https://gitlab.com/raymww/raytracing.git

We (Max Hartfield and Raymond Wang) implemented all required functionality including Phong Shading, Light attenuation and shadow rays, reflection, refraction, Ray-triangle intersection and use of barycentric coordinates, and Cube Mapping.

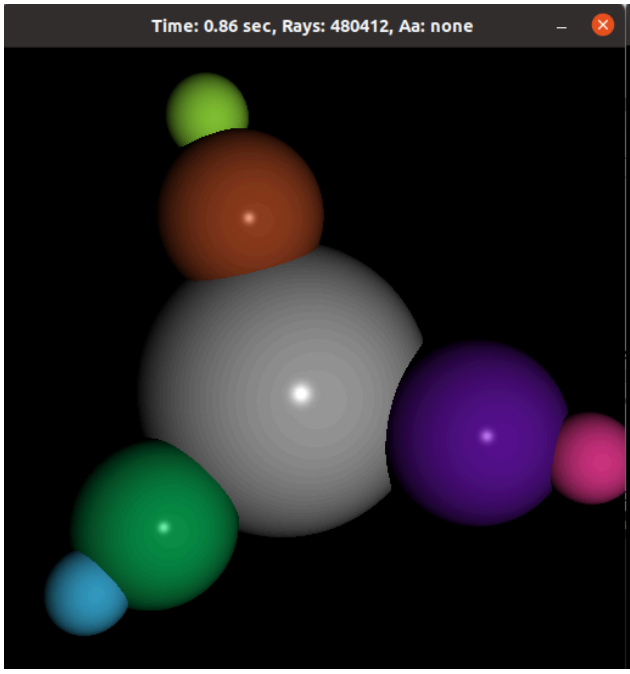
Phong Shading:

For Phong Shading, we implemented based on the formulas given in class for the attenuation of the different types of light rays, and looped through all the lights to attenuate for each light and the shadows from each light. This section was fairly straightforward and we didn't have many issues with this.

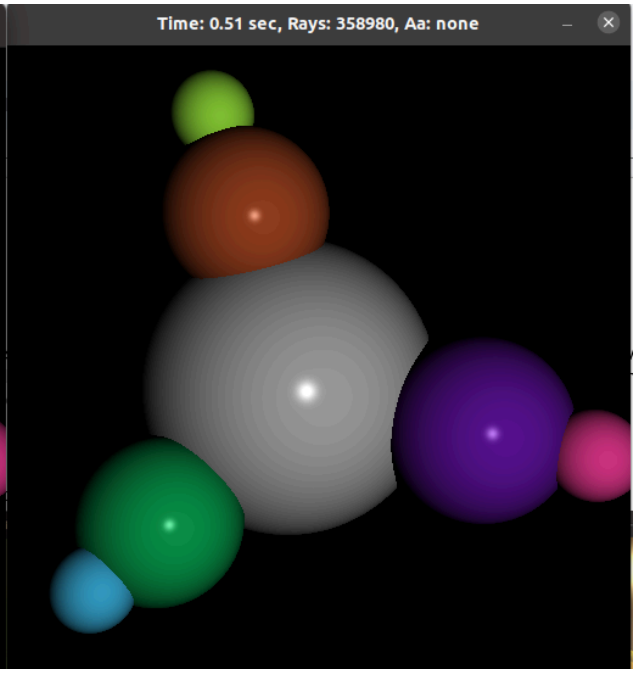
```
glm::dvec3 Material::shade(Scene *scene, const ray &r, const isect &i)
const {
 glm::dvec3 sum = glm::dvec3(0.0, 0.0, 0.0);
 //multiple lights
 for ( const auto& pLight : scene->getAllLights() )
 {
   // diffuse term
   glm::dvec3 temp = glm::dvec3(0.0, 0.0, 0.0);
   temp += kd(i) * pLight->distanceAttenuation(r.at(i)) *
max(glm::dot(pLight->getDirection(r.at(i)), i.getN()), 0.0);

   // specular term
   temp += ks(i) * pLight->distanceAttenuation(r.at(i))
     * pow(max(glm::dot(glm::reflect(pLight->getDirection(r.at(i)) * -1.0,
i.getN()), r.getDirection() * -1.0), 0.0), shininess(i));

   ray shadowRay(r.at(i) + pLight->getDirection(r.at(i)) * RAY_EPSILON,
pLight->getDirection(r.at(i)), glm::dvec3(0, 0, 0),
       ray::SHADOW);
   // shadow ray
   isect shadowI;
   if(scene->intersect(shadowRay, shadowI)) {
     temp *= pLight->shadowAttenuation(shadowRay, shadowI);
   }
   temp *= pLight->getColor();
   sum += temp;
```

```
}
return ke(i) + (ka(i) * scene->ambient()) + sum;
```

