Group Members: Remus Wong (rcw2448), Marlowe Johnson (msj879)

Phong shading and lighting

Material::shade()

Basically follows the pseudo code from the slides. We sum together all the components and do separate calculations for each light in the scene to determine the phong shading. We also make sure to use RAY_EPSILON when finding the shadow attenuation. For distance attenuation it was found using “atten = 1 / (constantTerm + linearTerm*d + quadraticTerm*d*d)” and we made sure to cap the attenuation at 1 to make sure the scene would not be blown out.

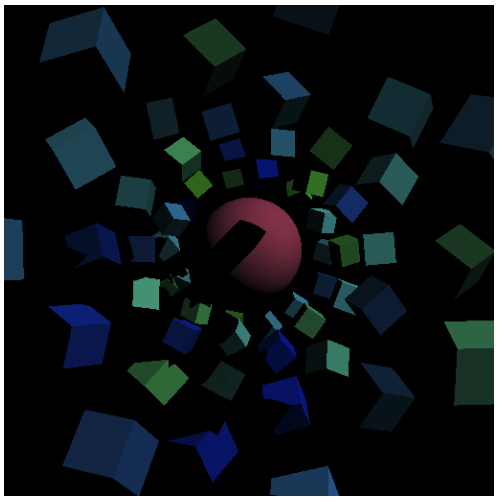
```
const Material& m = i.getMaterial();

glm::dvec3 phong(0,0,0);
glm::dvec3 pointQ = r.at(i.getT());
phong += m.ke(i) + m.ka(i) * scene->ambient();
for ( const auto& pLight : scene->getAllLights() ) {
    // Calc attenuation
    double distatten = pLight->distanceAttenuation(pointQ);
    glm::dvec3 shadatten = pLight->shadowAttenuation(r, pointQ+pLight->getDirection(pointQ)*RAY_EPSILON);
    glm::dvec3 atten = distatten * shadatten;
    // Calc diffuse
    glm::dvec3 diffuse = kd(i) * max(0.0, dot(i.getN(), pLight->getDirection(pointQ)));
    // Calc specular
    glm::dvec3 R = glm::reflect(pLight->getDirection(pointQ)*-1.0, i.getN());
    glm::dvec3 V = r.getDirection()*-1.0;
    glm::dvec3 specular = ks(i) * pow(max(0.0, dot(R, V)), m.shininess(i));
    phong += atten*(diffuse + specular) * pLight->getColor();
}
return phong;
```

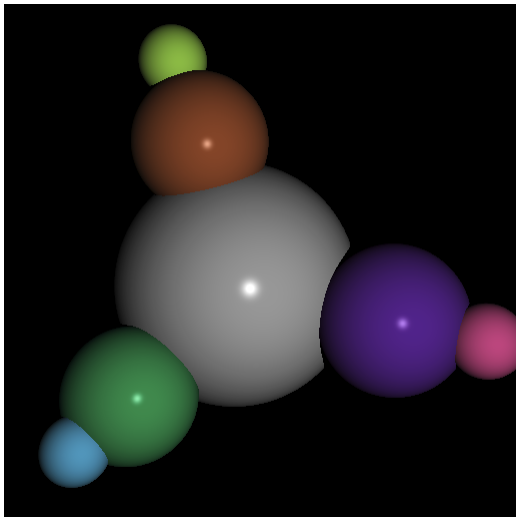
PointLight::shadowAttenuation()

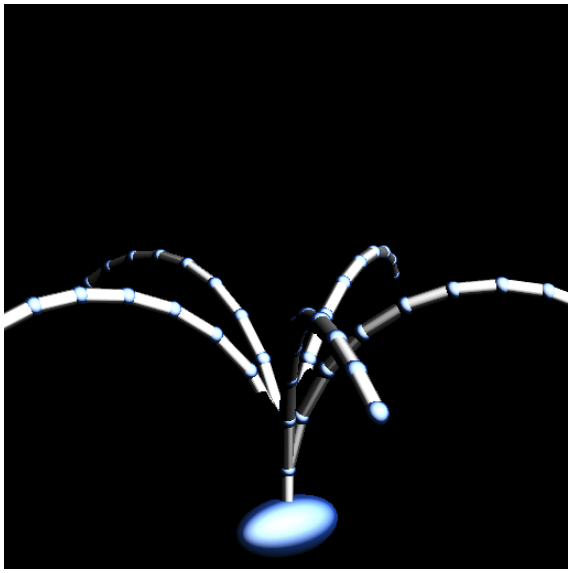
To implement shadow attenuation we had to make sure that we check if we hit any object on our way to the light source. We did not implement soft shadows so we only set the shadow attenuation to 1 or 0. The implementation is basically the same for directional lights.

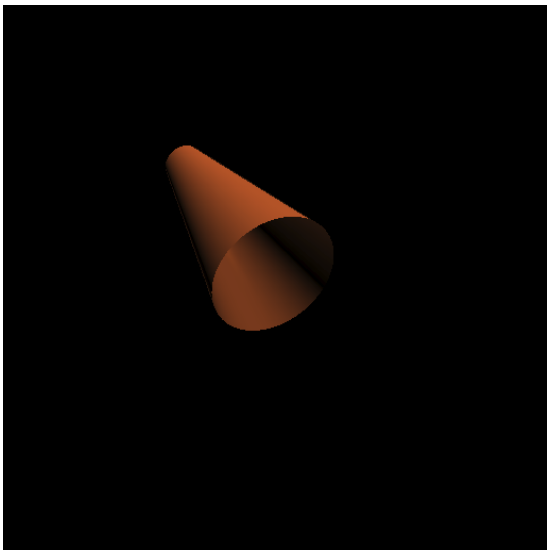
```
glm::dvec3 shadAtten;
glm::dvec3 d = glm::normalize(getDirection(p));            // pointQ ->
ray shadRay(p,d,glm::dvec3(1,1,1),ray::SHADOW);            // ray from
isect hitT;                                                // informati
bool check = scene->intersect(shadRay, hitT);
double dist2light = glm::length(position - p);
double dist2Q = glm::length(getDirection(p) * hitT.getT());
if (dist2light > dist2Q)
    // our surface hit is before the light, apply attenuation
    shadAtten = glm::dvec3(0,0,0);
else
    // our surface is behind the light, no need to apply attenuation
    shadAtten = glm::dvec3(1,1,1);
return shadAtten;
```

