Lab Journal

Introduction

This lab journal was written to indicate how each of the worksheets where implemented throughout the course. The journal also contains screenshots as results of the implementation of each part to satisfy the lab's deliverables along side with relevant code snippets and my personal reflection of some questions. The implementation and writing of the report was only conducted by myself, although some help was gotten in the lab by the TAs of the course as well as speaking about solutions with some of my peers.

> **Rendering Lab 1**
>
> ### Lab 1: Rendering Introduction
>
> Expected due date: 07-09-2022
>
> **Lab Purpose:**
> Ray tracing is the easiest and the most general technique for visualizing 3D models. It is easy to create many sophisticated lighting effects using ray tracing, but ray tracing is still slower than rasterization. Hence, raytracing is used primarily for rendering of photorealistic images in applications where image quality rather than a high frame rate is the primary concern. The purpose of the exercises for the first weeks is to help you learn how a ray tracer works. A ray tracing framework is available on DTU Learn which provides a coding infrastructure such that you only need to implement the essential parts.
>
> **Learning Objectives**
>
> - Implement ray casting (tracing rays from eye to first surface intersection).
> - Use a pinhole camera model for generating rays in a digital scene.
> - Compute intersection of rays with three-dimensional primitive objects (planes, triangles, spheres).
> - Compute shading of diffuse surfaces using Kepler's inverse square law and Lambert's cosine law.
>
> **Deliverables**
> Renderings of the default scene (e.g. preview, flat reflectance shading, and shading of diffuse objects). Compare preview and ray casting using the default flat reflectance shading to ensure that ray generation and ray-object intersection work as expected. Include relevant code snippets. Worksheet deliverables, such as these, should be included in your lab journal at the final hand-in.
>
> #labnotes #todo
>
> ---
>
> **Useful Links:**
>
> Course Book
> Lab Worksheet
>
> ---

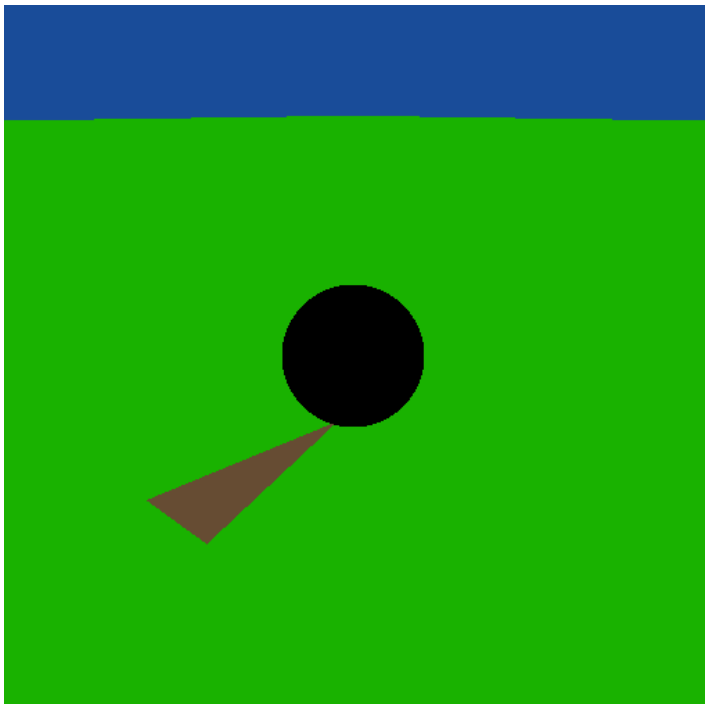
Task 0: Getting the Framework Running

The first task of the course is to get the framework that has been provided by the course coordinator to build and compile. This is a harder task than was anticipated. There are two ways to work on the course. One is in person in the preconfigured lab computers the other is a road of hours of despair to get the binary's pathing to work. Ultimately the solution to get the framework working on my home computer is as follows:

1. Delete all present C++ redistributables through Control Panel.

2. Install the DXSDK
3. Install the Nvidia CG Toolkit
4. Install the C++ 2015-2022 redistributables.
5. Do a clean Install of CLION as Visual Studio doesn't work.
6. Get the MinGW freeglut binaries
7. Follow the steps provided by the course coordinator to get the pathing right.
8. Delete all float variable references in `sqrt` functions in the quaternion file so that instead of `sqrtf()` they are simply `sqrt()`.
9. Delete all float variable references in `fmax` functions in the quaternion file so that instead of `fmaxf()` they are simply `fmax()`.

The resulting image when running the framework is shown below:



Task 1: Using the Raytracing to Change the Color of the Screen to Red

By following the lab's instructions we need to alter the render method inside the `RenderEngine.cpp` file in order to loop over every pixel on the screen and color them red. The first step is to change the `RayCaster.cpp` file to color the pixel sent in to the `compute_pixel` function red. This is desired to be done first to know that our image is changing while implementing the loop. To change the color of the computed pixel we simply alter the `make_float` function call parameters into a 3 dimensional float array using the red coordinates of (1, 0, 0). This change is shown in the code below:

```
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y) const
{
  float3 result = make_float3(1.0f, 0.0f, 0.0f);
  return result;
}
```

Now going back to the `RenderEngine.cpp` file we can change the inner loop of the function `render` to loop over all the screen's pixels. The for loop will change the image array as shown by the code below:

```
void RenderEngine::render()
{
  for(int y = 0; y < static_cast<int>(res.y); ++y)
  {
    for (int x = 0; x < static_cast<int> (res.x); x++){
        float2 pixelCoords = make_float2(x / res.x, y / res.y);
                pixelCoords = pixelCoords * 2 - 1; // [-1, 1]
                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;
}
```

The result of this first task should be a screen covered in red pixels when pressing the `r` key on the keyboard once the program is ran. Which it indeed is as shown in the image below:

Task 2: Implementing a Pinhole Camera

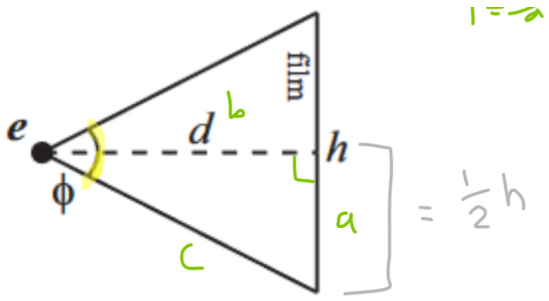
In order to implement a pinhole camera we need to change the `camera.cpp` file. Specifically we need to change the three functions; `set`, `get_ray_dir`, and `get_ray`. The first function we can change is the camera set function. This is where taking the sister course 02561 Computer Graphics helps as the implementation using WebGL is much more straightforward. What we need to do is implement the method view matrix form WebGL. This needs three variables changed in the `set` camera function; `ip_normal`, `ip_xaxis`, `ip_yaxis`, and `fov`. The following explanation is given by the course conductor:

Given pixel index, camera resolution, and the size of the film, we can find the image plane coordinates (x_ip, y_ip) of a point in the pixel of index (i, j). The directions of the axes in the camera coordinate system are defined by an orthonormal basis which includes the viewing direction $\vec{v}$. With this basis, the point with coordinates (x_ip, y_ip) in the image plane has coordinates $q_ip = (x_ip, y_ip, d)$ in the camera coordinate

system, and the direction of a ray from the eye through this point is given by a change of basis from the camera coordinate system to the usual coordinate system. The orthonormal camera basis $\langle \vec{b_1}, \vec{b_2}, \vec{v} \rangle$ is:

$$\vec{v} = \frac{p-e}{|p-e|}, \ \vec{b_1} = \frac{\vec{v} \times \vec{u}}{|\vec{v} \times \vec{u}|}, \ \vec{b_2} = \vec{b_1} \times \vec{v}$$

If we start with what we know from the assumptions of $h = 1, w = 1$ we know that we can find the FOV or ϕ by using Pythagoras theorem on the triangle shown in the below picture and doubling the result:



Then we have to make sure that the FOV we are getting is in degrees by multiplying our result by $\frac{180^\circ}{\pi}$
Which is proof to how we can find the FOV by using the following code:

```
float phi = 2.0f * atan((h/2.0f) / d);
fov = (phi * (180.0f * M_PIf)) / 10.0f;
```

However we also need to set the other camera variables in our `set` function. These variables are the normalized vectors of the parameters with the cross product normalized vector of each other as seen in the following code, but it is important to remember to flip the image since the plane is behind the eye point:

```
ip_normal = normalize(lookat - eye);
ip_xaxis = normalize(cross(ip_normal, up));
ip_yaxis = -normalize(cross(ip_normal, ip_xaxis));
```

Now we have to change the `get_ray_dir` so that the direction of the ray from the image coordinates Have the change of basis as explained in the "Ray Generation Using a Pinhole Camera" supplement course instructions. We get the rebased coordinates from the given formula:

$$q = \begin{bmatrix} \vec{b_1} & \vec{b_2} & \vec{v} \end{bmatrix} q_{ip} = \vec{b_1} x_{ip} + \vec{b_2} y_{ip} + \vec{v} d$$

Which in code looks like the following:

```
float3 Camera::get_ray_dir(const float2& coords) const
{
    float3 q = ip_xaxis * coords.x + ip_yaxis * coords.y + ip_normal * cam_const;
        return normalize(q);
}
```

Lastly we need to modify the `get_ray` function which is simply changing the parameters that go into the ray. The parameters of the Ray object are; the origin of the ray, the direction, the ray type that specifies which closest-hit/any-hit pair will be used when the ray hits a geometry object, the `tmin` / `tmax` members

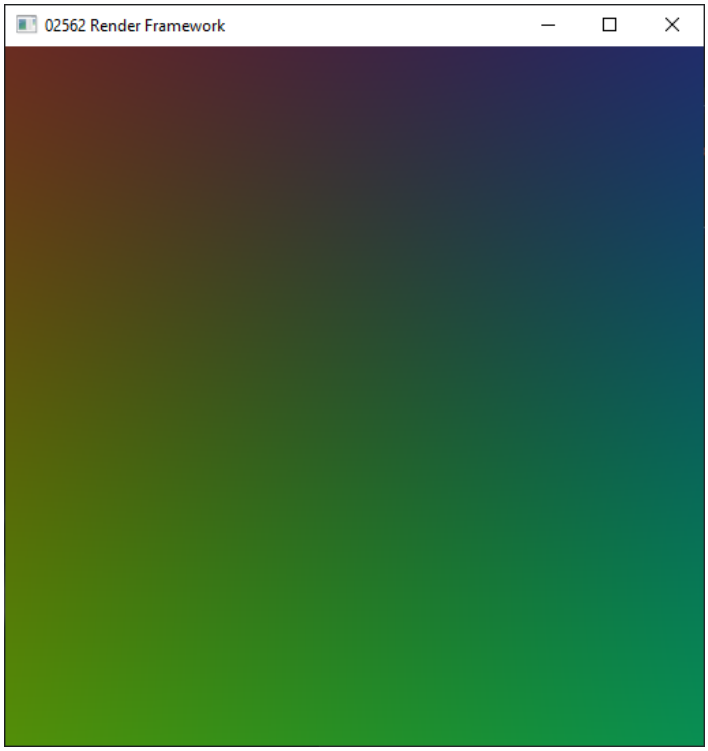
specify the interval over which the ray is valid. This solution is a bit simpler than the one provided in the "Learn Raytracing in a weekend" course supplement material:

```
Ray Camera::get_ray(const float2& coords) const
{
    return Ray(eye,get_ray_dir(coords),0,0,RT_DEFAULT_MAX);
}
```

Now we have to change the coloring of the pixels to match the ray direction by changing the `compute_pixel` function in the `RayCaster.cpp` file again. Here we use the ray we get the ray from the scene object and it's camera variable. Then to color the pixel the direction of the ray with the added bias the code will be as follows:

```
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y) const
{
  float3 result = make_float3(1.0f, 0.0f, 0.0f); // Default color is red
  float2 pos = make_float2(x, y) * win_to_ip + lower_left;
  Ray r = scene->get_camera()->get_ray(pos);
  result = r.direction*0.5+0.5;

  return result;
  }
```

If we run the `RayTracing` application again and hit r on the keyboard we get the following result:



Task 3: Implement Conditional Acceptance on Hit

We will now be implementing a way to check if the ray being casted has hit a primitive or not. We have to implement a function that checks all the primitives geometry attribute and compare it to the current casted ray to decide if that ray hit the object. You would think that this would be using the bounding box

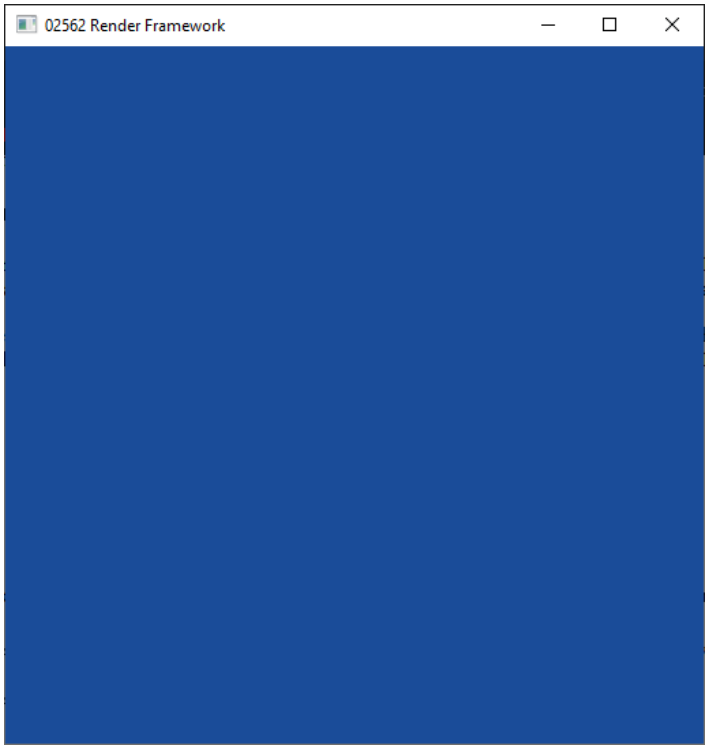
of the object but I couldn't get it to work with the bounding box. Instead it's a simpler solution that just calls the `intersect` function as shown below:

```
hit.has_hit = false;
  for (unsigned int i = 0; i < primitives.size(); i++) {
      if (primitives[i]->geometry->intersect(r, hit, primitives[i]->prim_idx)) {
          // Set the new max distance
          r.tmax = hit.dist;
      }
  }

return hit.has_hit;
```

Going back to the `compute_pixel` function in the `RayCaster.cpp` file again. With this ray we can use the scene object again which has the `closest_hit` function which returns a `HitInfo` object. This `HitInfo` object holds a Boolean if it has hit an object. We can use this flag to determine the color of the pixel that has been hit by getting the shader of whatever was hit using the `get_shader` function. Our code thus looks as follows:

```
float3 RayCaster::compute_pixel(unsigned int x, unsigned int y) const
{
  float3 result = make_float3(1.0f, 0.0f, 0.0f); // Default color is red
  float2 pos = make_float2(x, y) * win_to_ip + lower_left;
  Ray r = scene->get_camera()->get_ray(pos);
  HitInfo hit;
  scene->closest_hit(r, hit);

  if (hit.has_hit) {
    result = get_shader(hit)->shade(r, hit);
    }
    else {
        result = get_background();
    }

  return result;
  }
```

From our implementation we get the following result showing that the ray did not hit anything as the background color is blue:



Task 4: Ray Intersections for Spheres, Triangles, and Planes

Lets start off with setting up the intersection for the planes. A plane is a simple 2D object that stretches to infinity or a set max distance. We have to edit the `Plane.cpp` files in order to set them up correctly. For now all we have to do is implement the `intersect` function we called in our primitive loop during Task 3. A plane is intersected if the direction of the ray is parallel with the normal of the plane. As such you can find if a line intersect a plane by the following formula:

$$l \cdot n \neq 0$$

Or the special case where the line is on the plane and thus it is intersected at every point which can be found with the following:

$$(p_0 - l_0) \cdot n = 0$$

However, during the lecture we learned of a way to derive the distance to the intersection point using the available variables to the function. As such we can determine the hit variable using the following code:

```
// If the ray is parallel to the plane, there is no intersection
if(dot(r.direction, onb.m_normal) == 0.0f){
    return false;
}

// Distance to the intersection point by applying p = r
float t = -(dot(r.origin, onb.m_normal) + d) / dot(r.direction, onb.m_normal);

  // Check if the intersection point is within the ray's range
  if (t >= r.tmin && t <= r.tmax) {
      hit.has_hit = true;
      hit.dist = t;
      hit.position = r.origin + r.direction * t;
      hit.geometric_normal = normalize(onb.m_normal);
```

```
        hit.shading_normal = normalize(onb.m_normal);
        hit.material = &material;
        return true;
    }

    return false;
```

Now we can move on to triangles which act in a similar manner as planes, note that this implementation is already done in the framework but can still be implemented for practice. The algorithm that will be implemented to check for ray-triangle intersect will be that of the Möller-Trumbore intersection algorithm shown below:

```
bool Triangle::intersect(const Ray& r, HitInfo& hit, unsigned int prim_idx) const
{
    // Implementation of the Möller-Trumbore algorithm    const float epsilon = 0.0001f;
    optix::float3 edge1, edge2, h, s, q;
    float  a, f, u, v;
    edge1 = v1 - v0;
    edge2 = v2 - v0;
    h = optix::cross(r.direction, edge2);
    a = optix::dot(edge1, h);
    if (a > -epsilon && a < epsilon)
        return false;    // This ray is parallel to this triangle.
    f = 1.0f / a;
    s = r.origin - v0;
    u = f * optix::dot(s, h);
    if (u < 0.0 || u > 1.0)
        return false;
    q = optix::cross(s, edge1);
    v = f * optix::dot(r.direction, q);
    if (v < 0.0 || u + v > 1.0)
        return false;
    // At this stage we can compute t to find out where the intersection point is on the
line.
    float t = f * optix::dot(edge2, q);
    if (t > epsilon) // ray intersection
    {
        hit.has_hit = true;
        hit.dist = t;
        hit.position = r.origin + t * r.direction;
        hit.geometric_normal = optix::normalizecross(edge1, edge2);
        hit.shading_normal = hit.geometric_normal;
        hit.material = &material;
        return true;
    }
    else // This means that there is a line intersection but not a ray intersection.
        return false;

  return false;
}
```

Another possible implementation would be to first get the β and the γ from the supplementary function taken from Listing 1 of the Ray-triangle intersection bonus material from the course.

