TNCG15: Advanced Global Illumination and Rendering: Lecture 6 (”The scene”)

Mark Eric Dieckmann,
MIT group, ITN
Campus Norrköping
SE-60174 Norrköping

October 9, 2018

Outline of the lecture

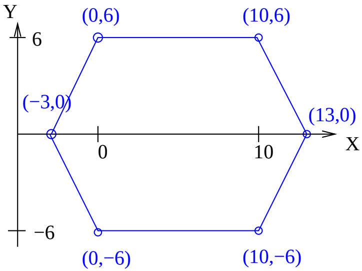
We discuss in this lesson the details of the scene, its content and the proposed structure of your code.

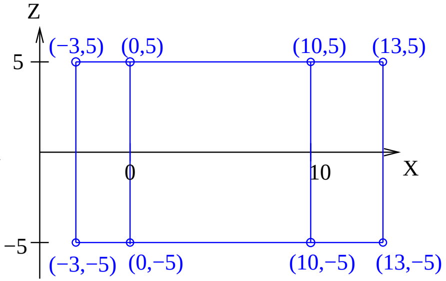
- We define the room and its layout.
- We introduce the camera, the pixels, the rays and the triangles.
- We discuss spherical and tetrahedral objects.
- We specify how we can compute intersections between rays and triangles or spheres.
- We discuss how we compute the reflected and refracted rays.

The world

The world is a hexagonal room. The roof and floor are separated by the distance 10 along z and they lie in the planes $z = 5$ (roof) and $z = -5$ (floor). The walls are oriented orthogonally to the (x,y)-plane.

The figure below shows the view of the flat from the top (left) and from the side (right).



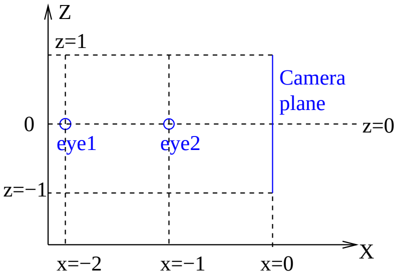


You can represent the world by a triangle mesh. The floor and the roof contribute a total of 2*6 triangles, while each of the 6 walls contributes 2 triangles: The triangle mesh has 24 triangles.

The camera

We know from TNM046 that we can have a camera coordinate system that can move relative to the world.

We keep in our project the camera system fixed (you can make it movable if you wish). The camera location and orientation is:

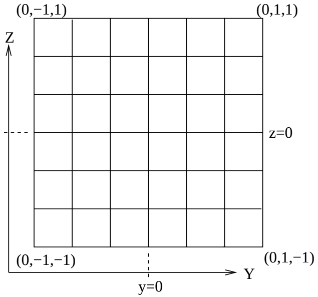


Camera plane: $x = 0$. The points (x,y,z) that define the plane are: $\mathbf{c}_1 = (0, -1, -1)$, $\mathbf{c}_2 = (0, 1, -1)$, $\mathbf{c}_3 = (0, 1, 1)$ and $\mathbf{c}_4 = (0, -1, 1)$.

We want to be able to switch between two eye positions: $\mathbf{e}_1 = (-2, 0, 0)$ and $\mathbf{e}_2 = (-1, 0, 0)$.

The camera plane

The camera is subdivided into pixels. If we look at the camera from the front (opposite to the eyes) we see



Each pixel has the same size along y and along z: $\Delta_y = \Delta_x = \Delta$.

The resolution is 800×800 pixels and the side length of each pixel $\Delta = 0.0025$.

We will give one color value (RGB) to each pixel.

Filling all pixels with colors is the goal of the renderer.

We write out the RGB values for all pixels and display the data set as an image.

The ray

Rays are the key object for ray-tracing. A ray is a straight line with a starting point $\mathbf{p}_s$ and an end point $\mathbf{p}_e$:

$$\mathbf{x}(t) \equiv (x(t), y(t), z(t)) = \mathbf{p}_s + t\ (\mathbf{p}_e - \mathbf{p}_s)\ ,\ 0.0 \leq t \leq 1.0$$

The direction of the ray is $\mathbf{p}_e - \mathbf{p}_s$.

