Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

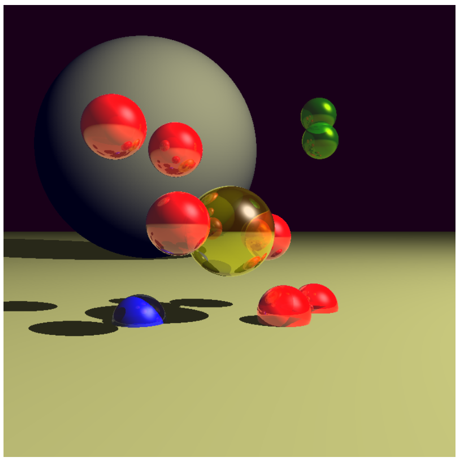
Übungen zur Vorlesung Informatik I (D-ITET)
Dozent: F. Friedrich
`http://informatik1.ee.ethz.ch`

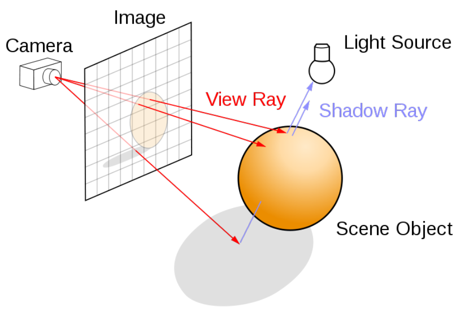
Problem set # 13 **16.12. – 22.12.2013**

Problem 13.1. Ray-tracer

Introduction

A ray-tracer is a program that simulates a camera in a three-dimensional scene. As a result it provides a visualisation of the three-dimensional scene as two-dimensional image. Depending on the complexity of the implemented algorithm, ray-tracing can provide photo-realistic results. But even a quite limited implementation can already provide marvellous images.
We recommend you to consult the web-pages that we reference on the course home-page in order to get a basic understanding of the algorithm. Here we only sketch the idea roughly and then give hints for the implementation. Looking at a three dimensional scene with a camera amounts to sensing light intensity and color for each pixel on the camera image plane. This can in principle be modelled by simulating all light sources of the three dimensional scene and count all photons that eventually reach a camera pixel. Obviously, this corresponds to a huge amount of data and processing. Moreover, only a relatively small amount of data contributes to the image. A much simpler and computational more feasible model is the reverse as follows: for each pixel in the camera plane, the ray from the camera to the scene (”primary ray“) is checked for intersection with all possible objects of the scene. The closest intersection is taken (if any) and the light intensity of the corresponding point is computed by looking on all rays (”secondary rays”) that contribute to the light at this point in space such as direct rays from light sources („shadow rays”), reflection and refraction rays.


Self made ray-tracing image (with reflection and refraction)



Principle (source: Wikipedia)

Building Blocks

The building blocks of the ray-tracer algorithm that we provide as pseudo-code below are

- Intersection of rays with objects, only taking intersections on the ”positive side” of a ray into account
- Computing normal vector on an object
- Computing a reflected ray
- Projection of a light vector on the normal vector, for light intensity computation
- Color addition and multiplication

Objects that we have to consider in the scene comprise of

- Shapes like Balls (center, radius) and Planes (normal, distance) with some material properties such as colors

- Rays (starting point, normalized direction)
- Light source (position, color)

Light contribution at each pixel

Having identified the closest intersection of a ray with the objects, the color contribution to the respective pixel comprises of

(i) The ambient color of the respective shape

(ii) If the shadow ray (i.e. the ray from the intersection point to the light source) is not intersected by another object: the diffusive component $\vec{n} \cdot \vec{s} \cdot dc \cdot lc$ where $\vec{n}$ is the shape normal, $\vec{s}$ the direction from hit-point to the light, dc is the shape diffusive color and lc the light color.

(iii) If the shadow ray is not intersected by another object: the specular light component $(\vec{r} \cdot \vec{s})^8 \cdot sc \cdot lc$ where $\vec{r}$ is the reflected ray direction, $\vec{s}$ the direction from hit-point to the light, sc is the shape specular color and lc the light color.

(iv) The reflective light contribution requires a recursive call to the algorithm with the reflected ray as starting ray. The recursion should be stopped after a certain amount of iterations.

(v) The refractive light contribution

We consider the handling of (i) and (ii) "mandatory" for this exercise, (iii) and (iv) optional and (v) we do not consider any further.

Raytracer Algorithm

```
Input: ray
Result: color = light contribution on ray
set pixel color = Black (= (0,0,0) in RGB) ;
Find intersection of ray with an object (use Intersection Algorithm from Appendix below);
if intersection then
    color = color + object ambient color;
    compute intersecting object normal at intersection point;
    compute shadow ray as ray from hit point to light;
    compute reflected ray from incoming ray and hit normal;
    check for intersection of shadow ray with object
    (hint: add a small fraction of normal vector to the starting point of shadow ray to avoid
    self-intersection at t=0);
    if no intersection then
        color = color + lightcolor * diffuseObjectColor * (shadowrayDirection * hitnormal);
        color = color + lightcolor * specularObjectColor * (reflectedrayDirection * shadowray)^8;
    else
        (hit point is in shadow, no light added)
    end
    if recursion depth not exceeded then
        call this raytracer algorithm with reflected ray;
        color = color + result * reflectionColor;
    end
else
    color = color + background color;
end
```

Tasks

With files `vector3.cpp` and `bitmaps.cpp` we provide you with basic functionality on three dimensional vectors and on bitmaps.