Now that we have both the triangle and the plane showing up on the scene we lastly need to implement the sphere. The first simple test to check the distance between the center of the sphere and the ray, it this distance is larger than the radius of the sphere it means there is no intersection. We also learned during the lectures that using a calculated distance to the intersections we can find the points of intersection using the following code:

```
// Setup the quadratic equation
float a = dot(r.direction, r.direction);
a = 1.0f; // a = 1 according to the lecture slides
float b = 2.0f * dot((r.origin - center), r.direction);
float b_2 = b / 2; // b/2
float c = dot((r.origin - center), (r.origin - center)) - (radius * radius);

// Calculate the discriminant
float discriminant = (b_2 * b_2) - c;

// No intersection if b^2 - c < 0
if (discriminant < 0.0f) {
    return false;
}

// There is no need to handle the case where the discriminant is zero separately but here is
the code anyway
if (discriminant == 0.0f){
    // If the discriminant is zero the ray grazes the sphere
    // One intersection point}
else if(discriminant > 0.0f){
    // If the discriminant is greater than zero the ray intersects the sphere at two points
    // Two intersection points: The entry point and the exit point
}

// Calculate distance to the intersection point(s)
float t_1 = -b_2 - sqrt(b_2 * b_2 - c);
float t_2 = -b_2 + sqrt(b_2 * b_2 - c);

bool hit_1 = false;
bool hit_2 = false;

// Check if the intersection point(s) are within the ray's tmin and tmax
if (t_1 >= r.tmin && t_1 <= r.tmax) {
    hit_1 = true;
}

if (t_2 >= r.tmin && t_2 <= r.tmax) {
    hit_2 = true;
}

if (hit_1 && hit_2){
    // The closest intersection is the smallest of the two solutions (t_1 and t_2)
    float t = (t_1 < t_2) ? t_1 : t_2;
```

```
    float t = (t_1 < t_2) ? t_1 : t_2;
    hit.has_hit = true;
    hit.dist = t;
    hit.position = r.origin + r.direction * t;
    hit.geometric_normal = normalize(hit.position - center);;
    hit.shading_normal = normalize(hit.position - center);;
    hit.material = &material;
    return true;
} else if (hit_1){
    // The closest intersection is t_1
    float t = t_1;
    hit.has_hit = true;
    hit.dist = t;
    hit.position = r.origin + r.direction * t;
    hit.geometric_normal = normalize(hit.position - center);;
    hit.shading_normal = normalize(hit.position - center);;
    hit.material = &material;
    return true;
} else if (hit_2){
    // The closest intersection is t_2
    float t = t_2;
    hit.has_hit = true;
    hit.dist = t;
    hit.position = r.origin + r.direction * t;
    hit.geometric_normal = normalize(hit.position - center);;
    hit.shading_normal = normalize(hit.position - center);;
    hit.material = &material;
    return true;
} else {
    // No intersection
    return false;
}
```

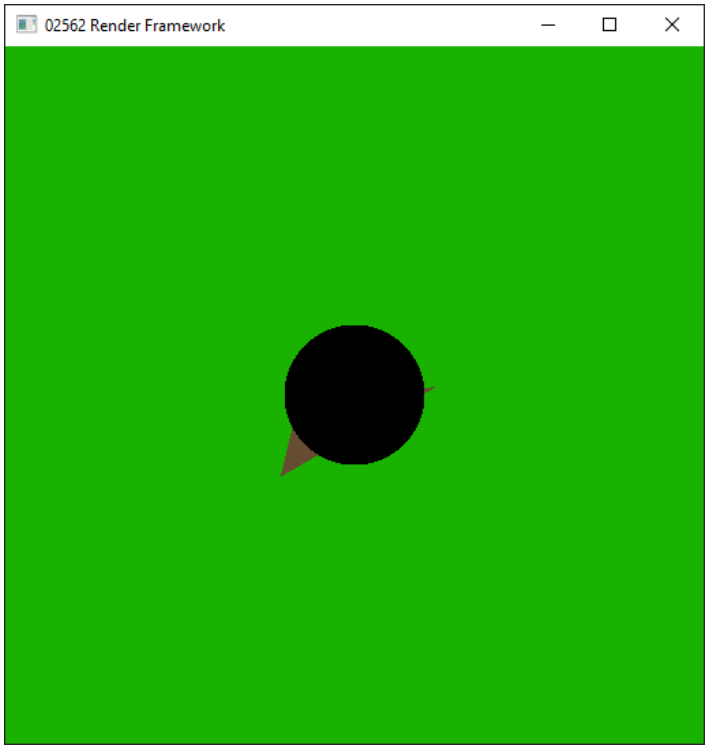
After the implementation of the three different intersection tests we get a rendered scene similar to that of the one provided in the framework. There are possible improvements in where our plane and triangle tests are showing up even if they are seen from behind (below). Although, they do appear reflected so it means our test is working albeit as perhaps it's not supposed to. There is also no depth test as of yet in our scene meaning that if the triangle is supposed to be hidden behind the sphere our program doesn't understand this and simply draws the triangle (or sphere depending on which was called first) on top of the sphere. Never the less, the result from this task can be seen below:

We can fix this by implementing the z-buffer as described in the book that simply tracks the z-depth position on the `hit` object. Then we only return a hit if it's the first thing that is hit. Except for our plane is the thing that is always being hit first so there's an offset by 1. Or instead we can implement the triangle-ray intercept function like in Lecture 1. As shown in the following code:

```
// Edges of the triangle
    float3 e0 = v1 - v0;
    float3 e1 = v2 - v0;

    // Face normal
    float3 n = cross(e0, e1);

    // Ray-plane intersection
    float t = dot((v0 - r.origin), n) / dot(r.direction, n);

    if (t >= r.tmin && t <= r.tmax ){
        // Decomposition of r(t) - v0
        float3 r_t = r.origin + t * r.direction - v0;

        // Barycentric coordinates of r(t)
        float beta = (dot((cross(v0 - r.origin, r.direction)), e1)) /
                     (dot(r.direction, n));

        float gamma = -(dot((cross(v0 - r.origin, r.direction)), e0)) /
                      (dot(r.direction, n));

        float alpha = 1 - beta - gamma;

        if(beta >= 0 && gamma >= 0 && beta + gamma <= 1){
            hit.has_hit = true;
            hit.dist = t;
```

```
            hit.position = r.origin + t * r.direction;
            hit.geometric_normal = normalize(n);
            hit.shading_normal = normalize(n);
            hit.material = &material;
            return true;
        }
    }

    return false;
```

The result which looks much better is shown below:



Task 5: Shading the Diffuse Surface

The last task in this first worksheet is to get the lighting to work, the lighting is done using a technique called diffused surfaces. This technique will use ***Kepler's Inverse Square Law*** to understand the intensity of the point light source. As well as, ***Lambert's Cosine Law*** to understand the brightness fall off on the the angle of hit by the ray of light which we learned in Lecture 2. Lets start by implementing Kepler's law in the `sample` function inside the `PointLight.cpp` file. The deliverables in this function will be the: direction towards the light (`dir`), the radiance received from the direction (`L`), and return whether or not the position is in shadow. Here we need to make sure that light from a point source falls off with the square of the distance to the point. Then we need to check that if shadows are enabled (by the user pressing `1` on the keyboard) then the function must cast a ray and trace it to any direction checking if it's behind an object. The function returns true if **NOT** in shadows! The code for the `sample` function can be seen below:

```
bool PointLight::sample(const float3& pos, float3& dir, float3& L) const
{
  // The distance from the light to the point
  float dist = length(light_pos - pos);
```

```
    // The direction from the point to the light
    dir = normalize(light_pos - pos);
    L = intensity / (dist * dist);

    // If shadows are enabled, check if the point is in shadow
    if (shadows) {
        Ray shadow_ray = Ray(pos, dir, 0.001f, dist);
        HitInfo shadow_hit;
        if (tracer->trace_to_any(shadow_ray, shadow_hit)) {
            return false;
        }
    }

  return true;
}
```

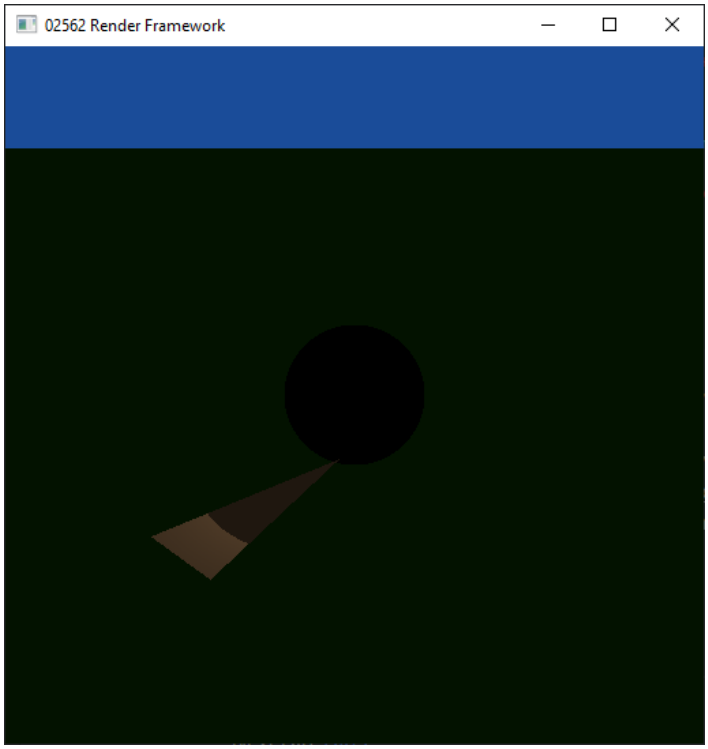
We now can implement Lambert's Law in the `shade` function of the `Lambertian.cpp` file. This function will need to find the reflected light L_r using the diffuse reflectance p_d from the angle of incidence θ from the normalized surface normal $\vec{n}$ to the direction of the light $\vec{w}$ from the formula below:

$$L_r = \frac{P_d}{\pi} L_i(\vec{n} \cdot \vec{w})$$

Which in code translates to the following:

```
for (unsigned int i = 0; i < lights.size(); ++i) {
    float3 dir, L;
    if (lights[i]->sample(hit.position, dir, L)) {
        float cos_theta = dot(dir, hit.shading_normal);
        if (cos_theta > 0.0f) {
            result += (rho_d * M_1_PIf) * L * cos_theta;
        }

    }
}
```

At this point in the scene we should have a scene with a rendered sphere triangle and plane with two different types of shading. The result of this worksheet can be seen below:



Next Lab: Lab 2

Rendering Lab 2

Lab 2: Raytracing

Expected due date: 14-09-2022

Lab Purpose:
Local illumination, like the result from the first worksheet, is rendered more quickly and in equal quality using rasterization techniques. However, the moment we start looking at global illumination effects (shadows, reflections, refractions, etc.), ray tracing has a number of advantages. Rasterization techniques for global illumination effects work well for some special cases only,1 whereas ray tracing is simpler to implement and works well in general.

Learning Objectives:

- Implement ray tracing.
- Render hard shadows by tracing shadow rays to a point light.
- Render reflections and refractions by tracing rays recursively.
- Compute shading of surfaces using the Phong illumination model.

Deliverables:
Renderings of the default scene (e.g. shading of diffuse objects, mirror ball, glass ball, Phong ball, and glossy ball). Include relevant code snippets. Explain the helper function `get_ior_out` and your final glossy shader `shade`. Please insert all this into your lab journal.

#labnotes #todo

Useful Links:

Course Book

Previous Lab

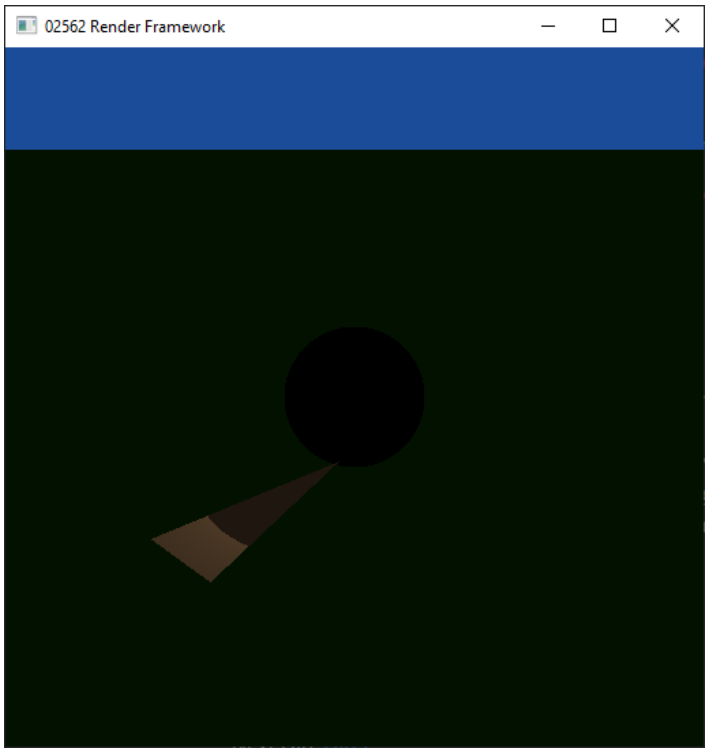
Lab Worksheet

Task 1: Finish Shadows From Last Worksheet

The shadows tracing in the point light was implemented last worksheet the code responsible for the shadows is shown below:

```
// Shadow ray cutoff variables
float epsilon = 0.0001f; // 10^-4
float t_max = distance - epsilon; // ||p-x|| - epsilon

if (shadows) {
    Ray shadow_ray = Ray(pos, dir, 0.0f, epsilon, t_max);
    HitInfo shadow_hit;
    if (tracer->trace_to_closest(shadow_ray, shadow_hit)) {
        return false;
    }
}
```

The result can be seen below:



Task 2: Implement Reflection

In this task we are going to make the black ball of death in the middle of the scene into a mirrored sphere. The way we are going to do this is by:

1. Implementing the the function `trace_reflected` in the file `RayTracer.cpp`.
2. Uncomment the call to `Mirror::shade` in the `shade` function of the file `Glossy.cpp` file.

The second step is quite self explanatory so lets focus on explaining the implementation for the `trace_reflected` function. Here what we want to do is is change the `out` object that we are modifying inside the function. The `out` object is the ray that is reflected off the reflective surface. Then there is also the `out_hit` object which contains the object hit in case if the ray has hit anything. This `out_hit` object is

updated not by us but by the `closest_hit` function. The only thing we do have to update in the function is update the the depth test for the ray or else the same rays will be casted for ever since the if statement in the `Mirror::shade` function will never be hit. The code for the `trace_reflected` is shown below:

```
// Initialize the reflected ray
out.origin = in_hit.position;
out.direction = reflect(in.direction, in_hit.geometric_normal);
out.tmin = in.tmin;
out.tmax = RT_DEFAULT_MAX;
out.ray_type = in.ray_type;

// Trace the reflected ray
// Check if the reflected ray hit anything
if (scene->closest_hit(out, out_hit)) {
    // Set out_hit.ray_ior and out_hit.trace_depth
    out_hit.ray_ior = in_hit.ray_ior; // Maybe 0?
    out_hit.trace_depth = in_hit.trace_depth + 1;
    return true;
}

return false;
```

This fixes our black ball of death in the middle of the scene into a nice mirrored ball. As shown by the screenshot below:

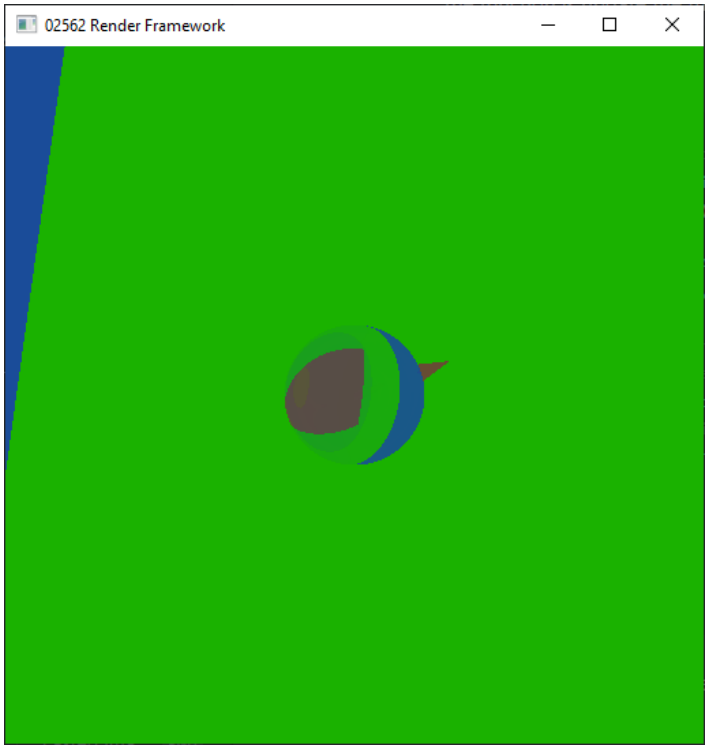
Task 3: Implement Refraction

We are now implementing glass material objects. This means that light will travel through the ball and refract. Refraction is the process of light slightly bending as it travels through a medium. For this implementation we are going to be changing the first overloaded `trace_refracted` function in the `RayTracer.cpp` file. In this function, just like the `trace_reflected` function, we have a ray of light hitting

an object. However, now instead of reflecting a ray out we needs to refract light passing through an object. Here we need to remember that this function should be continuously called until the in ray is no longer in the object. The refraction function can be seen below:

```
float3 normal;
const float n_1 = in_hit.ray_ior;
const float n_2 = get_ior_out(in, in_hit, normal);

const bool ref= refract(out.direction, in.direction, normal, n_2 / n_1);

// Refract will return false if full internal reflection occurs
if (!ref) {
    return false;
}
out.origin = (in_hit.position);
out.tmax = RT_DEFAULT_MAX;
out.tmin = in.tmin;
out_hit.ray_ior = n_2;

if (trace_to_closest(out, out_hit)) {
    out_hit.trace_depth = in_hit.trace_depth + 1;
    return true;
}

return false;
```

In the code above we make use of the `get_ior_out` helper function in the same `RayTracer.cpp` file. This function is what determines the new **Index of Refraction** for the next ray. This index is what is used to determine the direction that light will bend in an object. Light bends when going through a medium. What this function does is that it first checks that the normal and the direction of the incoming ray are not parallel. If the If they're not then it changes the normal to be the opposite of the normal to simulate light refracting in a medium incrementing the index by 1 (this indicates that there is internal reflection). Otherwise if the ray is coming directly into the object it checks for that object's material rule for direct rays. The result of the refraction is seen quite clearly in the screenshot below in where the triangle is being wrapped by the light traveling through the glass ball:



Task 4: Implement the Phong Illumination Model

Like in the sister course Computer Graphics we need to implement Phong Shading in order to render specular surfaces such as the mirror ball. In order to implement the **Phong Illumination** we need to use the formula presented in the Lecture Slides. The formulas variables are:

- p_d and p_s are the diffuse and specular reflectance $(p_d + p_s \leq 1)$
- s is the Phong shininess exponent
- $\vec{w}_o$ is the unit length direction vector towards the observer (eye)
- $\vec{w}_i$ is the direction of the incidence ray.
- $\vec{w}_r$ is the direction of the reflection ray.
- $\vec{n}$ is the unit length normal of the surface at intersect point.

The formula for the modified energy conserving Phong Model is shown below:

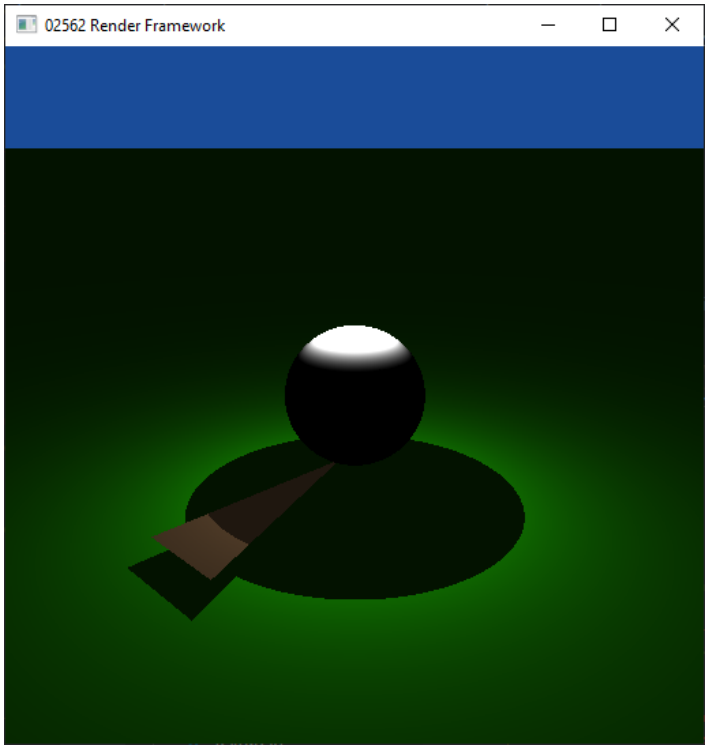
$$L_r = (k_d + k_s \cos^2 \alpha) L_i \cos\theta = (\frac{p_d}{\pi} + p_s \frac{s+2}{2\pi} (\vec{w}_o \cdot \vec{w}_r)^s) L_i (\vec{w}_i \cdot \vec{n})$$

So the code implementation of the formula above will look something like what is shown below:

```
float3 result = Lambertian::shade(r, hit, emit);

float3 dir = make_float3(0.0f, 0.0f, 0.0f);
float3 reflection = make_float3(0.0f, 0.0f, 0.0f);
float3 light_dir = make_float3(0.0f);
float3 L_i = make_float3(0.0f);
float3 radiance = make_float3(0.0f, 0.0f, 0.0f);

float spec = 0.0f;

// Loop through all the lights in the scene
```

```
for(auto light: lights){
    // Sample the light
    if(light->sample(hit.position, light_dir, L_i)){
        // Calculate the reflection vector
        reflection = reflect(-light_dir, hit.shading_normal);

        // Calculate the dot product between the reflection vector and the ray direction
        spec = dot(reflection, -r.direction);
        if(spec > 0.0001f){
            radiance += ( rho_d * M_1_PIf
                    + rho_s * ((M_1_PIf * (s + 2) ) / 2)
                    * pow(dot(-r.direction, reflection), spec)
            ) * L_i * dot(r.direction, hit.shading_normal);
            result += radiance;
        }
    }
}
```

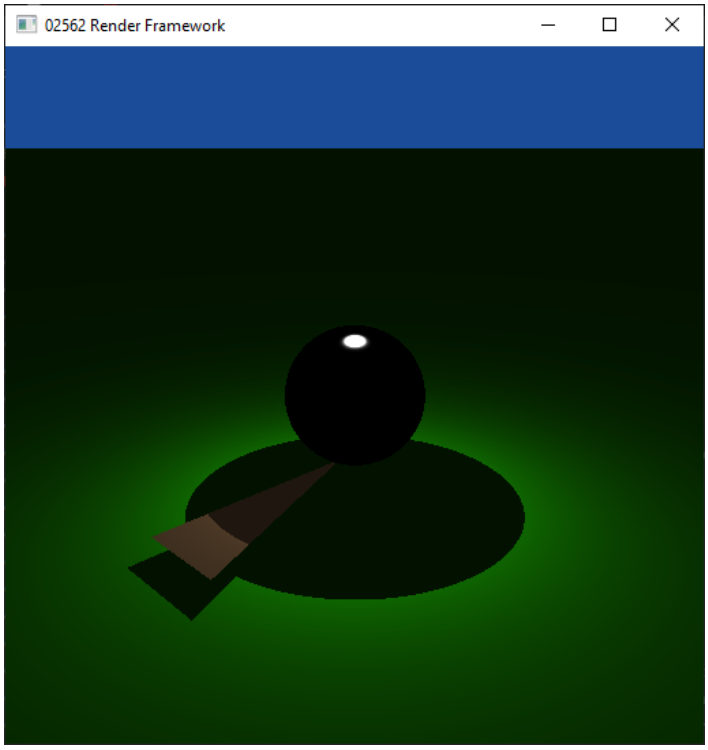
I would like to take a second to reflect on my implementation here, I compared my implementation to that of my peers in the lab and I noticed that their render result looked less washed out by the light. Having only a specular reflection at the top of the sphere. I am a bit conflicted with my results as I enjoy the way that the light appears to be much closer to the ball and I believe that is what the intended result should look like. My evidence for this is the huge shadow that the ball casts, which would indicate that the source of the light is quite close to the ball. The result of my implementation is shown below:



However after speaking with a TA it seems that the correct solution will be an implementation where I simply return the `spec` calculated with the `pow` function of the `spec` and `s` as follows:

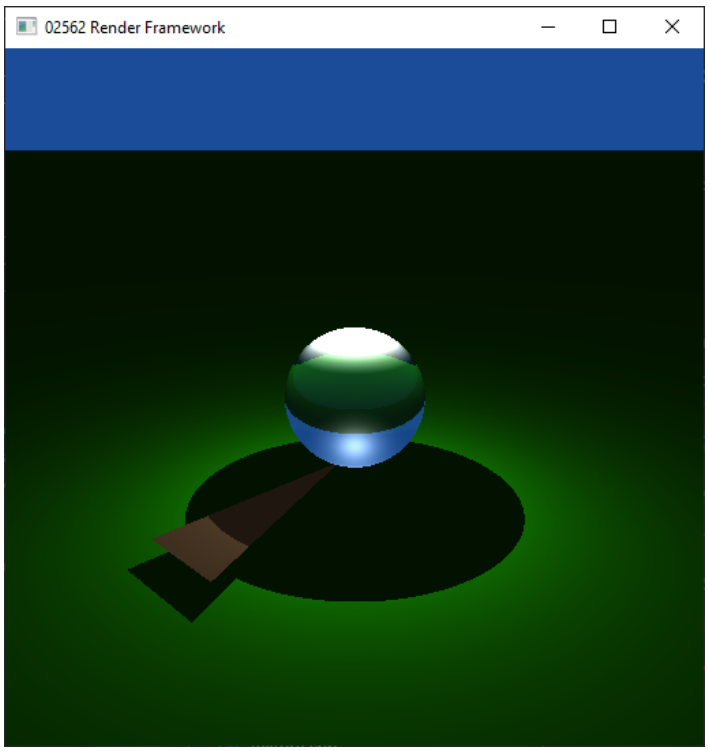
```
result += rho_s * L_i * pow(spec, s);
```

Which gives the small specular reflection at the top of the sphere as shown below:

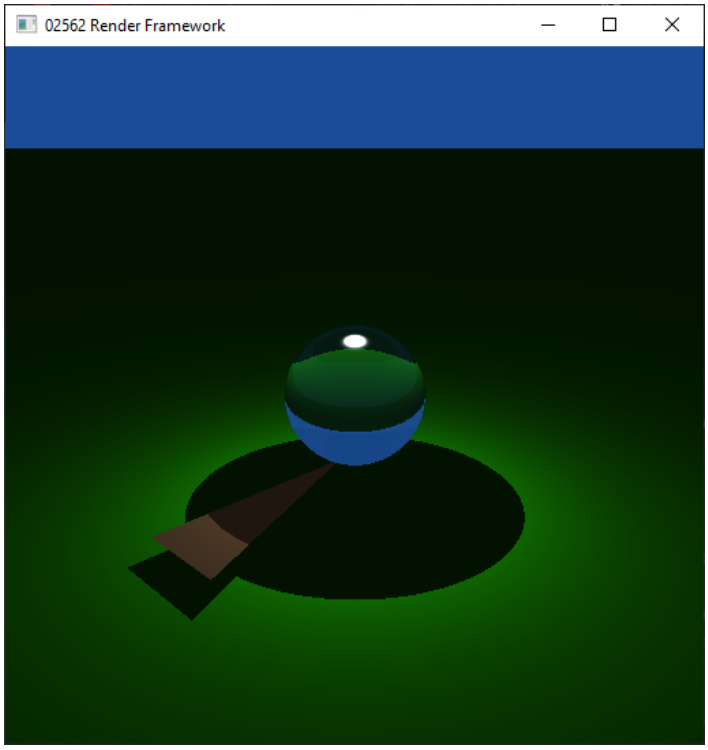


Task 5: Combine the Phong Illumination Model with Glass Shader

Taking a look at how the `shade` function works in the `Transparent.cpp` file works, it combines the reflected and refracted tracing into a single shading. Where the returned shading is a combination of both results. We can trivially combine this in the `Glossy.cpp` file by adding the result of the Phong Illumination Model to the shading. The code snippet is skipped for this task as it is just copying and pasting and adding the Phong call that we had before. The results from this lab are shown in the image below:



The results with the TA altered Phong Illumination model are shown below:



Next Lab: Rendering Lab 3

Rendering Lab 3

Lab 3: Textures

Expected due date: 21-09-2022

Lab Purpose:
If we compare rendered images to photographs, flatly colored surfaces with no visual detail is one of the first giveaways that an image is a rendering. To obtain realism in rendered images, it is important to add visual detail. One way to add visual detail is to use texturing. This set of exercises is about how to do texturing in ray tracing.

Learning Objectives:

- Use texture mapping (mapping an image to a surface) to heighten the level of visual detail.
- Compute texture coordinates using inverse mapping.
- Apply bilinear interpolation for texture magnification filtering.

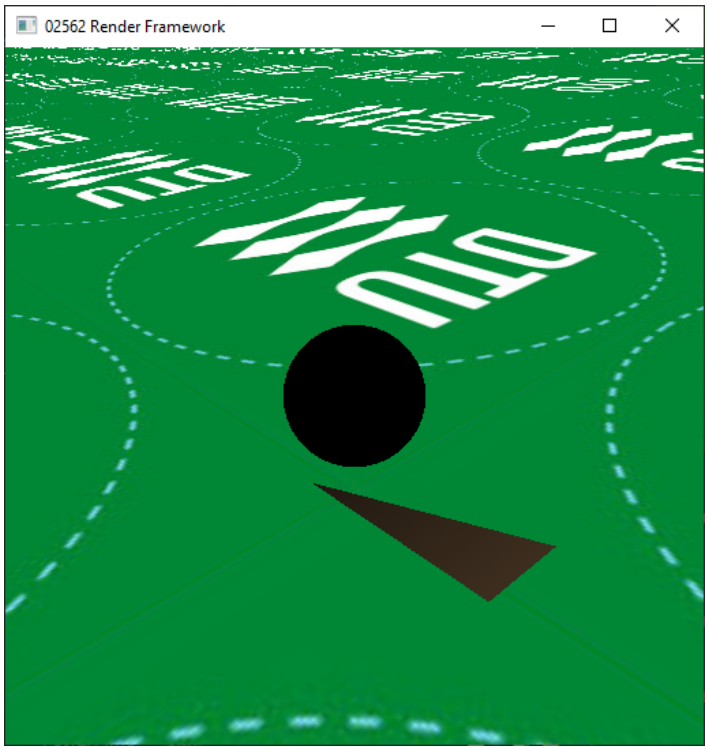
Deliverables:
Renderings of the default scene with a textured plane (e.g. using look-up of nearest texel, bilinear magnification filtering, and using nearest texel or bilinear filtering with a 10-fold magnified texture). Rendering of the default scene with a different choice of texture. In addition, provide the explanations and comparisons mentioned above. Include relevant code snippets. Please insert all this into your lab journal.

#labnotes #todo

Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: How the Framework Loads Textures

The main loop of the program was explained in Lecture 1. Before the main loop can occur though the program does some initialization tasks like setting up GL and and GLUT dependencies. In between the initialization of the GLUT and GL dependencies is the initialization of loading files into the program with the use of the `RenderEngine.cpp` file's `load_files` function. The `load_files` function is responsible for initializing the scene and scene variables. In the first part of the function it initializes things like the triangle meshes for objects and in the second part initializes the scene variables such as the eye position and light position which we implemented in previous labs. In this second part of the function it also initializes scene objects like the sphere, triangle, and plane. The initialization of such objects depend on their material file (as denoted with the file extension `.mtl`). The location of such material file is sent in to their respective `add_x` functions in where x is the type of object. If we look at the `add_plane` function the last argument is the texture scale we will use in the implementation of texture coordinate (see Task 3). More importantly this function make a call (through the call of another function `mt_load` in the `obj_load.cpp` file) to `read_material_library`. This function reads the material file specified in the sent in location. It does it by reading the file twice, once to setup memory space, and then once again to populate that memory. In the material file there is a variable which specifies which image file to use for the diffuse reflectivity `k_d` for the 'green' material. This is where the texture image is specified for objects with the green material. We can simply change the path of the file from `grass.jpg` to whatever we desire. Doing this can produce results like the one shown below:



Task 2: Explain Texture Color

Color in rendering is done by manipulating the diffuse and emission properties of that object. As we saw in Task 1 we can change the diffusion to change what the color would be when implementing diffuse shading practices like we implemented in previous Labs. Rendering of these objects uses a combination

of their diffuse color specified and their emission color (and by extent the ambient lighting color) to make the actual color of that object.

Task 3: Compute Texture Coordinates

In this task we will be computing texture coordinates (u, v) for the plane in the function `get_uv` of the `Plane.cpp` file. This is a straightforward endeavor as the available variable in the function are the orthonormal basis of the plane. This is the needed variables along side the hit position and the origin of the plane in order to find the texture coordinates by using the formula below in where x is the intersection point, x_o is the plane's origin, $\vec{b_1}$ is the orthogonal tangent of the plane, and $\vec{b_2}$ is the orthogonal binormal:

$$x = x_o + u\vec{b_1} + v\vec{b_2}$$

Using the knowledge that $\vec{n} = \vec{b_1} \times \vec{b_2}$ we know that we can get (u, v) by using the formula below in where s is the texture scale:

$$u = s(\vec{b_1} \cdot (x - x_o))$$

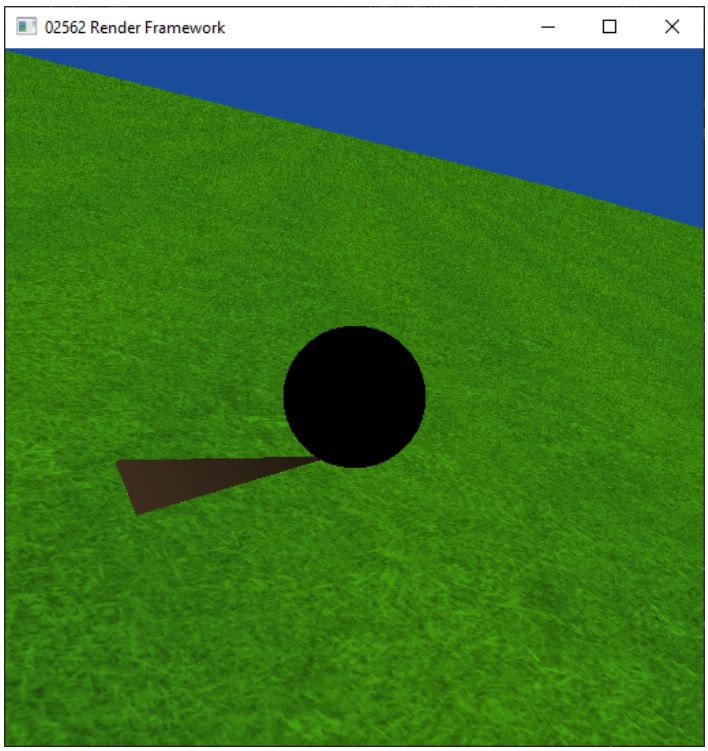
$$v = s(\vec{b_2} \cdot (x - x_o))$$

We can call the `get_uv` function by checking at the function of finding our intersection point of the plane, whether or not the material of the plane has a texture by calling the `material.has_texture` function as shown in the code below:

```
// Check if the plane has a texture in order to calculate the texture coordinates
if (material.has_texture) {
    get_uv(hit.position, hit.texcoord.x, hit.texcoord.y);
}
```

Task 4: Rendering Grass Texture on the Plane

We can test our texture coordinates from Task 3 by running the program and pressing `x` on the keyboard which turns on textures. The result can be seen in the image below:



Task 5: Implement Texture Look-Up Function

The texture look-up function is responsible for translating the texture coordinates (u, v) into texture image indices i and thereafter (U, V) image index coordinates. For the first implementation we want to map texture coordinates into image coordinates using nearest neighbor for texture filtering and repeat mode for edge clamping. Texture coordinates have a possible range of $[0, 1] \times [0, 1]$ while image space coordinates have a range of $[0, W] \times [0, H]$ where $W \times H$ is the image resolution. In texture repeat mode we learned that the mapping of these coordinates is found by the following formulas:

Two new sets of coordinates:

- (s, t) texture space $[0, 1] \times [0, 1]$.
- (a, b) image space $[0, W] \times [0, H]$, where $W \times H$ is the image resolution.

In texture repeat mode: $s = u - \text{floor}(u)\,, \quad t = v - \text{floor}(v)\,.$

$a = s * W\,, \qquad b = t * H\,.$

Once we have the image space coordinates we can find the image index coordinates using the following formulas:

$$U = (int)(a + 0.5) \mod W$$

$$V = (int)(b + 0.5) \mod H$$

Then we can find the index of the image array by:

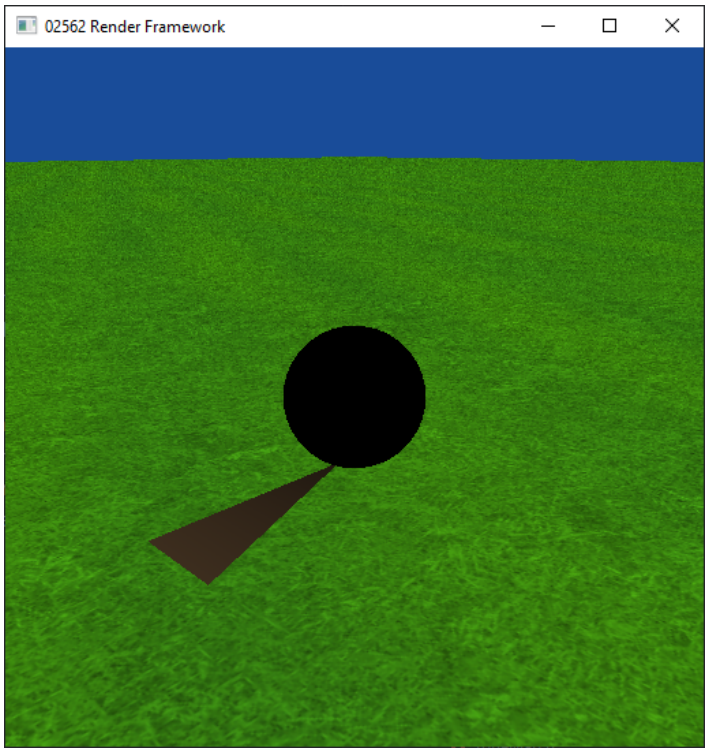
$$i = U + V \times W$$

Before we compute the above formulas it is important that when doing the implementation we remember that the texels are reversed in the texture array in the v-direction. So our implementation in code will look like:

```
float u = texcoord.x;
float v = -texcoord.y; // Revert the vertical axis of the texture array

// The following is done in texture repeat mode

// Texture coordinates in texture space [0,1] x [0,1]
float s = u - floor(u);
float t = v - floor(v);

// Texture coordinates in image space [0,width] x [0,height]
float a = s * width;
float b = t * height;

// Nearest neighbor filtering [0,1] x [0,1]
unsigned  int U = static_cast<unsigned int>(a + 0.5f) % width;
unsigned  int V = static_cast<unsigned  int>(b + 0.5f) % height;

unsigned  int index = U + V * width;

return fdata[index];
```

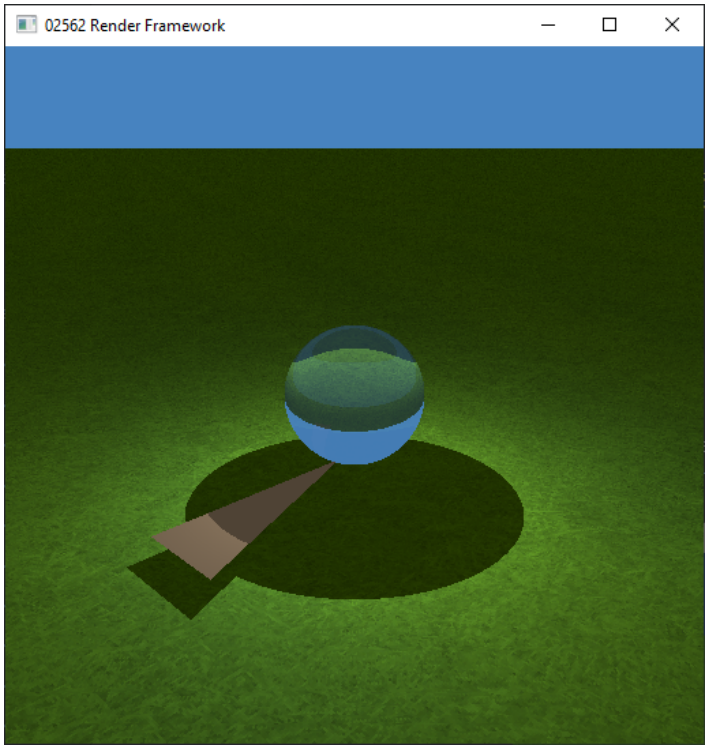
Task 6: Render the Scene and Compare the Results

When rendering the default scene we can really see how far we have come in our implementation of ray tracing rendering. We started off with a flat scene from our view camera as shown below:



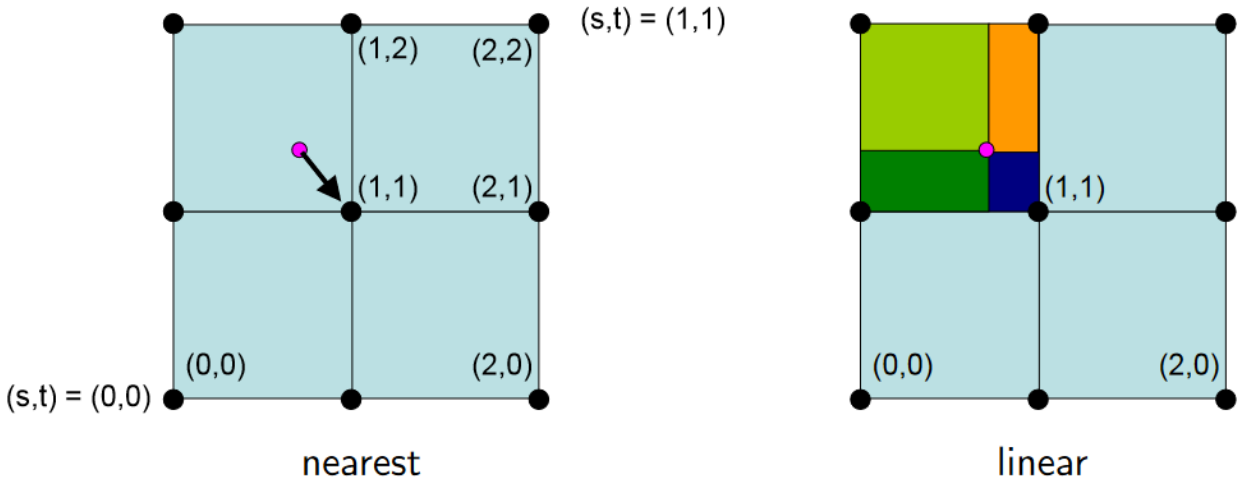
Now we have implemented; textured grass thanks to our plane-ray intercept implementation, we have shading on our objects using the diffuse technique we learned, and we have scene lighting thanks to the Phong illumination model. Not only that we have just implemented texture filtering in the previous task. Texture filtering improves the grass texture by making it blurry by interpolating the neighbor values. This

effect is more obvious on pixels closer to the camera but can also be seen in the smoothing of the plane and background transition. The result of the tasks implemented so far in the lab can be seen below:



Task 7: Implement Bilinear Interpolation

In this next task we are going to implement a different type of texture filtering called bilinear interpolation. This method is called so because instead of taking the index result of the nearest coordinate like in the previously implemented technique. We are now taking the weighted average of texels in two directions (up, down) and (left, right) hence the name bilinear. This concept is better explained by looking at the picture below:



We can find the weights for the interpolation using the following formulas:

$$c_1 = a - U$$

$$c_2 = b - V$$

We also need to modify how we get our (U, V) image index coordinates which are now found using:

$$U = (int)a$$

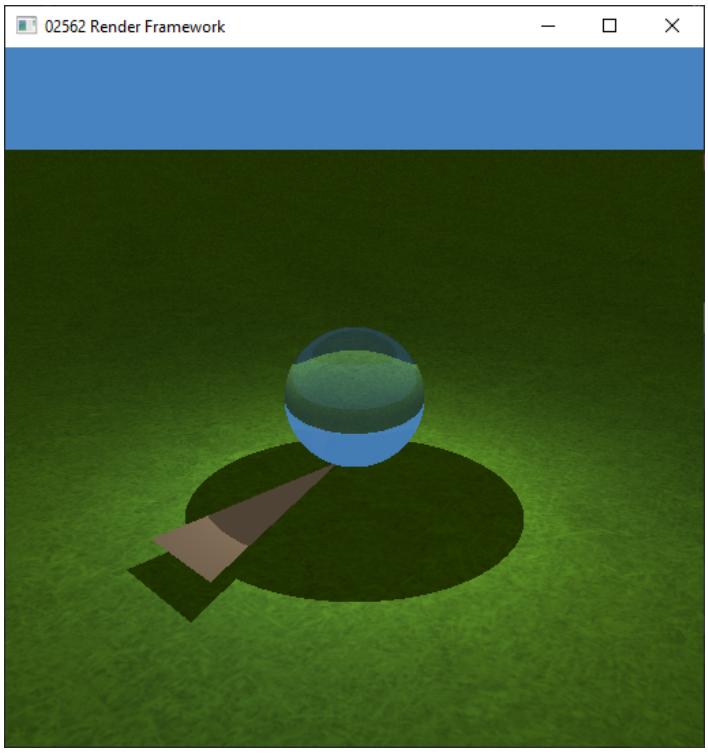
$$V = (int)b$$

With the above variables we can now get the corner indexes using the corner coordinates. These corner indexes can be used in the OptiX function `lerp` which conducts a linear interpolation across two axis using the weight c_1 which we defined above. We then have to interpolate the results of the two axis interpolations using the different weight c_2 . This will result in a bilinear interpolation across two axis. The code implementation for bilinear interpolation can be seen below:

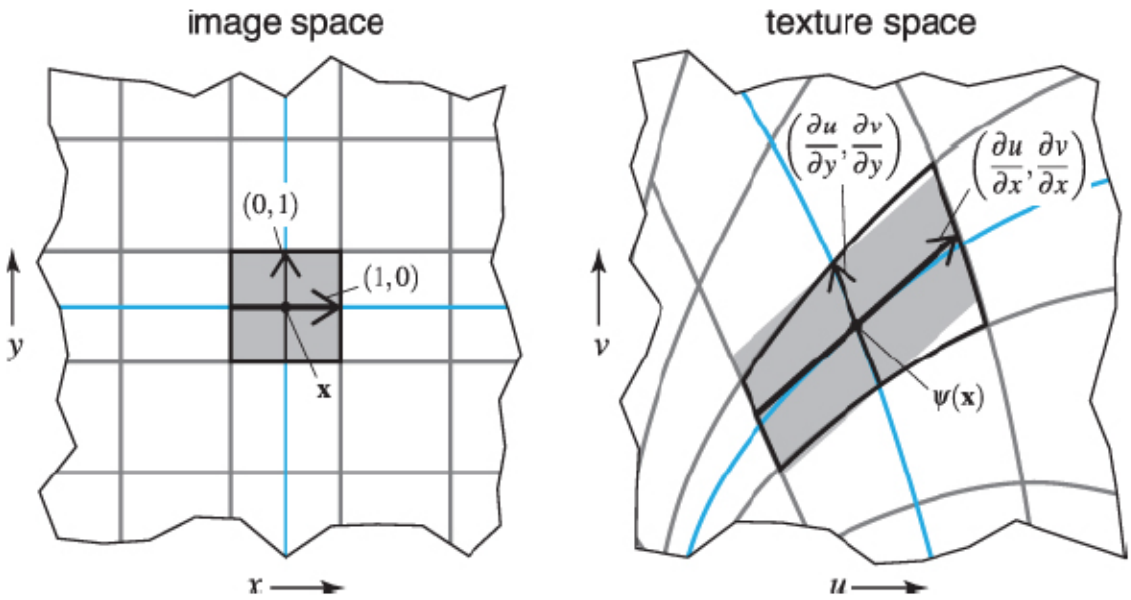
```
float4 Texture::sample_linear(const float3& texcoord) const
{
  if(!fdata)
    return make_float4(0.0f);

  // Toggle comment the following line to switch to nearest neighbor filtering
  //return sample_nearest(texcoord);
    float u = texcoord.x;
    float v = -texcoord.y; // Revert the vertical axis of the texture array

    // The following is done in texture repeat mode
    // Texture coordinates in texture space [0,1] x [0,1]    float s = u - floor(u);
    float t = v - floor(v);

    // Texture coordinates in image space [0,width] x [0,height]
    float a = s * width;
    float b = t * height;

    // Bi-linear filtering [0,1] x [0,1]
    unsigned  int U = static_cast<unsigned int>(a);
    unsigned  int V = static_cast<unsigned  int>(b);

    // Weight factors
    float c1 = a - U;
    float c2 = b - V;

    // Get corner texture coordinates
    unsigned  int U1 = U % width;
    unsigned  int U2 = (U + 1) % width;
    unsigned  int V1 = V % height;
    unsigned  int V2 = (V + 1) % height;

    // Get corner texture indices
    unsigned  int index00 = U1 + V1 * width;
    unsigned  int index10 = U2 + V1 * width;
    unsigned  int index01 = U1 + V2 * width;
    unsigned  int index11 = U2 + V2 * width;

    return lerp(
    lerp(fdata[index00], fdata[index10], c1),
    lerp(fdata[index01], fdata[index11], c1),
    c2);
}
```

Task 8: Comparison of Bilinear and Nearest Neighbor Interpolations

If we compare the results of Bilinear interpolation and the results of nearest neighbor interpolation, we can see a stark difference in results. We see it more noticeable in pixels in the distance, where as in nearest neighbor these pixels were still quite pixelated now in the bilinear interpolation we see them much smoother. The results of the bilinear interpolate implementation can be seen below:

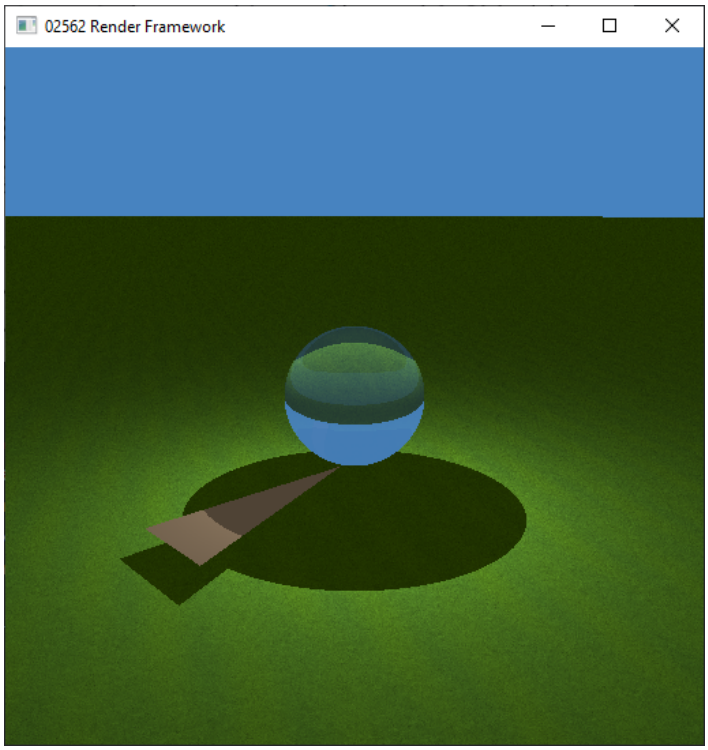


The conversion between image space and texture space we have conducted in the implementation of both interpolation methods can be overviewed in the image from the course book below:



If we refer to the grey color in the image above as the data available for the texture, we can imagine that if the texture data is less than our pixel size, we won't have enough data to draw a pixel. This will cause every pixel to have the same interpolated value, making an image like we had at the beginning of the

course with just a green plane. If the opposite was true and the gray texture data was larger than our pixel size we have more data per pixel so more pixel can have different value leading to less pixelization that we see at the edges of texture data. We can test this theory by changing the scale of our texture, as explained in Task 1, to be 2 instead of 0.2 magnifying how much texture data we have by a factor of 10. In the magnified result below we see that the texture on the plane is much smoother:



Next Lab: Rendering Lab 4

Rendering Lab 4

Lab 4: Meshes and Lighting

Expected due date: 28-09-2022

Lab Purpose:
One of the key strengths of computer graphics techniques is that they work in general. Ray tracing is, for example, well known in numerous fields of research. The distinguishing feature of ray tracers in computer graphics is their ability to efficiently handle nearly arbitrary scenes. This set of exercises helps you load and efficiently render large triangle meshes.

Learning Objectives:

- Accelerate rendering techniques using spatial data structures.
- Implement ray tracing of a triangle mesh (indexed face set).
- Use a BSP tree for efficient space subdivision.
- Interpolate normals and texture coordinates across a triangle

Deliverables:
Renderings of the Cornell box, the Utah teapot, and the Stanford bunny. Also provide the explanations, comparisons, and performance measurements mentioned above. Include relevant code snippets. Please insert all this into your lab journal.

#labnotes #todo

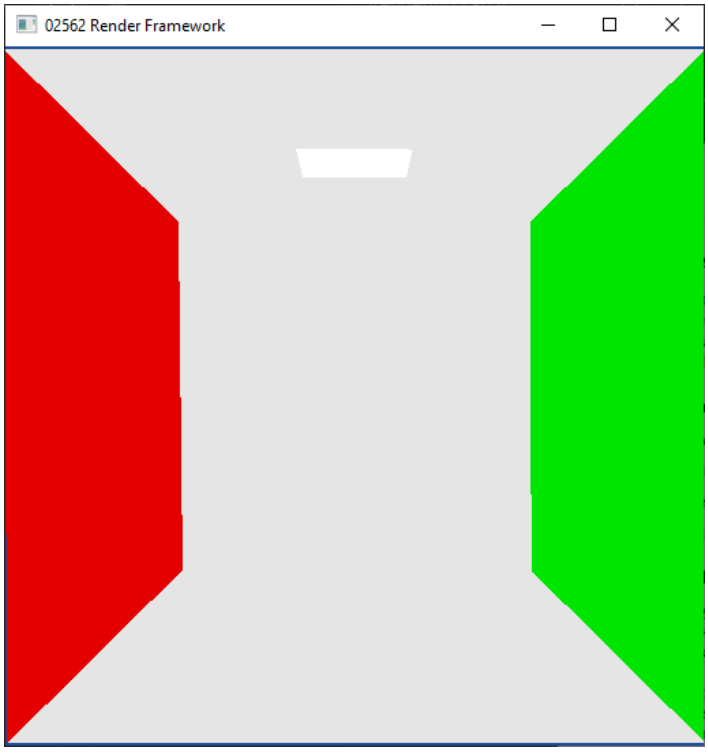
Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Load the Cornell Box

The Cornell Box scene is very popular scene for testing lighting and shading in rendering model implementations like our raytracing approach. As such, we would like to load it into our engine. If we look back at Lab 3 Task 1 we remember that `load_files` function was responsible for loading meshes into the program. The way it does that is by looping over `argc` which is provided by arguments sent into the program. We can provide this argument as a command line argument which in CLion, (the compiler I am using for the course) is done through the Run/Debug configurations and then under "program arguments". Here we can add objects to the scene instead of the default scene by sending in the object path we would like to load. In our case we want to load the Cornell Box and blocks which we do with the following arguments:

`../models/CornellBlocks.obj`
`../models/CornellBox.obj`

The result of loading the two objects can be seen below:



Task 2: Ray-Mesh Triangle Interception

If we try to user or raytracing on the Cornell scene we do not get anything except for the background. This is due to our ray intercept not implemented for triangle meshes. We can implement the interception under `intersect` function in the `TriMesh.cpp` file. The implementation is similar to how we did intercepting of triangles (because meshes are made up of triangles) except now we have vertices declared

in an indexed face set. In fact we can reuse the code we had for the hit information which is why Lab 1 suggested to implement the `intersect_triangle` function within the `Triangle.cpp` file. The code implementation of the `intersect` function in `TriMesh.cpp` can be seen below:

```
// Get the triangle vertices from the geometry object
float3 v0 = geometry.vertex(face.x);
float3 v1 = geometry.vertex(face.y);
float3 v2 = geometry.vertex(face.z);

// Get the triangle intersection variables from the helper function
float3 n;
float t;
float beta;
float gamma;
ifintersect_triangle(r, v0, v1, v2, n, t, beta, gamma) {
    if (beta >= 0 && gamma >= 0 && beta + gamma <= 1) {
        hit.has_hit = true;
        hit.dist = t;
        hit.position = r.origin + t * r.direction;
        hit.geometric_normal = normalize(n);
        hit.shading_normal = normalize(n);
        hit.material = &materials[mat_idx[prim_idx]];
        return true;
    }
}

return false;
```

The results from this task are the same as Task 1 just can be rendered using the raytracing when pressing `r` on the keyboard.

Task 3: Flat Shading

Currently if we try to render our scene using any lighting we cannot see anything this is because the Cornell box uses an area light which we haven't implemented at this point. In order to implement an area light we need to alter the `sample` function in the `AreaLight.cpp` file. Area lights act in a similar manner to point lights however area lights come from a rectangle area made up of four vertices as well as a direction of the light unlike point lights which emit all around. Rays are emitted by an Area light from an area defined by the mesh of that area, usually a rectangle. The area light thus emits rays at every point of the mesh in a direction of a hemisphere. If we start by the basic principle of rendering light given by the equation below:

$$L_o(\vec{w}_o = L_e(\vec{w}_o) + \int f(\vec{w}_i, \vec{w}_o) L_i(\vec{w}_i)(\vec{w}_o \cdot \vec{n}) d\vec{w}_i$$

Which in human speech translates to the light out L_o form a point in the view direction $\vec{w}_o$ is equal to the emissive light in the view direction added to the reflected light in the view direction and every direction. The integral part is saying that we are taking the result of everything between two points and add them up for every point in a hemisphere, multiplying each value by the fractional size of the point's are for the hemisphere. The hemisphere is surrounding the normal of the surface we are looking at. This gets pretty complicated because technically there are an infinite number of points in a hemisphere. If we were to

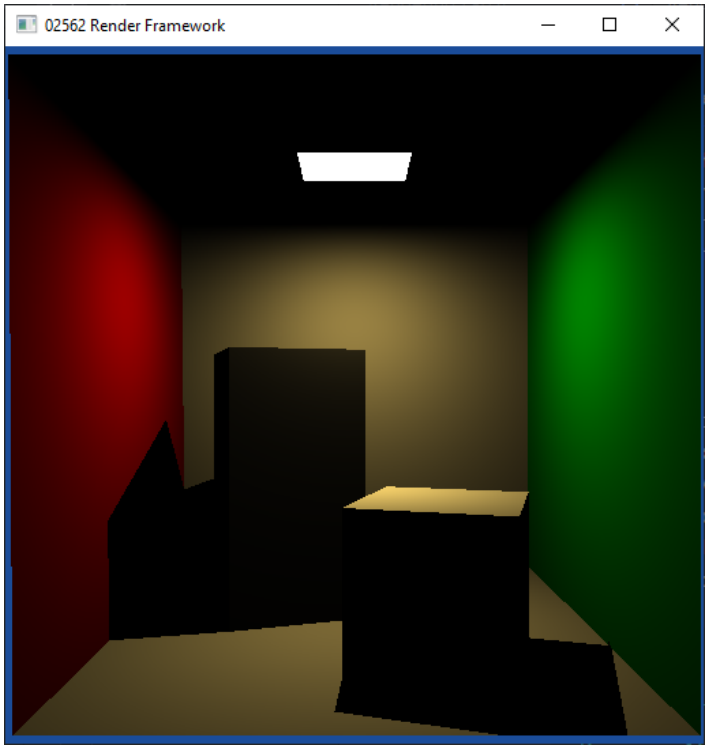
implement a true area light we would need to find out how much light is coming from that direction by multiplying it by the **Bidirectional Reflectance Distribution Factor (BDRF)** which is how much light is reflected towards the view direction of the light coming in from the point on the hemisphere. Instead the course looks into approximating an area light by implementing it like a point light with only one sample. This is done with the nicer formula shown below where the variable t_i indicates the triangle index:

$$L_r = f_r \frac{V}{r^2}(\vec{w_i} \cdot \vec{n}) \sum_{t_i=1}^{N} (-\vec{w_i} \cdot \vec{n_{et_i}}) L_{et_i} A_{et_i}$$

In the formula above V is the visibility which is calculated at the center of the bounding box for an area light approximation. The variable f_r is the diffuse point of the entire scene which is $\frac{p_d}{\pi}$ something that we have implemented in Lab 1 and thus is not visible at the implementation for any of the following tasks. However, for this part of the lab will implement an area light the same way we implemented a point light using; the mesh bounding box's center as the light position, the `get_emission` helper function as the emission light, and the normalized sum of face normals as the face normal, as described by the lab requirements. The `sample` function is shown in the code below:

```
// Use the center of the axis aligned bounding box of the area light mesh as the position.
float3 light_pos = mesh->compute_bbox().center();
float3 normal = normalize(light_pos - pos);
float distance = length(light_pos - pos);
dir = normalize(light_pos - pos);

// Shadow ray cutoff variables
float epsilon = 0.0001f; // 10^-4
float t_max = distance - epsilon; // ||p-x|| - epsilon

// If shadows are enabled, check if the point is in shadow
if (shadows) {
    Ray shadow_ray = Ray(pos, dir, 0.0f, epsilon, t_max);
    HitInfo shadow_hit;
    if (tracer->trace_to_closest(shadow_ray, shadow_hit)) {
        return false;
    }
}

// The normalized sum of the vertex normals can be used as the face normal.
float3 face_normal_sum = make_float3(0.0f);
for(unsigned int i=0; i < normals.no_vertices(); i++){
    face_normal_sum += normals.vertex(i);
}
face_normal_sum = normalize(face_normal_sum);

for (unsigned int i = 0; i < mesh->geometry.no_faces(); i++) {
    // L_r = fr * V * L_e * (w_i dot n) * ((-w_i dot n_e) / r^2)  * dA_e
    L += get_emission(i) * dot(dir, normal) * (dot(-dir, face_normal_sum) / (distance *
distance)) * mesh->face_areas[i];
}

return true;
```

One thing that is missing from this implementation is the shading of the roof which kind of ruins the shading in my personal opinion, I tried fixing this by having the center point being a bit higher but to no success. Fixing this might require the plane interception implementation to be altered so that planes cannot be intercept from behind. The results from the implementation of the area light can be seen in the screenshot below:



Task 4: Loading the Utah Teapot with Directional Lights

In this part we are loading a much more complex mesh object called the "Utah Teapot". This teapot can be loaded in the same way that was done in Task 1. We now also have to implement yet another type of lighting called directional light source. We implement this light source in the `sample` function in the `Directional.cpp` file. A directional light source is a global light source from a direction in the scene. This lighting affects all objects the same where just objects declare their specific lighting variables. This implementation is straightforward as the formula for calculating L_r using a directional light source can be seen below where $\vec{w}_e = -\vec{w}_i$:

$$L_r = f_r V L_e(-\vec{w}_e \cdot \vec{n})$$

From the formula we can see that the f_r as well as the V we have implemented in previous tasks (currently the implementation for V is only possible values of 0 or 1 which is a possible future improvement). So the part we need to implement here for the directional light is the $(-\vec{w}_e \cdot \vec{n})$ and L_e is given. The implementation code can be seen below:

```
// The direction toward the light is opposite to the light direction
dir = -light_dir;

// Shadow ray cutoff variables
float epsilon = 0.0001f; // 10^-4
float t_max = RT_DEFAULT_MAX - epsilon; // cannot be calculated because of the infinite
light source
```

```
// If shadows are enabled, check if the point is in shadow
if (shadows) {
    Ray shadow_ray = Ray(pos, dir, 0.0f, 0.001f, t_max);
    HitInfo shadow_hit;

    if (tracer->trace_to_any(shadow_ray, shadow_hit)) {
        return false;
    }
}

L = emission;

return true;
```

From the result of our new light source we can see that we are having flat shading in our scene because we are using the normal set in our triangle mesh to be that of the geometric normal. This makes the shading not smooth and transition based on these normals. The result of our directional lighting with flat shading is shown below:

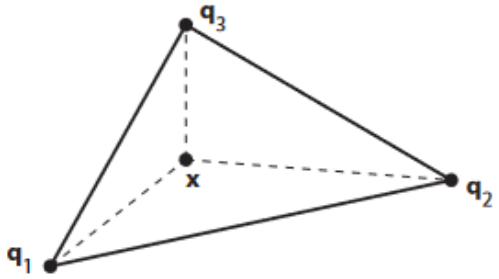
Task 5: Phong Shading

If we want to have better shading we will need to change the `shading_normal` variable we had set in our `intersect` function of the `TriMesh.cpp` file, to instead of being the geometric normal to be an interpolated value from each triangle. We can figure out an interpolated point on a triangle using the barycentric coordinates as weights as shown in the formula below:

Linear interpolation across triangles

- A point $\boldsymbol{x}$ in a triangle is given by a weighted average of the triangle vertices $(\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3)$:

$$\boldsymbol{x} = \alpha \boldsymbol{q}_1 + \beta \boldsymbol{q}_2 + \gamma \boldsymbol{q}_3 \quad , \quad \alpha + \beta + \gamma = 1 \ .$$



