracted);
}
```

get_ior returns the index of refraction for the material.

2.2.2 Phong Reflection



(a) Phong (b) Glossy

Figure 15: The default scene rendered with (a) only Phong highlights and (b) a combination of reflection, refraction and highlights.

So far our scene can only render either reflection or refraction. To solve this we create a shader that combines both of these and also includes specular Phong highlights.

The Phong highlights can be seen in figure 15a. The code for it is below.

```
//Phong.cpp
float3 Phong::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  float3 rho_d = get_diffuse(hit);
```

```
  float3 rho_s = get_specular(hit);
  float s = get_shininess(hit);

  float3 result = make_float3(0);
  for(Light* light : lights){
     float3 Lr = make_float3(0.0f);
     float num_samples = light->get_no_of_samples();
     for (int i = 0; i < num_samples; i++) {
        float3 light_dir, L;
        if (!light->sample(hit.position, light_dir, L)) continue;

        float3 n = hit.shading_normal;
        float3 wi = light_dir;
        float3 wr = optix::reflect(-wi, n);//2 * dot(wi, n) * n - wi;
        float3 wo = -normalize(r.direction);
        Lr += (rho_d * M_1_PIf + rho_s * (s + 2) * (0.5f * M_1_PIf) *
            pow(fmax(0,dot(wo, wr)), s)) * L * fmax(0,dot(wi, n));
     }
     result += (Lr/num_samples);
  }
  return result + Emission::shade(r, hit, emit);
}
```

The combined shader can be seen in figure 15b. The code for it is below.

```
//Glossy.cpp
float3 Glossy::shade(const Ray& r, HitInfo& hit, bool emit) const
{
    float R;
    Ray reflected, refracted;
    HitInfo hit_reflected, hit_refracted;
    tracer->trace_reflected(r, hit, reflected, hit_reflected);
    tracer->trace_refracted(r, hit, refracted, hit_refracted, R);
    float3 result = R * (Phong::shade(r, hit, emit) +
        shade_new_ray(reflected, hit_reflected)) + (1.0f - R) *
        shade_new_ray(refracted, hit_refracted);
    return result;

}
```

2.3 Worksheet 3

In this worksheet we add textures to the objects. The framework loads the texture specified in the corresponding .mtl file. When the raytracer detects an intersection with an object it interpolates the texture coordinates at that position and samples the texture at the specified

coordinates. It takes either the color of nearest pixel or it interpolates linear between adjacent pixels, depending on the settings. The color is then used to determine the color of the reflected light. The code for calculating the texture (uv) coordinates of the plane is as follows:

```
//Plane.cpp
bool Plane::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
    [...]
   if (material.has_texture) {
      float u, v;
      get_uv(hit.position, u, v);
      hit.texcoord = make_float3(u, v, 0); //TODO: change
   }

   return true;
}

void Plane::get_uv(const float3& hit_pos, float& u, float& v) const
{
   float3 xx0 = hit_pos - position;
   u = dot(onb.m_tangent, xx0) * tex_scale;
   v = dot(onb.m_binormal, xx0) * tex_scale;
}
```

The result looks like this:

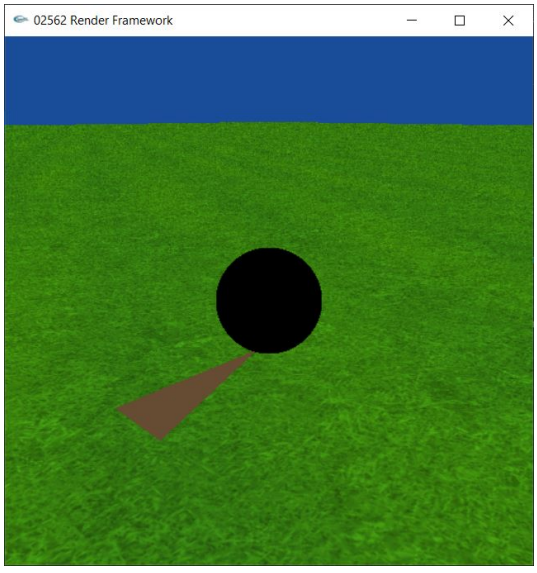
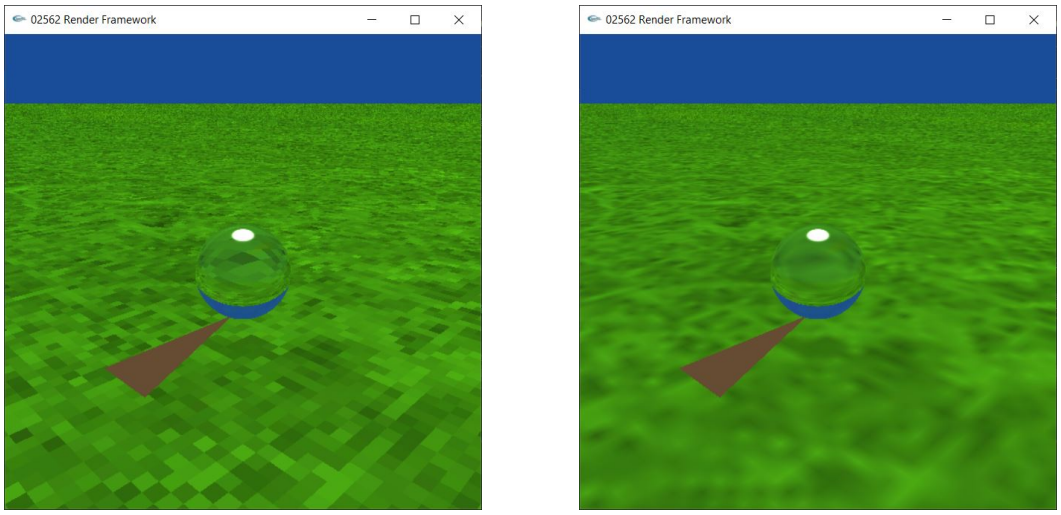


Figure 16: The default scene with a texture on the plane in preview mode.

As already explained, the texture can either be sampled using nearest neighbor or linear interpolation. A comparison is show in figure 17. The image is scaled there in order to make the difference more visible. Scaling magniies the pixel in the texture by spacing out the uv coordinates.



(a) Nearest Neighbor (b) Linear Interpolation

Figure 17: The default scene rendered with a scaled up texture on the plane. (a) uses nearest neighbor sampling, (b) linear interpolation-

Lastly we added a texture to the sphere. For this we had to implement uv coordinates for the sphere first.

