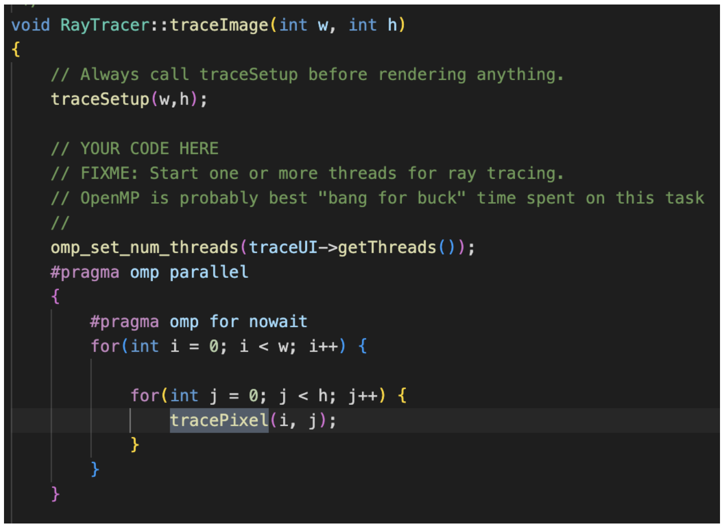
Project 1: RayTracer

By: Eric Wang, Andy Hsu

Initial steps

Our implementation started by shooting rays into the scene. This was done by calling tracePixel for each pixel in the scene. We added multithreading using OpenMP to make the ray tracer more efficient.

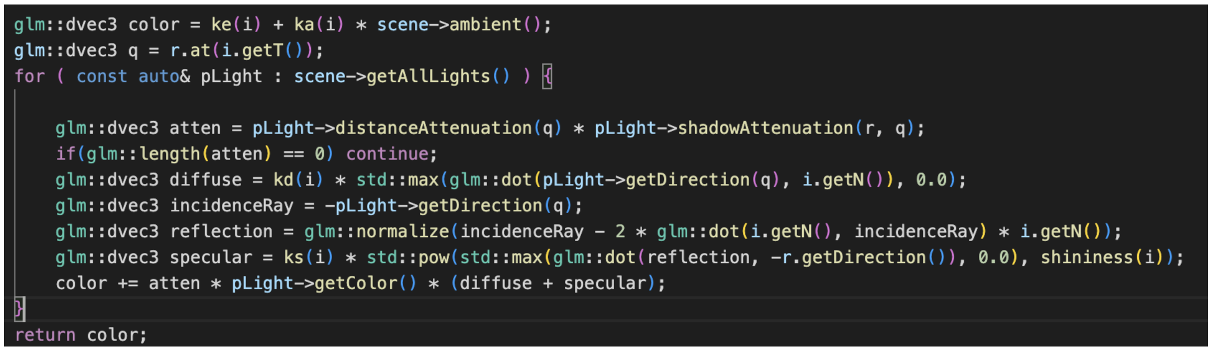


```
void RayTracer::traceImage(int w, int h)
{
    // Always call traceSetup before rendering anything.
    traceSetup(w,h);

    // YOUR CODE HERE
    // FIXME: Start one or more threads for ray tracing.
    // OpenMP is probably best "bang for buck" time spent on this task
    //
    omp_set_num_threads(traceUI->getThreads());
    #pragma omp parallel
    {
        #pragma omp for nowait
        for(int i = 0; i < w; i++) {

            for(int j = 0; j < h; j++) {
                tracePixel(i, j);
            }
        }
    }
```

Phong Shading

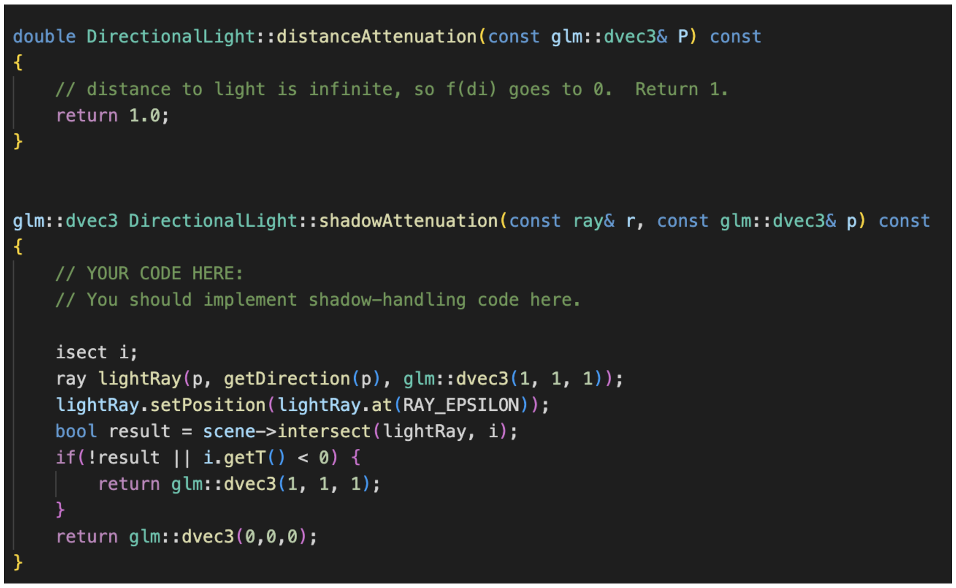
To implement Phong shading, we used the method described in class, adding up diffuse, specular, ambient light terms for each light in the scene. This required us to implement distance attenuation for point light and shadow attenuation for both point light and directional light. For distance attenuation and shadow attenuation we simply implemented the algorithms described in class. We didn't account for objects being transmissive in our shadow attenuation, which causes some of our images to be different from the solution.