Easy2 Ours



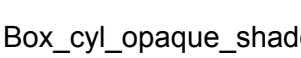
Easy2 Solution



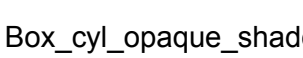
Light attenuation and shadow rays:
Shadow implementation was done through casting a ray starting at the position of intersection and in the direction of the light. For point light if the ray intersects, we check if the point of intersection was before or after the light. If it was before, then we know it is in shadow and if it is after it is not in shadow. We can do this by comparing the lengths of the vectors from the object position to the new point of intersection and the object position to the light position. The one algorithmic difference we have is the transmissive shadow attenuation because it was not required

```
glm::dvec3 DirectionalLight::shadowAttenuation(const ray &r,
                                               const isect &i) const {
```

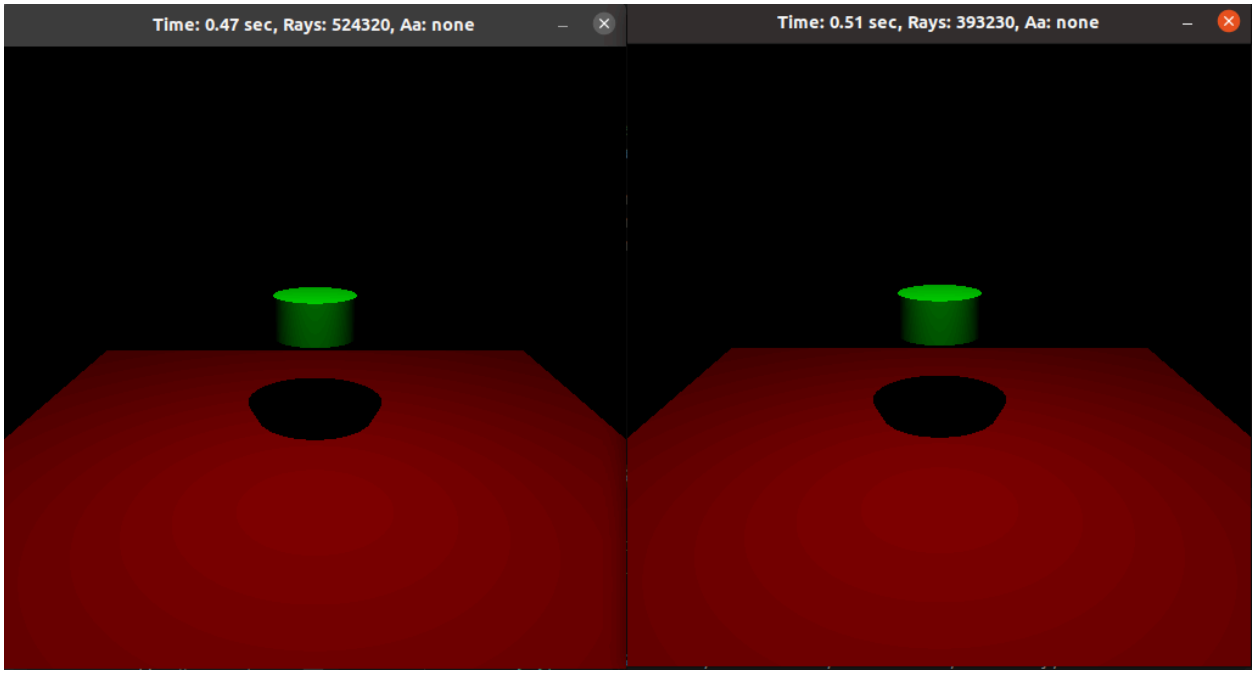
```
 if(i.getMaterial().Trans()){
   return glm::dvec3(1, 1, 1);
 }
 //we check if it intersects before we call this function
 return glm::dvec3(0, 0, 0);
}

double PointLight::distanceAttenuation(const glm::dvec3 &P) const {
 double d = length(position - P);
 return min(
     1.0, 1 / (constantTerm + (linearTerm * d) + (quadraticTerm * pow(d,
2))));
}
glm::dvec3 PointLight::shadowAttenuation(const ray &r,
                                         const isect &i) const {
 glm::dvec3 p = r.at(i);
 if(i.getMaterial().Trans()){
   return glm::dvec3(1, 1, 1);
 }
 double lenLight = glm::length(position - r.getPosition());
 double lenI = glm::length(p - r.getPosition());
 //if intersect point is between object and light we know its in shadow
 return lenI >= lenLight ? glm::dvec3(1, 1, 1) : glm::dvec3(0, 0, 0);
}
```