```
//Shpere.cpp
bool Sphere::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
    [...]
   if (material.has_texture) {
      hit.texcoord = make_float3(hit.shading_normal.x,
          hit.shading_normal.y, hit.shading_normal.z);
   }
   return true;
}
```

```
//InvSphereMap.cpp
void InvSphereMap::project_direction(const float3& d, float& u, float& v)
    const
{
    u = acos(d.z) / (M_PIf);
   v = atan2(d.y, d.x) / (2 * M_PIf);
```

```
}
```

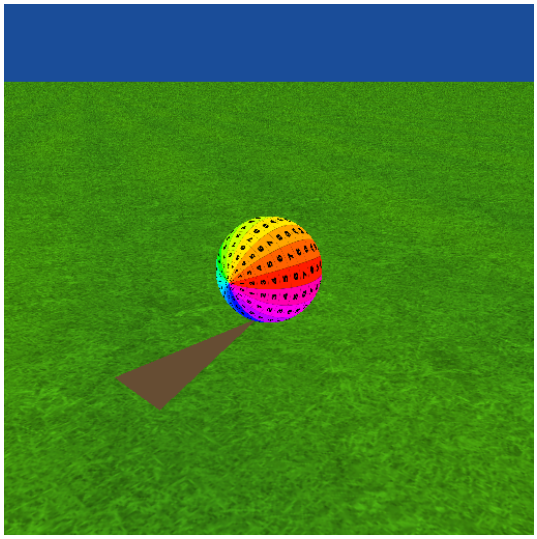
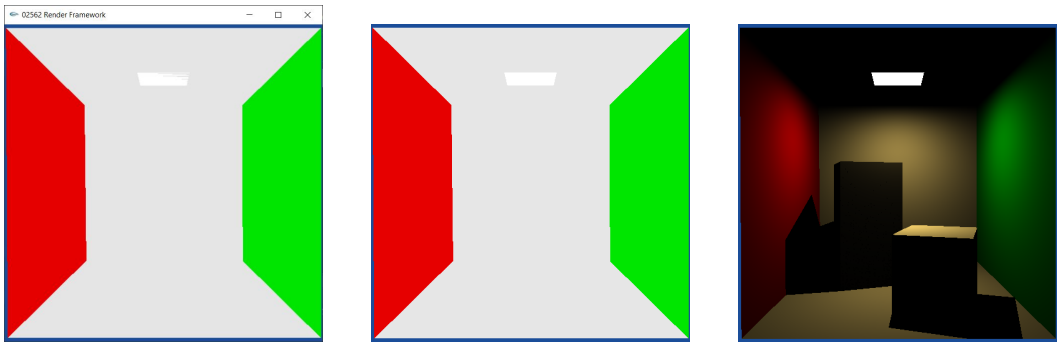


Figure 18: The default scene with textures on the plane and the sphere.

2.4 Worksheet 4

2.4.1 Triangle Meshes



(a) Preview (b) Raytraced (c) Raytraced and shaded

Figure 19: The CornellBox (a) in preview, (b) raytraced and (c) raytraced and shaded.

In this exercise we had to expand the raytracer to be able to handle triangle meshes. First we had to implement the ray-mesh intersection test. The code below already performs interpolation on the surface normal to create a smooth appearance.

```
//TriMesh.cpp
bool TriMesh::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx)
    const
{
   const uint3& face = geometry.face(prim_idx);
```

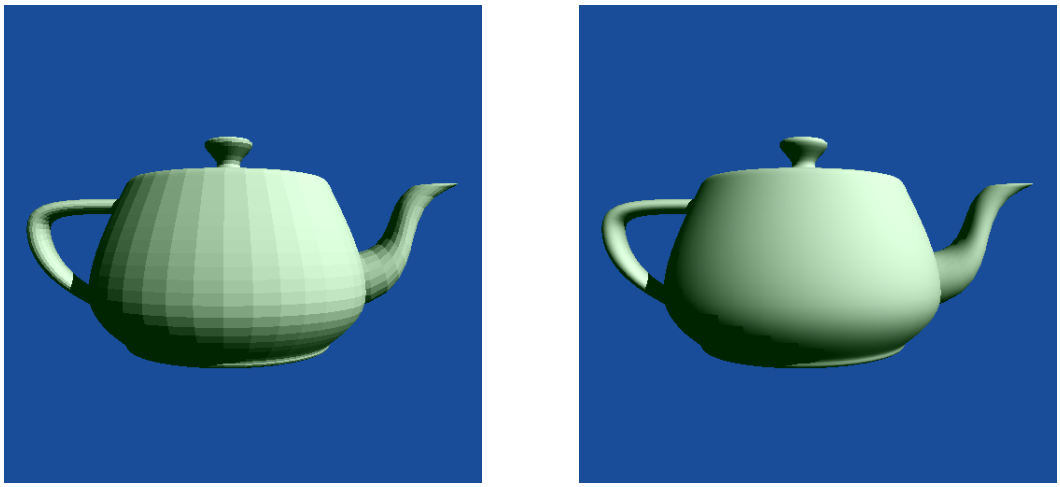
```
    float3 normal, n;
    float t, beta, gamma;

    if (!optix::intersect_triangle(r, geometry.vertex(face.x),
        geometry.vertex(face.y), geometry.vertex(face.z), n, t, beta, gamma)
        || t < r.tmin || t > r.tmax) { return false; }

    //set hit parameter
    hit.has_hit = true;
    hit.dist = t;

    hit.geometric_normal = normalize(n);
    hit.shading_normal = has_normals() ? normalize((1 - beta - gamma) *
        normals.vertex(face.x) + beta * normals.vertex(face.y) + gamma *
        normals.vertex(face.z)) : normals.vertex(face.x);

    hit.position = r.origin + t * r.direction;
    hit.material = &materials[mat_idx[prim_idx]];

    return true;
}
```

We also implemented an area light and a directional light.