(a) Implement a struct `Ray` for rays and an abstract class for shapes with the following interface:

```
class Shape{
public:
    // material consistency like
    // color, reflectivity etc. omitted
    // constructor omitted

    virtual bool Intersect (const Ray& ray, double& t) const =0;
    virtual Vector GetNormal(const Vector& at) const =0;
};
```

Add colors and other material components according to what you feel is necessary. In order to make things not too complicated in the very beginning, we recommend to only take into account ambient and diffusive light components for the shapes. Implement `Ball` and `Plane` as Extensions of `Shape` and provide functionality for `Intersect` and `GetNormal` for both classes according to the formulas we provide in Appendix 2.

(b) Before you go on with the algorithm, test the intersection member functions, provide trace output for found intersections and check against intuition. Take simple cases first, e.g. Ball at (0,0,0) with radius 1, Ray from (0,0,-3) with direction (0,0,1). Check that directions of the normal are correct.

(c) For the representation of the scene, we suggest the following interface:

```
class Scene{
public:
    // the pixel plane is fixed at z=0, y,x = -0.5..0.5
    Vector cameraVector;
    Vector lightVector;
    Color lightColor;
    Color backgroundColor;

    Scene(Vector camera, Vector light, Color lc, Color bc):
        cameraVector(camera), lightVector(light), lightColor(lc), backgroundColor(bc)
    {}

    void AddShape(Shape* shape);
    Color TraceRay (const Ray& ray, int iter) const;

};
```

Implement `AddShape` to add shapes such as balls and planes to the scene. You may want to implement a helper function

```
Shape* Scene::Intersect (const Ray& ray, double& t) const;
```

as part of the private interface of `Scene` that returns the closest intersecting shape in the scene for a given ray according to according to the algorithm provided in Appendix 1. Test this function with a couple of objects before you continue.

d Implement `TraceRay` containing the core algorithm for each ray.

Start with a very simple scene with only one ball. For example, the following setup contains a ball in the visible domain of the renderer. Note that the pixel plane is considered fixed at $z = 0$, $x, y \in [-0.5, 0.5]$.

```
Scene scene(Vector(0,0,-dist), Vector(10,10,-1), White, Color(0.1,0,0.1));
// add a ball at (0,0,4) with radius 0.5
```

Loop over all pixels of the image and call the algorithm for each ray from a pixel. For example:

```
for (int x = 0; x<sizex; ++x)
{
    for (int y = 0; y<sizey; ++y)
    {
        // we are sitting at (0,0,-dist) and are looking at a plane
        // that sits around (0,0,0) and expands from (-0.5, -0.5, 0) to (0.5, 0.5, 0)
        Vector planePoint(double(x)/sizex-0.5, -double(y)/sizey+0.5, 0);
        Ray ray(planePoint, planePoint - scene.cameraVector);
        bitmap(x,y) = AdjustColor(scene.TraceRay(ray));
    }
}
```

We wish you fun with this exercise and hope you enjoy it. Please do not hesitate to send us your nice resulting images.

Appendix 1: Intersection Algorithm

```
Input: ray
Result: closest intersecting object, if any, and intersection position $t$ on ray
set t = infinity;
foreach object do
    intersect ray with object;
    if intersection distance $t_0$ on ray $< t$ then
        memorize object;
        set $t = t_0$;
    end
end
If object was found, then return memorized object and $t$, return null otherwise,
```

Appendix 2: Basics of Geometry

Let us refresh shortly our knowledge of the intersection of ball and ray. We assume a ray is given as starting point $\vec{s} \in \mathbb{R}^3$ and normalized direction $\vec{d} \in \mathbb{R}^3$. Moreover, we assume a ball given by center $\vec{c} \in \mathbb{R}^3$ and radius $r \in \mathbb{R}^+$. Then the surface of the ball is given as solutions $\vec{x}$ of the equation

$$||\vec{x} - \vec{c}||^2 = r^2$$

Replacing $\vec{x}$ by $\vec{s} + t \cdot \vec{d}$ yields

$$||\vec{s} + t \cdot \vec{d} - \vec{c}||^2 = r^2$$

Substituting $\vec{v} = \vec{s} - \vec{c}$ and expanding yields

$$\vec{v} \cdot \vec{v} - r^2 + 2\, t \cdot \vec{v} \cdot \vec{d} + t^2 \cdot \vec{d} \cdot \vec{d} = 0$$

Therefore the solutions are given as

$$t_{1,2} = -\vec{v} \cdot \vec{d} \pm \sqrt{(\vec{v} \cdot \vec{d})^2 - \vec{v} \cdot \vec{v} + r^2}.$$

if the discriminant is smaller than zero then there is no intersection. Additionally we are not interested in intersections with $t < 0$, i.e. those that lie behind the starting point of the ray. And, moreover, we are only interested in the smaller of the two intersection point, provided it is not negative.

An even simpler computation of the intersection point can be provided for the plane, which can for instance be given by the normal vector $n \in \mathbb{R}^3$ and distance $a \in \mathbb{R}$ as all $\vec{x} \in \mathbb{R}^3$ with

$$\vec{x} \cdot \vec{n} - a = 0.$$

Replacing $\vec{x}$ by $\vec{s} + t \cdot \vec{d}$ yields

$$\vec{s} \cdot \vec{n} + t \cdot \vec{d} \cdot \vec{n} - a = 0.$$

If $\vec{d} \cdot \vec{n} \neq 0$ then this yields

$$t = \frac{a - \vec{s} \cdot \vec{n}}{\vec{d} \cdot \vec{n}}.$$