Box_cyl_opaque_shadow Ours



Box_cyl_opaque_shadow Solution

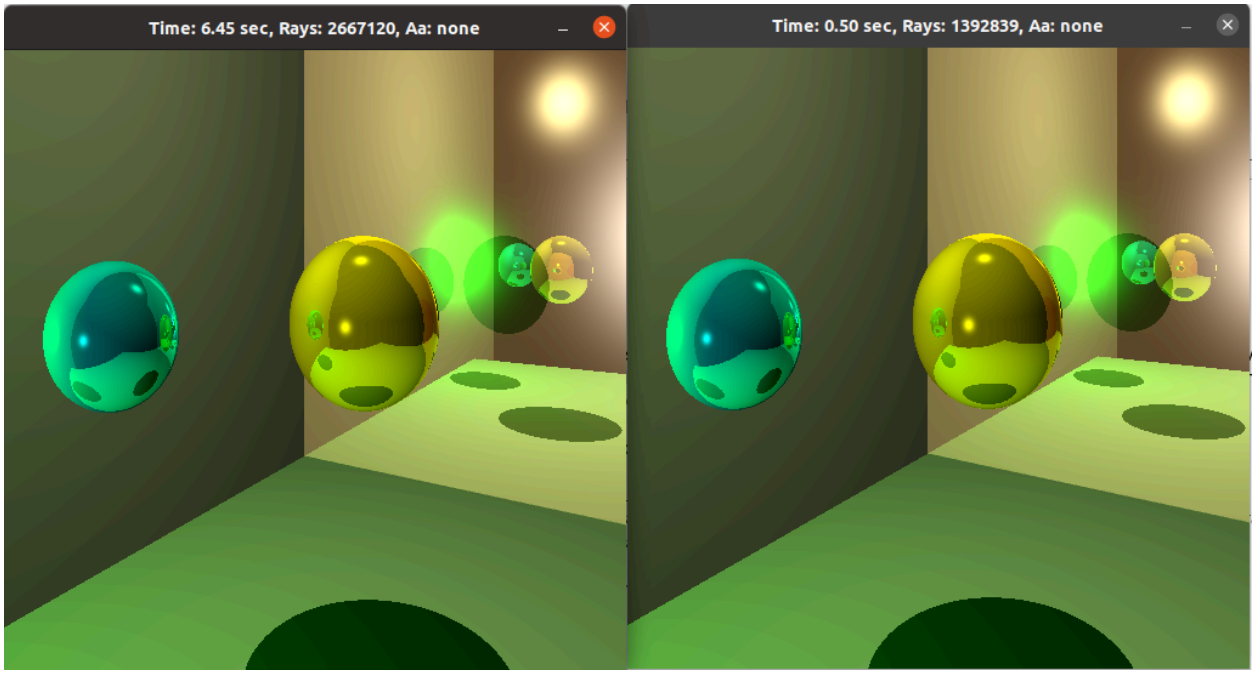


Reflection: Reflection was done by making a new recursive call of traceRay and adding those color values to the existing color values. The new ray was cast with position at the point of intersection and direction in the reflect direction. The reflect direction was easy to get since glm has a function for us and we didn't have to do any complicated math like shown on the slides.

```
//reflection
    if(m.Refl()) {
      glm::dvec3 reflectDir = glm::reflect(r.getDirection(), i.getN());
      ray reflect(r.at(i) + reflectDir * RAY_EPSILON, reflectDir, thresh,
              ray::REFLECTION);
      colorC += m.kr(i) * traceRay(reflect, thresh, depth - 1, t);
    }
```

Reflection2(depth 2) Ours

Reflection2(depth 2) Solution

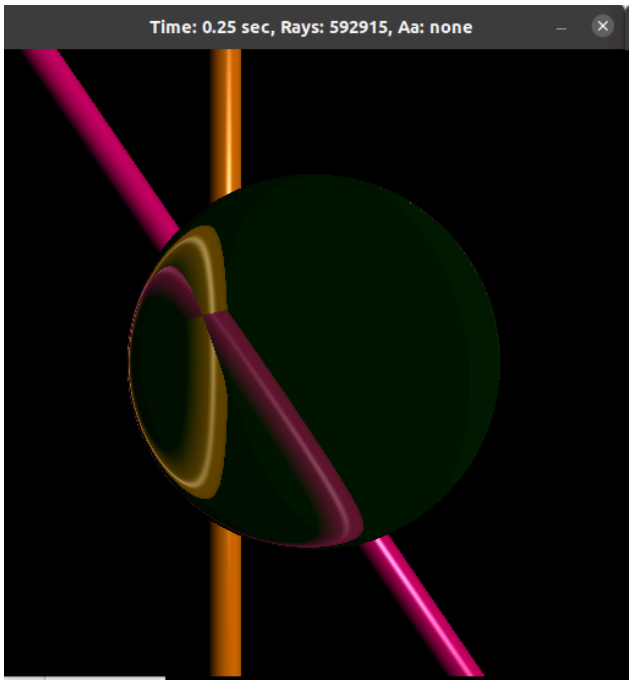


Refraction: Refraction was implemented in a similar way to reflection. It was done by making a new recursive call of traceRay and adding those color values to the existing color values. The new ray was cast with position at the point of intersection and direction in the refract direction. The refract direction was easy to get since glm has a function for us and we didn't have to do any complicated math like shown on the slides. Depending on if we are entering the object or leaving the object, ni, nt, and the normal get assigned different values.