Phong shading (implemented in shade() function):

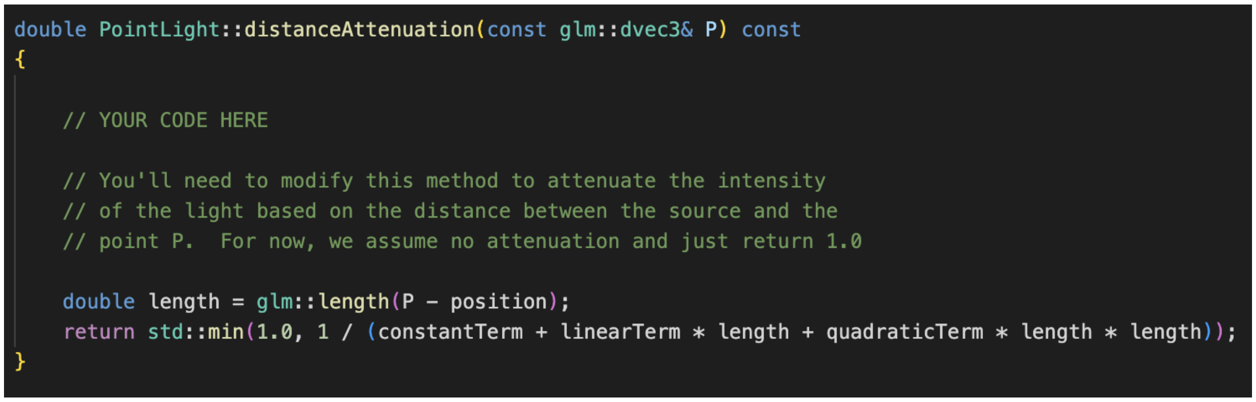


```cpp
glm::dvec3 color = ke(i) + ka(i) * scene->ambient();
glm::dvec3 q = r.at(i.getT());
for ( const auto& pLight : scene->getAllLights() ) {

    glm::dvec3 atten = pLight->distanceAttenuation(q) * pLight->shadowAttenuation(r, q);
    if(glm::length(atten) == 0) continue;
    glm::dvec3 diffuse = kd(i) * std::max(glm::dot(pLight->getDirection(q), i.getN()), 0.0);
    glm::dvec3 incidenceRay = -pLight->getDirection(q);
    glm::dvec3 reflection = glm::normalize(incidenceRay - 2 * glm::dot(i.getN(), incidenceRay) * i.getN());
    glm::dvec3 specular = ks(i) * std::pow(std::max(glm::dot(reflection, -r.getDirection()), 0.0), shininess(i));
    color += atten * pLight->getColor() * (diffuse + specular);
}
return color;
```