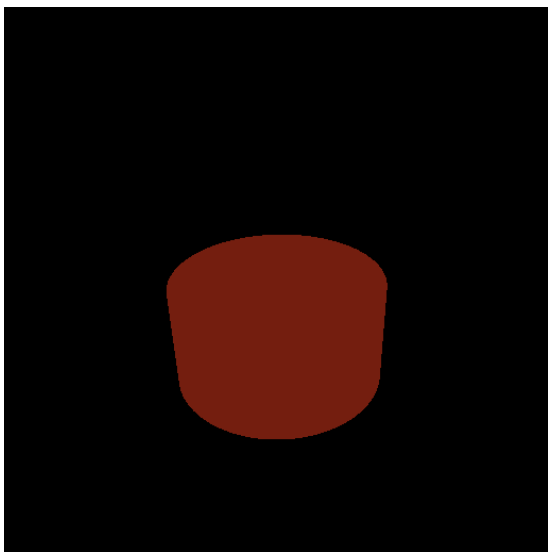
Examples:

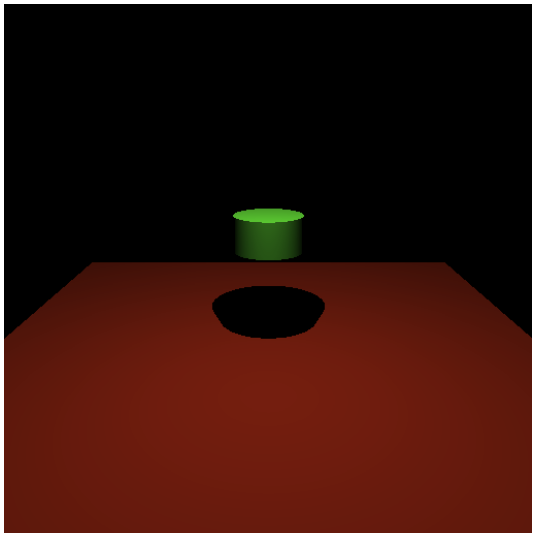










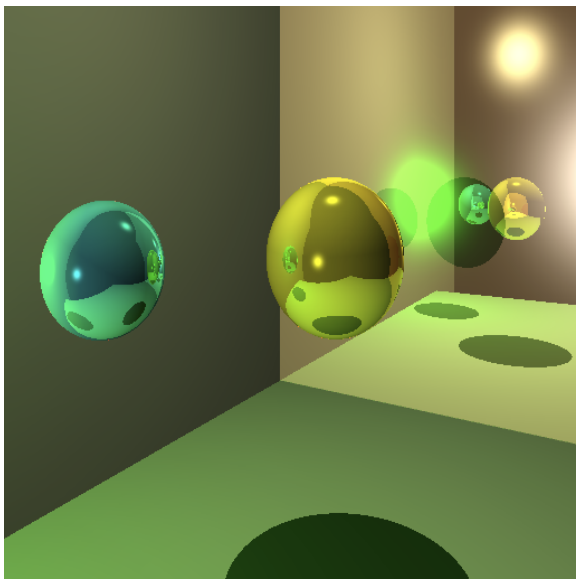


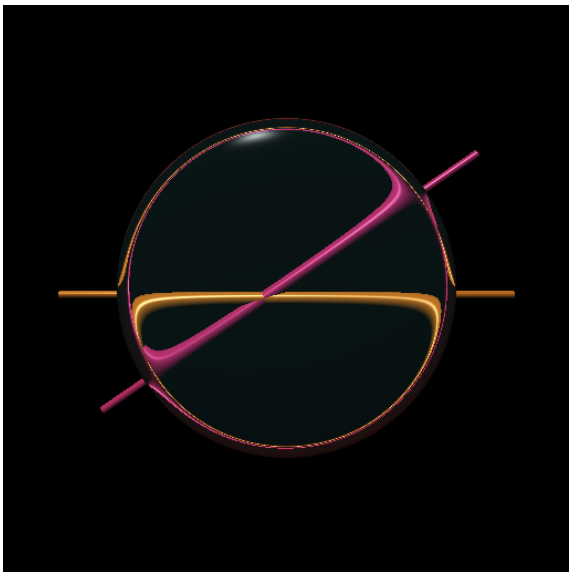
Reflection/Refraction

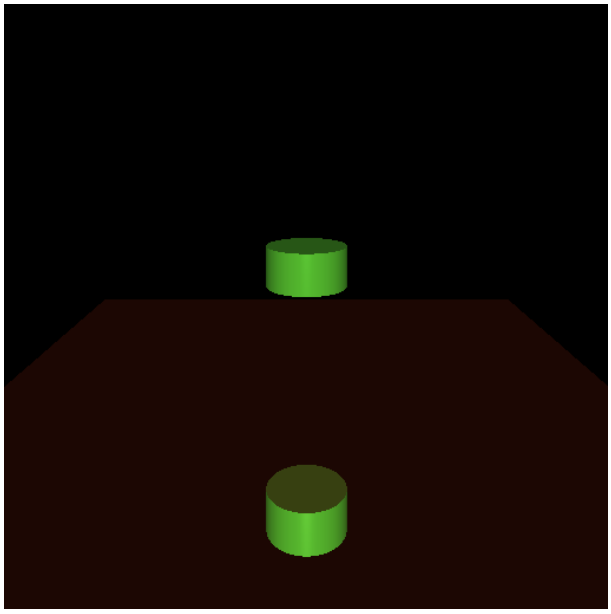
This part was not very difficult in terms translating the math to code but it did take a long time to debug the refraction model. One of the main hurdles was realizing that I had to flip the i normal when exiting an object. I also added the RAY_EPSILON when calling reflection and refraction rays by pushing the point slightly in the direction of the reflection/refraction vector to avoid self collision.