```
//AreaLight.cpp
bool AreaLight::sample(const float3& pos, float3& dir, float3& L) const
{
  const IndexedFaceSet& normals = mesh->normals;
  L = make_float3(0.0f);
  float3 intensity = make_float3(0);
  for (int i = 0; i < mesh->get_no_of_primitives(); i++) {
    //intensity += get_emission(i) / mesh->face_areas[i];
    uint3 face = normals.face(i);
    float3 normal = normalize(normals.vertex(face.x) +
        normals.vertex(face.y) + normals.vertex(face.z));
    intensity += dot(-dir, normal) * get_emission(i) * mesh->face_areas[i];
  }

  dir = (mesh->compute_bbox().center() - make_float3(0,0.001f,0)) - pos;
  float dist = length(dir);
  dir = normalize(dir);

  //shadows
  if (shadows) {
    Ray shadowray = Ray(pos, dir, 0, 0.001f, dist);
```

```
    HitInfo hit;
    tracer->trace_to_any(shadowray, hit);
    if (hit.has_hit) { return false; }
  }

  //lighting
  L = intensity / pow(dist, 2);

  return true;
}
```

```
//Directional.cpp
bool Directional::sample(const float3& pos, float3& dir, float3& L) const
{//shadows
   if (shadows) {
      Ray shadowray = Ray(pos, -light_dir, 0, 0.001f);
      HitInfo hit;
      tracer->trace_to_any(shadowray, hit);
      if (hit.has_hit) { return false; }
   }

   //lighting
   L = emission;
    dir = -light_dir;

   return true;
}
```



(a) (b)

Figure 20: The Utah Teapot rendered using (a) Flat Shading and (b) Phong Shading. *(Note: The lighting appears green due to a mistake in the light initialization that was later fixed)*

2.4.2 Space Subdivison

In this exercise we enable a BSP Tree spatial subdivision to speed up the rendering process. This works by dividing the space in smaller subspaces along the axis (at least in this case). Each subspace can be divided even further. When doing intersection test the raytracer first checks if the ray crosses the subspace an object is in. If not then the intersection test doesn't even have to be performed, which gives us a hugh speed boost. The render times can be seen in the table below

Scene	#Triangle	Time without BSP Tree	Time with BSP Tree	Speed-Up factor
Cornell Box	36	3.911s	1.951s	x2
Utah Teapot	6320	57.081s	1.353s	x44
Stanford Bunny	69451	827.943s	0.837s	x1000

Table 1: *Note: the speedup factor for the bunny seems extrem, but I performed the test multiple times with similar results*

In general it's pretty clear that the greater the number of triangles, the greater the benefit from the speedup. This is as expected.

Figure 21: The Stanford Bunny rendered using a BSP Tree.

The code for the BSP Tree intersection is below.

```
bool BspTree::closest_hit(Ray& r, HitInfo& hit) const
{
    intersect_min_max(r);
    HitInfo hitTmp;
    closest_plane(r, hitTmp);

    intersect_node(r, hit, *root);

    hit = !hitTmp.has_hit || (hit.has_hit && hit.dist < hitTmp.dist) ? hit
        : hitTmp;