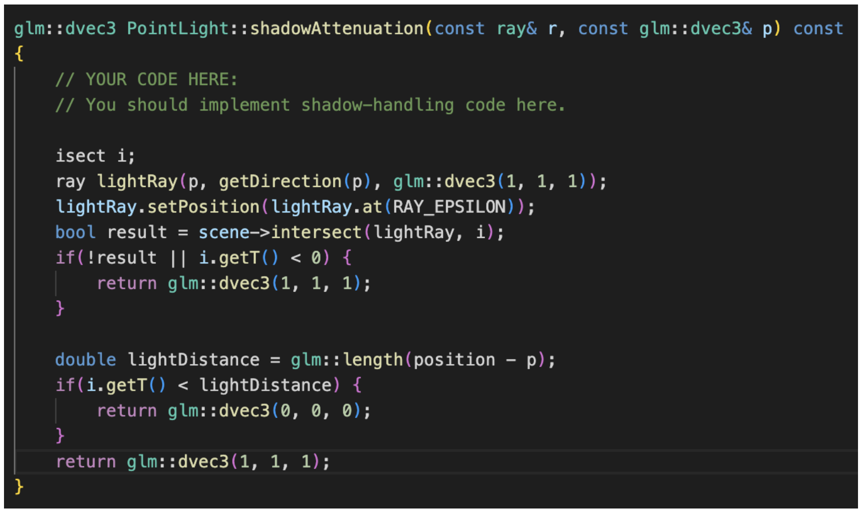
Distance and shadow attenuation (implemented in light.cpp):



```cpp
double DirectionalLight::distanceAttenuation(const glm::dvec3& P) const
{
    // distance to light is infinite, so f(di) goes to 0.  Return 1.
    return 1.0;
}


glm::dvec3 DirectionalLight::shadowAttenuation(const ray& r, const glm::dvec3& p) const
{
    // YOUR CODE HERE:
    // You should implement shadow-handling code here.

    isect i;
    ray lightRay(p, getDirection(p), glm::dvec3(1, 1, 1));
    lightRay.setPosition(lightRay.at(RAY_EPSILON));
    bool result = scene->intersect(lightRay, i);
    if(!result || i.getT() < 0) {
        return glm::dvec3(1, 1, 1);
    }
    return glm::dvec3(0,0,0);
}
```



```cpp
double PointLight::distanceAttenuation(const glm::dvec3& P) const
{

    // YOUR CODE HERE

    // You'll need to modify this method to attenuate the intensity
    // of the light based on the distance between the source and the
    // point P.  For now, we assume no attenuation and just return 1.0

    double length = glm::length(P - position);
    return std::min(1.0, 1 / (constantTerm + linearTerm * length + quadraticTerm * length * length));
}
```

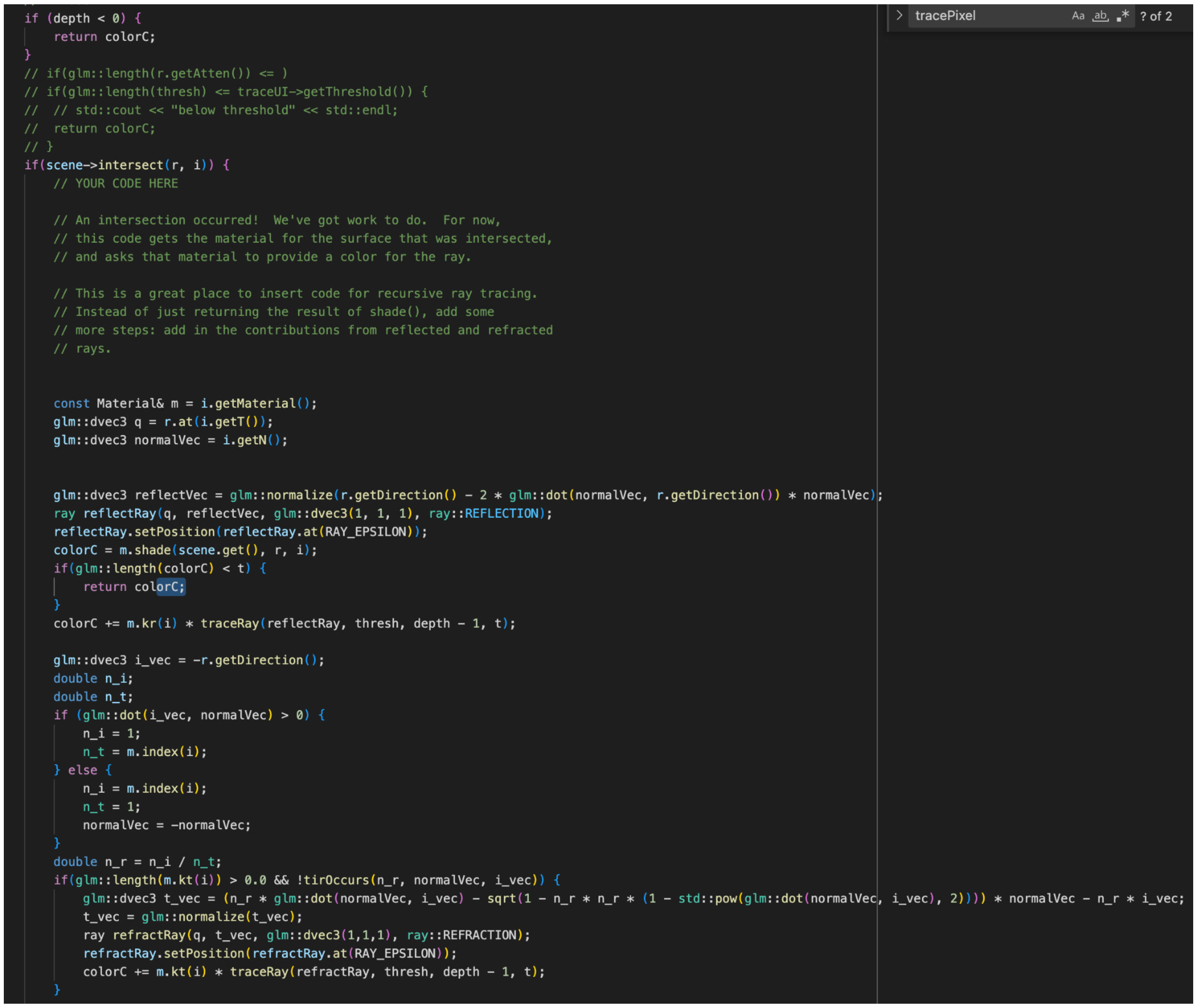


```
glm::dvec3 PointLight::shadowAttenuation(const ray& r, const glm::dvec3& p) const
{
    // YOUR CODE HERE:
    // You should implement shadow-handling code here.

    isect i;
    ray lightRay(p, getDirection(p), glm::dvec3(1, 1, 1));
    lightRay.setPosition(lightRay.at(RAY_EPSILON));
    bool result = scene->intersect(lightRay, i);
    if(!result || i.getT() < 0) {
        return glm::dvec3(1, 1, 1);
    }

    double lightDistance = glm::length(position - p);
    if(i.getT() < lightDistance) {
        return glm::dvec3(0, 0, 0);
    }
    return glm::dvec3(1, 1, 1);
}
```

