Table of Contents

Algorithms for Assignment 4 Classes and Methods

This document describes the algorithms that need to be implemented for the successful completion of assignment 4.

Method: minimumIntersection(self,direction,objectList):

Input: direction is the vector describing the direction of the ray; objectList is a list of objects composing the scene.

Output: Returns a list of tuples (k, t_0) where k is the position in the list of an object that the ray intersects, and t_0 is the minimum *t*-value of the intersection the ray makes with the object. This list is sorted in increasing order of the *t*-values.

Algorithm:

- create empty intersection list
- for each object k in the list:
  - M^{-1} = inverse of matrix T associated with object
  - transform the ray with M^{-1} in the following way: $T_e = M^{-1}e$, where e is the position of the camera, and $T_d = M^{-1}d$, where d is the direction of the ray
  - $t_0 =$ object.intersection (T_e, T_d)
  - if $t_0 \neq -1.0$ then add tuple (k, t_0) to intersection list
- sort intersection list in increasing order of t_0
- return intersection list

Shader: __init__ (self,intersection,direction,camera,objectList,light):

Input: intersection is the first (k, t_0) tuple from the intersection list; direction is the vector describing the direction of the ray; objectList is a list of objects composing the scene, and light is a lightSource object.

Output: Computes the shaded color for pixel (i, j) as instance variable `self.__color`

Algorithm:

- consider tuple (k, t_0) from intersection
- object $=$ objectList $[k]$
- t_0 is the *t*-value associated with object from tuple (k, t_0)
- M^{-1} = inverse of matrix T associated with object
- T_s = light position transformed with M^{-1}
- transform the ray with M^{-1} in the following way: $T_e = M^{-1} e$, where e is the position of the camera, and $T_d = M^{-1} d$, where d is the direction of the ray
- compute the intersection point as $I = T_e + T_d t_0$
- compute vector from intersection point to light source position as $S = (T_s - I)$, and normalize it
- compute normal vector at intersection point as

  $N =$ object.normalVector (I)
- compute specular reflection vector as $R = -S + (2 S \cdot N) N$
- compute vector to center of projection $V = T_e - I$, and normalize it
- compute $I_d = max\{N \cdot S, 0\}$ and $I_s = max\{R \cdot V, 0\}$
- $r =$ object.getReflectance()
- $c =$ object.getColor()
- $L_i =$ light.getIntensity()
- if the intersection point is not shadowed by other objects e.g. this is a call to helper method `__shadowed(object,` I , S `,objectList):`
  - compute $f = r[0] + r[1] Id + r[2] I_s^{r[3]}$
- else:
  - compute $f = r[0]$
- compute tuple `self.__color` = $(f(c[0] L_i[0], c[1] L_i[1], c[2] L_i[2]))$

Helper: method __shadowed(self,object,I,S,objectList):

Input: object is that which there is an intersection with; I is the intersection point; S is the vector to the light source, and objectList is a list of objects composing the scene.

Output: Returns true if the ray from the intersection point to the light source intersects with an object from the scene, and returns false otherwise.

Algorithm:

- M = matrix T associated with object
- compute $I=M(I+\epsilon S)$ where $\epsilon=0.001$. This operation detaches the intersection point from its surface, and then transforms it into world coordinates
- compute $S=MS$. This transforms S into world coordinates
- for object in objectList:
  - M^{-1} = inverse of matrix T associated with object
  - compute $I=M^{-1}I$. This transforms the intersection point into the generic coordinates of the object
  - compute $S=M^{-1}S$ and normalize S . This transforms the vector to the light source into the generic coordinates of the object
  - if object.intersection (I,S) $\neq -1.0$: (this means there is an intersection with another object)
    - return True
- return False