Assignment 4: Advanced Ray Tracing

The set of tasks will be about an advanced image-order ray-tracing graphics pipeline. This time, there is no strict minimal requirement, all features are optional and each feature gives you a certain number of points.

A new set of meshes (from the history of computer graphics) that should be used for the purpose of this assignment can be found here: https://www.cs.utah.edu/~natevm/newell_teaset/newell_teaset.zip

1. Implement triangle meshes support into your existing ray-tracing system: On top of the quadrics that you have already implemented for ray-object intersection, implement the intersection test for triangle meshes assignment (20 points).

```
//build the classified polygon table
polygon.dy = round(max_y) - round(min_y);
if (polygon.dy > 0 && max_y > 0 && min_y < height)
{
    Point3f v = model.vertexes[face.vertexIdx[0]].point;
    polygon.a = face.normal.x;
    polygon.b = face.normal.y;
    polygon.c = face.normal.z;
    polygon.d = -(polygon.a*v.x + polygon.b*v.y + polygon.c*v.z);
```

2. In a pre-processing step, for triangle meshes calculate **per-face normals** and from these calculate per-vertex normals by averaging the normals of all faces that contain the currently processed vertex. Do not forget to normalize these normals again (15 points).

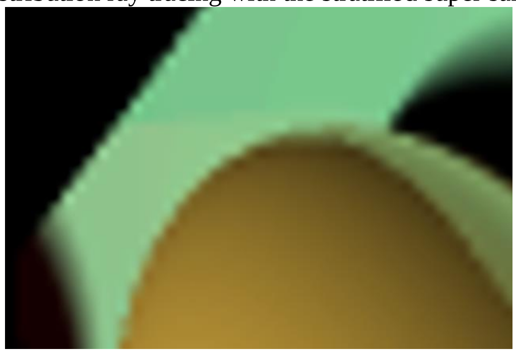
```
    //  calculate the normal vector determined by 3 points in the bin

    if (face.vertexIdx.size() > 2)
    {
        Point3f &a = vertexes[face.vertexIdx[0]].point,
            &b = vertexes[face.vertexIdx[1]].point, &c = vertexes[face.vertexIdx[2]].point;
        Vec3f& normal = normalize(cross(b - a, c - b));// find the normal and ultilize

        face.normal = normal;
        faces.push_back(face);

    }
}
```

3. In runtime, calculate shading with per-vertex normals that are interpolated using the barycentric coordinates. Do not forget to normalize these normals after each interpolation stage (10 points).

```
#pragma omp parallel for
    for (int i = 0; i < face_num; ++i)
    {
        Face& face = model.faces[i];
        int face_vertex_num = face.vertexIdx.size();
        for (int j = 0; j < face_vertex_num; ++j)
        {
            Vertex face_vertex = model.vertexes[face.vertexIdx[j]];
            Vec3f ray_direction = normalize(light_position - face_vertex.point);//
            Vec3f normal = face.normalIdx[j] >= 0 ?
                model.normals[face.normalIdx[j]]:face.normal;
            float cosine = dot(ray_direction, normal);// find the angle (cos) between the opposite direction of the
            // incident light and the normal to the surface / vertex normal
            if (cosine>0.0)face.color += kd*cosine*light_color;// the scattering color of the point source
            face.color += ambient_color;// increase the color of the environment
        }
        face.color /= face.vertexIdx.size();
        // the color of the small polygon takes the average color of the vertex

        // control color range from 0 to 1
        if (face.color.r > 1.0f)face.color.r = 1.0f;
        if (face.color.r < 0.0f)face.color.r = 0.0f;
        if (face.color.g > 1.0f)face.color.g = 1.0f;
        if (face.color.g < 0.0f)face.color.g = 0.0f;
        if (face.color.b > 1.0f)face.color.b = 1.0f;
        if (face.color.b < 0.0f)face.color.b = 0.0f;
    }
```