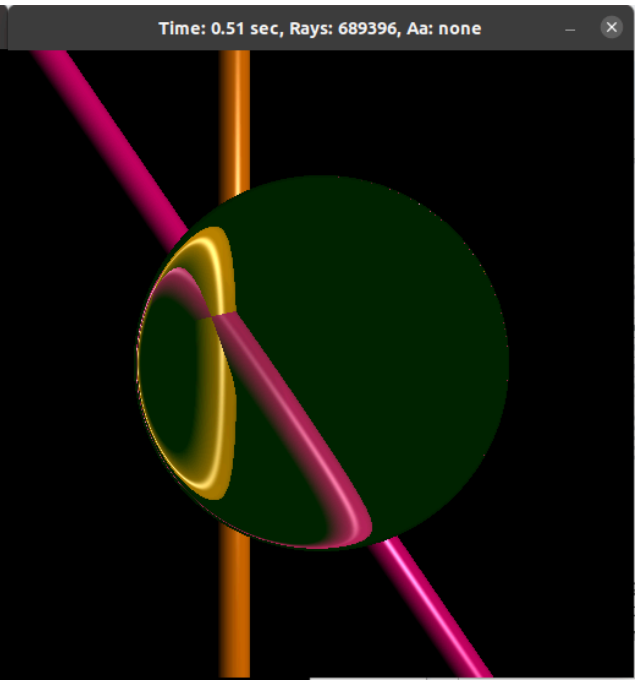
```
//refraction
   if(m.Trans()) {
     double ni = 0;
     double nt = 0;
     glm::dvec3 normal = i.getN();
     if (glm::dot(i.getN(), r.getDirection()) > 0) {
       ni = m.index(i);
       nt = 1;
       normal *= -1;
     } else {
       ni = 1;
       nt = m.index(i);
     }
     glm::dvec3 refractDir = glm::refract(r.getDirection(), normal, ni /
nt);
     if(glm::length(refractDir) != 0.0) {
       ray refract(r.at(i) + refractDir * RAY_EPSILON, refractDir, thresh,
         ray::REFRACTION);
```

```
        colorC += m.kt(i) * traceRay(refract, thresh, depth - 1, t);
    }
  }
```

sphere_refract Ours



sphere_refract Solution



Ray-triangle intersection and use of barycentric coordinates:

Triangle ray intersections were implemented pretty straightforwardly from what was learned in class, using cross products and the normals to determine whether the intersection is within the triangle or not, as well as then using the given formulas to calculate barycentric coordinates. We then interpolate as outlined by the function, interpolating normals, colors, and uv coordinates.