Reflection/Refraction

Reflection and refraction was implemented in the traceRay method. The reflection vector was found by using the law of reflection similar to the implementation in glm. Refraction was implemented using Snell's law and the formula for refraction given in class. Total internal reflection was also implemented using the formula given in class - this was done in the tirOccurs() method. Recursive calls were made on the reflection and refraction vectors, with these return values being multiplied by the kr and kt values of the material, respectively. These two values were then added to the return value of the shade() function to get the color of the pixel and the return value of traceRay(). Recursive calls were only made if the depth parameter was greater than or equal to zero and if the magnitude of the vector returned from shade() (or in other words the intensity) was greater than the specified threshold value. We also had to shift the starting point of our reflection and refraction vectors by RAY_EPSILON to account for floating point errors.

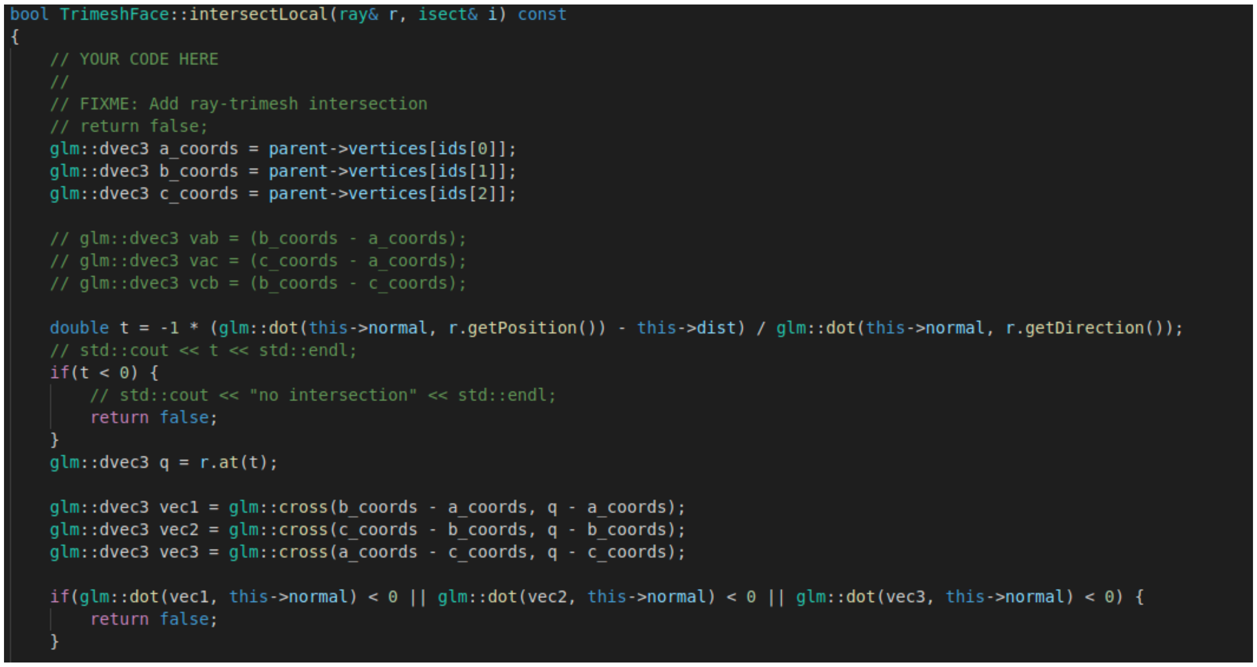
traceRay():



```cpp
if (depth < 0) {
    return colorC;
}
// if(glm::length(r.getAtten()) <= )
// if(glm::length(thresh) <= traceUI->getThreshold()) {
//  // std::cout << "below threshold" << std::endl;
//  return colorC;
// }
if(scene->intersect(r, i)) {
    // YOUR CODE HERE

    // An intersection occurred!  We've got work to do.  For now,
    // this code gets the material for the surface that was intersected,
    // and asks that material to provide a color for the ray.

    // This is a great place to insert code for recursive ray tracing.
    // Instead of just returning the result of shade(), add some
    // more steps: add in the contributions from reflected and refracted
    // rays.


    const Material& m = i.getMaterial();
    glm::dvec3 q = r.at(i.getT());
    glm::dvec3 normalVec = i.getN();


    glm::dvec3 reflectVec = glm::normalize(r.getDirection() - 2 * glm::dot(normalVec, r.getDirection()) * normalVec);
    ray reflectRay(q, reflectVec, glm::dvec3(1, 1, 1), ray::REFLECTION);
    reflectRay.setPosition(reflectRay.at(RAY_EPSILON));
    colorC = m.shade(scene.get(), r, i);
    if(glm::length(colorC) < t) {
        return colorC;
    }
    colorC += m.kr(i) * traceRay(reflectRay, thresh, depth - 1, t);

    glm::dvec3 i_vec = -r.getDirection();
    double n_i;
    double n_t;
    if (glm::dot(i_vec, normalVec) > 0) {
        n_i = 1;
        n_t = m.index(i);
    } else {
        n_i = m.index(i);
        n_t = 1;
        normalVec = -normalVec;
    }
    double n_r = n_i / n_t;
    if(glm::length(m.kt(i)) > 0.0 && !tirOccurs(n_r, normalVec, i_vec)) {
        glm::dvec3 t_vec = (n_r * glm::dot(normalVec, i_vec) - sqrt(1 - n_r * n_r * (1 - std::pow(glm::dot(normalVec, i_vec), 2)))) * normalVec - n_r * i_vec;
        t_vec = glm::normalize(t_vec);
        ray refractRay(q, t_vec, glm::dvec3(1,1,1), ray::REFRACTION);
        refractRay.setPosition(refractRay.at(RAY_EPSILON));
        colorC += m.kt(i) * traceRay(refractRay, thresh, depth - 1, t);
    }
```