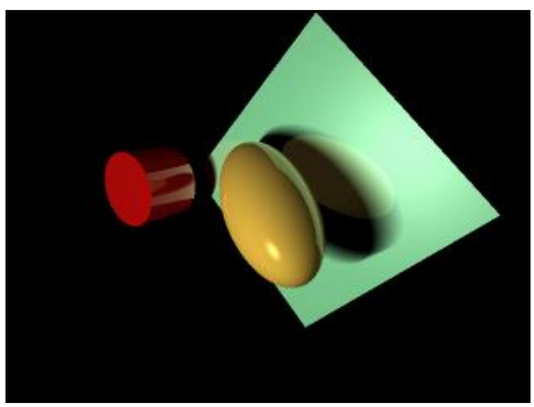
---

8. Implement antialiasing using distribution ray tracing with the stratified super sampling technique (15 points).



```
        float row_subdivide[2] = {0.0, 0.5};
        float col_subdivide[2] = {0.0, 0.5};

#pragma omp parallel for schedule(dynamic, 1)
        for (int rowAntiAlias = 0; rowAntiAlias < num_offsets; rowAntiAlias++) {
          for (int colAntiAlias = 0; colAntiAlias < num_offsets; colAntiAlias++) {

            Point3D origin(0, 0, 0);
            Point3D imagePlane;

            imagePlane[0] = (-double(width)/2 + j + col_subdivide[colAntiAlias])/factor;
            imagePlane[1] = (-double(height)/2 + i + row_subdivide[rowAntiAlias])/factor;
            imagePlane[2] = -1;
```

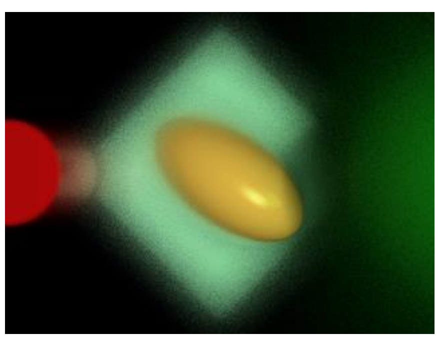
9. Implement soft shadows using distribution ray tracing (20 points).



```cpp
    ////////////////////////////////////////////////////////////////////////////
    // --ADVANCED RAY TRACING--
    // Shadows, INCLUDING soft shadows
    ////////////////////////////////////////////////////////////////////////////
    Colour rayCol;

    Vector3D l;

    for (float i = -1.0; i < 1.0; i += 0.05) {
      l = curLight->light->get_position() - ray.intersection.point;
      l[0] += i;
      l[1] += i;
      l[2] += i;
      double t_val = l.length();
      l.normalize();


      Ray3D r = Ray3D(ray.intersection.point + 0.005 * l, l);
      traverseScene(_root, r);      // shoot the ray
      curLight->light->shade(ray);

      bool isNotInShadow = (r.intersection.none || t_val < r.intersection.t_value);

      if (isNotInShadow) {
        rayCol = rayCol + 0.025 * ray.col;
      }

    }
    ray.col = rayCol;
    curLight = curLight->next;
  }
```

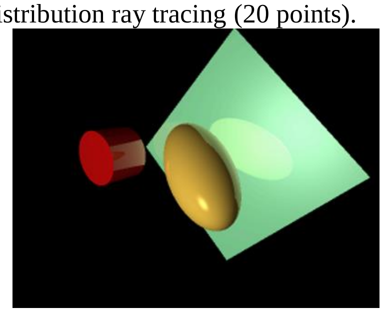
```cpp
void Raytracer::initPixelBuffer() {
  int numbytes = _scrWidth * _scrHeight * sizeof(unsigned char);
  _rbuffer = new unsigned char[numbytes];
  _gbuffer = new unsigned char[numbytes];
  _bbuffer = new unsigned char[numbytes];
  for (int i = 0; i < _scrHeight; i++) {
    for (int j = 0; j < _scrWidth; j++) {
      _rbuffer[i*_scrWidth+j] = 0;
      _gbuffer[i*_scrWidth+j] = 0;
      _bbuffer[i*_scrWidth+j] = 0;
    }
  }
}
void Raytracer::flushPixelBuffer( char *file_name ) {
  bmp_write( file_name, _scrWidth, _scrHeight, _rbuffer, _gbuffer, _bbuffer );
  delete _rbuffer;
  delete _gbuffer;
  delete _bbuffer;
}

Colour Raytracer::shadeRay( Ray3D& ray ) {
  Colour col(0.0, 0.0, 0.0);
  traverseScene(_root, ray);

  if (!ray.intersection.none) {
    computeShading(ray);
```

