Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Computer graphics and interaction

« Raytracing »

April 6^{th} - May 19^{th}

Author
Rémi Domingues

Teachers
Christopher Peters

Scholar year 2014-2015

Introduction

This document describes the implementation and output of a raytracer managing direct light, shadows and camera translation and rotation.

The model used in this experiment is a cubic room available at `http://www.graphics.cornell.edu/online/box` and described in the file *TestModel.h*. This model is defined by a set of colored triangles.

2 Intersection of Ray and Triangle

By representing rays as lines in 3D, defined by a starting point and a direction (3D vector), we are able to compute the intersection between a ray and a triangle as described below.

This can be expressed as a line-plane intersection, where we check that the intersection lies in the triangle using the vectors $T_1\vec{T}_3$ and $T_1\vec{T}_2$.

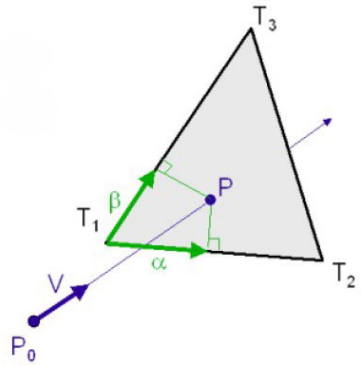

FIGURE 1 – Ray-triangle intersection

These computations have been implemented in the methods *vec3 GetIntersection(vec3 start, vec3 dir, Triangle triangle)* which returns the intersection between a ray (start point and direction) and a triangle, and *bool ClosestIntersection(vec3 start, vec3 dir, const vector<Triangle>& triangles, Intersection& closestIntersection)* which returns true if the ray intersects the given triangle, false otherwise, and stores the intersection in the given parameter.

3 Tracing Rays

Using the pinhole camera model, rays are traced from the camera, lying in position (0, 0, -3) and directed toward the z axis. Their closest intersection are then computed in order to display the closest triangle.

A focal length equal to the screen width has been used. With $width = height = 500$,

we obtain the following field of views :

$$\begin{aligned} FOV_{vertical} &= 2 * arctan(\frac{\frac{height}{2}}{f}) = 2 * arctan(\frac{500}{500}) = 90° \\ FOV_{horizontal} &= 2 * arctan(\frac{\frac{width}{2}}{f}) = 2 * arctan(\frac{500}{500}) = 90° \end{aligned} \tag{1}$$

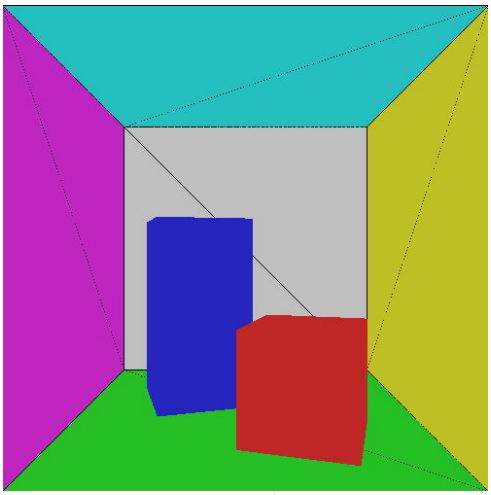

FIGURE 2 – Rendering output

4 Moving the Camera

A camera rotation on the y-axis has been introduced by multiplying the rays direction vector by the rotation matrix below, with θ the camera angle.

$$R = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{2}$$

Then, obtaining the axes direction using equation 3, we can apply a camera translation by adding the corresponding axes direction to the camera position.

$$\begin{aligned} \vec{x} &= (R[0][0], R[0][1], -R[0][2]) \\ \vec{y} &= (R[1][0], R[1][1], R[1][2]) \\ \vec{z} &= (-R[2][0], R[2][1], R[2][2]) \end{aligned} \tag{3}$$

These translations and rotations are triggered by pressing the arrow keys.