Trimesh intersection

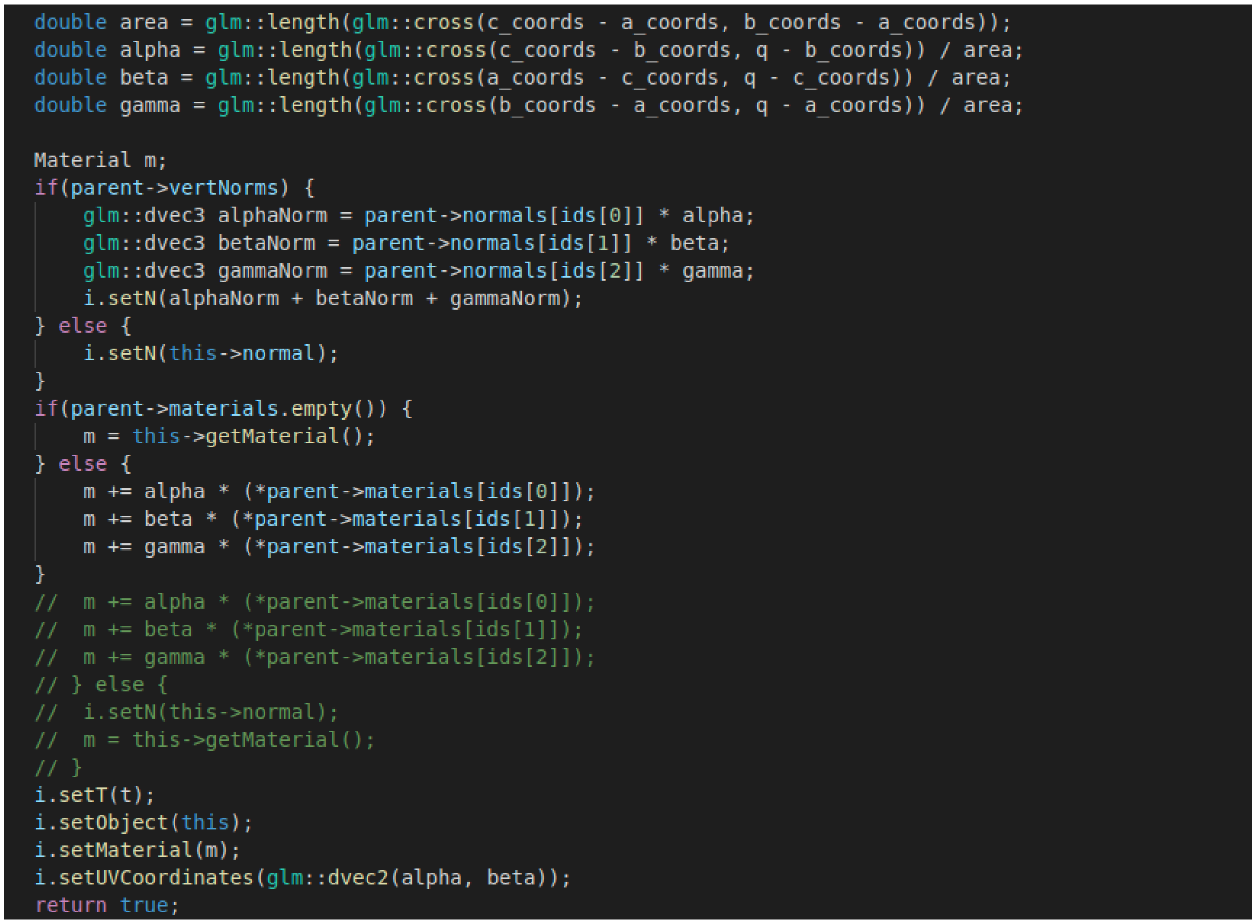
We implemented trimesh intersection in the trimesh.cpp file under the TrimeshFace::intersectLocal() function. We first grabbed the 3 coordinates of the triangle and then found the point of intersection between the ray and the plane that the triangle lies in. We checked if the point lies within the triangle using the Inside-Outside test.



```cpp
bool TrimeshFace::intersectLocal(ray& r, isect& i) const
{
    // YOUR CODE HERE
    //
    // FIXME: Add ray-trimesh intersection
    // return false;
    glm::dvec3 a_coords = parent->vertices[ids[0]];
    glm::dvec3 b_coords = parent->vertices[ids[1]];
    glm::dvec3 c_coords = parent->vertices[ids[2]];

    // glm::dvec3 vab = (b_coords - a_coords);
    // glm::dvec3 vac = (c_coords - a_coords);
    // glm::dvec3 vcb = (b_coords - c_coords);

    double t = -1 * (glm::dot(this->normal, r.getPosition()) - this->dist) / glm::dot(this->normal, r.getDirection());
    // std::cout << t << std::endl;
    if(t < 0) {
        // std::cout << "no intersection" << std::endl;
        return false;
    }
    glm::dvec3 q = r.at(t);

    glm::dvec3 vec1 = glm::cross(b_coords - a_coords, q - a_coords);
    glm::dvec3 vec2 = glm::cross(c_coords - b_coords, q - b_coords);
    glm::dvec3 vec3 = glm::cross(a_coords - c_coords, q - c_coords);

    if(glm::dot(vec1, this->normal) < 0 || glm::dot(vec2, this->normal) < 0 || glm::dot(vec3, this->normal) < 0) {
        return false;
    }
```