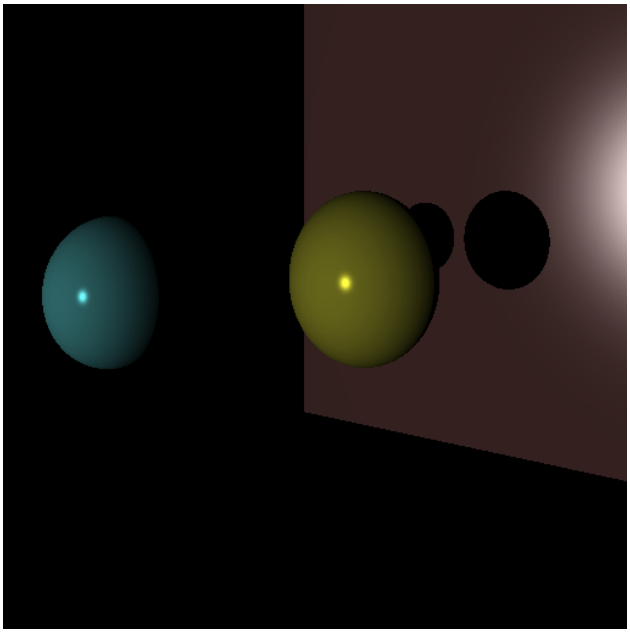
```
//Reflected Color
reflectVector = glm::reflect(rDir,plNorm);
reflectVector = glm::normalize(reflectVector);
ray refR(pointQ+reflectVector*RAY_EPSILON,reflectVector,glm::dvec3(1,1,1),ray::REFLECTION); // change ray type
colorRe = (m.kr(i) * traceRay(refR,thresh,depth-1,t));

//Refracted/Transmitted Color
double dotP = glm::dot(rDir,plNorm);
if (dotP < 0)
{
    // Entering object
    n_i = 1.0003;
    n_t = m.index(i);
}
else
{
    // Exiting object
    n_i = m.index(i);
    n_t = 1.0003;
    plNorm *= -1.0;
}
// check trans > 0 && notTIR (n_i, n_t, N, -d)
// check for total internal reflection
double eta = n_i/n_t;
double TIR = 1 - eta*eta * (1 - dotP*dotP);
if (TIR>=0 && glm::length(m.kt(i))>=0)
{
    refractVector = glm::refract(rDir,plNorm,eta);
    ray rafR(pointQ+refractVector*RAY_EPSILON,refractVector,glm::dvec3(1,1,1),ray::REFRACTION); //change ray
    glm::dvec3 refractCol = traceRay(rafR,thresh,depth-1,t);
    colorRf =  m.kt(i) * refractCol;
}
colorC = colorO + colorRe + colorRf;
```

Correct examples:





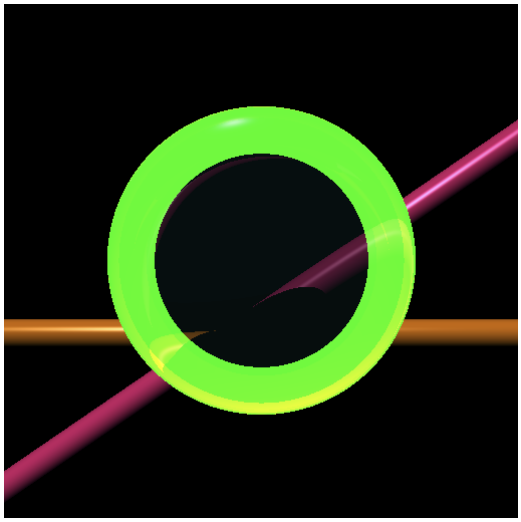




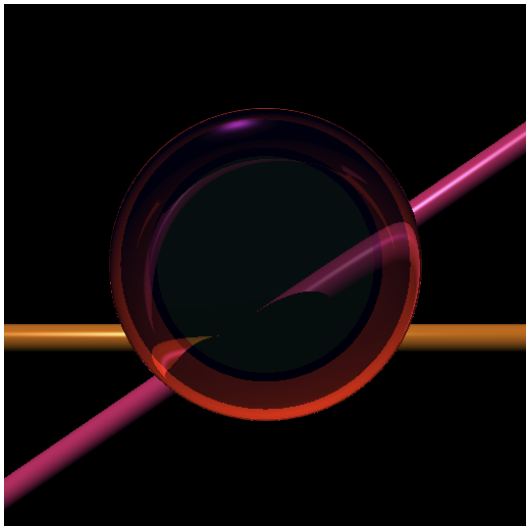
BUG. Refraction seemed to behave very inconsistently when building the project with CMAKE is release mode. When building with plain CMAKE, the refractions would look the same as the output. However, in release mode I would get different outputs outlined in a few examples section below. This seemed to only affect refractions in the /simple/ folder.

Bugged release mode examples (Sphere_refraction3):