We have a bundle of rays that share the eye point $\mathbf{e}$ as starting point $\mathbf{p}_s$.

Each ray should go through one pixel $\Rightarrow$ we can determine its direction.

A pixel of our camera can be selected by the pair of integers (i, j) where $0 \leq i, j < 800$. Its value for $x = 0.0$ (orthogonal to the x-axis).

The lowest values for y, z are obtained for $i, j = 0$. Since $\Delta = 0.0025$ that coordinate is $\mathbf{x}_p(0, 0) = (0.0, -0.99875, -0.99875)$.

Thus $\mathbf{x}_p(i, j) = (0, i \cdot 0.0025 - 0.99875, j \cdot 0.0025 - 0.99875)$.

The ray

Equation for the ray between the eye point $\mathbf{e}$ and the point $\mathbf{x}_p(i,j) = (0, i \cdot 0.0025 - 0.99875, j \cdot 0.0025 - 0.99875)$ on the pixel (i,j):

$$\mathbf{x}(t) = \mathbf{e} + t\,\left((0, i \cdot 0.0025 - 0.99875, j \cdot 0.0025 - 0.99875) - \mathbf{e}\right).$$

We are between the eye point and the pixel point for all $0 \leq t \leq 1$.

We follow the ray until it hits a wall. Our camera is contained in the room and the room is closed. At least one intersection exists with $t > 1$.

Our room is composed of triangles $\Rightarrow$ we need a good algorithm that computes the intersection of a ray with a triangle.

We use the Möller-Trumbore algorithm.

Möller Trumbore algorithm

The triangle is defined by the three corner points $\mathbf{v}_0$, $\mathbf{v}_1$ and $\mathbf{v}_2$.

We introduce the barycentric coordinates (u, v) with $u \geq 0$, $v \geq 0$ and $u + v \leq 1$.

A point in the triangle is given by $T(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2$.

We intersect the triangle with our ray $\mathbf{x}(t) = \mathbf{p}_s + t\ (\mathbf{p}_e - \mathbf{p}_s)$.

$$
\begin{gathered}
(1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2 = \mathbf{p}_s + t\ (\mathbf{p}_e - \mathbf{p}_s) \\
\Rightarrow (-u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2 - t\ (\mathbf{p}_e - \mathbf{p}_s) = \mathbf{p}_s - \mathbf{v}_0 \\
\Rightarrow u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0) - t\ (\mathbf{p}_e - \mathbf{p}_s) = \mathbf{p}_s - \mathbf{v}_0
\end{gathered}
$$

This equation can be solved using Cramer's rule (not done here).

Möller Trumbore algorithm

We define: $\mathbf{T} = \mathbf{p}_s - \mathbf{v}_0$, $\mathbf{E}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{E}_2 = \mathbf{v}_2 - \mathbf{v}_0$, $\mathbf{D} = \mathbf{p}_e - \mathbf{p}_s$, $\mathbf{P} = \mathbf{D} \times \mathbf{E}_2$ and $\mathbf{Q} = \mathbf{T} \times \mathbf{E_1}$.

The solution of the rewritten equation

$$u\mathbf{E}_1 + v\mathbf{E}_2 - t\mathbf{D} = \mathbf{T}$$

is given by

$$\begin{pmatrix} \mathrm{t} \\ \mathrm{u} \\ \mathrm{v} \end{pmatrix} = \frac{1}{\mathbf{P} \cdot \mathbf{E}_1} \begin{pmatrix} \mathbf{Q} \cdot \mathbf{E}_2 \\ \mathbf{P} \cdot \mathbf{T} \\ \mathbf{Q} \cdot \mathbf{D} \end{pmatrix}$$

or, for our purposes

$$t = \frac{\mathbf{Q} \cdot \mathbf{E}_2}{\mathbf{P} \cdot \mathbf{E}_1}$$

We obtain the intersection point directly with u, v

The suggested basic code structure

You should implement the following classes/struct's in Java or C++:

- *Vertex*:
  It has three spatial coordinates x, y, z and w (for homogeneous coordinates).
- *Direction*:
  It has the three components of a direction vector x, y, z.
- *ColorDbl*:
  It has three double precision variables that contain the intensities in the red, green and blue channel.
- *Ray:*
  It has two instances of *Vertex*, which are the ray's starting point and end point. You can put the vertices into a vertex list and use references to these points in *Ray*. *Ray* contains a reference to the triangle on which the end point is located. The ray color is a *ColorDbl*.