Phong interpolation

We implemented Phong interpolation in the TrimeshFace::intersectLocal() function after running the Inside-Outside test to test for triangle intersection. If we passed the Inside-Outside test, we then had a check to see if the triangle had per-vertex normals. If it passes the test, then we calculate the normal of the surface using the normals at the vertices. Then, we check if there is more than one material and if so calculate the new material.



```
double area = glm::length(glm::cross(c_coords - a_coords, b_coords - a_coords));
double alpha = glm::length(glm::cross(c_coords - b_coords, q - b_coords)) / area;
double beta = glm::length(glm::cross(a_coords - c_coords, q - c_coords)) / area;
double gamma = glm::length(glm::cross(b_coords - a_coords, q - a_coords)) / area;

Material m;
if(parent->vertNorms) {
    glm::dvec3 alphaNorm = parent->normals[ids[0]] * alpha;
    glm::dvec3 betaNorm = parent->normals[ids[1]] * beta;
    glm::dvec3 gammaNorm = parent->normals[ids[2]] * gamma;
    i.setN(alphaNorm + betaNorm + gammaNorm);
} else {
    i.setN(this->normal);
}
if(parent->materials.empty()) {
    m = this->getMaterial();
} else {
    m += alpha * (*parent->materials[ids[0]]);
    m += beta * (*parent->materials[ids[1]]);
    m += gamma * (*parent->materials[ids[2]]);
}
//  m += alpha * (*parent->materials[ids[0]]);
//  m += beta * (*parent->materials[ids[1]]);
//  m += gamma * (*parent->materials[ids[2]]);
// } else {
//  i.setN(this->normal);
//  m = this->getMaterial();
// }
i.setT(t);
i.setObject(this);
i.setMaterial(m);
i.setUVCoordinates(glm::dvec2(alpha, beta));
return true;
```