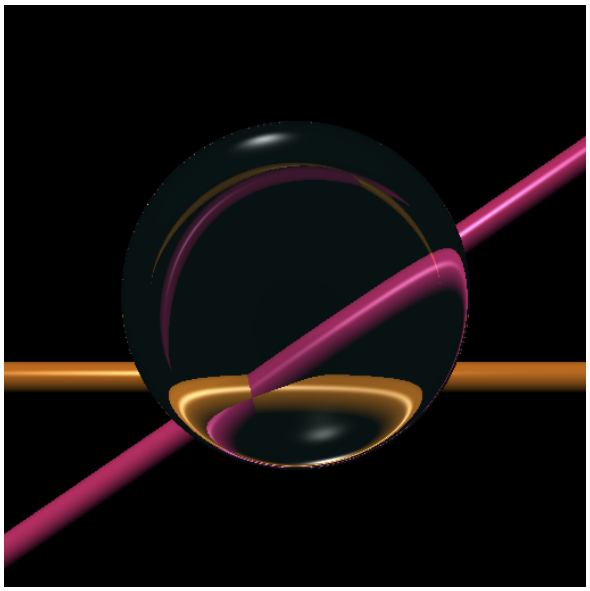
Release mode output1



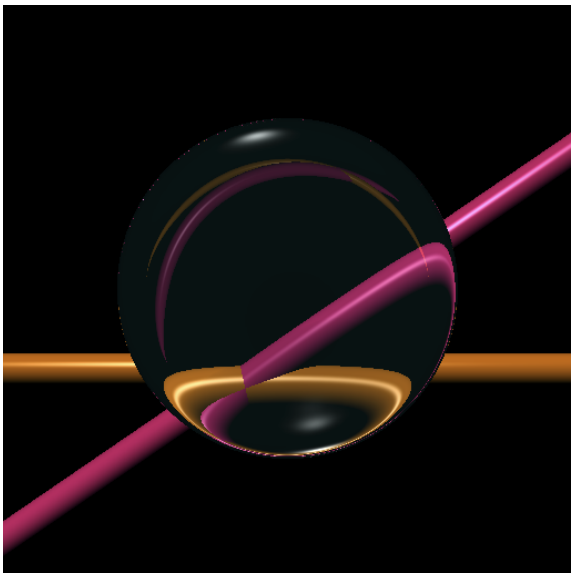
Release mode output2



Debug mode output



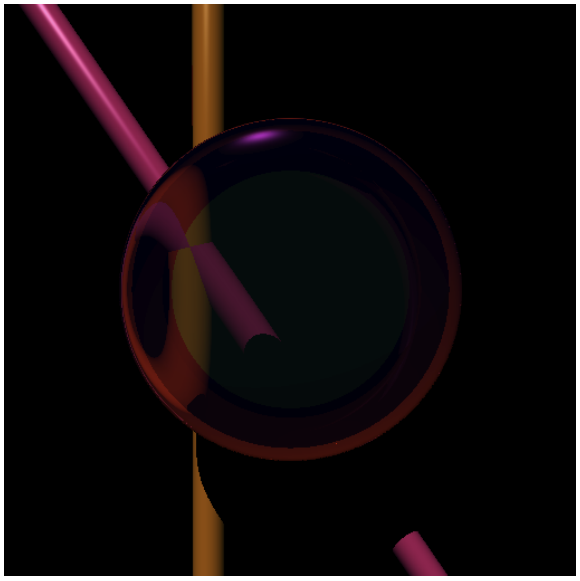
Solution output

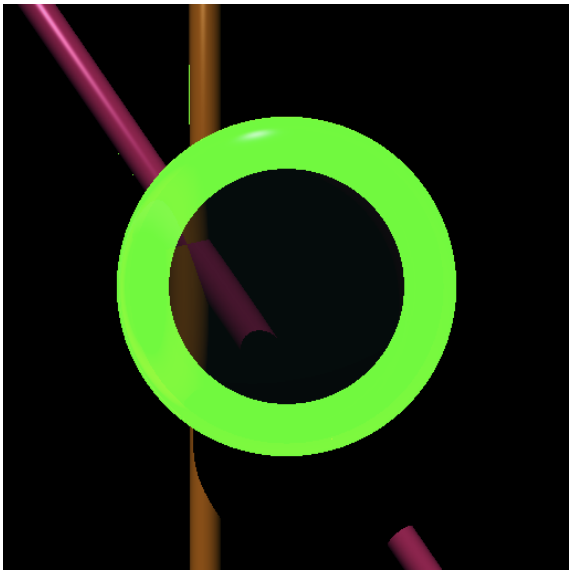


Bugged release mode examples (Sphere_refraction7):

Release output1

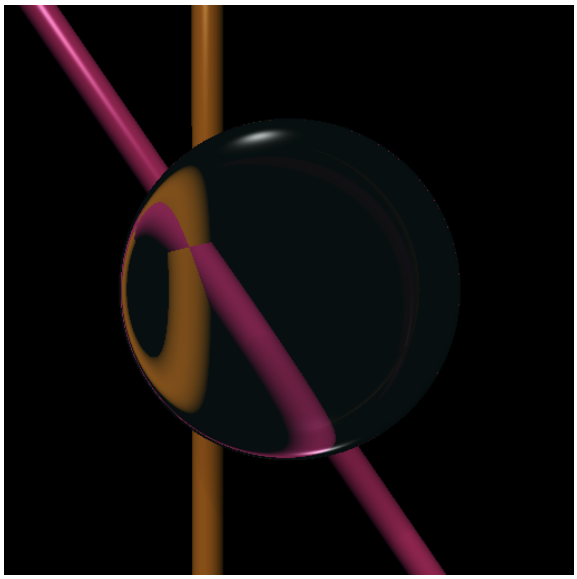
Release output2

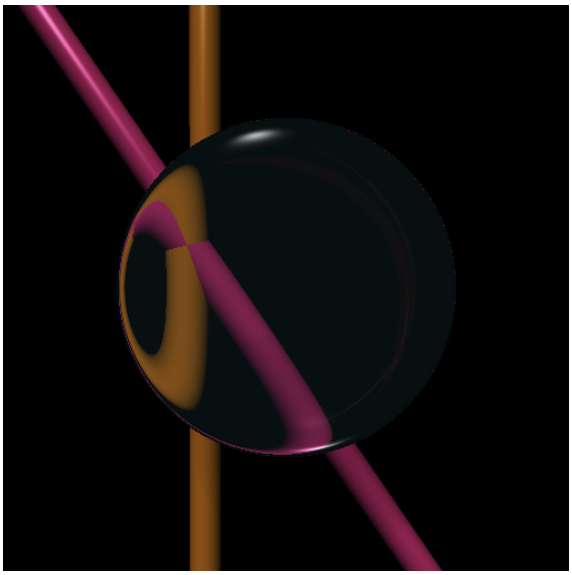




Debug output

Solution output





Cubemaps

Cubemap::getColor()

The important bulk of mapping the cubemap is done in these lines. Once the greatest magnitude component is found we then proceed to remap the direction vector from a (-1,1) range to the UVs from a (0,1) range and call the appropriate map. It should be noted that in some scenes the front/back maps are flipped but Dr. Abraham said that this was ok as long as the vertical orientation is preserved.

```
dir = r.getDirection();
dir[greatest] = abs(dir[greatest]);
glm::dvec2 coord(0,0);
if(greatest == 0){ // X-axis
    coord = glm::dvec2(dir[1], dir[2]);
    coord[0] = 0.5 * (dir[2] / dir[greatest] + 1);
    coord[1] = 0.5 * (dir[1] / dir[greatest] + 1);
    return r.getDirection()[greatest]>0 ? tMap[0]->getMappedValue(coord) : tMap[1]->getMappedValue(coord);
}
else if(greatest == 1) { // Y-axis
    coord = glm::dvec2(dir[0], dir[2]);
    coord[0] = 0.5 * (dir[0] / dir[greatest] + 1);
    coord[1] = 0.5 * (dir[2] / dir[greatest] + 1);
    return r.getDirection()[greatest]>0 ? tMap[2]->getMappedValue(coord) : tMap[3]->getMappedValue(coord);
}
else if(greatest == 2) { // Z-axis
    coord[0] = 0.5 * (dir[0] / dir[greatest] + 1);
    coord[1] = 0.5 * (dir[1] / dir[greatest] + 1);
    return r.getDirection()[greatest]>0 ? tMap[4]->getMappedValue(coord) : tMap[5]->getMappedValue(coord);
}
return r.at(1.0);
```

