raytracer2

Petter Ljungqvist Hilkka Tamminen Anni Toppila

Last updated: December 12, 2013

Abstract

Raytracing is a method for generating images of three dimensional worlds. Artificial "rays" are shot from a camera (virtual eyepoint) trough every pixel of an image, and if the ray hits an object, the pixel's color is calculated from scene parameters at that intersection point. Any mathematical surface that can be intersected by a line can be rendered using raytracing.

We have been focusing on rendering a realistic static image in this project. Performance has naturally been taken into account, but we have tried not to compromise the finale image.

Contents

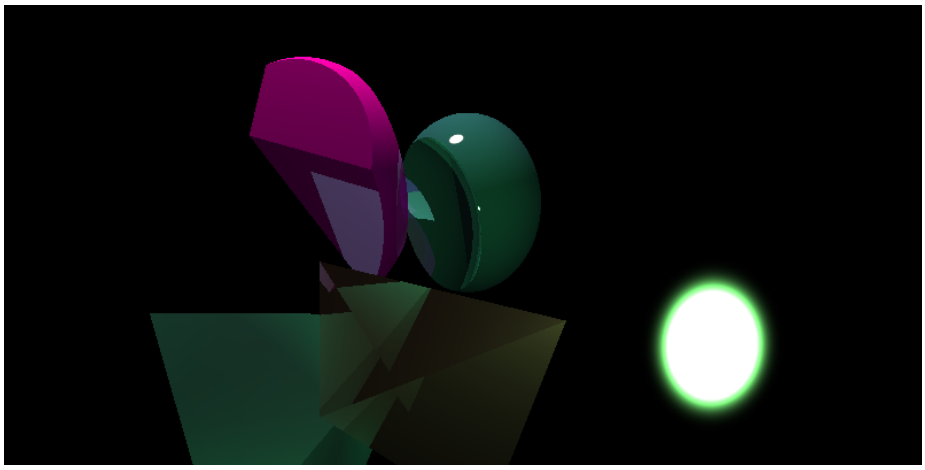

Figure 1: Example picture

1 Instructions for compiling and use

The Raytracer has been tested to work on Aalto's Unix computers, but it should work on other platforms as well. For compiling simply go to the src folder and write the command `make`, and the program will compile to a file called raytracer.

To run the program, simply write ./raytracer. The program will ask first for the scene file, then for the name of the file where you want to save the picture. The scene file is where all the information about the picture is stored, and for testing purposes you can use for instance `src/scene.txt`, which is uploaded in SVN. Alternatively, you can give the scene file as an argument to `raytracer` and then type in the picture name, or even give both as arguments eg

```
./raytracer scene.txt pic.ppm
```

The program will read the scene file and create a picture with the given file name.

2 Program architechture

The project is built up of three main parts; a 3D-world, a rendering control and a input file reader.

2.1 World

The base of the architecture is a singleton class called **World**. World contains everything in the 3D-world. The abstract objects present in the world are called Thing and Light. The actual act of ray tracing happens in World as well. This means that the World doesn't need to be rebuilt for capturing an image from another position.

The abstract class **Thing** is inherited from all objects which can be seen in the finale picture. The way they are seen depends on everything else (and itself) present in the world. The other type of objects that influence picture inherit the abstract class **Light**. Lights do not actually appear in the picture, they just

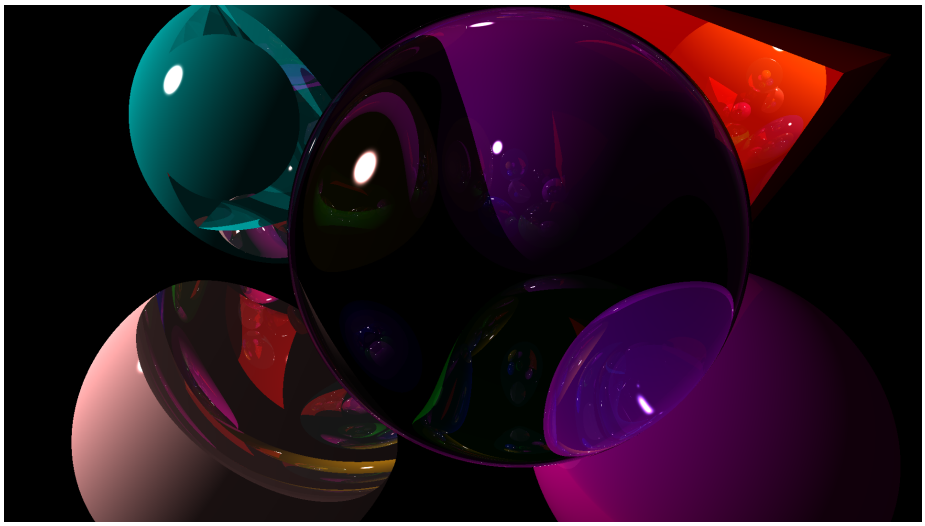

Figure 2: Example picture

contribute with light. Though the Lights aren't seen they can add some kind of Thing to World when they are created.