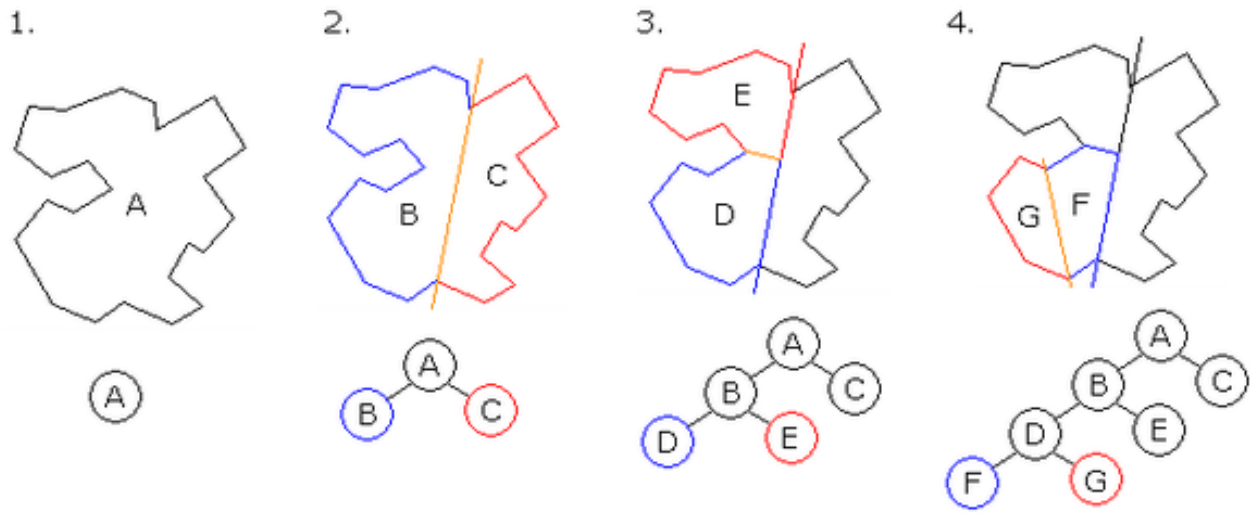
If we go back to Task 2 we had already implemented the barycentric coordinated so we simply have to modify how we decide what shading normals to use. This decision implementation can be seen below:

```
// Get the triangle intersection variables from the helper function
float3 n;
float t;
float beta;
float gamma;
ifintersect_triangle(r, v0, v1, v2, n, t, beta, gamma) {
    if (beta >= 0 && gamma >= 0 && beta + gamma <= 1) {
        hit.has_hit = true;
        hit.dist = t;
        hit.position = r.origin + t * r.direction;
        hit.geometric_normal = normalize(n);
        hit.material = &materials[mat_idx[prim_idx]];

        float alpha = 1 - beta - gamma;

        // If the mesh has vertex normals, use them to compute the shading normal
                if(has_normals()) {
            float3 n0 = normals.vertex(normals.face(prim_idx).x);
            float3 n1 = normals.vertex(normals.face(prim_idx).y);
            float3 n2 = normals.vertex(normals.face(prim_idx).z);
            hit.shading_normal = normalize(alpha * n0 + beta * n1 + gamma * n2);
        } else {
            hit.shading_normal = hit.geometric_normal;
        }

        return true;
    }
}
```

We can also replace the triangle vertices (q_1, q_2, q_3) by vertex texture coordinate to get (u, v) coordinates for texturing. This will be looked into at a later time. For now, below are the results to the task which we can see a much smoother shading result:

Task 6: Axis-Aligned Binary Space Partitioning Tree

The rendering of the raytraced teapot took `65.97` seconds, this of course can vary depending on hardware available to the rendering computer but the problem comes from the way we are handling rendering of our meshes. Currently we are rendering all of the mesh and then looping over every primitive in order to find an intersection through the use of the method we implemented in Lab 1 `closest_hit` in the `Accelerator.cpp` file. This will test every triangle in the mesh at every pixel on the screen. This teapot is made up of 6,320 primitives and our screen size is by default 512 x 512 pixels. This means our loop minimum has to be looping 1,656,750,080 times just to do the intersection tests. A good way to reduce the amount of intersection tests is to make it so that there are less triangles needed to test yet keeping the ones that are relevant to the scene. A way we can do this is by implementing an axis aligned **Binary Space Partitioning Tree (BSP Tree)** also referred to as a **KD Tree**. We can see such implementation of a BSP Tree in the `BspTree.cpp` file. The way that this BSP works is by dividing the primitive objects into a binary tree in a fashion as shown in the figure below:



The way that the BSP knows where to do the division is by using a cost function which samples a defined number of random placements and computes a cost for each of the placements. The accepted plane to do the division is therefore the one with the lowest cost. The cost is determined by the number of objects it has split that subdivision into left and right sides. The way that the tree's depth is determined is by a set of stop criteria which include; a max number of objects per leaf, or a max level reached. The way we use this BSP is by implementing the following two functions within the `BspTree.cpp` file:

`closest_hit`

```
closest_plane(r, hit);

if (intersect_min_max(r)) {
    intersect_node(r, hit, *root);
    return true;
}

return false;
```

`any_hit`

```
if (any_plane(r, hit))
    if (intersect_min_max(r)){
            intersect_node(r, hit, *root);
        return true;
    }

return false;
```

Task 7: Rendering Times Using BSP

Now that we are using spatial subdivision of the model object we are now rendering the same teapot in just `0.053` seconds. This is a whopping `65.92` seconds faster! If we compare the render speed and triangle speed of rendering the teapot and the other models that are available we get the following results:

Model	No of Triangles	Render Time	BST Time
Teapot	6320	65.968	0.054
Boxes	36	0.469	0.065
Bunny	69451	999.958	0.109

From the results table we can see that as the higher triangle count the higher the time it took to render the scene using our looping method. We also saw that using the BST was an improvement factor of ~1000%. Except for one case, the Cornwell boxes. This is due to the scene being so triangle simple consisting of only 36 triangles including the blocks. That means our program is spending more time dividing the scene that is actually gained from just rendering it.

Next Lab: Rendering Lab 5

Rendering Lab 5

Lab 5: Radiometric and Photometric Concepts

Expected due date: 05-10-2022

Lab Purpose:
The purpose of this exercise is to get acquainted with radiometric and photometric concepts.

Learning Objectives:

- Use radiometric concepts to describe how light energy is emitted and received.
- Explain the different radiometric quantities.
- Use solid angles in radiometric calculations.
- Convert between radiometric and photometric units.

Deliverables:
Solutions for Parts 1–7 and, optionally, Parts 8–9. Optional parts give extra credit. Please insert everything into your lab journal.

#labnotes #todo

Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Photons

A small 25 W 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.

The energy of the photon is related to the angular frequency of the wave ω as shown by the formula below:

$$E = H\omega = hf = \frac{hc}{\gamma}$$

$$H = \frac{hc}{\gamma}$$

$$f = 2\pi\omega$$

So for our problem we know:

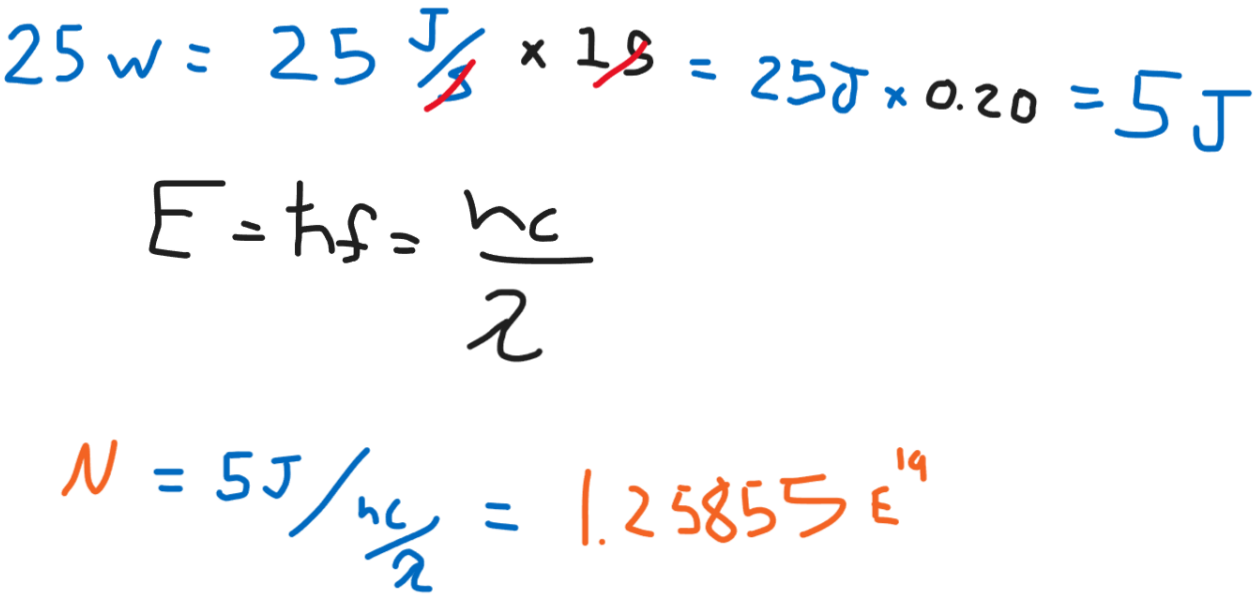
$E = 25$ Watts $= 25 \times 20\% = 25 \times 0.20 = 5$ Watts
$c = 2.9979e^{8}$ m/s
$h = 6.626e^{-34}$
$\gamma = 500$ nm $= 500e^{-9}$ m
$f = 2\pi 500e^{-9}$

So we can rewrite our formula to find the change of base:

$$E = \frac{hc}{\gamma} = \frac{6.626 \times 10^{-34} \times 2.9979 \times 10^{8}}{500 \times 10^{-9}} = 3.97282 \times 10^{-19}$$

$$\frac{5}{3.97282 \times 10^{-19}} = 1.25855 \times 10^{19}$$

So for our problem there are approximately 1.25855×10^{19} photons emitted every second. You can solve this a different way in the screenshot below below showing how watts are just a measurement of J/s. From now on the results will be shown in handwritten screenshots as shown below:



Task 2: Light Bulbs

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) Use W for Watt, J for Joule, m for meter, s for second and sr for steradian.

- Radiant flux
- Radiant intensity
- Radiant exitance
- Emitted energy in 5 minutes

Radian Flux

Radian flux is a straightforward calculation in where we know the formula for Flux is given by the following:

$$\phi = \epsilon P = \epsilon VC$$

Where we are given all the variables assuming that it is ideal conditions the efficiency of the light bulb would be $\epsilon = 100\% = 1.00. The the other variables \$V = 2.4_V$ and $C = 0.7_A$ so the result of the Radian Flux would be as shown in the screenshot below:

Radian Intensity

Now that we have our Flux we can get our Intensity I given the formula below:

$$I = \frac{\phi}{\Omega}$$

For the solid angle Ω we assume that we can get it by following the following formula:

$$\Omega = \frac{A}{r^2}$$

So our intensity is given by the following:

$$I = \frac{\Phi}{\Omega} = \frac{1.68\,W}{A/r^2} = \frac{1.68\,W}{4\pi r^2/r^2} = \frac{1.68\,W}{4\pi\,sr} = 0.13369\,W/sr$$

Radiant Exitance

With Flux calculated we can also now figure out the exitance with the following formula:

$$M = \frac{\phi}{A}$$

Where we find the area of a sphere using the following:

$$A = 4\pi r^2$$

Meaning that we can find our exitance M using the following:

$$A = 4\pi r^2 = 4\pi\left(1\times10^{-2}\right)^2 = 1.2566\times10^{-3}\,m^2$$

$$M = \frac{\Phi}{A} = \frac{1.68\,W}{1.2566\times10^{-3}\,m^2} = 1.3369\times10^{3}\,W/m^2$$

Energy Emitted in 5 Min

Using the formula for Flux we know that we calculated the power which is a function of energy per period of time, which for the previous formulas we used 1 second. We can then figure out what the energy is by using the flux and $t = 5min = 5 \times 60 = 300s$ which would give us the formula and result as shown below:

$$Q_J = \phi(5\times60)_s = 0.13369\,\frac{J/s}{sr}(5\times60)_s = 4.0107\times10^{1}\,J/s$$

Task 3: Irradiance

The light reaching an observer L_i at a distance r is given by the intensity of the light I with the following formula:

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

For the case we are given an observer 1m away at a measurement sensor $A = 6mm$ so then

$$\theta = \cos^{-1}\left(1 - \frac{1}{2\pi}\right) = 0.572 \text{ rad} = 32.77^\circ$$

$$L_i = \frac{\Phi \cos\theta}{4\pi r^2} = \frac{1.68 \text{ J/s} \cos(32.77^\circ)}{4\pi (1\text{m})} = 1.12413 \times 10^{-1} \frac{\text{J/s}}{\text{m}}$$

$$\Phi_i(A, \Omega) = \int_A \int_\Omega R(x_i, \vec{\omega}_i) L_i(x_i, \vec{\omega}_i) \cos\theta \, d\omega_i dA_i = ?$$

$R = 555\text{nm}$? $x_i = 1\text{m}$?

$A = \pi\left(\frac{6\text{mm}}{2}\right)^2 = 2.82743 \times 10^{-5}$ $\vec{\omega}_i = 0,0,0$?

Task 4: Illuminance

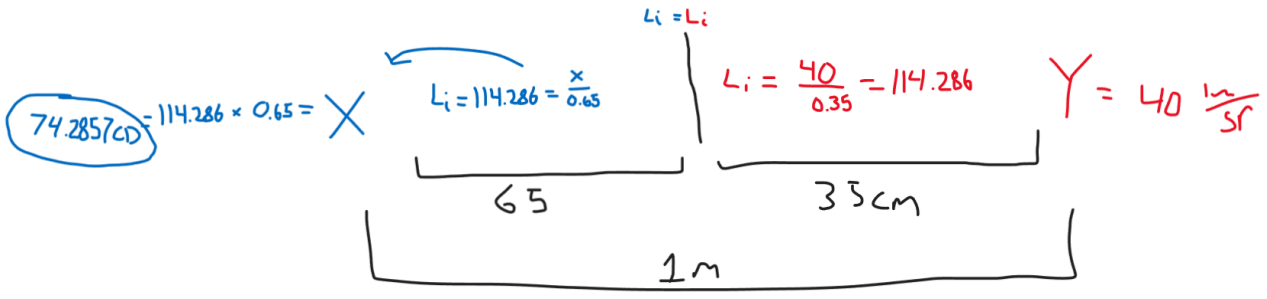
We have calculated the irradiance at a distance before so that is the same formula (seen below as E_R) what we have to test for in this task is the calculation of the Photometric value illuminance. To get Illuminance we multiply the calculated irradiance by a constant of 685 by the value of the luminous efficiency curve at that wave length. So our formula for this task is shown below:

$$E_R = \frac{P}{4\pi r^2} = \frac{200 \times 0.20}{4\pi (2)^2} = 0.795775 \text{ W/m}^2$$

$$E_p = E_R \cdot 685 \cdot V(\lambda) = 0.795775 \cdot 685 \cdot 0.1 = 54.5106 \text{ lm/m}^2$$

Task 5: Intensity

If we know that the screen is being illuminated the same by both light sources and we know; how far away both sources are and the intensity of one of the sources. Then we can figure out the illuminance of the light source in which we know the intensity for by following the picture shown below (the problem is technically solved from right to left):



Task 6:

The formulas for a diffuse light source where given to be the following where ϕ is the emitted light and B is the radiosity:

$$\Phi = L A \pi = 5000 w \times (0.1 \times 0.1) = 50 w$$

$$B = L \pi = 5000 w \pi = 15708 \ W/m^2$$

Task 7:

I'm a bit confused by the question since if an emitter is non diffuse does that mean it's a mirror surface or a black body surface. If the object is a mirror then the amount of flux per solid angle or intensity from a surface in a certain direction would always be what came in if the surface was a perfect mirror. Lambert's radiance we learned in a previous lecture stated how a surface has the same radiance when viewed from any angle therefore the radiant exitance will be the same as the radiance. Then finding the total flux for the light is the same as Task 6:

$$E = \pi 6000 W \times (0.1 \times 0.1) = 188.496 \cos\theta \ W$$

Task 8 (Optional):

Task 9 (Optional):

Next Lab: Rendering Lab 6

Rendering Lab 6

Lab 6: Radiometric and Photometric Calculations

Expected due date: 12-10-2022

Lab Purpose:
If you take a close look at your ray traced results, you might notice that shadow edges and geometry edges are jagged. The purpose of this set of exercises is to remove the jaggies such that you get nice, smooth, anti-aliased edges in your renderings. In addition, we will add indirect illumination using progressive unidirectional path tracing

Learning Objectives:

- Use simple sampling techniques for anti-aliasing and indirect illumination.
- Apply stratified sampling by subdividing pixels and jittering the sample point in each sub-pixel.

- Render anti-aliased edges using ray tracing.
- Do the first step of path tracing: stochastic sub-sampling of pixels.

Deliverables:
Renderings of the default scene using different pixel subdivision levels. Renderings of the Cornell box with anti-aliased edges and including indirect illumination. Include relevant code snippets, a table of render times, and list the number of samples per pixel. Provide the explanations and answers that we ask for in the assignment text. Please insert all this into your lab journal.

#labnotes #todo

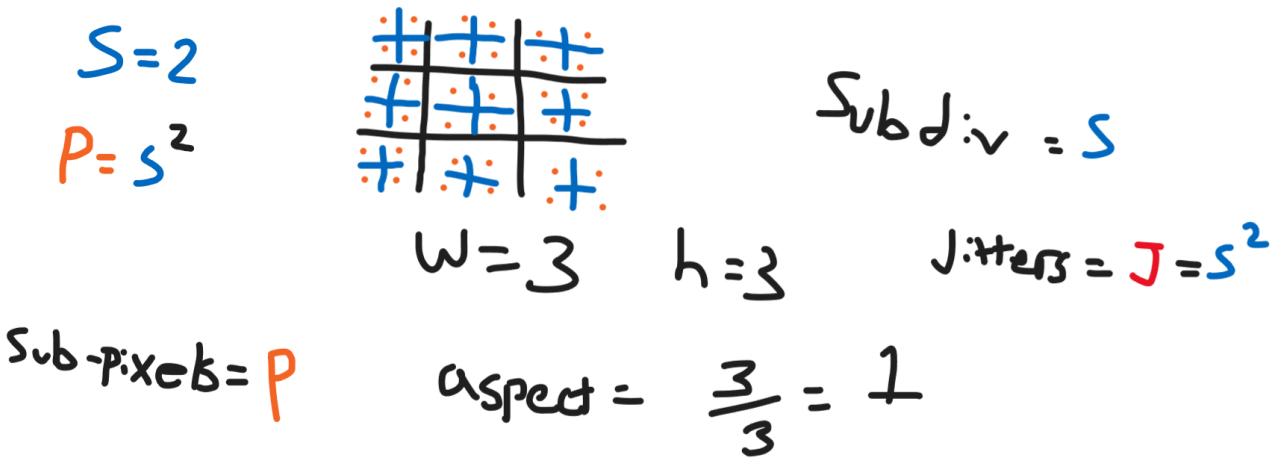
Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Pixel Subdivision

Anti-aliasing is a technique that attempts to minimize the amount of "jaggies" which are artifacts that occur on the edges of most objects and shadows. Currently the way that the framework implements anti-aliasing is by using a sub-division of the pixels on screen in sampling technique called stratified sampling. The way that the framework currently changes the level of sub divisions to do is by user input on the keyboard. From further analysis of the `keyboard` function in the `RenderEngine.cpp` file, we notice that specifically the keys '+' and '-' increase and decrease the number of samples to take per pixel respectively. They do this by calling the `increment_pixel_subdivs` function inside the `RayCaster.cpp` file. This function simply increases a variable and then calls `compute_jitters` which we will look at into more detail in Task 2.