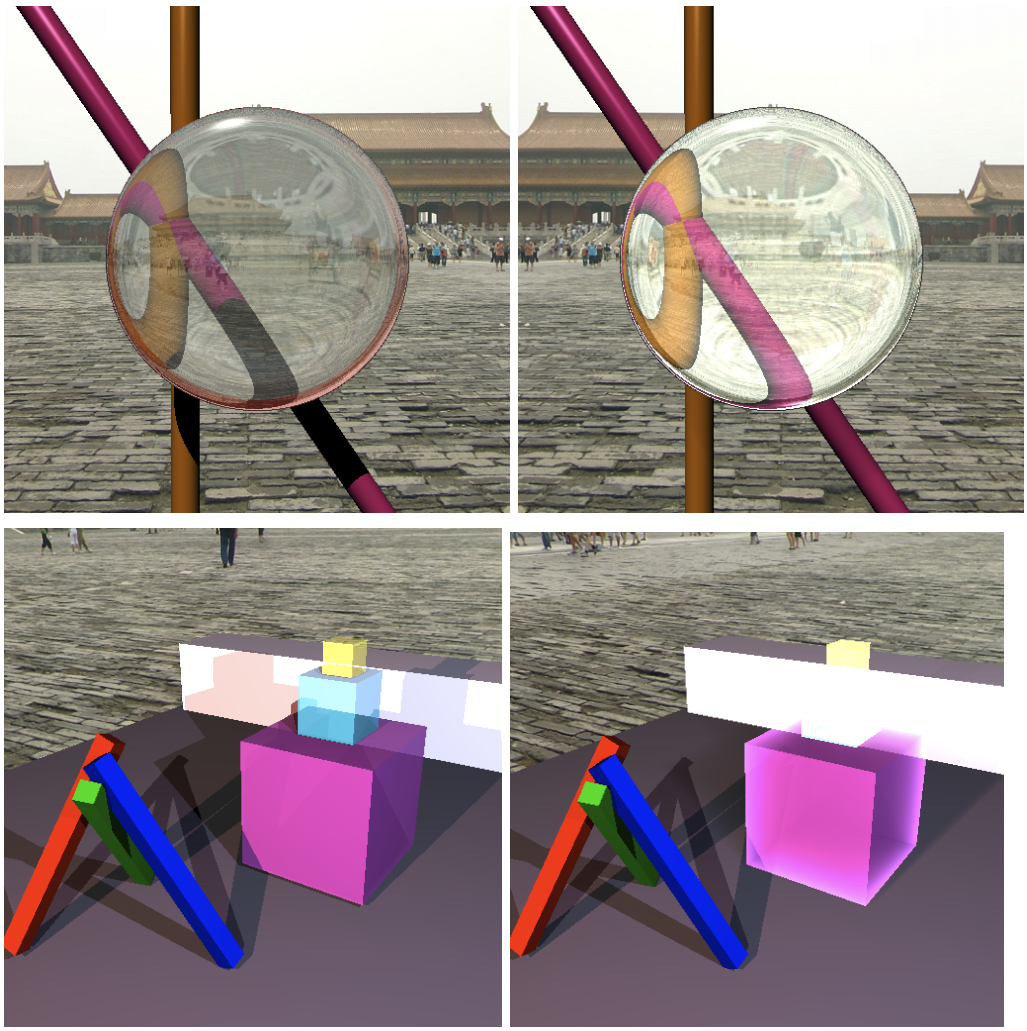
TextureMap::getMappedValue()

Mapping the coordinates to the bitmap is fairly simple as we just scale it to the width and height of the bitmap, and the code consists mainly of doing the bilinear interpolation.

```
double x = coord[0]*width;
double y = coord[1]*height;
int x1 = coord[0]*width;
int x2 = coord[0]*width+1;
x2 = glm::clamp(x2,0,width-1);
int y1 = coord[1]*height;
int y2 = coord[1]*height+1;
y2 = glm::clamp(y2,0,height-1);
glm::dvec3 pixQ11 = getPixelAt(x, y);
glm::dvec3 pixQ12 = getPixelAt(x1, y2);
glm::dvec3 pixQ21 = getPixelAt(x2, y1);
glm::dvec3 pixQ22 = getPixelAt(x2, y2);
glm::dvec3 fxy1;
glm::dvec3 fxy2;
if (x1!=x2) {
    fxy1 = (x2-x)/(x2-x1) * pixQ11 + (x-x1)/(x2-x1) * pixQ21;
    fxy2 = (x2-x)/(x2-x1) * pixQ12 + (x-x1)/(x2-x1) * pixQ22;
}
else{
    fxy1 = pixQ11;
    fxy2 = pixQ12;
}
glm::dvec3 bilerp;
if (y1!=y2)
    bilerp = (y2-y)/(y2-y1) * fxy1 + (y-y1)/(y2-y1) * fxy2;
else
    x1<=x2? bilerp = fxy1 : bilerp = fxy2;
return bilerp;
```

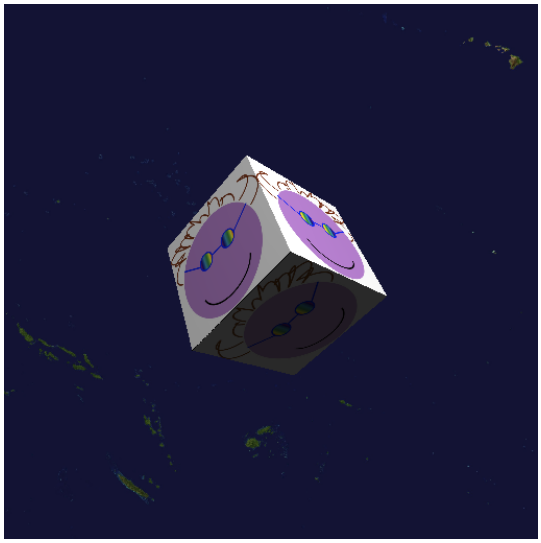
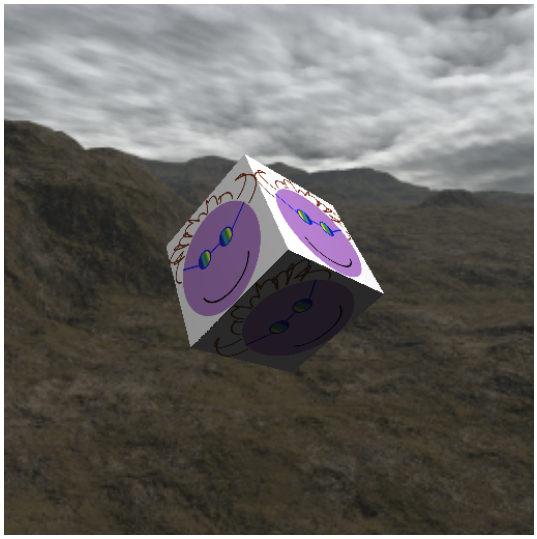
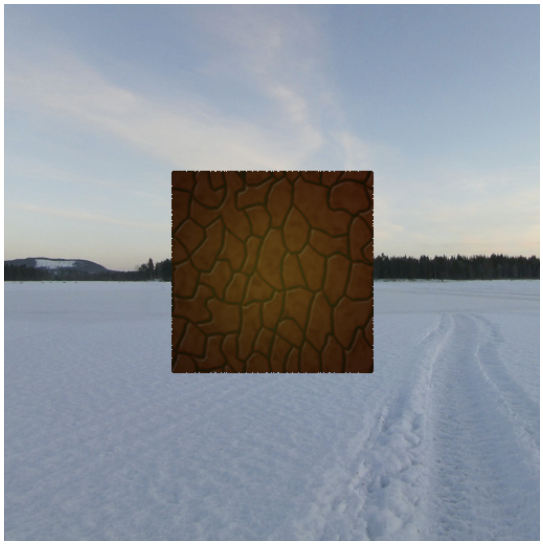
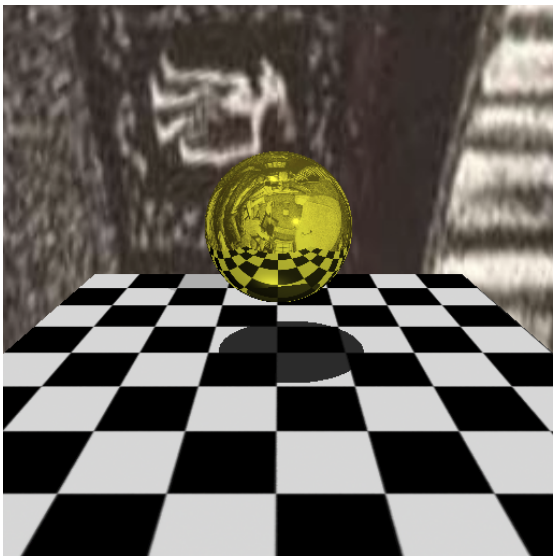
Differing examples:

(ours) (solution)



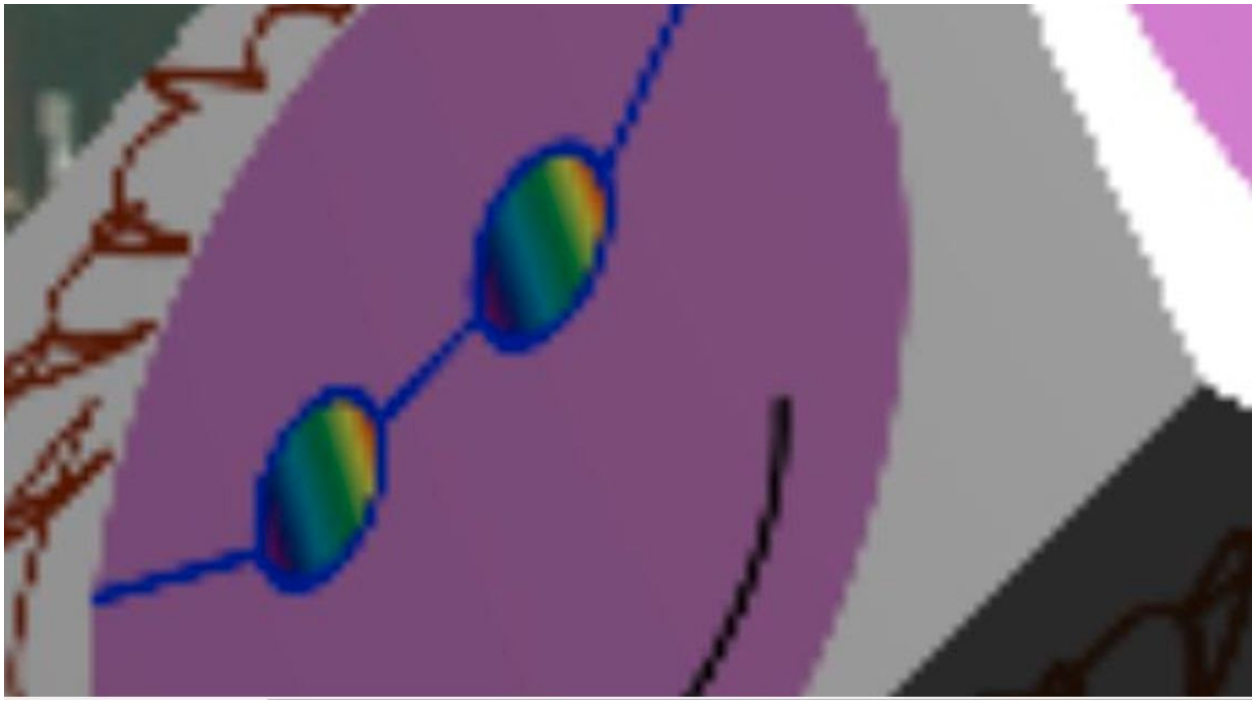
Some examples differed on the scenes that had refractive objects but this is a given because we only implement hard shadows and did not implement material absorption

Correct Examples:

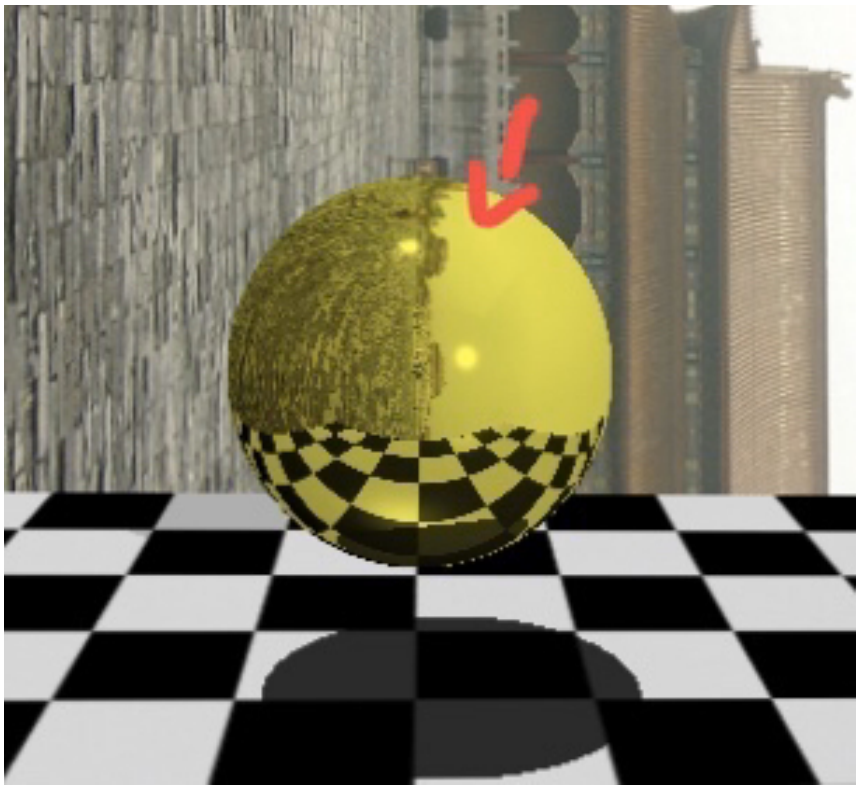
***note some of these have flipped x/z maps compared to the solution images**