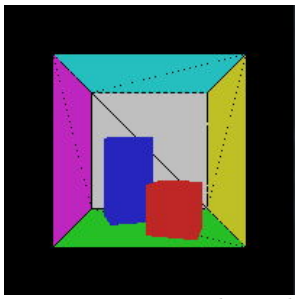
FIGURE 3 – Backward translation

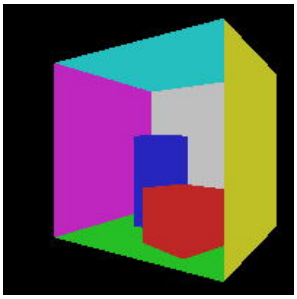
FIGURE 4 – Right rotation and translation

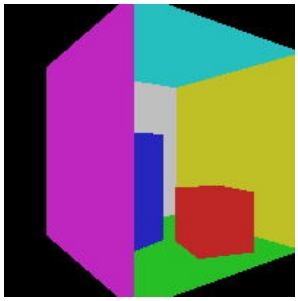
FIGURE 5 – Left rotations and translations

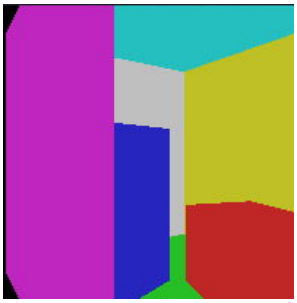
FIGURE 6 – Forward translation

5 Illumination

5.1 Direct light

Direct light is computed by estimating the light power reaching a triangle. This data is processed for every intersection computed in sections 2 and 3. This estimation only does not take into account obstacles between the light source and the intersection, which is why the renderings below do not contain any shadow.

The power D is explained in equation 4, using P the power of the light source, i.e. energy per time unit for each color component (14.f * vec3(1, 1, 1) for example), r the distance from the intersection to the light source, $\hat{n}$ the normal of the triangle and $\hat{r}$ a unit vector describing the direction from the surface point to the light source.

$$D = \frac{P * max(\hat{r}.\hat{n}, 0)}{4\pi r^2} \tag{4}$$

Using the same translation as described in section 4, the light source can be moved in the 6 possible directions using W, A, S, D, Q and E.

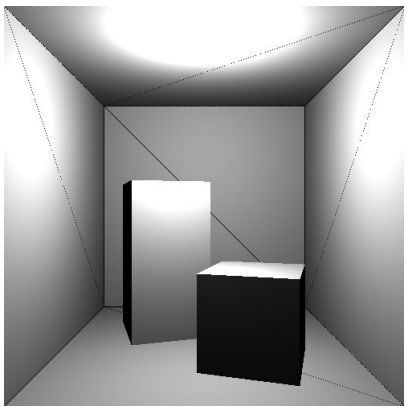
FIGURE 7 – Direct light

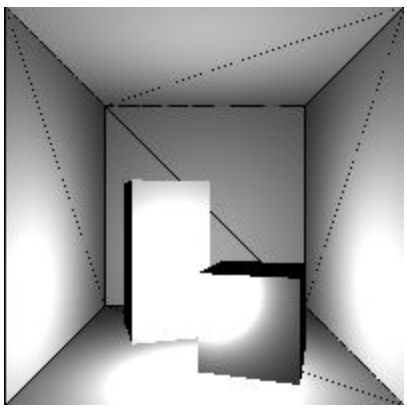
FIGURE 8 – Moving down

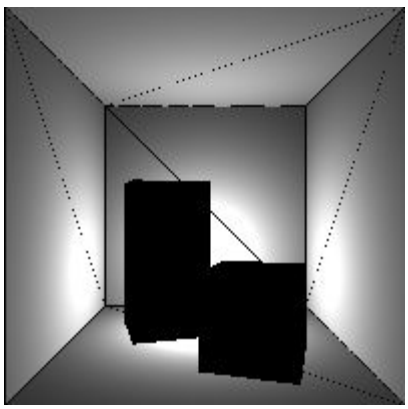
FIGURE 9 – Moving forward

Multiplying the triangle color by the power D for each color component reaching the intersection, we obtain the following renderings.

