MM 804

Assignment 2

Herb Yang

Nov. 2, 2015

Total marks: 90

Due date **Nov. 24, 2015** before midnight

Submit your assignment electronically to eClass. Organize your files properly into folders and zip up all the folders that correspond to a particular question into one single zip file for submission.

Important notes:

- Midnight denotes the beginning of a day. The due date means the time just before turning into the day specified. The latest time stamp of your files is used as the submission time.
- A late penalty of 10% per day applies to all late submissions. The maximum number of days late permitted is 2. After 2 days late, your assignment will be given a mark of **ZERO**.
- You are expected to complete the assignment on your own **without** collaboration with others. Discussions at the conceptual level but not at the coding level are permitted.
- Do not wait until the last minute to work on your assignment. Start as soon as possible. Debugging programs and understanding of the materials take time.
- For the programming part,
  - Your code should have sufficient comments to make it readable.
  - Include a README file to document features or bugs, if any, of your program.

Marking scheme:

- Full assigned mark - no observable bug
- Half of the assigned mark - observable bugs
- 0 marks - does not work at all

Raytracing: This problem helps you understand raytracing as discussed in class. A simple raytracing algorithm can be summarized in Algorithm 1. In the provided codes, there are three separate folders, namely, `camera`, `scene`, and `objects`. Each folder is for each of the following problems. To facilitate your work when you are working on one problem, say, `camera`, all the other missing codes are provided in the form of a library, which has been set up in the corresponding solution file. To test your program, you can open the provided scene files, i.e. files with extension `.txt`. The syntax of the scene file is specified in `parser.h`. Most of the syntax is self-explanatory. For the camera declaration, (ruX, ruY) and (llX, llY) refer to the coordinates of the upper right and lower left corners of the viewing rectangle, i.e. (b_u, b_v) and (a_u, a_v) using the same notation as in the lectures. The viewing distance, which is the distance of the image plane from the origin of the camera, is always positive.
Some useful classes such as `vector`, `ray`, and `color` are defined in the corresponding `.h` files, i.e. `vector3.h` , `ray.h` and `color.h`. Their implementations are in the supplied libraries.

Algorithm 1 A simple raytracing algorithm

```
for pixel p ∈ I do
  emit a ray
  if there is an intersection with object O then
    if O is diffuse then
      add local diffuse colour to shading
    end if
    if O is specular then
      add local specular colour to shading
      emit a reflected ray to determine indirect colour
      add indirect colour to shading
    end if
  end if
end for
```

1. (10 marks) Camera.

   (a) (5 marks) `void Camera::createONB(const Vector3& a, const Vector3& b)` - In `camera.h`, `basis` is declared as the `ONB basis`. `ONB` stands for orthonormal basis. `createONB` uses the inputs **a** and **b** to compute the basis. **a** is the gaze vector and **b** the view up vector. See text or notes for details.

   (b) (5 marks) `Vector3 Camera::fromCameraToWorld(Vector3 & p)` - transforms vector **p** specified in the ONB of the camera coordinate system to the world coordinate system. See text or notes for details.

   This part should be the easiest of all to complete.

2. (30 marks) Scene

   (a) (6 marks) `double Scene::rayToLight(Ray *ray, Vector3 *point, PointLight *ptLight )` - creates `ray` from `point` to point light `ptLight`. The return includes a ray (`ray`) and the distance from `point` to point light`ptLight`. Take a look at `ray.h`. You may find some provided functions useful to complete this function.

   (b) (10 marks) `Color Scene::getPointLightColor(Ray *ray, PointLight *ptLight, double distance)` - returns the colour of light if there is no obstruction. If there is an occlusion by an object, then returns black. Note, all objects are dynamically created and are subclasses of `geomObj`. The basic idea is to go through each object and check for intersection for each object. Remember, each object provides its own intersection function.

   (c) (8 marks) `geomObj* Scene::findFirstIntersection( Ray *ray, RTfloat *k, RTfloat k0, RTfloat k1)` - determines the first, i.e. closest, intersection of the ray with an object in the scene within a given interval $[k_0, k_1]$. This function uses the intersection routine of the object to find the closest distance. In fact, assuming that the intersection routine is implemented correctly, then this function simply goes through each object and then finds the one that intersects the ray.

   (d) (3 marks) `Ray Scene::reflectedRay(Ray ray, Vector3 point, Vector3 normal)` - `ray` is the incoming ray, `point` the point of intersection and `normal` the normal at the intersection. This function computes the reflected ray. See text for details.

(e) (3 marks) `double Scene::cosViewerToReflectedRay( Ray ray, Vector3 point, Vector3 normal, Camera cam)` - `ray` is the ray to the light from the point of intersection, `point` and `normal` are, respectively, the point of intersection and the corresponding normal, `cam` is the camera. This function returns the cosine of the angle between the direction to the viewer and the reflected ray (see Figure 1 below).

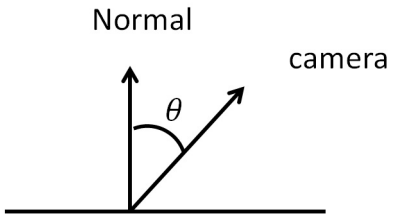


Figure 1. Angle between the viewer and the normal.

.