Bilinear interpolation examples (kind of hard to capture the detail to show off the bilerp but it would basically “smooth” out the colors of lines):

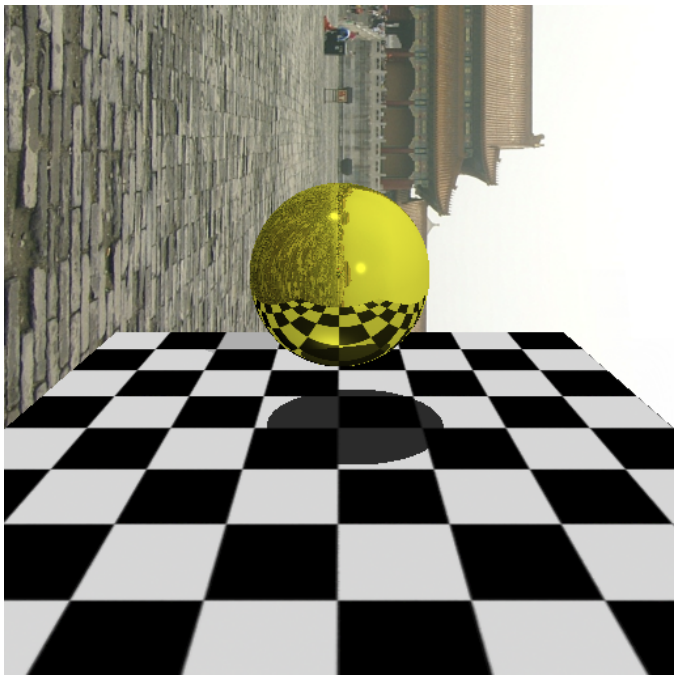


Slight bug that is apparent only in texture_map.ray. The refraction of the cubemap is slightly off. In the case of ForbiddenCity a slight bit of the sky is lighter than it should be as highlighted below.

(ours)



Solution



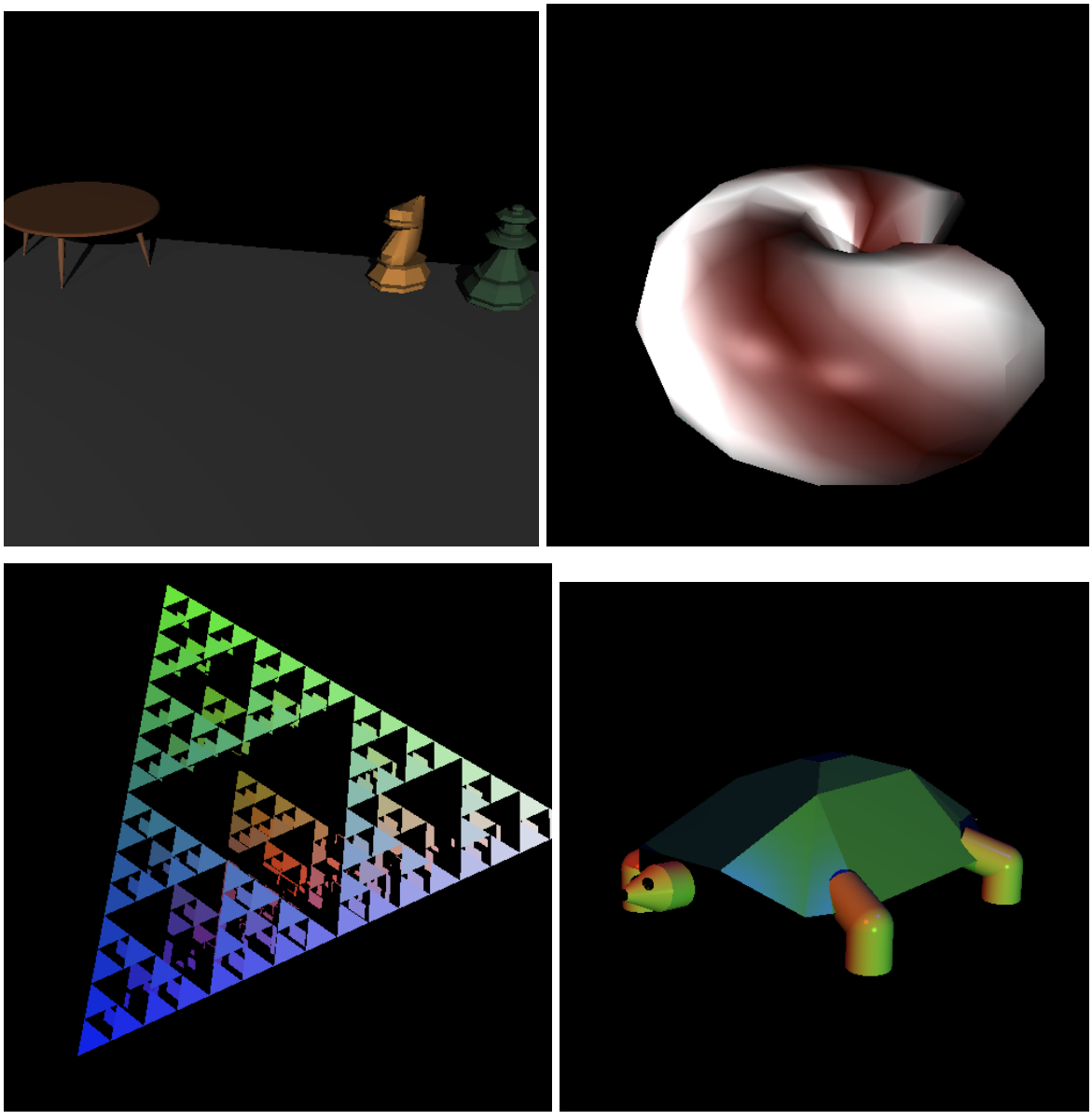
Trimesh

```
const glm::dvec3 tfNorm = normal;
i.setN(tfNorm);
i.setObject(this);
i.setMaterial(this->getMaterial());
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 midPoint =  (a_coords + b_coords + c_coords) * (1.0/3.0);
if (glm::dot(tfNorm,r.getDirection()) == 0.0) // Plane is perpendicular
    return false;
//calculate t value
double tIntersect = -1.0 * (double) (glm::dot(tfNorm,r.getPosition()) - glm::dot(tfNorm,a_coords))
/glm::dot(tfNorm,r.getDirection());
if (tIntersect < 0)
    return false;
glm::dvec3 pointQ = r.getPosition() + (tIntersect * r.getDirection());
// Do inside outside test to double-verify
//(c-b) x (q-b) . n >=0
//(a-c) x (q-c) . n >=0
bool insideOut = glm::dot(glm::cross(b_coords-a_coords,pointQ-a_coords),tfNorm) >= 0;
insideOut = insideOut && (glm::dot(glm::cross(c_coords-b_coords,pointQ-b_coords),tfNorm) >= 0);
insideOut = insideOut && (glm::dot(glm::cross(a_coords-c_coords,pointQ-c_coords),tfNorm) >= 0);
if (!insideOut)
    return false;

//area from Point a b c
glm::dvec3 area = glm::cross(b_coords-a_coords,c_coords-a_coords);
double aArea = (1.0/2.0) * glm::length(area);
// using Point B , C and Point Q get a subarea
glm::dvec3 aA = glm::cross(c_coords-b_coords,pointQ-b_coords);
double xA = glm::length(aA) * (1.0/2.0);
// using Point A , C and Point Q get b subarea
glm::dvec3 aB = glm::cross(a_coords-c_coords,pointQ-c_coords);
double xB = glm::length(aB) * (1.0/2.0);
// using Point A , B and Point Q get c subarea
glm::dvec3 aC = glm::cross(b_coords-a_coords,pointQ-a_coords);
double xC = glm::length(aC) * (1.0/2.0);

//greek values
double qAlpha = xA/aArea;
double qBeta  = xB/aArea;
double qGamma = xC/aArea;

bool inTriangle = (qAlpha >=0.0 && qBeta>=0.0 && qGamma >= 0.0)
&& (qAlpha+qBeta+qGamma >= 1-RAY_EPSILON && qAlpha+qBeta+qGamma <= 1+RAY_EPSILON  );
if (!inTriangle)
    return false;
i.setT(tIntersect);
i.setUVCoordinates(glm::dvec2(qAlpha,qBeta));
if (parent->vertNorms)
{
    glm::dvec3 a_norm = parent->normals[ids[0]];
    glm::dvec3 b_norm = parent->normals[ids[1]];
    glm::dvec3 c_norm = parent->normals[ids[2]];
    glm::dvec3 kd_Q = (qAlpha * a_norm) + (qBeta * b_norm) + (qGamma * c_norm);
    kd_Q = glm::normalize(kd_Q);
    //use interpolated normal instead
    i.setN(kd_Q);
}
if (parent->materials.size()!=0)
{
    Material* aMat = parent->materials[ids[0]];
    Material* bMat = parent->materials[ids[1]];
    Material* cMat = parent->materials[ids[2]];
    Material mixedMat = (qAlpha *  *aMat);
    mixedMat += (qBeta *  *bMat);
    mixedMat += (qGamma *  *cMat);
    i.setMaterial(mixedMat);
}
```