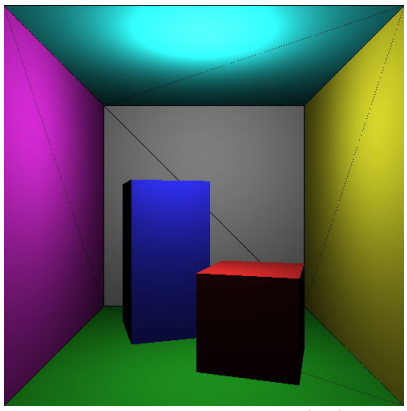
FIGURE 10 – Direct light

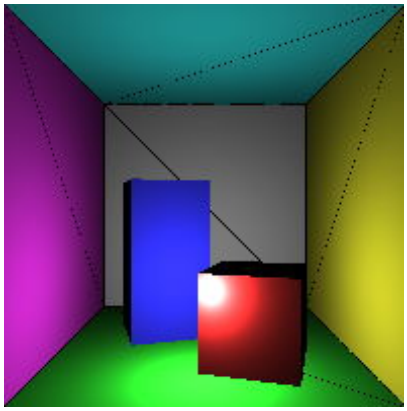
FIGURE 11 – Moving down

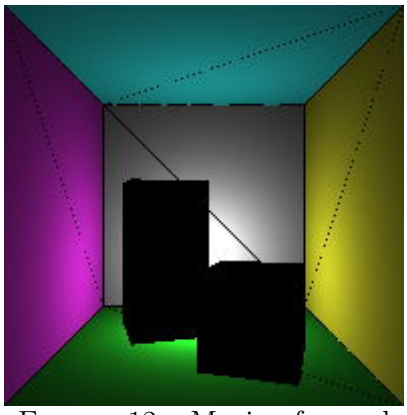
FIGURE 12 – Moving forward

5.2 Direct shadows

In order to make this scene more realistic, we render shadows by tracing rays from the light source to our intersections. If the intersection returned is closer than the distance between the intersection and the light source, this mean that light is hidden by another object. Light power is thus set to 0 for each color component.

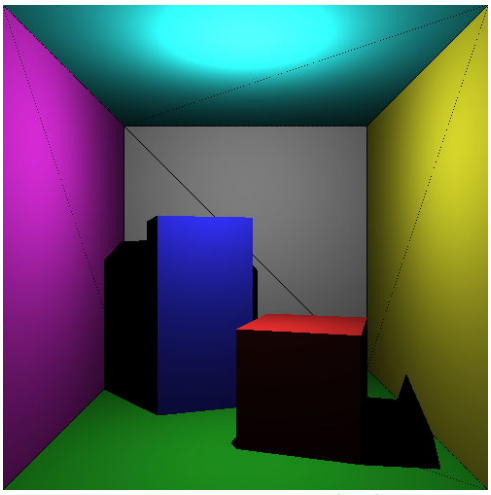
FIGURE 13 – Direct shadows

5.3 Indirect illumination

Eventually, to avoid black shadows and simulate the indirect illumination obtained by multiples bounces of rays from the light source, we add an indirect illumination constant to the power received by every intersection.

Note that computing those bounces would result in a rendering much more realistic but also in a loss of performances.

Therefore, the color of a triangle is now computed as follows, with N the indirect illumination constant and ρ the RGB vector of a triangle.

$$R = \rho * (D + N) \tag{5}$$

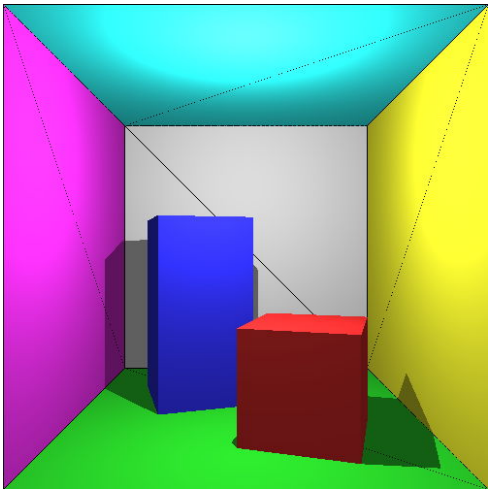

FIGURE 14 – Indirect illumination