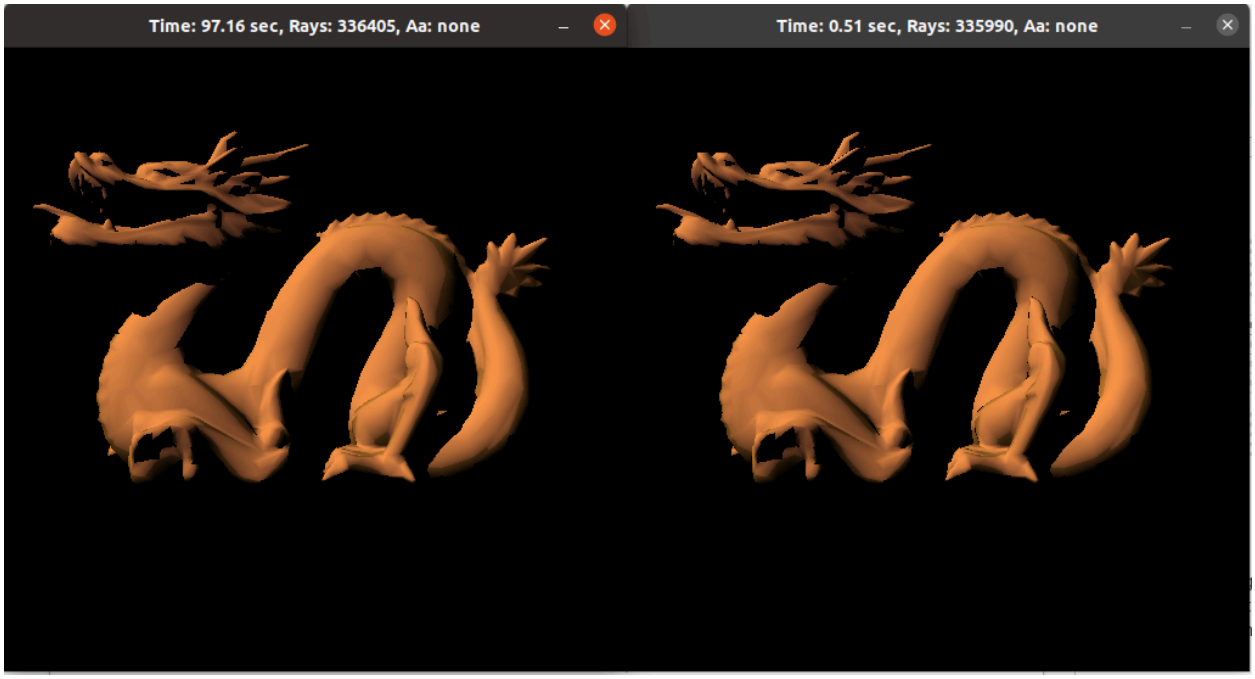
```
bool TrimeshFace::intersectLocal(ray &r, isect &i) const {
 glm::dvec3 Q;
 glm::dvec3 N;
 glm::dvec3 P = r.getPosition();
 glm::dvec3 a = parent->vertices[ids[0]];
 glm::dvec3 b = parent->vertices[ids[1]];
 glm::dvec3 c = parent->vertices[ids[2]];
```

```
 glm::dvec3 ab = b - a;
 glm::dvec3 bc = c - b;
 glm::dvec3 ca = a - c;

 N = glm::cross(ab, c - a);
 N = glm::normalize(N);
 i.setT((-1.0) * ((glm::dot(N, P)) - (glm::dot(N, a)))/(glm::dot(N,
r.getDirection())));

 Q = r.at(i);

 if(glm::dot(glm::cross(ab, Q - a), N) < 0 || glm::dot(glm::cross(bc, Q -
b), N) < 0 || glm::dot(glm::cross(ca, Q - c), N) < 0 || i.getT() <
RAY_EPSILON){
   // NOT IN TRIANGLE
   return false;
 }
  // in triangle, set barycentric
 double gamma = (glm::length(glm::cross(ab, Q - a)) / 2) /
(glm::length(glm::cross(ab, c - a)) / 2);
 double beta = (glm::length(glm::cross(ca, Q - c)) / 2) /
(glm::length(glm::cross(ab, c - a)) / 2);
 double alpha = 1 - gamma - beta;

 i.setBary(alpha, beta, gamma);

 if(!parent->uvCoords.empty()){
   glm::dvec2 uva = parent->uvCoords[ids[0]];
   glm::dvec2 uvb = parent->uvCoords[ids[1]];
   glm::dvec2 uvc = parent->uvCoords[ids[2]];
   glm::dvec2 uvi = glm::normalize((alpha * uva) + (beta * uvb) + (gamma *
uvc));
   i.setUVCoordinates(uvi);
 } else if (!parent->vertColors.empty()){
   // has color, need to interpolate
   assert(false);
   glm::dvec3 ca = parent->vertColors[ids[0]];
   glm::dvec3 cb = parent->vertColors[ids[1]];
```

```