The suggested basic code structure

- *Triangle*:

  The triangle is defined by three objects of the class *Vertex*.

  The *Triangle* has a color, which we represent by an instance of *ColorDbl*.

  The triangle's normal direction is stored in an instance of *Direction*.

  It has a method *rayIntersection(Ray arg)* that computes the intersection between a *Ray* and the *Triangle* with the Möller-Trumbore algorithm.

- *Scene*:

  Scene contains instances of *Triangle*. We use one *Scene* object that consists of 24 instances of *Triangle*. *Scene* objects are closed.

  The triangles of the floor and the ceiling should be white. Each of the 6 walls has a different color.

  It should have a method that determines which triangle is intersected by the *Ray arg* by calling successively the *rayIntersection(Ray arg)* method of each *Triangle*. It then passes references to the triangle and the intersection point to the *Ray arg*.

The suggested basic code structure

- *Pixel*:

  Pixel contains one instance of *ColorDbl* that holds the color and intensity for this pixel with a high dynamic range.

  Pixel has references to the rays that go through it. We use for now one.

- *Camera:*

  It contains two instances of *Vertex* (the eye points) and a variable that allows you to switch between both eye points.

  It contains a 2D array of size 800 $\times$ 800. Each element is a *Pixel*.

  Its method *render()* launches a ray through each pixel one at a time.

  The ray is followed through the scene and the radiance we give to the pixel is computed according to what we learnt in lectures 4 and 5.

  Initially and to test your code you follow the ray until it hits the first triangle and assign the color to the ray.

The suggested basic code structure

Camera contains a method *createImage()*:

The scene image is stored in the *ColorDbl* attribute of the *Pixel*s.

We want to convert the data into a 2D array (image) of RGB vectors with integer values. Each RGB value should be in the range 0 (no intensity) to 255 (maximum intensity).

First possibility, the scene is well-lit everywhere: We go through the red, green and blue intensity values of all pixels and we find the largest one i_{max}.
We extract the *ColorDbl* values of one pixel, multiply each value by $255.99/i_{max}$ and we truncate it. We write the result into the corresponding element of the image.

If we have bright spots in the scene, a linear scale would yield a dark room. In this case we first extract the *ColorDbl*'s from the pixels, take the square root of the red, green and blue values and write the result into an intermediate *ColorDbl* array. Then we convert them into the RGB array as specified in the aforementioned method.

Objects in the scene: A tetrahedron

Our code attributes one color to each pixel. Presently the color is that of the *Scene* triangle, which was hit by the ray that went through that pixel.

Now we want to put intransparent objects into the scene. We start off with polygonal objects like the scene.

We start off with the simplest polygon - the tetrahedron.

An instance of *Tetrahedron* contains 4 triangles. All triangles have the same color.

The method *rayIntersection(Ray arg)* of *Tetrahedron* checks if the ray intersects any of its triangles.

The ray will either not intersect the tetrahedron or it intersects two triangles. In the latter case we pick the intersection with the lower value of t (slide 9).

Objects in the scene: A sphere

A point with the coordinate vector $\mathbf{x}$ is located on the surface of a sphere with the radius r and the center $\mathbf{c}$ if

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

A ray with the starting point $\mathbf{o}$ and the normalized direction $\mathbf{l}$ is given by $\mathbf{x} = \mathbf{o} + d\mathbf{l}$. Each value d corresponds to one point $\mathbf{x}$ on the ray.