    return hit.has_hit;
}

bool BspTree::any_hit(Ray& r, HitInfo& hit) const
{
    intersect_min_max(r);
    return any_plane(r,hit) || intersect_node(r, hit, *root);
}
```

2.5 Worksheet 5

In this worksheet we had to do a series of calculations.

1)

$$\Phi = 25W * 20\% = 5W = 5J/s$$

$$E_{photon} = hc\lambda = hc/(500 * 10^{-9}m) = 6.626[Js] * 299,792,458[m/s] * 10^{-27}[1/m] = 1.98642483e-18[J]$$

$$N_{photons} = 5[J/s]/1.98642483e-18[J] = 2.51708493e18[1/s] = 2.5 * 10^{18}\frac{1}{s}$$

2) Assumed efficiency: 100%
Radiant Flux:

$$\Phi = 100\% * 2.4V * 0.7A = 1.68W$$

Radiant Intensity:

$$I = \Phi/\Omega = 1.68/4\pi[W/sr] = 0.13W/sr$$

Radiant exitance:

$$M = W/A = 1.68W/((1cm/2)^2 * \pi) = 1.68/(2.5e5 * \pi)[W/m^2] = 2.1e4W/m^2$$

Energy over 5 minutues:

$$\Phi * 5 * 60s = 1.68W * 300s = 504J$$

3) Assumption: the eye looks directly at the light: light_dir $\perp$ eye

$$w = dA/r^2 = ((6/2e-3m)^2)/1m^2 = (9e-6) * \pi = 28e-6$$

$$E = I * w = 0.13[W/sr] * 28e-6[sr] = 3.64W$$

4) Assumption: the table is facing towards the light: light_dir $\perp$ table

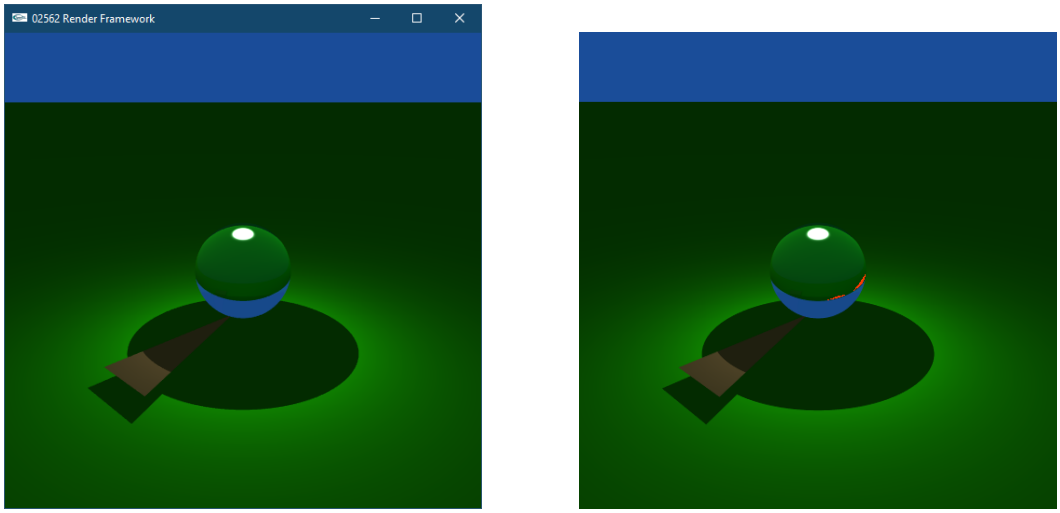
$$\Phi = 200W * 20\% = 40W$$

$$E = \Phi/4\pi * cos\Theta/r^2 = \Phi/4\pi^2 = 40/16\pi[W/m^2] = 2.5W/m^2$$

Sadly I didn't manage to finish this exercise.

2.6 Worksheet 6

2.6.1 Anti-Aliasing



(a) 9 Samples per pixel

(b) 16 samples per pixel

Figure 22: The default scene with anti-aliasing. (a) has 9 jitter sample per pixel, (b) has 16.

In this worksheet we had to implement anti-aliasing for the raytracer. This is done by rendering multiple rays per pixel, each with a slightly different starting position chosen at random. The subdivision level in thi program is changed with the '+' and '-' keys. This de-/increases the parameter subdivs and then calls the function compute_jitters(). The number of jttered samples per pixel is subdivs*subdivs The function resizes the vector jitters to the number of samples and creates random offset vectors for each sample. Each offset stays within one pixel.

```
//RayCaster.cpp
void RayCaster::increment_pixel_subdivs()
{
  ++subdivs;
  compute_jitters();
  cout << "Rays per pixel: " << subdivs*subdivs << endl;
}

void RayCaster::compute_jitters()
{
  float aspect = width/static_cast<float>(height);
  win_to_ip.x = win_to_ip.y = 1.0f/static_cast<float>(height);
  lower_left = (win_to_ip - make_float2(aspect, 1.0f))*0.5f;
  step = win_to_ip/static_cast<float>(subdivs);

  jitter.resize(subdivs*subdivs);
```

```
  for(unsigned int i = 0; i < subdivs; ++i)
    for(unsigned int j = 0; j < subdivs; ++j)
      jitter[i*subdivs + j] = make_float2(safe_mt_random() + j,
          safe_mt_random() + i)*step - win_to_ip*0.5f;
}
```

The compute_pixel() function in RayCaster has to be modified to make use of this:

```
//RayCaster.cpp
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y) const
{
    float2 ip_coords = make_float2(x, y) * win_to_ip + lower_left;

   Camera cam = *scene->get_camera();

   float3 result = make_float3(0);
   for (float2 jit : jitter) {
      HitInfo* hitInfo = new HitInfo();
      Ray ray = cam.get_ray(ip_coords+jit);
      scene->closest_hit(ray, *hitInfo);
      if (hitInfo->has_hit) {
         if (x == 150 && y == 450) {
            int a = 5;
         }
         result += get_shader(*hitInfo)->shade(ray, *hitInfo);
      }
      else {
         result += get_background();
      }
   }

   return result / (subdivs *subdivs);

}
```

The rendered results can be seen in figure 22. Increasing the number of samples also increases the render times. The render times for both the default scene and the Cornell Box at different subdivision levels can be seen here:

Rays per Pixel	Default	Cornell Box
1	0.299s	2.712s
4	0.468s	6.493s
9	1.666s	9.688s
16	2.744	

The table shows that the render time increases roughly linear with the number of pixels per ray, which makes sense. The anti-aliasing error should go down the higher the subdivision level is.

2.6.2 Progressive Path Tracing

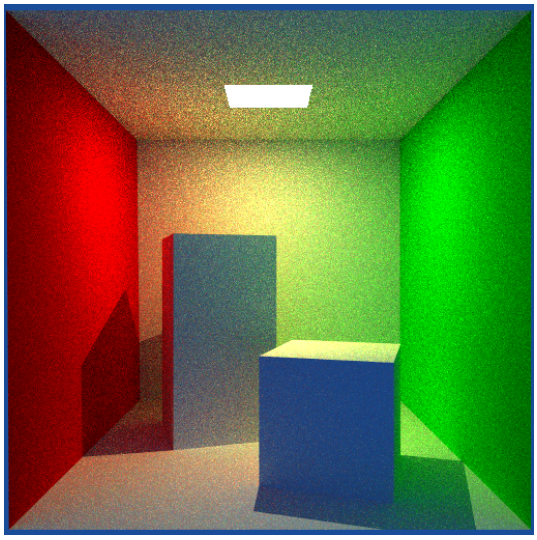


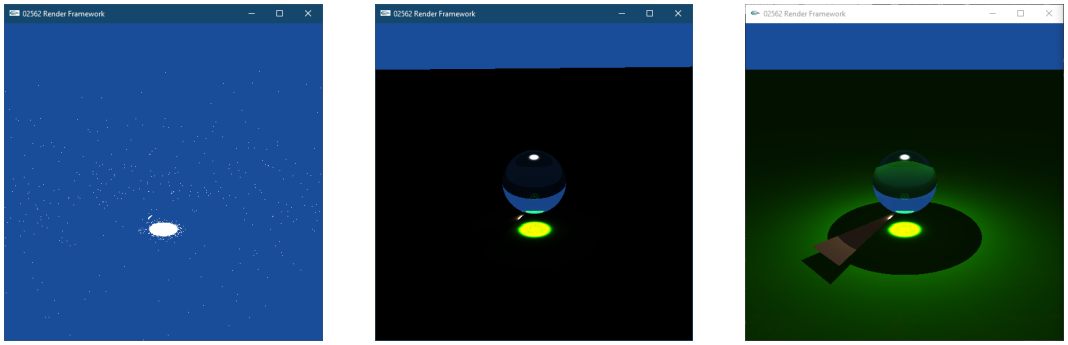
Figure 23: The Cornell Box rendered using Progressive Path Tracing and indirect illumination.

In this part of the exercise we had to complete the progressive path tracer, the results of which can be seen in figure 23. This path tracer slowly increases the number of samples per pixel over time. This is great, because it delivers very fast approximate results. The code is the following:

```
//sampler.h
inline optix::float3 sample_cosine_weighted(const optix::float3& normal)
{
   // Get random numbers
   float xi1 = mt_random_half_open();
   float xi2 = mt_random_half_open();

   // Calculate new direction as if the z-axis were the normal
   float theta = acos(sqrt(1 - xi1));
   float phi = 2 * M_PIf * xi2;

   optix::float3 wj = optix::make_float3(cos(phi) * sin(theta), sin(phi) *
       sin(theta), cos(theta));

   // Rotate from z-axis to actual normal and return
   rotate_to_normal(normal, wj);
```

```
    return wj;
}
```

2.7 Worksheet 7



(a) Photonmap (b) Only caustics (c) Shaded

Figure 24: Default scene rendered with photon mapping. (a) Shows the photon map, containing 20004 photons, (b) is a render of only the caustics, (c) shows the final result rendered with 1 sample pr pixel.

In worksheet 7 we had to integrate photon mapping into the project. First we had to calculate the photon flux phi for the point light:

```
//PointLight.cpp
bool PointLight::emit(Ray& r, HitInfo& hit, float3& Phi) const
{
    float3 direction;
    do
    {
        direction.x = 2.0f * mt_random() - 1.0f;
        direction.y = 2.0f * mt_random() - 1.0f;
        direction.z = 2.0f * mt_random() - 1.0f;
    } while (dot(direction, direction) > 1.0f);
    direction = normalize(direction);

    r = Ray(light_pos, direction, 0, 1e-4, RT_DEFAULT_MAX);
    if (!tracer->trace_to_closest(r, hit)) { return false; }

    Phi = intensity * 4 * M_PIf;

    return true;
}
```

Next, we had to call that function in the particle tracer. We also had to handle reflection and refraction for different surfaces and store the photons in the photon map:

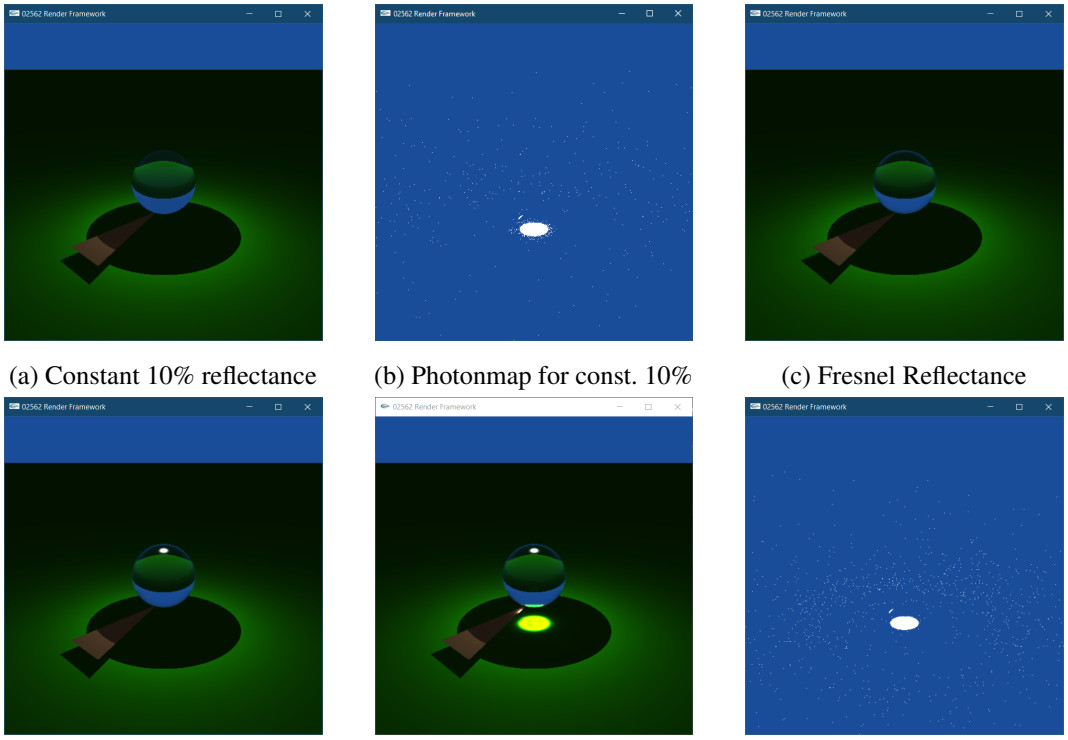
```
//ParticleTracer.cpp
void ParticleTracer::trace_particle(const Light* light, const unsigned int
    caustics_done)
{
  if(caustics_done)
    return;