   glm::dvec3 cc = parent->vertColors[ids[2]];
   glm::dvec3 ci = ((alpha * ca) + (beta * cb) + (gamma * cc));
   // setting material but with colors interpolated, so make a new
material
   Material mi = parent->getMaterial();
   mi.setDiffuse(ci);
   i.setMaterial(mi);
 } else {
   i.setMaterial(parent->getMaterial());
 }
 // normal interpolate
 // normal at i is alpha * normal(a) + beta * normal(b)
 if(!parent->normals.empty()){
   glm::dvec3 na = parent->normals[ids[0]];
   glm::dvec3 nb = parent->normals[ids[1]];
   glm::dvec3 nc = parent->normals[ids[2]];

   glm::dvec3 ni = glm::normalize((alpha * na) + (beta * nb) + (gamma *
nc));
   i.setN(ni);
 } else {
   i.setN(N);
 }
 i.setObject(this->parent);
 return true;
}
```

dragon Ours (under 180 sec)

dragon Solution



CubeMap:

Cubemaps were implemented by first using the most dominant axis of traversal to determine the face of the cubemap. We then normalize the other directions of traversal into UV coordinates, which are mapped to regular coordinates through getMappedValue and then bilinearly interpolated. We then added cubemap interpolation to when rays do not intersect. We encountered a couple difficulties here. The given slide's directions for UV as well as the direction for Z were not accurate and we had to modify the directions for the cubemap to show on the right faces. Additionally, we encountered a bug where there was a thin black line that was the result of not bounding the coordinates between the width / height on the interpolation.