The value of d for a ray point that is on the sphere surface is

$$(\mathbf{o} + d\mathbf{l} - \mathbf{c}) \cdot (\mathbf{o} + d\mathbf{l} - \mathbf{c}) = r^2$$

$$\Rightarrow d^2(\mathbf{l} \cdot \mathbf{l}) + 2d\,(\mathbf{l} \cdot (\mathbf{o} - \mathbf{c})) + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2 = 0.$$

The substitutions $a = (\mathbf{l} \cdot \mathbf{l}) = 1,\ b = 2\mathbf{l} \cdot (\mathbf{o} - \mathbf{c})$ and $c = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - r^2$ give

$$ad^2 + bd + c = 0 \Leftrightarrow d = -\frac{b}{2} \pm \sqrt{\left(\frac{b}{2}\right)^2 - ac}.$$

Computing reflected and refracted rays

We have information about the initial importance rays, the light source positions and the ray-surface intersections in the world coordinate system.

Directions of reflected and refracted rays are computed best in a local coordinate system:

- The origin is the ray-surface intersection point.
- One coordinate axis is given by the surface normal at this point.

First task: compute the other two axes such that we obtain a right-handed local coordinate system.

Second task: transform the incoming importance ray from the world coordinate system into this new local system.

Coordinate system definition

The axes of the global coordinate system are x, y and z.

Those of the local coordinate system are X, Y, Z.

Z is the surface normal at the ray-surface intersection point.

We determine X with the help of the ray direction I in the global coordinate system. We pick the part of I that is orthogonal to Z.

$I_\perp$ is the component of I that is orthogonal to Z. We compute it as

$$I_\perp = I - (I \cdot Z)\, Z \qquad \Rightarrow \qquad X = I_\perp / ||I_\perp||.$$

The coordinate system is right-handed: $Y = -X \times Z$.

Coordinate system transformation

We have obtained the direction vectors, which correspond to the axes of the local system, in the world coordinate system.

The direction of the incoming ray I is given in the world coordinate system.

We specified the reflection / refraction laws and the radiance computation in the local coordinate system (hemisphere around the ray-surface intersection point).

The incoming ray and the shadow rays have to be transformed from the world system into the local system and the reflected / refracted rays need to be transformed back.

We need the transformation matrix world system $\rightarrow$ local system and its inverse.

Coordinate system transformation

A general affine transformation is given by a 4×4 matrix. Direction and position vectors are given in homogeneous coordinates.

The translational part of the affine transformation is given by the vector from the origin of the world system $(x_0, y_0, z_0) = 0$ to the ray-surface intersection point (x_s, y_s, z_s).

The rotational part is related to the coordinates of the base vectors $X = (x_1, y_1, z_1)$, $Y = (x_2, y_2, z_2)$ and $Z = (x_3, y_3, z_3)$ in the world system.

$$\bar{\mathbf{M}} = \begin{pmatrix} x_1 & y_1 & z_1 & 0 \\ x_2 & y_2 & z_2 & 0 \\ x_3 & y_3 & z_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_s \\ 0 & 1 & 0 & -y_s \\ 0 & 0 & 1 & -z_s \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Homogenous coordinates in the world system $v_w = (x_w, y_w, z_w, 1.0)$ are transformed into the local coordinate system by the transformation $v_l = (x_l, y_l, z_l, 1)$ as $v_l = \bar{\mathbf{M}} v_w^t$. $\bar{\mathbf{M}}^{-1}$ does the opposite.

Light source and reflections

We introduce an area light source in our scene.

One of the triangles on the roof serves as a light source if you aim for a 3 or 4 as course grade.

A circular light source is used for the photon mapping part (discussed in the final lecture).

The light source should emit white light. You set the R, G and B components of the light source's radiance to the same value L_0.

Transparent objects and mirrors do not change the color of the light - you reflect the RGB components in the same way (the BRDF has the same coefficients for all color channels).

Diffuse reflectors can change the color of the reflected light.

Simplest way to introduce color: give the reflectance ρ of Lambertian reflectors different values for R, G, B.

Summary of the lecture

We discussed in this lesson the code components that we can already implement.

- We defined the room and its layout.
- We introduced the camera, the pixels, the rays and the triangles.
- We discussed spherical and tetrahedral objects.
- We specified how we can compute intersections between rays and triangles or spheres.
- We have discussed the light source and how colors affect the reflection.

You can start implementing parts of your code after this lecture - you know how the scene looks like, what objects you should implement and how you deal with perfectly reflecting (lecture 4) and transparent surfaces (lecture 5).