10. Implement depth of field effect using distribution ray tracing (20 points).



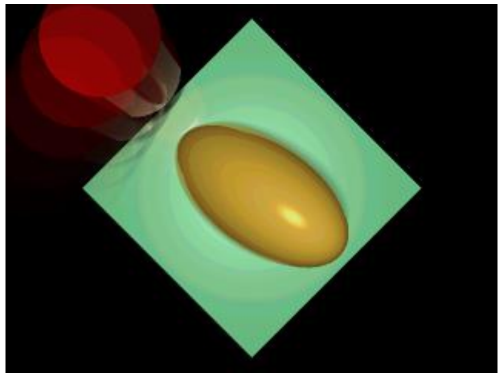
```
//////////////////////////////////////////////////////////////////
// --ADVANCED RAY TRACING--
// Depth of field
//////////////////////////////////////////////////////////////////
if (EXECUTE_DEPTH_OF_FIELD) {
  Vector3D ray_direction = imagePlane - origin;
  ray_direction.normalize();
  double t_val = FOCUS_PLANE_POINT_Z / ray_direction[2];


  Point3D intsctPtFocus = Point3D(t_val * ray_direction[0], t_val * ray_direction[1], t_val * ray_direction[2]);

  Colour DOFcolour;

#pragma omp parallel for schedule(dynamic, 1)
  for (int dof_iter = 0; dof_iter < DOF_RAYS_CAST; dof_iter++) {
    double angle = randomise(0, 2 * M_PI);
    double radius = randomise(0, APERTURE_SIZE);

    Point3D current_ray_origin = Point3D(radius * cos(angle), radius * sin(angle), 0);

    Ray3D ray;
    ray.origin = viewToWorld * current_ray_origin;
    ray.dir = viewToWorld * (intsctPtFocus - current_ray_origin);

    DOFcolour = DOFcolour + shadeRay(ray);
  }

  DOFcolour = (double) 1.0 / DOF_RAYS_CAST * DOFcolour;
  _rbuffer[i*width+j] += int(DOFcolour[0]*255/num_antialias_rays);
  _gbuffer[i*width+j] += int(DOFcolour[1]*255/num_antialias_rays);
  _bbuffer[i*width+j] += int(DOFcolour[2]*255/num_antialias_rays);
    if (EXECUTE_MOTION_BLUR)
      this->translate(cylinder, Vector3D(0.3, 0.5, 0));
  }
  if (EXECUTE_MOTION_BLUR)
    this->translate(cylinder, Vector3D(-0.3 * num_blurs, -0.5 * num_blurs, 0));

  flushPixelBuffer(fileName);
}
```

11. Implement glossy reflection using distribution ray tracing (20 points).



```
    Vector3D v = -ray.dir;
    v.normalize();

    Vector3D n = ray.intersection.normal;
    n.normalize();

    Vector3D reflectedVector = -v - 2 * ((-v).dot(n)) * n;    // the mirrored/reflected direction given n and v
    reflectedVector.normalize();

    Ray3D reflectedRay = Ray3D(ray.intersection.point + 0.005 * reflectedVector, reflectedVector);

    shadeRay(reflectedRay);

    if (reflectedRay.intersection.t_value > 0 && reflectedRay.intersection.t_value < 5.0) {
      double reflectionBackoff = fabs(1.0 / reflectedRay.intersection.t_value);
      if (reflectionBackoff > 0.75) {
        reflectionBackoff = 0.75;
      }
      col = ray.col + reflectionBackoff*reflectedRay.col;
    } else {

      col = ray.col;
    }
    col.clamp();
  }

  return col;
}
```

12. Implement motion blur using distribution ray tracing (20 points).



```
  for (int blur_iters = 0; blur_iters < num_blurs; blur_iters++) {
    // Construct a ray for each pixel.
#pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < _scrHeight; i++) {
      for (int j = 0; j < _scrWidth; j++) {
```

```
            if (!EXECUTE_MOTION_BLUR) {
              _rbuffer[i*width+j] += int(col[0]*255/num_antialias_rays);
              _gbuffer[i*width+j] += int(col[1]*255/num_antialias_rays);
              _bbuffer[i*width+j] += int(col[2]*255/num_antialias_rays);
            } else {
               executing motion blur
              _rbuffer[i*width+j] += int(col[0]*255)/num_antialias_rays/pow(2.0, num_blurs - blur_iters);
              _gbuffer[i*width+j] += int(col[1]*255)/num_antialias_rays/pow(2.0, num_blurs - blur_iters);
              _bbuffer[i*width+j] += int(col[2]*255)/num_antialias_rays/pow(2.0, num_blurs - blur_iters);
            }
          }
        }
      }
    }
  }
    if (EXECUTE_MOTION_BLUR)
      this->translate(cylinder, Vector3D(0.3, 0.5, 0));
  }
  if (EXECUTE_MOTION_BLUR)
    this->translate(cylinder, Vector3D(-0.3 * num_blurs, -0.5 * num_blurs, 0));

  flushPixelBuffer(fileName);
}
```