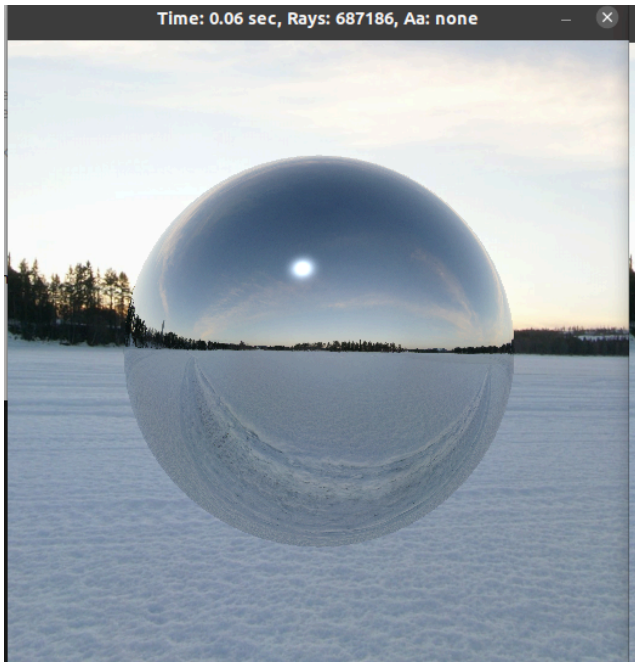
```
glm::dvec3 TextureMap::getMappedValue(const glm::dvec2 &coord) const {
 double x = coord.x * (getWidth() - 1);
 double y = coord.y * (getHeight() - 1);

 int x0 = (int) floor(x);
 int y0 = (int) floor(y);
 int x1 = x0 + 1;
 int y1 = y0 + 1;

 if(x1 == getWidth()){
```

```
    x1 -= 1;
    x0 -= 1;
  }
  if(y1 == getHeight()){
    y1 -= 1;
    y0 -= 1;
  }

  //getting nearest four pixels
  glm::dvec3 bottomLeft = getPixelAt(x0, y0);
  glm::dvec3 bottomRight = getPixelAt(x1, y0);
  glm::dvec3 topLeft = getPixelAt(x0, y1);
  glm::dvec3 topRight = getPixelAt(x1, y1);

  //ratios
  double rx = (x - x0) / (x1 - x0);
  double ry = (y - y0) / (y1 - y0);

  //interpolation
  glm::dvec3 cTop = topLeft * (1 - rx) + topRight * rx;
  glm::dvec3 cBottom = bottomLeft * (1 - rx) + bottomRight * rx;

  return cBottom * (1 - ry) + cTop * ry;
}
glm::dvec3 TextureMap::getPixelAt(int x, int y) const {
  //find index in data array
  int i = 3 * (getWidth() * y + x);
  return glm::dvec3(data[i] / 255.0, data[i + 1] / 255.0, data[i + 2] /
255.0);
}
glm::dvec3 CubeMap::getColor(ray r) const {
  // YOUR CODE HERE
  // FIXME: Implement Cube Map here
  glm::dvec3 dir = r.getDirection();
  double x = dir[0];
  double y = dir[1];
  double z = dir[2];
  bool xPositive = x > 0;
  bool yPositive = y > 0;
  bool zPositive = z > 0;
```