3. (50 marks) Objects

   (a) (5 marks) Disk. `bool Disk::intersect(RTfloat *k, Ray *ray, RTfloat k0, RTfloat k1)` - returns true if there is an intersection of `ray` with this object within the specified range $[k_0, k_1]$. When there is an intersection, `k` contains the corresponding value. See text for details.

   (b) (15 marks) Polygon

      i. (3 marks) `Plucker::Plucker(Vector3 a, Vector3 b)` - constructs the Plücker coordinates for a line which goes through point **b** and has direction **a**.

      ii. (2 marks) `double Plucker::operator% (const Plucker& v) const` - an overloaded operator that computes the permuted inner product of two lines given in Plücker coordinates.

      iii. (5 marks) `bool Poly::intersect(RTfloat *k, Ray *ray, RTfloat k0, RTfloat k1)` - returns true if there is an intersection of `ray` with this object within the specified range $[k_0, k_1]$. When there is an intersection, `k` contains the corresponding value. See text for details.

      iv. (5 marks) `bool Poly::pluckerTest(Ray *ray)` - returns true if the ray goes through the polygon. Otherwise, returns false. See text for details.

   (c) (30 marks) Cylinder. The cylinder can be either a hollow cylinder when `solidFlag = 0` or a solid cylinder when `solidFlag = 1`. For a solid cylinder, the caps are handled by the disk function that you implement above. For a hollow cylinder, you need to be careful with reflections inside the cylinder. The flag `outside` is for you to keep track of the side of the intersection.

      i. (5 marks) `Vector3 Cylinder::getNormal(Vector3 *point)` - returns the normal at `point` on the surface of the cylinder. See text for details. Note, $\mathbf{p} \perp \mathbf{l}$. From this constraint, you can derive the value of m and hence $\mathbf{p}$. If the cylinder were hollow, the intersection could be on the inside of the cylinder and hence, the normal should be reversed in direction.

ii. (25 marks) `bool Cylinder::intersect(RTfloat *k, Ray *ray, RTfloat k0, RTfloat k1)` - This function is probably the most challenging one in the whole assignment. It determines if there is an intersection with the cylinder within the specified range $[k_0, k_1]$. First, compute the two intersections, say, $t_1 \leq t_2$. If t_1, which is the minimum of the two, is on the inside of the cylinder, e.g. created because of reflection, then `outside=false`, `*k=t1` and the function returns true. If t_1 is not on the surface, then t_1 is the first intersection, which is on the outside of the cylinder and t_2, the second intersection, which is on the inside of the cylinder. t_2 can be ignored if $solidFlag = 1$. To help you complete this function, its pseudo code is given in Algorithm 2.

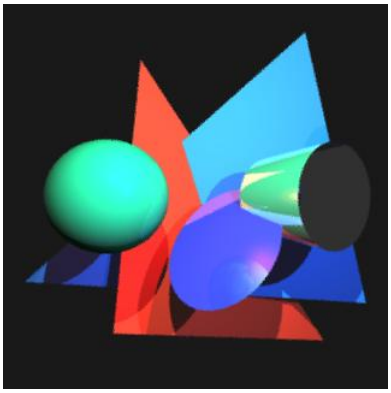
Algorithm 2 Intersection of cylinder

```
if the ray does not intersect the cylinder then
  return false
else
  compute the intersections, t1 and t2, of the ray with the cylinder, where t1 < t2.
  if fabs(t1) < ε then
    Comment: ε is some small value, say, 0.001.
    compute m
    if m ∈ [0,1] then
      set k = t1
      outside = false
      return true
    end if
  end if
  if t1 ∈ (k0,k1) then
    compute m
    if m ∈ [0,1] then
      set k = t1
      outside = true
      return true
    end if
  end if
  if not solid then
    if t2 ∈ (k0,k1) then
      compute m
      if m ∈ [0,1] then
        set k = t2
        outside = false
        return true
      end if
    end if
  end if
end if
return false
```

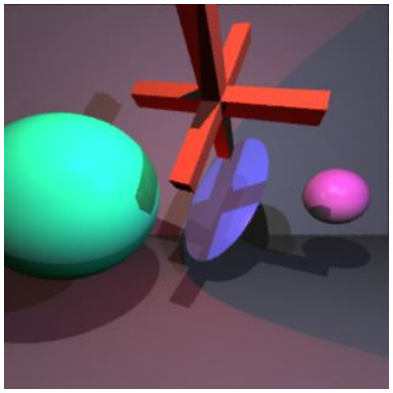
After you have successfully implemented this function, I strongly recommend you comment out different sections of the code to see the resulting effects. It helps you

better understand the mechanism of raytracing.

Some rendering results are shown in Figure 2.



a) Using 1c_1s_2t_1d_26.txt



Using scene_1d_2s_star2_tri_floor_wall.txt

Figure 2. Some raytracing results.

Remarks:

- As discussed in lectures, all rays must be converted to the world coordinate system.
- When you use overloaded functions defined in vector3.h or color.h, you may encounter '`no match for operator ...`' error message. The best way to overcome this problem is to add brackets to enclose the overloaded term, e.g. instead of p%q/r%s, where p, q, r, and s are of type Vector3, use p%q/(r%s) because '%' and '/' have the same precedence. Alternatively, you can decompose the statement into several statements, each of which uses only one overloaded function.
- Use 1 as the number of samples to raytrace. Using a higher number will slow down the raytracing program. You can try a higher number only after your program is working.
- To simplify typing in the name of the scene file, you may want to rename one using a simpler name during testing.