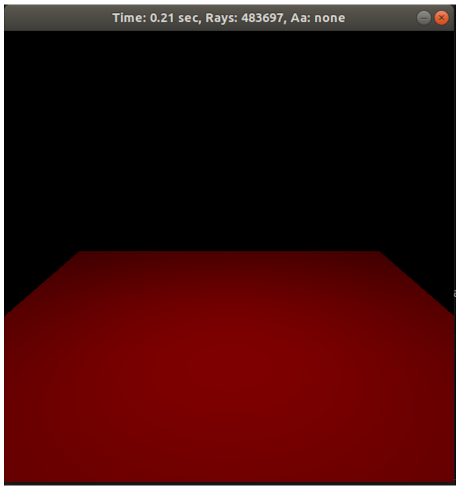
Performance, Bugs, and Future Work

We encountered a lot of bugs while working on this assignment. When working on Phong illumination, some of our refraction rays weren't being calculated correctly because we didn't take into account the transmissivity of objects. So, in our shadow attenuation code, if the light ray hits any object, we return a vector of zeros even if the object is somewhat transmissible. We were also having some trouble getting refraction to work in Phong Illumination because our normal vector was facing the wrong way whenever the light ray was leaving the object. Also, we ran into an issue when trying to do triangle intersection because when trying to calculate the 't' value, we were adding the 'dist' variable in the TrimeshFace class instead of subtracting the value. Whether or not we add or subtract 'dist' is based on how we calculate the variable and whether or not it is negated already. However, by adding it to t instead of subtracting it, we were getting some cases where our model wasn't detecting any intersection for some objects.

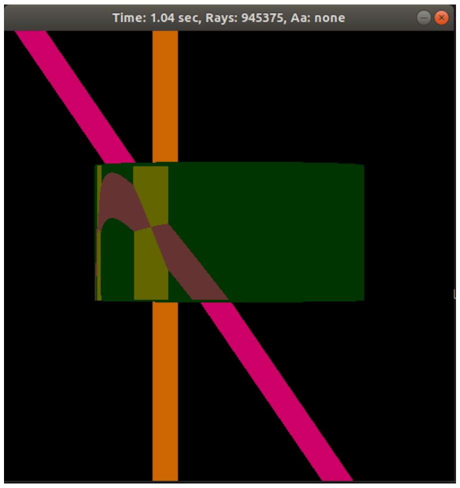
I think some things we could do in the future include adding transmissivity to shadow attenuation, so it isn't just returning either all ones or all zeros. Also, our model is running rather slow compared to the solution model. I know we haven't implemented k-d trees yet, but we could still probably optimize our model in some way to improve performance.

Output images:

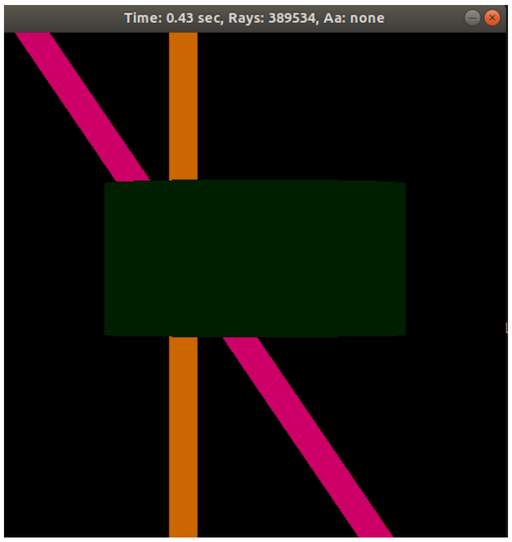
box.ray (Tests Phong shading for point light):



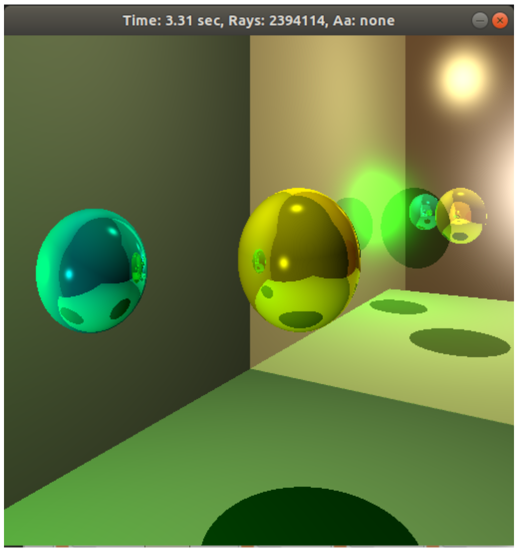
cylinder_refract_simple.ray - recursion depth 5 (Tests refraction):



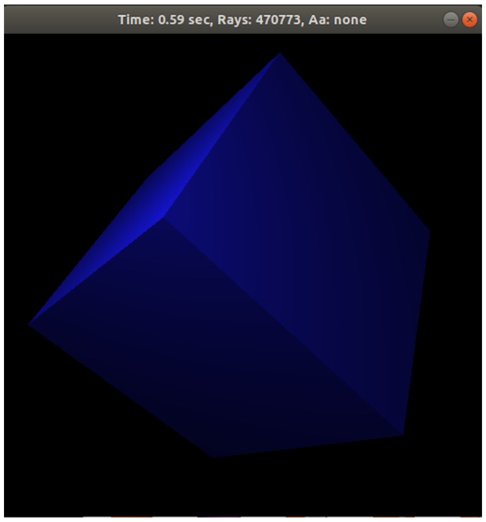
cylinder_refract_simple.ray - recursion depth 5, threshold 203 (Tests threshold):