Task 2: Jitters

As explained in Task 1, we reach the function `compute_jitters` whenever we increase/decrease the number of pixel subdivisions. What the framework is doing in this function is looping through each pixel, subdividing it into more pixels, and storing a random 2D vector for a ray position in the `jitters` array. The number of divisions determines how many sub-pixels we get by squaring the sub division level. This works in that each pixel is cut in half on each axis as many times as the sub division level says as explained in the picture below:



Task 3: Path-Tracing

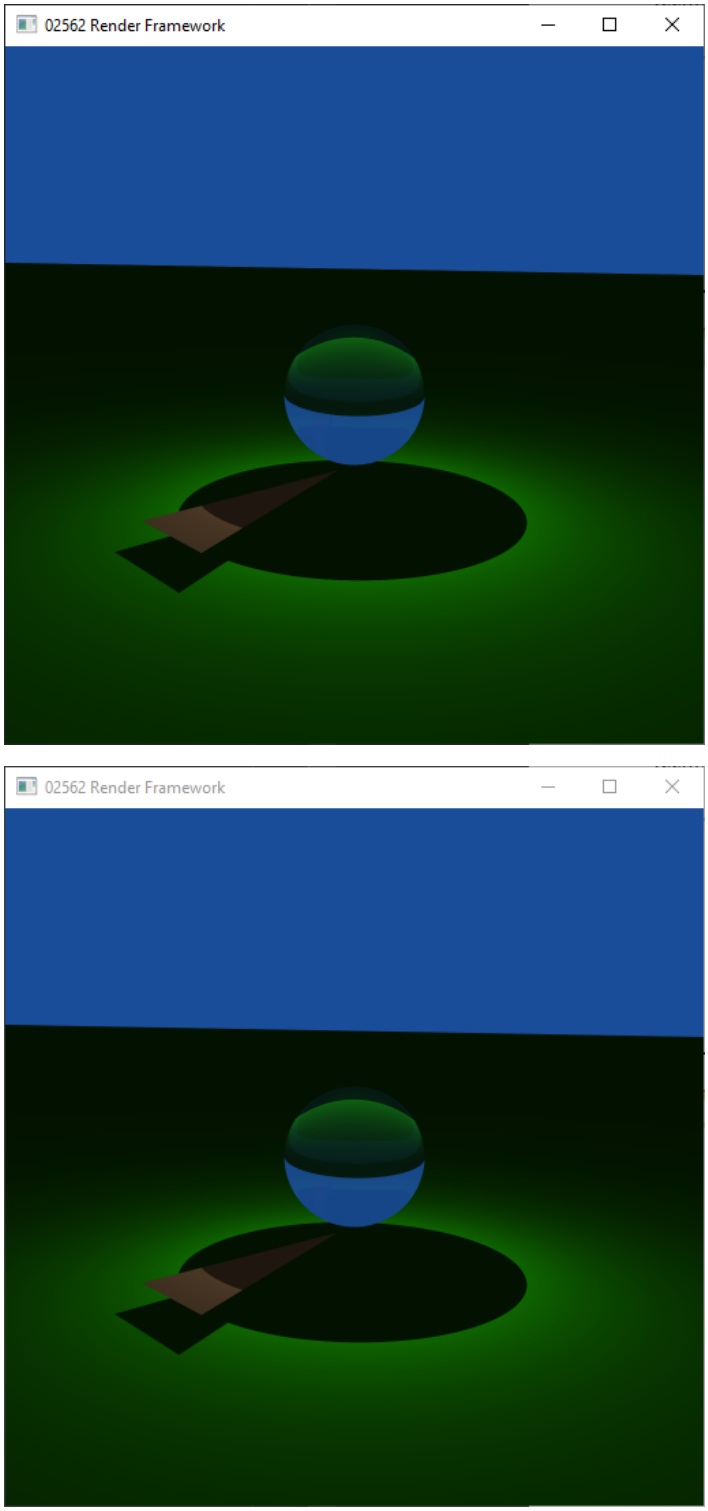
Now that we understand how the framework is subdividing the pixels we can implement the random ray casting at each of the jitter points created in the `jitters` array. We implement the ray casting inside the

`compute_pixel` function in the `RayCaster.cpp` file. We take the pseudo code given in the supplementary material provided on DTU Learn and replace our `compute_pixel` function with the following code:

```
float3 result;
float2 pos = make_float2(x, y) * win_to_ip + lower_left;

// Start of the Antialiasing implementation (Task 3)
// This implementation is based on the code from the supplementary material
for(unsigned int i = 0; i < subdivs; i++) {
    for (int j = 0; j < subdivs; ++j) {
        float2 jitter_pos = jitter[i * subdivs + j];
        Ray ray = scene->get_camera()->get_ray(pos + jitter_pos);
        HitInfo hit_info;
        if (scene->closest_hit(ray, hit_info)) {
            result += scene->get_shader(hit_info)->shade(ray, hit_info);
        } else {
            result += get_background(ray.direction);
        }
    }
}

// Sub-pixels = subdivs * subdivs (subdivs = 1 for no antialiasing)
return result / (subdivs * subdivs);
```

Task 4: Rendering the Default Scene

We now have a scene that can be rendered using different levels of anti-aliasing, by increase/decrease buttons (+/-) on the keyboard. If we render the scene with varying levels of subdivisions we notice that as the number of levels goes up so does the render time (from testing it seems that the increase is exponential), but the nicer the result looks. The highest level of rays-per-pixel that was tested was 1024 which took 46.43 seconds to render. In my opinion it seems that the benefits of the scene are not really worth it after 36 rays per pixel although the scene improves slightly the time to render is more noticeable. Below are the results from 36 rays-per-pixel and 1024-rays-per-pixel:



From the console we can see the render times and ray-per-pixel levels of some of what were tested:

```
Switched to shader number 1
Generating scene display list
Rays per pixel: 1
Raytracing.......... - 0.05 secs
Rays per pixel: 4
.Raytracing......... - 0.196 secs
Rays per pixel: 9
Raytracing.......... - 0.441 secs
Rays per pixel: 16
Raytracing.......... - 0.778 secs
Rays per pixel: 25
Raytracing.......... - 1.259 secs
Rays per pixel: 36
Raytracing.......... - 1.79 secs
Rays per pixel: 49
Raytracing.......... - 2.452 secs
Rays per pixel: 64
Raytracing.......... - 3.127 secs
Rays per pixel: 81
Raytracing.......... - 3.931 secs
Rays per pixel: 100
Raytracing.......... - 4.841 secs
Rays per pixel: 121
Raytracing.......... - 5.851 secs
Rays per pixel: 144
Raytracing..........
 - 6.963 secs
Rays per pixel: 169
Raytracing.......... - 8.18 secs
Rays per pixel: 196
Raytracing.......... - 9.494 secs
```

Task 5: Progressive Path Tracing

Progressive rendering is a useful implementation that will increase the fidelity of our scene without the need for us to manually change and restart the program. If we load up the Cornell scene once more we can test this progressive updates using the `t` key on the keyboard. When the key is pressed we can see our anti-aliasing improve. I personally enjoy the scene at a value of ~30 - 40 samples.

Task 6: Sampling of Indirect Illumination

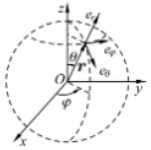
We can also increase the times we sample a weighted hemisphere and thus our indirect illumination of the scene by implementing the `sample_cosine_weighted` function inside the `sampler.h` file. The implementation is guided by the lecture slides which can be seen below:

Sampling a cosine-weighted hemisphere

- We sample ${}_{\perp}\vec{\omega}_{ij}$ on a cosine-weighted hemisphere using spherical coordinates (θ, ϕ) in the local tangent space of surface point with normal $\vec{n} = (n_x, n_y, n_z)$.

$$(\theta, \phi) = (\cos^{-1}\sqrt{1-\xi_1}, 2\pi\xi_2) \quad , \quad \xi_1, \xi_2 \in [0, 1) \quad \text{(random numbers)}\,.$$

- In Euclidian coordinates, the corresponding vector in tangent space is



$${}_{\perp}\vec{\omega}_{ij} = (x, y, z) = (\cos\phi\sin\theta, \sin\phi\sin\theta, \cos\theta)\,.$$

- We can now make a change of basis from tangent space to world space using a formula derived from quaternion rotation [Frisvad 2012, Duff et al. 2017]:

$$\vec{\omega}_{ij} = \left(x \begin{bmatrix} 1 - n_x^2/(1+|n_z|) \\ -n_x n_y/(1+|n_z|) \\ -n_x \operatorname{sgn}(n_z) \end{bmatrix} + y \begin{bmatrix} -n_x n_y/(1+|n_z|) \operatorname{sgn}(n_z) \\ (1 - n_y^2/(1+|n_z|)) \operatorname{sgn}(n_z) \\ -n_y \end{bmatrix} + z \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \right).$$

- Use the function `mt_random_half_open` to get a random number $\xi_1, \xi_2 \in [0, 1)$ and `rotate_to_normal` for the quaternion rotation.

The implementation of the slides into code can be seen below:

```
// Get random numbers
double e1 = mt_random_half_open();
double e2 = mt_random_half_open();

// Calculate new direction as if the z-axis were the normal
double theta = acos(sqrt(1 - e1));
double phi = 2 * M_PI * e2;

optix::float3 dir = spherical_direction(sin(theta), cos(theta), phi);

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

Task 7: Path traced Indirect Illumination

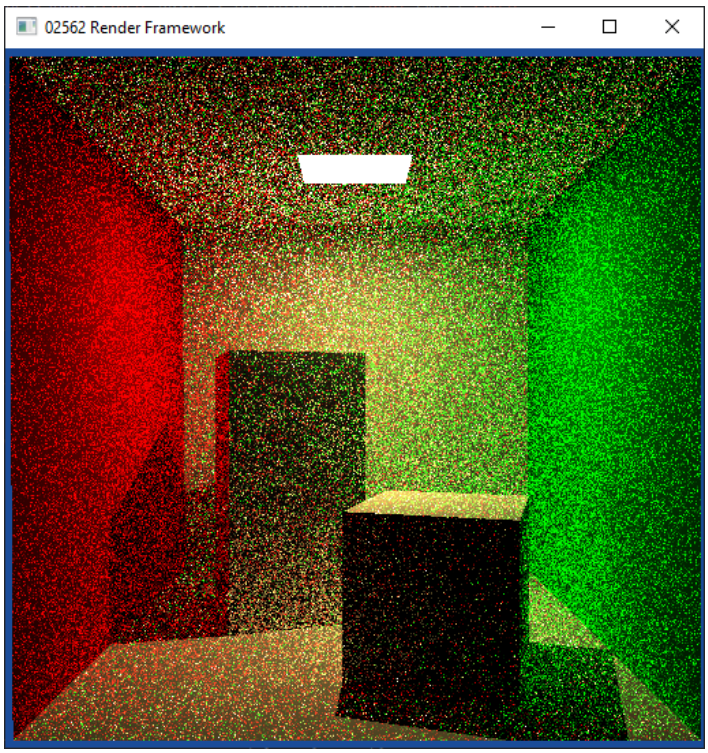
We now have to implement indirect Illumination to see the results of our sampling implementation. In order to do this we have to implement the `shade` function in the `MCGlossy.cpp` file and then press `3` on the keyboard to apply this shader. Here we want to start from our `Glossy.cpp` implementation and integrate the Monte Carlo sampling to it. The way that Monte Carlo is integrated is simply by casting a new path tracing ray and seeing if it has hit an object. If it has hit an object we shade the new ray and add it to the result. Other wise the original ray is the color for the object diffusion. According to the estimation formula we were provided in the lecture:

$$L_{indirect}, N = \rho_d(x)\frac{1}{N}\sum_{j=1}^{N} L_i(x, \vec{w_{ij}})$$

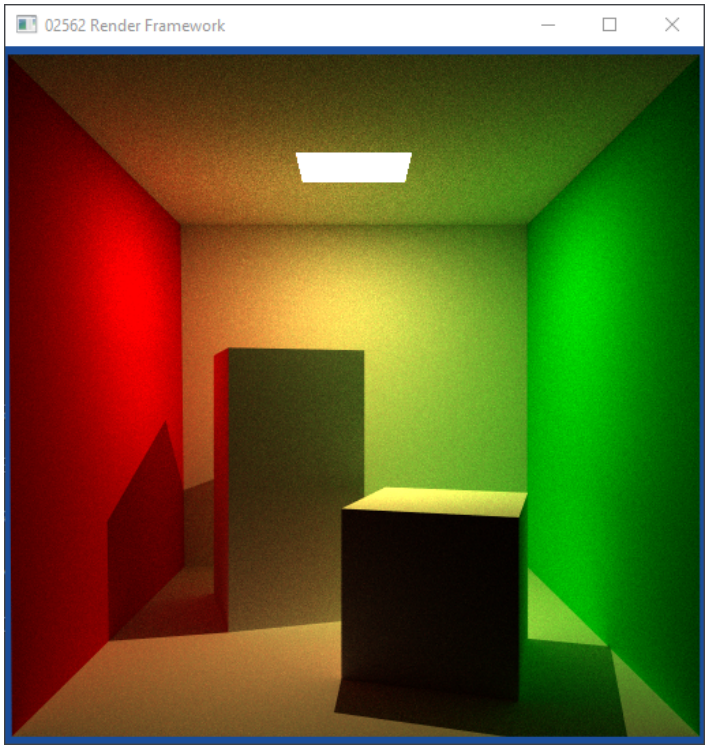
Therefore our code implementation would be:

```
// Step 1: chose a new ray direction
// Trance a new ray to see if it hit anything
Ray new_ray = Ray(hit.position, sample_cosine_weighted(hit.shading_normal), 0, 0.001f,
RT_DEFAULT_MAX);
HitInfo new_hit;

// Step 2: Trace ray to find point of intersection with the nearest surface
if (tracer->trace_to_closest(new_ray, new_hit)) {
    new_hit.trace_depth++;
    result += shade_new_ray(new_ray, new_hit, false);
}

return rho_d * result + Phong::shade(r, hit, emit);;
```

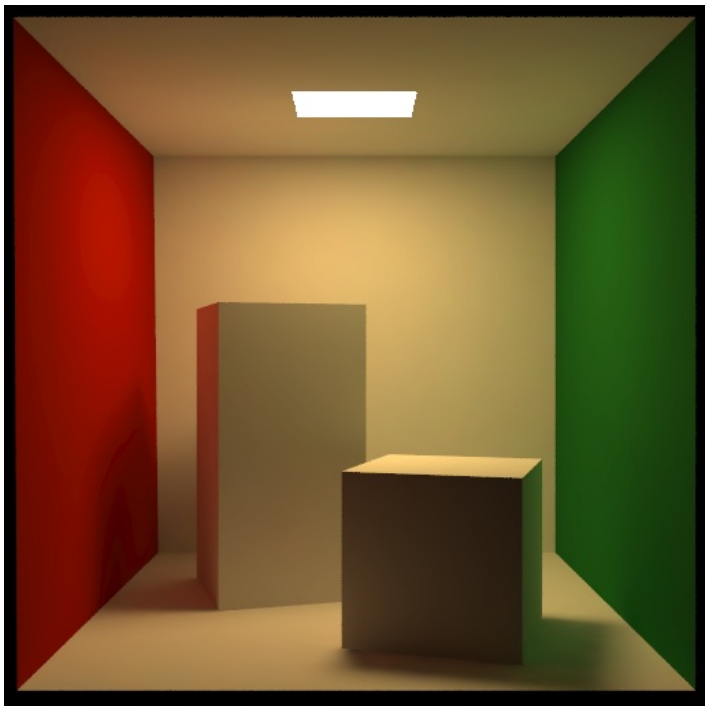
We then get a rendered image that gets progressively better as the number of sample of the hemisphere increase. The image goes from the results shown below with the initial sample:



To the rendered results after ~15 minutes of sampling:



Which if we compare the results of 15 min rendering to that of the Cornell preview of their scene, our rendering here is quite close. Keep in mind that their image has a different background color and thus a different border color as the framework default background color is "Cornflower Blue" while in the Cornell scene it is black. Our implementation is a lot more shiny which means that somewhere along the implementation there must have been some error in our glossy materials. Also our shadows are still implemented as hard shadows while Cornell uses soft shadow transitions. The Cornell Preview is shown in the image below:



Next Lab: Rendering Lab 7

Rendering Lab 7

Lab 7: Soft Shadows and Environment Mapping

Expected due date: 26-10-2022

Lab Purpose:
We will add some final touches to make the renderer fully capable of rendering path traced images and using 360 degrees photographs for environment lighting. This addition of the ability to render soft shadows makes the renderer more production relevant.

Learning Objectives:

- Monte Carlo evaluation of direct illumination
- Sample points on a triangle mesh.
- Using a panoramic texture to insert a background environment map.
- Render soft shadows including the effects of environment lighting

Deliverables:
Cornell box image with blocks that cast soft shadows, and an image with mirror and glass balls instead of the blocks. An image of a bunny or a teapot placed in a photographed environment where it casts soft shadows onto the ground. Please insert code snippets and images into your lab journal.

#labnotes #todo

Useful Links:
Course Book
|Previous Lab
Lab Worksheet

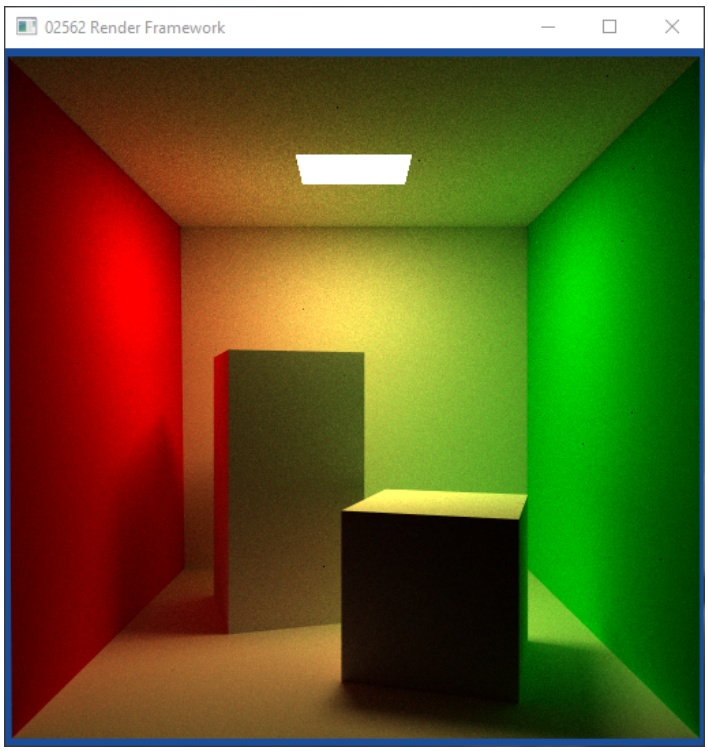
Task 1: Implementing a True Area Light

In our previous labs we have implemented and estimation of Area lights in order to get the scene rending as we had not implemented path tracing. Now that we have path tracing of a hemisphere we can use that to implement a true area light with soft shadows. We do this by changing our implementation of `sample` and `shade` functions in `AreaLight.cpp` and `Lambertian.cpp` files respectively. In order to get the implementation of an Area Light we have to follow three steps. First we take a random triangle on the mesh of the light. Then we sample random points on the hemisphere of that triangle and move to a different triangle. We repeat these steps until we have a result that we are satisfied with. Since we have implemented the sampling of the hemisphere we simply have to implement the taking of a random point on the triangle which we implement with the interpolation of a point to be consistent with smooth shading, although this might not matter as the sampling of a hemisphere should take care of this but a TA told me to do it. The implementation of the Area Light is shown below:

```
// Get random triangle from the light mesh
double rand_index = mt_random_half_open() * mesh->geometry.no_faces();
uint3 rand_face = mesh->geometry.face(rand_index);

double xi_1 = mt_random_half_open();
double xi_2 = mt_random_half_open();

float sqrt_xi_1 = sqrt(xi_1);
float u = 1 - sqrt_xi_1;
```

```
float v = (1 - xi_2) * sqrt_xi_1;
float w = xi_2 * sqrt_xi_1;

// Interpolate across the face triangle for smooth shading
float3 q0 = mesh->geometry.vertex(rand_face.x),
        q1 = mesh->geometry.vertex(rand_face.y),
        q2 = mesh->geometry.vertex(rand_face.z);

// Setup light ray source
float3 light_pos = q0 * u + q1 * v + q2 * w;;
float3 const temp_dir = light_pos - pos;
float const dir_dot = dot(temp_dir, temp_dir);
float const distance = sqrt(dir_dot);
dir = temp_dir / distance;

// Shadow ray cutoff variables
float epsilon = 0.0001f; // 10^-4zz
float t_max = distance - epsilon; // ||p-x|| - epsilon

// This is also calculating V otherwise known as the visibility
// If shadows are enabled, check if the point is in shadow
if (shadows) {
    Ray shadow_ray = Ray(pos, dir, 0.0f, epsilon, t_max);
    HitInfo shadow_hit;
    if (tracer->trace_to_any(shadow_ray, shadow_hit)) {
        return false;
    }
}

for (unsigned int i = 0; i < mesh->geometry.no_faces(); i++) {
    L += dot(-dir, normalize(mesh->normals.vertex(i))) / (distance * distance) *
get_emission(i) * mesh->face_areas[i];;
}

return true;
```

Below is the result of the lighting after letting the sampling happen for ~15 min, which indicates that smooth shadows are indeed happening:



We can use the Cornell scene with instead of blocks having a mirror ball (on the left) and a glass ball (on the right) in the scene to really test our raytracing engine as is the purpose of the Cornell box standard. We do this by loading the `../models/CornellBox.obj` along side with the mirror ball `../models/CornellLeftSphere.obj` and the glass ball `../models/CornellRightSphere.obj`. The results of this scene could show that our render engine can handle caustic illumination which is when light gets refracted/reflected, creating a distorted lighting where the ray ends up. This testing was skipped as loading the balls in and doing the path tracing was taking too much time.

Task 2: Changing Light Probes and Environment Mapping

IN this task we are going to explore how the render engine handles environment mapping by using a new HDR light probe which can be downloaded from https://polyhaven.com/. I decided to use the light studio 6 HDR as it looked like an adequate place for the Utah teapot. We load the HDR by pointing to it in the `RenderEngine.cpp` file under the variable `bgtex_filename`. This will load the HDR file by checking if the file name has a `hdr` file extension and using the `load_hdr` function which sets the background texture to the HDR. We then have to turn off the default light and change the image resolution by changing the corresponding variable(s) in the constructor function for the `RenderEngine.cpp` file. The resolution of the HDR is 4096 x 2048 but I'm not sure that's what the resolution we're supposed to place here is.

Task 3: Saving the Camera Configuration

Right now our teapot and background are in a weird point of view. The background is way too zoomed in and the teapot is huge. Instead we want to setup our camera so that the background and teapot are in a nice place we do this by zooming out the background (by changing the FOV) by pressing `shift+z` on the keyboard. We also click the mouse wheel down and drag to change the size of the teapot. Once we are satisfied with the result we can save the camera settings by pressing `shift+S` on the keyboard then next time we launch the program we hit `shift+L` on the keyboard to load the camera values.

Task 4: Implementing a Holdout Shader

Right now if we render the scene the teapot is black due to no shading. We want to implement a shading technique in where the teapot can be affected by it's environment map and vice-versa. This technique is known as a Holdout shader which we can implement in the `shade` function of the `Holdout.cpp` file. This shader calculates ambient occlusion by sampling with the trusty hemisphere
over each surface point of the object and making the background a distant ambient area light. We need an object to shade to project the shadows so we implement that first by uncommenting the Panoramic Light code in the `init_tracer` function of the `RenderEngine.cpp` file. As for the implementation of the Holdout shader. The job of the holdout shader is to calculate ambient lighting which is simply a float. The ambient light is calculated by once an object is it hit will sample the hemisphere for the number of specified samples and darken the pixel if the sampled ray hits anything. The render is taking too much time to accomplish so instead of the teapot I will be using the Cornell Blocks to test the implementation. The formula used for the implementation can be seen below:

$$L_N = L_{env}(\vec{w}\frac{1}{N}\sum_{j=1}^{N} V\vec{w_j'})$$

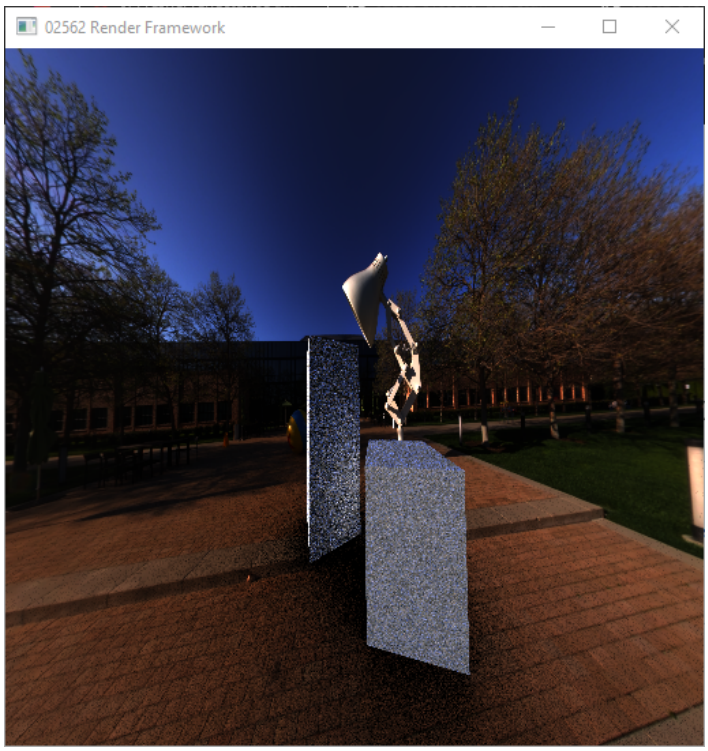
Where:

$$\vec{w_j'} = \frac{x_{l,j} - x}{||x_{l,j} - x||}$$

We therefore get the code implementation of:

```
float3 L_env = tracer->get_background(r.direction);
float3 x = hit.position;

for (int i = 0; i < samples; ++i) {
    float3 dir = sample_cosine_weighted(hit.shading_normal);
    Ray shadow_ray = Ray(hit.position, dir, 0, 0.001f, RT_DEFAULT_MAX);
    HitInfo shadow_hit = HitInfo();
    if(tracer->trace_to_closest(shadow_ray, shadow_hit)) {
        ambient -= 1.0f;
    } else {
         ambient += 1.0f;
    }
}

ambient /= samples;

return ambient * L_env;
```

We then get results in where the shading on the objects is correct but shadows are not being portrayed correctly to the ground. Below are the results of rendering for about ~30 min:



Next Lab: Rendering Lab 8

Rendering Lab 8

Lab 8: Fresnel Reflectance and Volume Shading

Expected due date: 02-11-2022

Lab Purpose:
In previous exercises, we assumed that the reflection from a transparent surface such as glass is always 10%. We also assumed that all light is reflected from or transmitted through a transparent object. This is an idealized model. In reality, the reflectance (the amount of reflection) depends on the angle of incidence of the ray and some light is lost due to absorption. The transmittance (the amount of light that is not absorbed) depends on the distance that the ray travels through the medium. In this set of exercises, we include physically based Fresnel reflectance and absorption in the ray tracing framework

Learning Objectives:

- Implement shaders for rendering transparent and glossy objects.
- Simulate the angle-dependency of reflectance as it appears in transparent and glossy objects.
- Simulate absorption of light as a function of the distance that a ray moves through a medium.
- Use Fresnel's equations for reflection and Bouguer's law of exponential attenuation.

Deliverables:
Renderings of the default scene (e.g. using Fresnel reflectance with highlights, with and without absorption). Compare to previous results. Include relevant code snippets. Please insert all this into your lab journal.

#labnotes #todo

Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Fresnel Reflectance

In previous worksheets we were assuming a static 10% reflectance from a transparent object. In this exercise we will implement a more realistic approach of reflection in where the amount of reflection depends on the angle of incidence. To do this we will be using a technique called **Fresnel Reflectance** and implement it in the `fresnel.h` file. We will be using the `trace_refracted` with the available parameter `R` at the bottom of the `RayTracer.cpp` to send in the argument for returning reflectance. We will have to revert back to using the default scene by setting the render resolution back to 512 x 512 and the `bgtex_filename` back to empty as well as setting the `use_defult_light` back to true. Now that we have the default scene back we can try the implementation of the Fresnel Reflectance using the following formulas:

$$r_{\perp} = \frac{n_i \cos\theta_i - n_t \cos\theta_i}{n_i \cos\theta_i + n_t \cos\theta_t}$$

$$r_{||} = \frac{n_t \cos\theta_i - n_i \cos\theta_t}{n_t \cos\theta_i + n_i \cos\theta_t}$$

$$R = \frac{1}{2}(|r_{\perp}|^2 + |r_{||}|^2)$$

where n_i is the refractive index of the medium from which the incident ray reaches the surface, n_t is the refractive index of the medium that the ray transmits into, θ_i is the angle of incidence, and θ_t is the angle of refraction. Thus the `fresnel_R` function will be:

```
float r_s = fresnel_r_s(cos_theta1, cos_theta2, ior1, ior2);
float r_p = fresnel_r_p(cos_theta1, cos_theta2, ior1, ior2);
float r_s2 = r_s * r_s;
float r_p2 = r_p * r_p;
return (r_s2 + r_p2) / 2.0f;
```

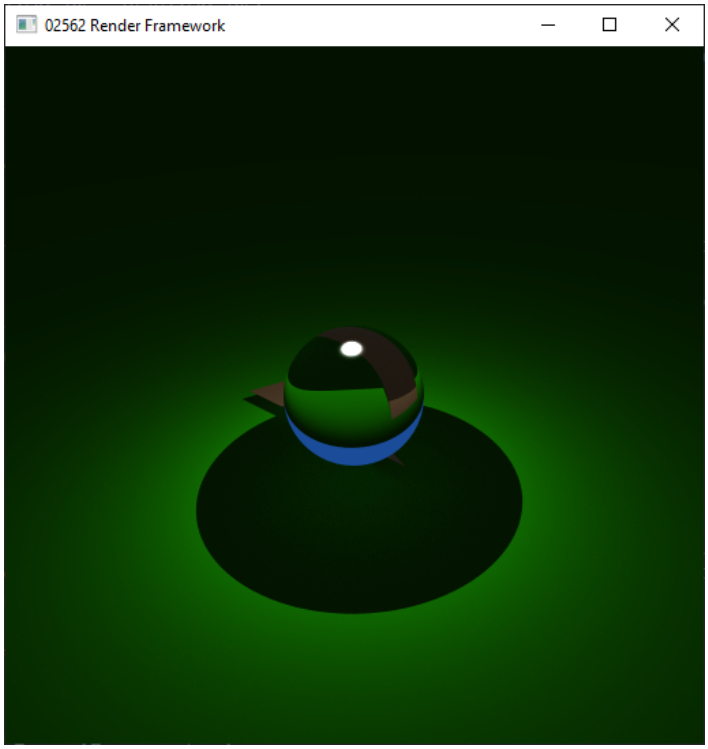
This uses the two helper function in the same header file in order to get the perpendicular polarized component and the parallel polarized component of the Fresnel reflectance as shown below:

```
inline float fresnel_r_s(float cos_theta1, float cos_theta2, float ior1, float ior2) {
    // Compute the perpendicularly polarized component of the Fresnel reflectance
    float top = ior1 * cos_theta1 - ior2 * cos_theta2;
    float bot = ior1 * cos_theta1 + ior2 * cos_theta2;
    float Rs = top / bot;
    return Rs;
}

inline float fresnel_r_p(float cos_theta1, float cos_theta2, float ior1, float ior2) {
    // Compute the parallelly polarized component of the Fresnel reflectance
    float top = ior2 * cos_theta1 - ior1 * cos_theta2;
    float bot = ior2 * cos_theta1 + ior1 * cos_theta2;
    float Rp = top / bot;
    return Rp;
}
```

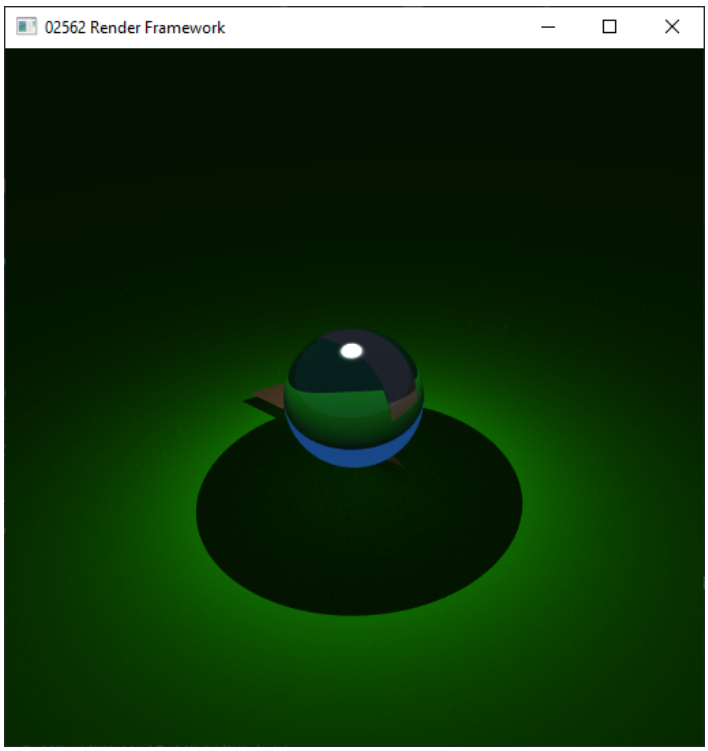
Then in order to test it we use the `trace_refracted` function as shown below:

```
float3 normal;
out_hit.ray_ior = get_ior_out(in, in_hit, normal);

// Test for full internal reflection (return false if it occurs)
if (refract(out.direction, in.direction, normal, out_hit.ray_ior / in_hit.ray_ior)) {
    out.origin = in_hit.position;
    out.tmin = in.tmin;
    out.tmax = RT_DEFAULT_MAX;
    out.ray_type = in.ray_type;
    out_hit.trace_depth = in_hit.trace_depth + 1;
    float cos_theta_i = dot(-in.direction, in_hit.shading_normal);
    float cos_theta_t = dot(out.direction, -in_hit.shading_normal);

    R = fresnel_R(cos_theta_i, cos_theta_t, in_hit.ray_ior, out_hit.ray_ior);

    return trace_to_closest(out, out_hit);
} else {
    R = 1.0f;
    return out_hit.has_hit;
}
```

If we compare the results from when we had a transparent ball in the `Glossy.cpp` as shown below:



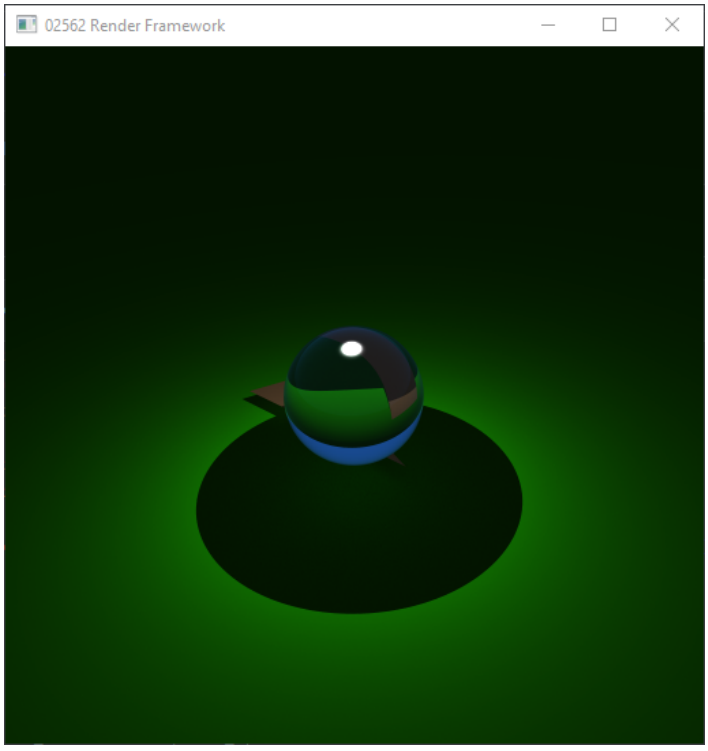
To the results when we have a static R of 10% as shown below:



We would get washed out colors on the other side of the ball regardless of where the camera was facing. The result would look good but it is not how a real glass ball would act in where only the edges should be washed out. The diffusion of the colors is because the sphere is reflecting the sky.

Task 2: Compare Fresnel Reflectance

If we now use the R calculated by the newly implemented Fresnel Reflectance formulas we can see the difference in the washing out of the colors. We are now getting a sphere that changes the transparency of the refracted image depending on what angle it is viewed from. We can see from the picture below that towards the edges of the sphere the reflectance is more pronounced than towards the center of the sphere:



Task 3: Absorption Using Bouguer's Law

Bouguer's law of exponential attenuation is used for computing the transmittance T_r of a light ray that moves through a medium a certain distance s. The formula for Bouguer's Law is shown below:

$$L = L_o T_r = L_o \, exp(-\sigma_t s)$$

Where σ_t is called the extinction coefficient. In a transparent medium, the extinction coefficient equals the absorption coefficient ($\sigma_t = \sigma_a$). We can let the user specify the absorption coefficient by setting the diffuse reflectance ρ_d using the following formula making it more intuitive to set:

$$\sigma_a = \frac{1}{\rho_d} - 1$$

We can then implement Bouguer's Law in the function `shade` and `get_transmittance` in the `Volume.cpp` file. The implementation can be seen in the code below:

```
float3 Volume::shade(const Ray &r, HitInfo &hit, bool emit) const {
    // If inside the volume, Find the direct transmission through the volume by using
    // the transmittance to modify the result from the Transparent shader.

    // L = L_o * T = L_o * exp(-sigma_t * d)
    float3 sigma = get_transmittance(hit);
    float3 L_o = Transparent::shade(r, hit, emit);

    // If inside
    if(dot(hit.geometric_normal, r.direction) > 0.0f) {
        float3 absorption = -sigma * hit.dist; // In transparent, sigma_t = sigma_a
        return L_o * expf(absorption);
    }

    return L_o;
}

float3 Volume::get_transmittance(const HitInfo &hit) const {

    float3 transmittance = make_float3(1.0f);

    if (hit.material) {
    // Compute and return the transmittance using the diffuse reflectance of the material.
    // Diffuse reflectance rho_d does not make sense for a specular material, so we can use
    // this material property as an absorption coefficient. Since absorption has an effect
    // opposite that of reflection, using 1/rho_d-1 makes it more intuitive for the user.
        float3 rho_d = make_float3(hit.material->diffuse[0], hit.material->diffuse[1],
hit.material->diffuse[2]);
        float3 sigma = (1.0f / rho_d) - 1.0f; // Extinction coefficient
        transmittance = sigma;
    }

    return transmittance;
}
```

Task 4: Changing the Material of the Glass Ball

We can now add absorption to the glass ball by opening the file `default_scene.mtl` and setting the diffuse reflectance K_d of the glass material to the color red (because that's what was shown). We also need to set the `illum` parameter to 12 as this will ensure that the framework calls the volume shader as shown in the code below:

```
newmtl glass
Ka 0.0 0.0 0.0
Kd 1.0 0.0 0.0
Ks 0.3 0.3 0.3
Ns 42
Ni 1.5
illum 12
```

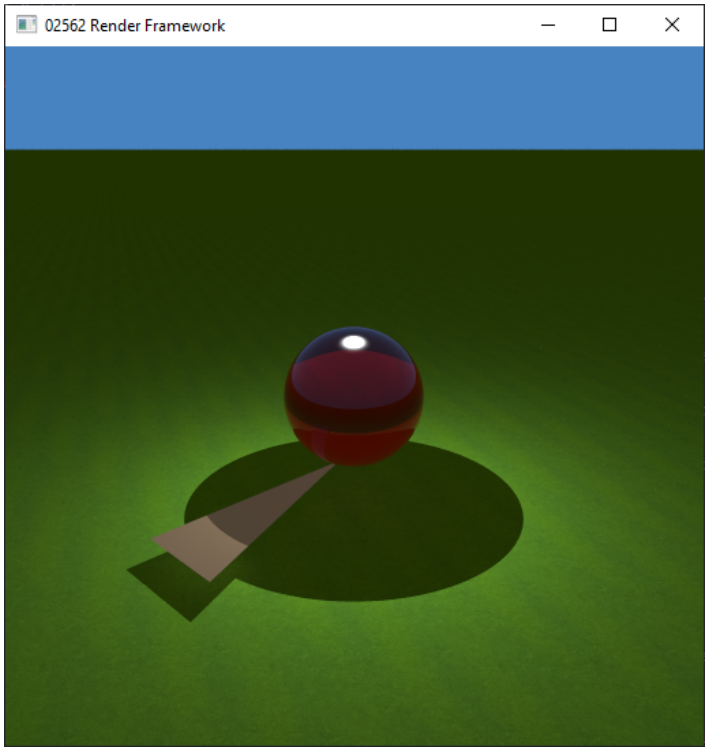
Task 5: Testing the Implementation

We can now test the scene by implementing the `shade` function in the `GlossyVolume.cpp` file. The implementation of this shader should be a combination of the glossy shader and a Phong shader to include the highlight. This implementation will just take the results of the Phong shader without the Lambertian ρ_d and attach the results of the Volume shading. The stitched code can be seen below:

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

float3 result = Volume::shade(r, hit, emit);
float3 reflection = make_float3(0.0f, 0.0f, 0.0f);
float3 light_dir = make_float3(0.0f);
float3 L_i = make_float3(0.0f);
float3 radiance = make_float3(0.0f, 0.0f, 0.0f);

float spec = 0.0f;

// Loop through all the lights in the scene
for(auto light: lights){
    // Sample the light
    if(light->sample(hit.position, light_dir, L_i)){
        // Calculate the reflection vector
        reflection = reflect(-light_dir, hit.shading_normal);

        // Calculate the dot product between the reflection vector and the ray direction
        spec = dot(reflection, -r.direction);
        if(spec > 0.0001f){
            radiance += ( rho_d * M_1_PIf
                        + rho_s * ((M_1_PIf * (s + 2) ) / 2)
                          * pow(dot(-r.direction, reflection), spec)
                       ) * L_i * dot(r.direction, hit.shading_normal);
            result += rho_s * L_i * pow(spec, s);
        }
    }
}
```

```
return  result;
```

Then rendering the scene shows the following result when the gamma is increased to a visible image:



Next Lab: Rendering Lab 9

Rendering Lab 9

Lab 9: Photon Mapping

Expected due date: 09-11-2022

Lab Purpose:
Light that goes through one or several reflections and refractions before reaching a diffuse surface is referred to as caustics. The smaller the light source, the harder it is to find caustics by chance in unidirectional path tracing. If we have idealized perfectly specular materials and use an idealized singular source, such as a point light or a directional light, the chance of finding caustics is zero. In this worksheet, we will use photon mapping to include caustics in our renderings

Learning Objectives:

- Implement photon mapping.
- Use rejection sampling to emit photons from a point light in random directions.
- Transform irradiance to reflected radiance.
- Render caustics.

Deliverables:
Renderings of the default scene (e.g. caustics illumination only and the complete result with and without absorption) and a rendering of the photons in the caustics photon map as dots. Include relevant code snippets and render settings: number of photons in the map, number of photons in each radiance estimate, and number of samples per pixel. Please insert all this into your lab journal.