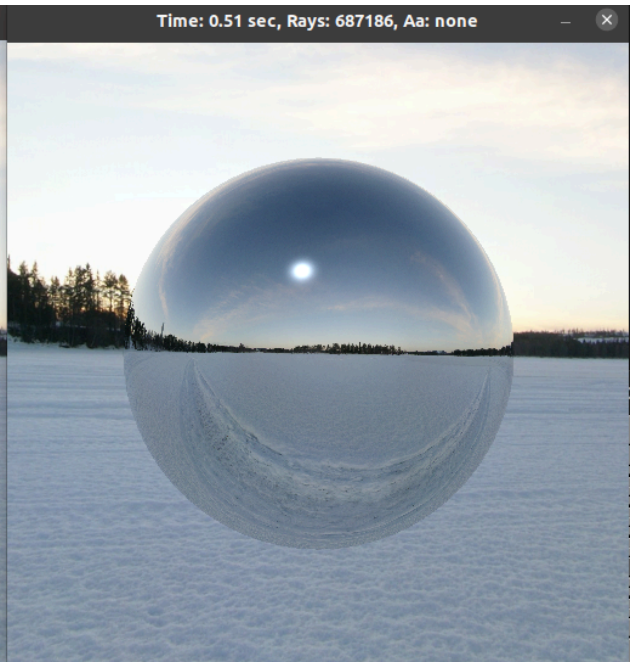
```
double absX = fabs(x);
double absY = fabs(y);
double absZ = fabs(z);
double dominant, u, v;
int i = 0;

//handling all 6 faces
if(xPositive && absX >= absY && absX >= absZ) {
  dominant = absX;
  u = y;
  v = z;
  i = 0;
} else if(!xPositive && absX >= absY && absX >= absZ) {
  dominant = absX;
  u = y;
  v = -1.0 * z;
  i = 1;
} else if(yPositive && absY >= absX && absY >= absZ) {
  dominant = absY;
  u = z;
  v = x;
  i = 2;
} else if(!yPositive && absY >= absX && absY >= absZ) {
  dominant = absY;
  u = -1.0 * z;
  v = x;
  i = 3;
} else if(!zPositive && absZ >= absY && absZ >= absX) {
  dominant = absZ;
  u = y;
  v = x;
  i = 4;
} else {
  dominant = absZ;
  u = y;
  v = -1.0 * x;
  i = 5;
}
//scale to [0, 1]
u = .5 * (u / dominant + 1);
```

```
 v = .5 * (v / dominant + 1);
 return tMap[i]->getMappedValue(glm::dvec2(v, u));
}
```

sphere_reflect with iceriver Ours



sphere_reflect with iceriver Solution



Issues we encountered and fixed:

Phong shading: We were calculating sum in the light for loop and multiplying the entire sum instead of individual sums by shadow attenuation and color of light. Fixed by making a temp variable to store individual sums.

Light attenuation and shadow rays: Some of the shadow scenes weren't working. Fixed by using RAY_EPSILON.

Reflection: Scene wasn't showing because the base case for trace ray was wrong. Fixing the base case to depth has to be below zero. Also some of the reflection looked wrong this was due to the direction being not multiplied by -1 correctly.

Refraction: Was barely showing refraction, fixed by changing the normal direction depending on if we are entering or exiting an object. Also, for some reason the object was in shadow in some

places when it shouldn’t be so we decided to make it so that if the material has refractive properties to never be in shadow.

Ray-triangle intersection and use of barycentric coordinates: Fully black scene due to the intersect function returning true and always being in shadow when it wasn’t intersecting. Fixed by returning false if t was less than RAY_EPSILON. Also we calculated alpha and beta and gamma in the wrong order so we had to swap some values.

Cube Mapping: Setting U V from the slides made the wrong face show up. We fixed this by trial and error and adjusting until the faces were correct. Also for reflect + cube map we had a random thin black line through the object. This was due to us clamping the coordinates so we got rid of this and it worked.

Known bugs: In objmesh, treelog and glider have the correct textures and everything but the color is warped. This is the only bug we could find and everything else works.

Future work: We finished everything for A1, but for A2 we will work on Anti-aliasing and k-d tree.

Optimizations Made: We added multi-threading support to the casting of rays. This reduced our runtime significantly, and made dragon run at 90 seconds, down from 550.

Algorithmic Differences that result in differences from the solution: We did not implement transmissive attenuation because it wasn’t required. This made our refract scenes to be slightly darker than the solution.