Time: 0.43 sec, Rays: 389534, Aa: none

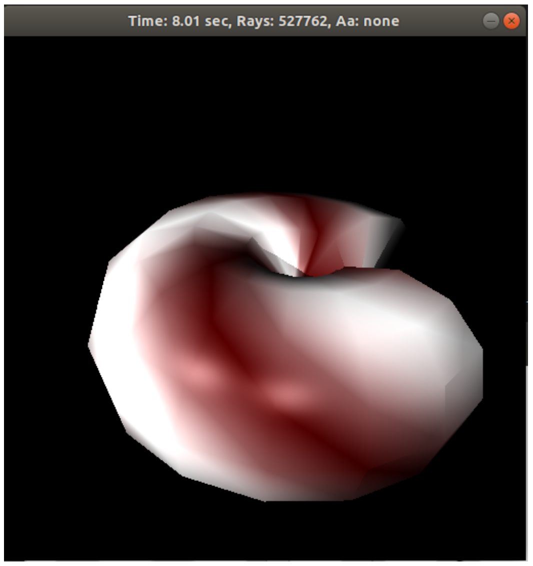
reflection2.ray - recursion depth 5 (Tests reflection, shadows from multiple light sources):


Time: 3.31 sec, Rays: 2394114, Aa: none

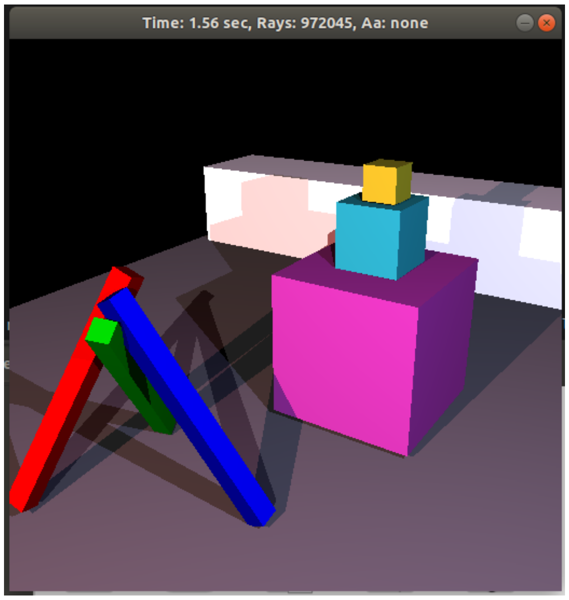
cube.ray (Tests trimesh intersection):


Time: 0.59 sec, Rays: 470773, Aa: none

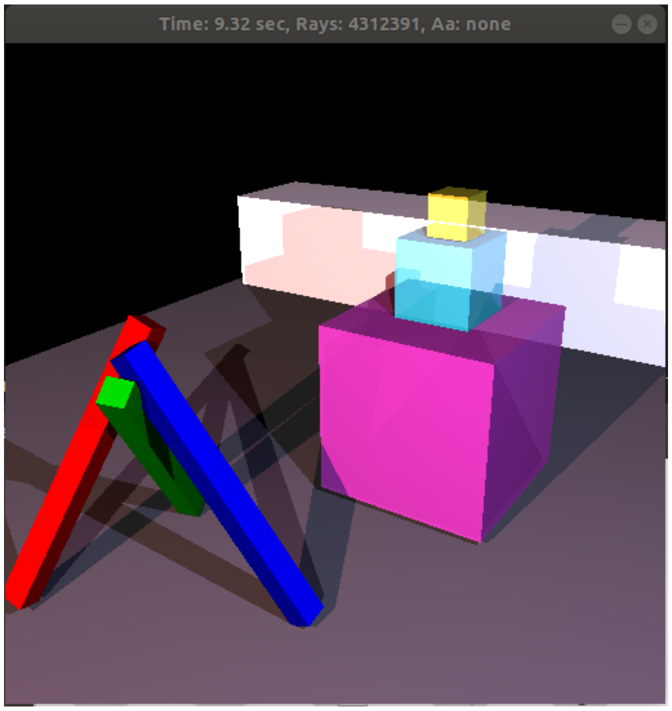
shell.ray (Tests phong interpolation):


Time: 8.01 sec, Rays: 527762, Aa: none

recurse_depth.ray (Tests various recursion depths) - recursion depth 0:


Time: 1.56 sec, Rays: 972045, Aa: none

recurse_depth.ray - recursion depth 5:


Time: 9.32 sec, Rays: 4312391, Aa: none