#labnotes #todo

Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Emitting Photons

One last course guided addition we will make to the rendering engine is the mapping of photons traveling through our glass ball in order to shine a spot on the ground. We will use the map that is created by the photons hitting a diffuse surface by storing it in a map file. First we need to emit these photons from a point light by implementing the `emit` function of the `PointLight.cpp` file. The implementation is done with reference to the lecture slides as shown below:

```
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);
tracer->trace_to_closest(r, hit);

if (hit.has_hit) {
    Phi = intensity * 4 * M_PIf;
    return true;
}

return false;
```

Now that our point light is emitting the photons we can implement the the shooting of these photons by editing the `trace_particle` function of the `ParticleTracer.cpp` file, by calling `light->emit` which we had just implemented. The emit code is shown below:

```
// Emit a photon from the light source
if(!light->emit(r,hit, Phi))
    return;
```

Task 2: Trace Photons to First Diffuse Surface

We now need to implement the forwarding of the photon from the mirror surface and transparent surface that was hit. Here we use a **Russian Roulette** probabilistic to chose either reflection or refraction when forwarding from the transparent surface. We also specify a return call if the photon does not hit

anything from this trace reflected or trace refracted call. We then also update the ray and hit info if something is hit. The implementation to the Transparent forwarding is shown below:

```
// Forward from transparent surfaces here
Ray out; // The ray that is transmitted or reflected
HitInfo out_hit;
float R = 0.0f; // The reflectance

if(!trace_refracted(r, hit, out, out_hit, R)){
    return; // Nothing was hit
};

// Russian Roulette to decide whether to reflect or refract
double p = mt_random();
if(p < R){
    // Reflect
    if(!trace_reflected(r, hit, out, out_hit)){
        return; // Nothing was hit
    }
}

r=out;
hit=out_hit;

break;
```

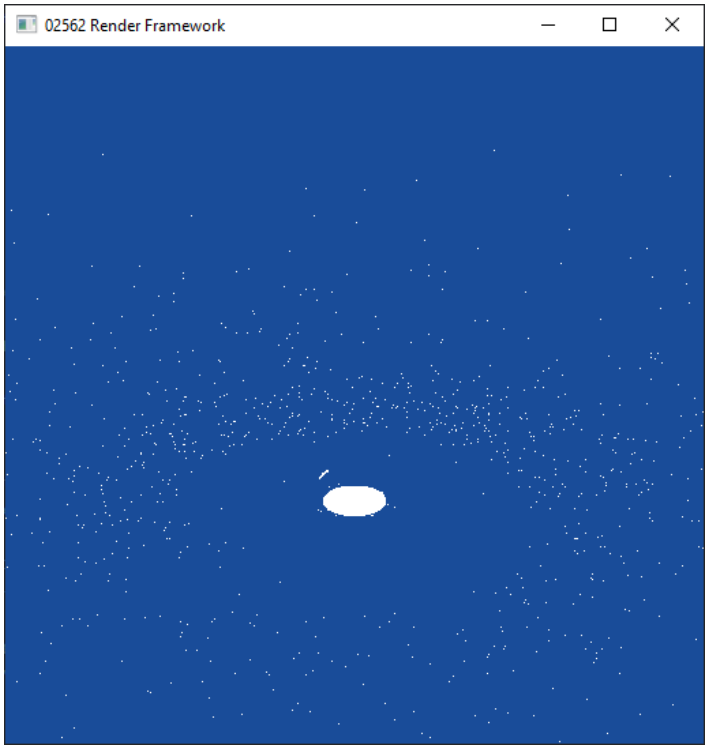
Task 3: Storing Photons in a Caustics Photon Map

Now we need to store the photons that have made it to a diffuser surface, as gathered by the while loop. We need to store the photons that have hit 1 or more surfaces on their journey from the point light. Here we use the `store` helper function from the `PhotonMap.h` file. We need to remember that when storing the photons using this helper function the convention is that the photon direction is pointing back towards where the photon came from. The code to store the photon in the photon map is shown below:

```
if(hit.trace_depth > 0){
    caustics.store(Phi, hit.position, -r.direction);
}
```

Task 4: Generate the Photon Map

If we have implemented everything correctly we can now start the program and press `2` on the keyboard to get the scene in the point of view of the photons. Every photon that has hit a diffuse surface will be a white point. The results of generating the photon map can be seen below:



Task 5: Shading from Photon Map

Now we can implement a shader for these photons that are emitted from the point light by implementing the function `shade` in the `PhotonCausics.cpp` file. The important information that we have to remember is that photons carry irradiance as we calculated in Lab 5. We want the shader to return radiance so if we remember from Lab 5 in order to get radiance we use the formula shown below:

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

Which now we need to remember that our lighting formula is the following formula:

$$L = L_e + \sum_{p \in \Delta A} fr \frac{\Delta \phi_p(x, \vec{w_p'})}{\Delta A}$$

Where our estimated fr is given by:

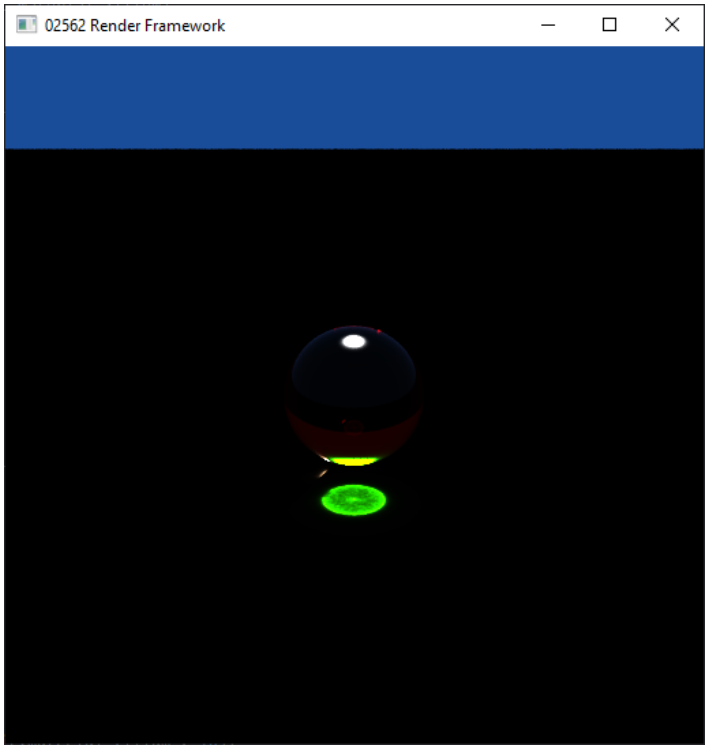
$$fr = \frac{\rho_d}{\pi}$$

The implementation of the `shade` function can be seen below:

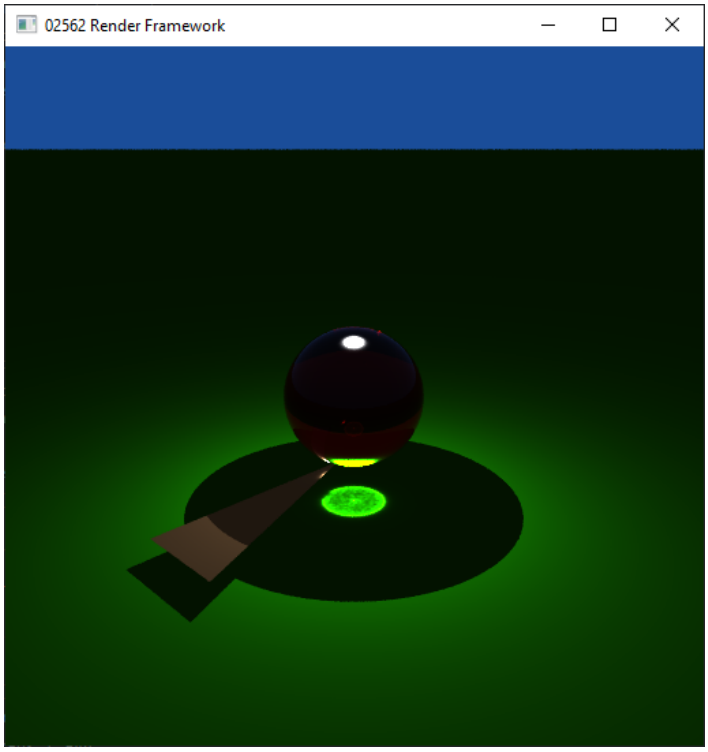
```
float3 irradiance = tracer->caustics_irradiance(hit, max_dist, photons);

// L = I / r^2
float3 radiance = irradiance / (max_dist * max_dist);

// fr = rho_d / PI
float3 fr = rho_d * M_1_PIf;

float3 L = Lambertian::shade(r, hit, emit);
float3 L_r = radiance * fr;
```

```
return L + L_r; // The combination of the two shading models
return  L_r; // Just the caustic shading

return Lambertian::shade(r, hit, emit);
```

The scene as rendered with only the caustic shading is shown below:



The scene rendered with both the caustic shading and the other forms of illumination is shown below:

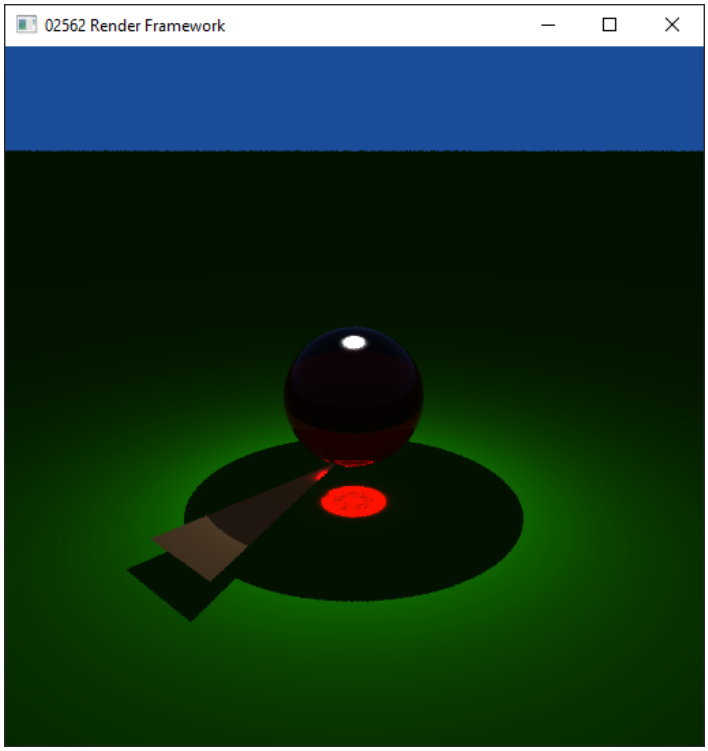


Task 6: Implement Absorption of Photons

If we look at the photon map we made we see that some photos are all over the place. This is due to the absorption of photons not working correctly. If we want the absorption of photons to work we need to copy our implementation of the `get_transmittance` from the `Volume.cpp` file and place it in the `ParticleTracer.cpp` file of the same name. We also need to handle the absorption if the medium that the photon is passing through is an absorbing medium (`illum` > 10). This implementation is straight forwards as we are just copying and pasting code. We only need to change the case 12 to the implementation shown below:

```
if (hit.material->illum > 10){
    float3 trans = expf(-get_transmittance(hit) * hit.dist);
    phi *= trans;
}
```

The result of this worksheet is shown below:



Next Lab: Rendering Lab 10

Rendering Lab 10

Lab 10: {title}

Expected due date: 16-11-2022

Lab Purpose:
The purpose of this worksheet is to show the links between rendering using software toolsets for 3D computer graphics (we will use Blender) and the rendering techniques implemented during the course. When using the rendering tools integrated into 3D modelling software, we can more easily compose a

nice scene and prepare content for being rendered. However, integrated rendering tools are often also limited in their capabilities and therefore often restrict our freedom to operate.

Learning Objectives:

- Render a scene using a production-ready path tracer.
- Use a high dynamic range (HDR) panoramic image as environment lighting.
- Cast shadows on the environment.
- Use a principled shader for simulating light-material interaction in a global illumination context.

Deliverables:
Metallic sphere and glass cube in a photographed environment. One or more other objects placed in a photographed environment and shaded with a realistic material shader. Provide the explanation and the discussion listed above.

#labnotes #todo

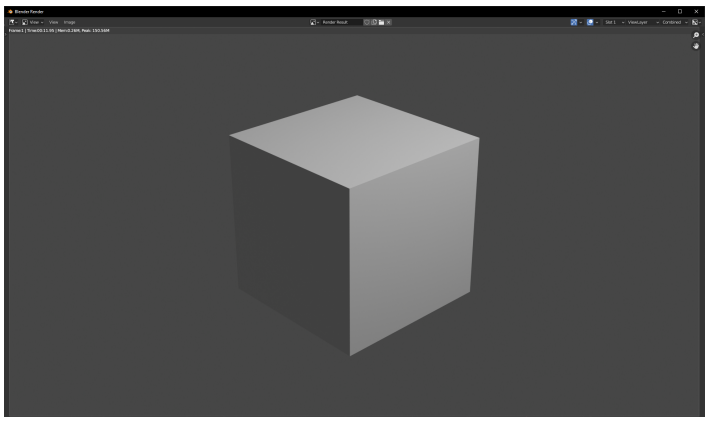
Useful Links:
Course Book
Previous Lab
Lab Worksheet

Task 1: Rendering a Scene in Blender

This first task is to introduce the student to blender by getting to render the default scene. The default scene is what appears when you open up blender and it just consists of a camera, a cube, and a point light. If we want to render the image we simply go to the render tab and press render (or we can press `F12` on the keyboard). We can enable using the GPU by following the steps:

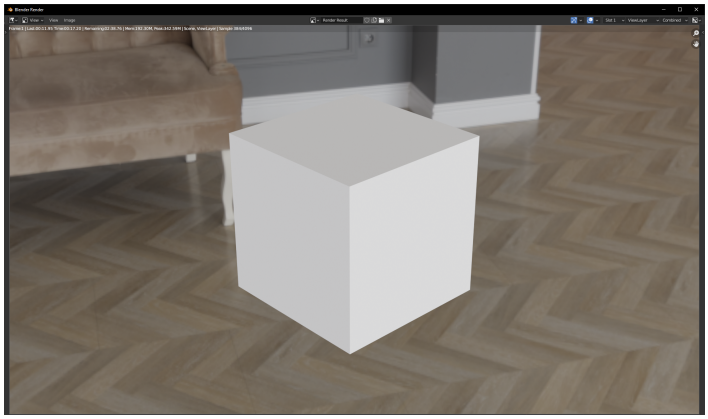
1. Edit -> Preferences -> System
2. Then under Cycles render devices make sure the GPU is selected.
3. Then go to scene render properties in the bottom right panel (the little camera)
4. Select Cycles as the render engine.
5. Select GPU Compute as the device.

We now are rendering the default scene using Cycles as the engine and the result is shown below:



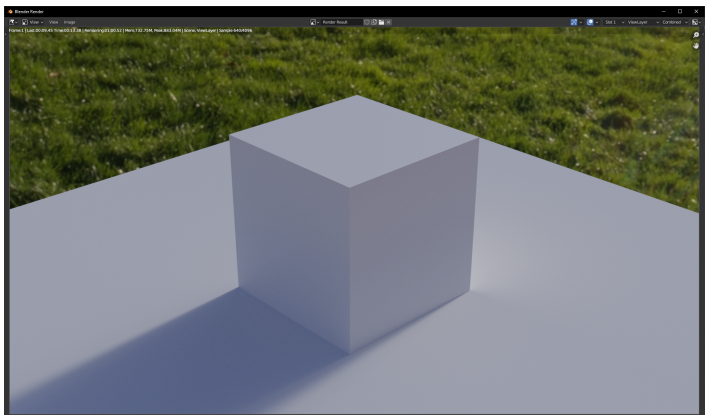
Task 2: Adding a Panoramic Texture

We can add the HDR panoramic texture we used for Lab 7 by clicking on the cube and adding the HDR image as a texture. We then go to world properties and selecting the color to be the image we specified before. Now when we render the scene we see that the background is our environment image as shown in the image below:



Task 3: Adding a Holdout Plane to the Scene

Now like before we want to render the scene we had in Lab 7 with the holdout plane being used to cast shadows unto. We do this by clicking the add button on the top of the preview screen, then clicking mesh, then plane. We can then scale an position this object by clicking on it on the right of the screen and clicking object properties. For this we use a (x, y) scale of 5.00 and a z-axis transformation of -1.00. We also want to make it invisible by clicking the visibility dropdown menu inside of the object properties and make it invisible in the render (or both in the render and the viewports). Then in the same menu we make it able to catch shadows by enabling the `Shadow Catcher` attribute. We then get rid of our point light and render the scene. Keep in mind that the plane is made invisible later as here it was just to test if it is working, also the other scene didn't project any shadows so we revert to using the one handed out by the coordinator. The render is shown below:

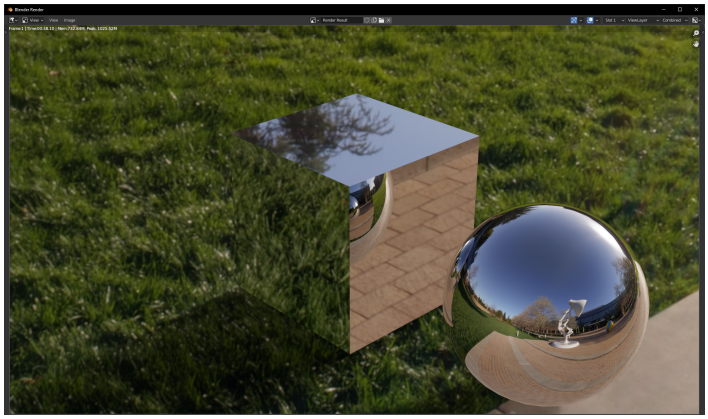


Task 4: Add a Sphere and Render Using BSDF

We can add a sphere in the same way we added an transformed the plane. However, here there a bit of new properties that we require to change. First we can change the amount of subdivisions by selecting the sphere and using the menu at the bottom left of the preview screen. We can also right click the sphere and select smooth shading to implement the smooth shading on the object.

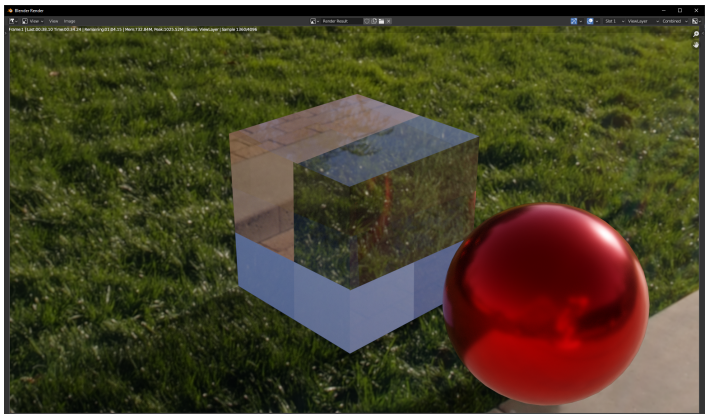
As for changing the material properties of an object it is done in the material tab right above where we changed the texture. Here we want to make both objects a mirror object by adding a material to the object. Then changing the subsurface method to Christensen-Burley and changing the properties to full metallic, full specular, and no roughness. We need to remember to make the plane object properties also invisible to rays by unchecking the properties under visibility, ray visibility, of all except the camera.

The results of the mirror ball and cube are shown below:



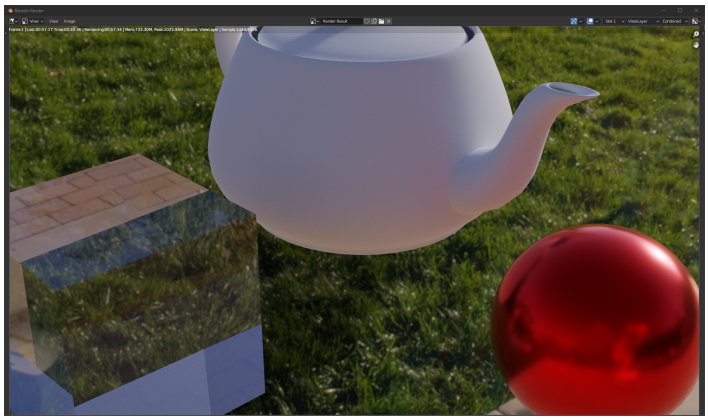
Task 5: Metallic Ball and Glass Cube

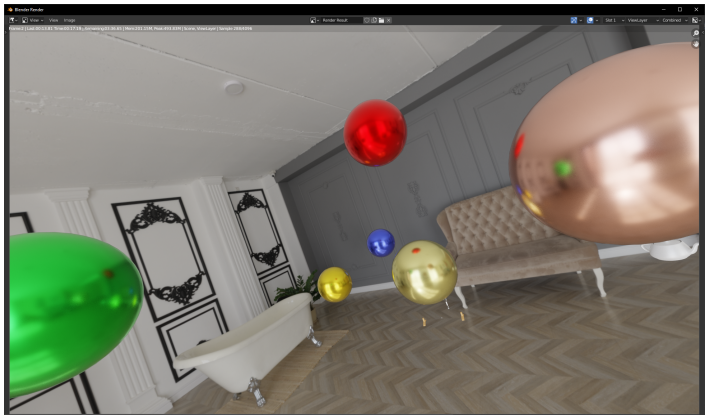
We were shown how to make the metallic Christmas ornament and glass cube in class so I will not go over how to do it here as it is redundant. However, it is important to reflect that what we are doing here is setting up the parameters that we have coded into our implementation of the render engine. We discuss this more during Task 7. The results of this task are shown below:



Task 6: Import a Triangle Mesh

Importing a mesh is done by going to file then import then `wavefront` object then selecting the object file. For this we decide to use the Utah Teapot. We then just have to place it somewhere in the scene to not obstruct the other objects. The result is shown below:





Task 7: Comparison of Render Engines

There are quite a bit of pros using a third party rendering engine such as blender. For one having a tool that is standardized allows creative people to come up with optimization for workloads. This is especially true for tools like blender which have established themselves as standard in the industry. People know what the tool can do an thus provide others with knowledge on how to create certain effects. The sampling and rendering with the GPU is something that our framework desperately needs as some of the more complex scenes like having the glass and metallic spheres in the Cornell box was talking up to 12 hours to render. Where as Blender I kept making as many metallic spheres as I wanted. I think that the last thing we implemented in the framework was something that blender could use as photon mapping was a really cool effect. Also the framework's preview camera is something that blender could use as having that really simple render as a basis on what the camera could potentially see is a really nice quality of life thing to have. Currently what needs to be improved in my framework implementation is the user input. Something that rendering in blender was quite nice to quickly change model properties unlike our framework that needs to alter the `mtl` files manually.

Next Lab: Rendering Project