  // Shoot a particle from the sampled source
  Ray r;
  HitInfo hit;
  float3 phi;

  light->emit(r, hit, phi);

  if (!hit.has_hit || !scene->is_specular(hit.material)) {
     return;
  }
  // Forward from all specular surfaces
  while(scene->is_specular(hit.material) && hit.trace_depth < 500)
  {
    switch(hit.material->illum)
    {
    case 3: // mirror materials
      {
        // Forward from mirror surfaces here
      Ray r_out;
      HitInfo hit_out;
      if (!trace_reflected(r, hit, r_out, hit_out)) {
         return;
      }

      r = r_out;
      hit = hit_out;
      }
      break;
    case 11: // absorbing volume
    case 12: // absorbing glossy volume
      {
        // Handle absorption here (Worksheet 8)
      }
    case 2: // glossy materials
    case 4: // transparent materials
     {
```

```
      // Forward from mirror surfaces here
      Ray r_out;
      HitInfo hit_out;
      float p;
      trace_refracted(r, hit, r_out, hit_out, p);
      if(mt_random()<=p){
         hit_out.has_hit = false;
         trace_reflected(r, hit, r_out, hit_out);
      }
      if (!hit_out.has_hit) {
         return;
      }
      r = r_out;
      hit = hit_out;
     }
      break;
    default:
      return;
    }
  }

  // Store in caustics map at first diffuse surface
  caustics.store(phi, hit.position, -r.direction);
}
```

The results can be seen in figure \ref{fig:ws7}.

2.8 Worksheet 8

2.8.1 Fresnel Reflectance



(a) Constant 10% reflectance (b) Photonmap for const. 10% (c) Fresnel Reflectance

(d) Fresnel with highlights (e) Fresnel with caustics (f) Fresnel Photonmap

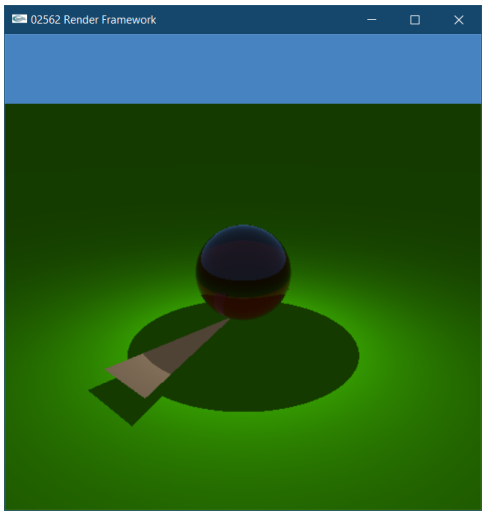
Figure 25: Before implementing Fresnel Reflectance (25a, 25b), and after (25c, 25d, 25e, 25f).

In this part of the worksheet we had to calculate the Fresnel Reflectance based on the angle of incidents instead of assuming it to be a constant 10%. Figure 25 shows the results. This is the relevant code:

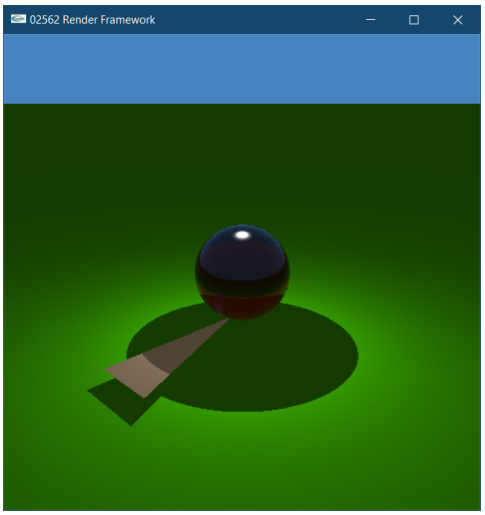
```
//Raytracer.cpp
bool RayTracer::trace_refracted(const Ray& in, const HitInfo& in_hit, Ray&
    out, HitInfo& out_hit, float& R) const
{
   if(trace_refracted(in, in_hit, out, out_hit) || !(out.direction.x == 0
       && out.direction.y == 0 && out.direction.z == 0)) {
        float cosOi = dot(normalize(in_hit.shading_normal),
            normalize(in.direction));
        float cosOt = dot(normalize(in_hit.shading_normal),
            normalize(out.direction));
        float ni = in_hit.ray_ior;
        float nt = out_hit.ray_ior;
        float rperp = (ni * cosOi - nt * cosOt) / (ni * cosOi + nt * cosOt);
```

```
        float rpara = (nt * cosOi - ni * cosOt) / (nt * cosOi + ni * cosOt);
        R = 0.5f * (pow(abs(rperp), 2) + pow(abs(rpara), 2));
    }else { //total internal reflection
        R = 1.0f;
        return false;
    }
}
```

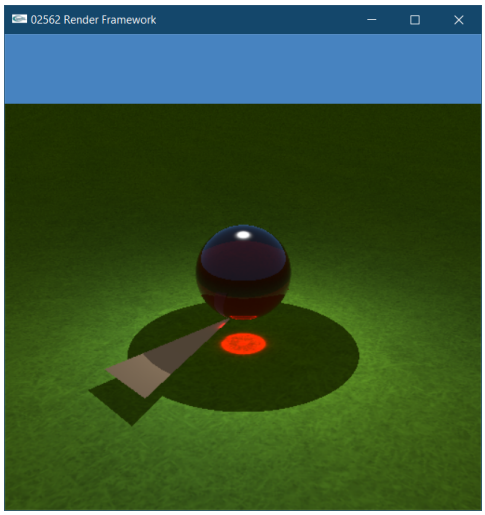
2.8.2 Absorption



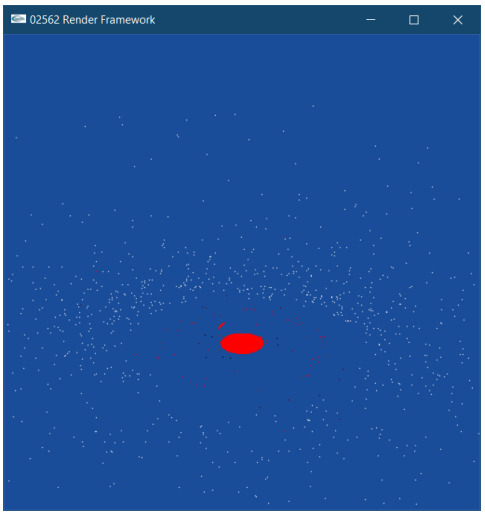
(a) Glass shpere with absorption



(b) Shpere with absorption and highlights



(c) Shpere with absorption and caustics



(d) Photonmap for absorbing sphere

Figure 26: Default scene with glass sphere rendered with volumetirc absorption.

The second part of this worksheet was to calculate the absorption that is happening while rendering transparent objects. For this we had to implement Bouguer's law o transmittance:

```
//Volume.cpp
float3 Volume::get_transmittance(const HitInfo& hit) const
{
   if (hit.material)
   {
      float3 rho_d = make_float3(hit.material->diffuse[0],
          hit.material->diffuse[1], hit.material->diffuse[2]);
      float3 result = make_float3(
         (rho_d.x != 0) ? exp((-1 / rho_d.x - 1)*hit.dist) : 0.0f,
         (rho_d.y != 0) ? exp((-1 / rho_d.y - 1)*hit.dist) : 0.0f,
         (rho_d.z != 0) ? exp((-1 / rho_d.z - 1)*hit.dist) : 0.0f
      );
      return result;
  }
  return make_float3(1.0f);
```

We then had to extend this to also include Phong highlights:

```
//GlossyVolume.cpp
float3 GlossyVolume::shade(const Ray& r, HitInfo& hit, bool emit) const
{
   if (hit.trace_depth >= max_depth)
      return make_float3(0.0f);

   float R;
   Ray reflected, refracted;
   HitInfo hit_reflected, hit_refracted;
   tracer->trace_reflected(r, hit, reflected, hit_reflected);
   tracer->trace_refracted(r, hit, refracted, hit_refracted, R);
   float3 result = R * (shadePhong(r, hit, emit) + shade_new_ray(reflected,
       hit_reflected)) + (1.0f - R) * shade_new_ray(refracted,
       hit_refracted);

   if (dot(hit.shading_normal, r.direction) > 0) {
      result *= get_transmittance(hit);
   }

   return result;
}

float3 GlossyVolume::shadePhong(const Ray& r, HitInfo& hit, bool emit)
    const
{
   float3 rho_s = get_specular(hit);
   float s = get_shininess(hit);
```

```
    float3 result = make_float3(0);
    for (Light* light : lights) {
        float3 Lr = make_float3(0.0f);
        float num_samples = light->get_no_of_samples();
        for (int i = 0; i < num_samples; i++) {
            float3 light_dir, L;
            if (!light->sample(hit.position, light_dir, L)) continue;

            float3 n = hit.shading_normal;
            float3 wi = light_dir;
            float3 wr = optix::reflect(-wi, n);//2 * dot(wi, n) * n - wi;
            float3 wo = -normalize(r.direction);
            Lr += (rho_s * (s + 2) * (0.5f * M_1_PIf) * pow(fmax(0, dot(wo,
                wr)), s)) * L * fmax(0, dot(wi, n));
        }
        result += (Lr / num_samples);
    }
    return result + Emission::shade(r, hit, emit);
}
```

And finally, we also implemented transmittance for the photon map:

```
//ParticleTracer.cpp
float3 ParticleTracer::get_transmittance(const HitInfo& hit) const
{


    if (hit.material)
    {
        float3 rho_d = make_float3(hit.material->diffuse[0],
            hit.material->diffuse[1], hit.material->diffuse[2]);
        float3 result = make_float3(
            (rho_d.x != 0) ? exp((-1 / rho_d.x - 1) * hit.dist) : 0.0f,
            (rho_d.y != 0) ? exp((-1 / rho_d.y - 1) * hit.dist) : 0.0f,
            (rho_d.z != 0) ? exp((-1 / rho_d.z - 1) * hit.dist) : 0.0f
        );

        return result;
    }
    return make_float3(1.0f);
}
```

The results can be seen in figure 26.

2.9 Worksheet 9

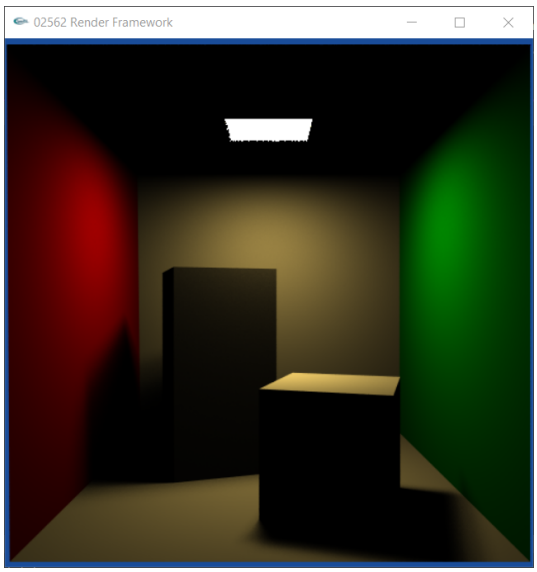


Figure 27: The Cornell Box rendered using an area light casting soft shadows.

In this worksheet we first had to implement area lights which cast softer shadows. This is more realistic since there are no real point lights in reality. The results of this can be seen in figure 27, the relevant code is below.