TrimeshFace::IntersectLocal()

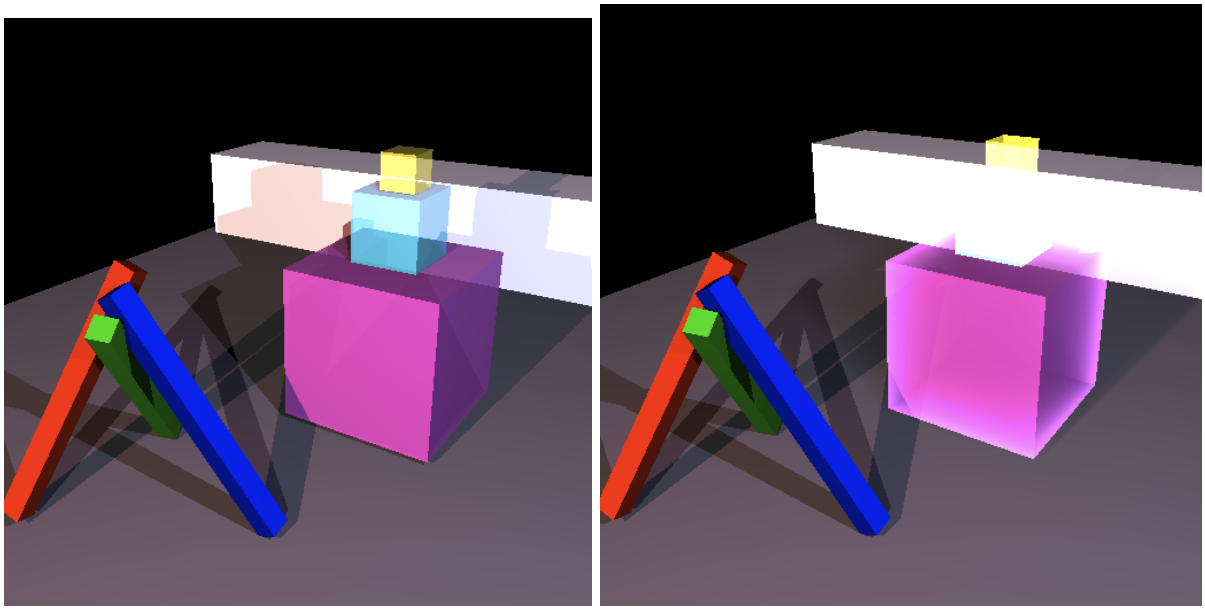
Trimesh code follows basically triangle intersection checking. Using the ray position and direction + the normal of the triangle and a point on the triangle the t-intersect value is calculated. Then the point of intersection (if it exists) is calculated, point Q. This point is used to do outside-inside tests and calculate subareas of the triangle. One those values are obtained the greek values (alpha,beta,gamma) are calculated from the subareas divided by the area. Greek values are checked for rules. Then interpolation of normals and materials are conducted using the greek values and values for the i-sect are properly set.

Correct examples:



Differing examples:

(ours) (solution)



This is expected as we did not implement material absorption.

Issues Encountered

- For some images, parts of the object contained a small grainy area throughout the object leaving it to, from afar look darker than usual, and from up close have thousands of tiny holes in the object

- Some images were completely black after all ray tracing was completed, though this was due to an issue of handling of the materials in trimesh

- Some of the given solutions for the ray files were quite simply incorrect

- Jumbled trimesh images

- Lots of wrongly flip vectors when doing calculations part of the raytrace method

Known bugs

1. Inconsistent output after running CMAKE in release mode. This affects our refraction scenes.
2. Cubemap reflection warping. In the texture_map scene the reflection on the sphere is not completely the same as the output image

Future Work.

At this current point in time anti aliasing is not planned anymore nor acceleration structures. Most likely threading will be attempted to implement in order to reduce time spent