```
//AreaLight.cpp
bool AreaLight::sample(const float3& pos, float3& dir, float3& L) const
{
  const IndexedFaceSet& normals = mesh->normals;
  L = make_float3(0.0f);

  //calculate light position using a russian roulette algorith
  uint3 faceIDs = mesh->geometry.face((int)(mt_random_half_open() *
      mesh->geometry.no_faces()));
  float3 bar_coord = normalize(make_float3(mt_random(), mt_random(),
      mt_random()));
  bar_coord /= (bar_coord.x + bar_coord.y + bar_coord.z);
  float3 light_pos = mesh->geometry.vertex(faceIDs.x) * bar_coord.x +
      mesh->geometry.vertex(faceIDs.y) * bar_coord.y +
      mesh->geometry.vertex(faceIDs.z) * bar_coord.z;

  //calculate direction and distance
  dir = light_pos - pos;
  float dist = length(dir);
  dir = normalize(dir);

  //calculate intensity
```

```
  for (int i = 0; i < mesh->get_no_of_primitives(); i++) {
    //intensity += get_emission(i) / mesh->face_areas[i];
    uint3 face = normals.face(i);
    float3 normal = normalize(normals.vertex(face.x) +
        normals.vertex(face.y) + normals.vertex(face.z));
    intensity += dot(-dir, normal) * get_emission(i) * mesh->face_areas[i];
  }

  //shadows
  if (shadows) {
    Ray shadowray = Ray(pos, dir, 0, 0.0001f, dist-0.0001f);
    HitInfo hit;
    tracer->trace_to_any(shadowray, hit);
    if (hit.has_hit) { return false; }
  }

  //lighting
  L = intensity / pow(dist, 2);

  return true;
}
```

Figure 28: The Utah Teapot rendered using an environment map. The shadow on the bottom is due to a holdout plane.

The second part of the exercise was to setup a scene using the Utah teapot and a HDRI environment map. We also had to add a holdout plane which is an invisible piece of geometry used to calculate shadows that objects cast on the scene. The result is pictured in figure 28. The code is below.

```
//Holdout.cpp
float3 Holdout::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  float ambient = 0.0f;
```

```
  float3 result = make_float3(0);
  for (int i = 0; i < samples; i++) {
   float3 wj = sample_cosine_weighted(hit.shading_normal);
   HitInfo hitTemp;
   Ray rn = Ray(hit.position, wj, 0, 1e-4, RT_DEFAULT_MAX);
   ambient += tracer->trace_to_closest(rn, hitTemp) ? 0 : 1;
  }
  return ambient / samples * tracer->get_background(r.direction) ;
}
```

2.10 Worksheet 10

Figure 29: The teapot shaded using a the MERL shader with a pearl BRDF

In this worksheet we haad to render the Utah teapot, or any other obejct, using a BRDF from the MERL database. This database contains measured BRDFs for many different materials. I decided on using a pearl paint BRDF. The results can be seen in figure 29. The relevant code snippets can be found below.

```
//BRDF.cpp
float3 lookup_brdf_val(const float* brdf, const float3& n, const float3&
    normalized_wi, const float3& normalized_wo)
{
   float3 b1 = make_float3(0);
   float3 b2 = make_float3(0);

   onb(n, b1, b2);

   Matrix3x3 m = Matrix3x3();
   m.setRow(0, b1);
   m.setRow(1, b2);
   m.setRow(2, n);
```

```
   float3 i = m*normalized_wi;
   float3 o = m * normalized_wo;

   float theta_half, phi_half, theta_diff, phi_diff;
   vectors_to_half_diff_coords(i, o, theta_half, phi_half, theta_diff,
       phi_diff);

   return lookup_brdf_val(brdf, theta_half, phi_half, theta_diff, phi_diff);
}
```

```
//BRDF.cpp
float3 integrate_brdf(float* brdf, unsigned int N)
{
  float3 sum = make_float3(0.0f);
  float3 n = make_float3(0.0f, 0.0f, 1.0f);

  for (int i = 0; i < N; i++) {
     sum += lookup_brdf_val(brdf, n, normalize(sample_cosine_weighted(n)),
         normalize(sample_cosine_weighted(n)));
  }

  return sum * M_PIf / N;
}
```

```
//MerlShader.cpp
float3 MerlShader::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  if(hit.trace_depth >= max_depth)
    return make_float3(0.0f);

  const ObjMaterial* m = hit.material;
  MerlTexture* tex = brdfs && m && m->has_texture ? (*brdfs)[m->tex_name] :
      0;
  float3 rho_d = get_diffuse(hit);
  float3 result = make_float3(0.0f);

  if (tex == 0 || !tex->has_texture()) { return make_float3(1,0,1); }

  //Direct Lighting
  float3 dir, L;
  for (Light* light : lights) {
     float3 resultL = make_float3(0);
     for (int i = 0; i < light->get_no_of_samples(); i++) {
        if (light->sample(hit.position, dir, L)) {
```

```
            resultL += tex->brdf_lookup(hit.shading_normal,
                normalize(-r.direction), normalize(dir)) * L * fmax(0.0f,
                dot(hit.shading_normal, dir));
        }
      }

      result += resultL / light->get_no_of_samples();
    }

    //Indirect Lighting
    float pdf = (rho_d.x + rho_d.y + rho_d.z) / 3;
    if (mt_random() < pdf) {
     Ray rayTemp = Ray(hit.position,
         normalize(sample_cosine_weighted(hit.shading_normal)), 0, 0.001f,
         RT_DEFAULT_MAX);
     HitInfo hitTemp;
     hitTemp.trace_depth = hit.trace_depth + 1;
     tracer->trace_to_closest(rayTemp, hitTemp);
     result += tex->brdf_lookup(hit.shading_normal, normalize(-r.direction),
         rayTemp.direction) * M_PIf * shade_new_ray(rayTemp, hitTemp, false)
         / pdf;
    }
    return result;
}
```