Thing has a subclass called **SimpleThing** that is also a abstract class. SimpleThings are built up of only one material and colour. They can be combined in different manners with a subclass of SimpleThing called **CSG** (Combined Solid Geometry). That means that a CSG can be used in another CSG.

A small UML-diagram of the world can be found in appendix B.

2.2 Picture

The class **Picture** handles the capturing of the picture. It positions the camera, builds the raster and controls the rendering of the image. Picture is independent of the world it is possible to render different kinds of pictures.

2.3 Scene File

The scene file is read by the function FileReading::read(istraem file)and is described in appendix A.

3 Data structures and algorithms

3.1 Reading the Scene File

To build a world using the *FileReading::read(std::istream file)* function a file (or other kind of stream) is needed. The reader reads the stream word by word and uses its *sub*-functions to interpret the objects. All functions make use of the braces, { }. The `constructive_solid_geometry` function uses the same function as the environment `thing` to add objects to itself. This recursion makes it possible to build complex *CSG*s with many objects.

The file reader is not necessary to build a World, it can be done through Worlds own functions (also used by the reader), but the scene file input is quite easy and has also helped us a lot with testing.

3.2 Building the World

The world is initialized at start-up. Then the different objects (the *Thing*s and *Light*s) in the world are built and added to the world after they are ready. The *camera* is not part of the world, it just makes use of the worlds ray tracing capabilities. That means that many cameras can work on the same World object, and there is only one World per instance of the programme since the class is a singleton.

The class World as it is built is not capable of moving its objects, just removing and adding new once.

3.3 Prepering the Raster

The whole act of ray-tracing begins from the *Picture* class. Picture had a *Camera* that contains information about the set-up (position, direction, projection, angles, size etc.). The raster (the *Ray*s) for the image is calculated by Picture and sent to the World instance using its function *sendRay*. sendRay returns the colour generated by the ray. All the colours from the rays are stored in a class called *PictureData*.

3.4 Ray Tracing

The class Worlds function sendRay takes a Ray (contains two vectors, origin and direction). sendRay first loops through all the Things in the world and checks, with the Things checkRay function, whether the ray hits it or not. checkRay returns a lu (the length unit used throughout the project) which represents the distance the ray has to travel before hitting the object. From the point where the ray hits the object closest to its starting point (smallest value of the returned lu), if it hits any, three different rays are sent out. The shadow-ray, the reflection-ray and the refraction-ray. All these rays will return a colour and the final colour for send Ray to return is

$$c = i_{sh}c_{sh} + i_{rl}c_{rl} + i_{rr}c_{rr} \tag{1}$$

where the cs are the colours and the is the indices of the different phenomena. The is should add up to one. The colour of three different rays are calculated somewhat differently:

3.4.1 Shadow-Ray

The shadow-ray represents the scattered lights. It is calculated for each light source in the world and all the colours added together. The colour contribution from each light source is calculated the following way:

First a base colour is calculated by

$$c_{sh,b} = k\frac{c_{mat}c_{light}}{r^2} \tag{2}$$

where r is the distance to the light and k is a coefficient that we have chosen to be 0.1. After that a ray from the the point to the light is checked for shading objects. If there are any not transparent objects in the way the or if the object itself is in the way of the light only the base colour is returned. If there aren't any objects in the way though a light colour is added to the base colour. The light colour is

$$c_{sh,l} = (1-k)\frac{c_{mat}c_{light}}{r^2}\mathbf{n}\cdot\mathbf{v}_l \tag{3}$$

where $\mathbf{n}$ is the objects normal in the point and $\mathbf{v}_l$ is the unit vector pointing at the light source from the point.

If the point is shaded by transparent objects the light in (3) is shaded accordingly.

3.4.2 Reflection-Ray

The reflection-rays colour is generated by using the Worlds sendRay function from the point in the opposite direction of the incoming rays mirror image through the objects normal in the point.

3.4.3 Refraction-Ray

The refraction-ray goes into the object. The direction is calculated with the formula [1]

$$\frac{\sin\alpha_1}{\sin\alpha_2} = \frac{n_2}{n_1}. \tag{4}$$

A ray is then sent in that direction through the object and from there it continues in again another direction determined with the same formula (4). The part of the ray that goes through the object is shaded according to the objects materials properties:

$$c_{shade} = (c_{mat}k_{tansl})^r\, c_{outray} \tag{5}$$

where k_{transl} is the translucence coefficient and r the distance the ray travels in the object. c_{outray} is the colour of the ray going out on the other side of the object. That colour is again calculated with Worlds sendRay.

3.5 Writing the Image to a File

At the end the PictureData can write out the colour raster to a simple `.ppm`-file.

3.6 Constructive Solid Geometry

The constructive solid geometry (or CSG) class combines two *SimpleThing*s in one of three different combination types, union, intersection and complement union. A CSG has only one material. The materials in its SimpleThins are ignored.

The functions that the CSG needs to be able to calculate for these three types are

checkRay(ray:Ray&,max:lu):lu checks if a ray hits the object and returns the distance the ray has to travel before hitting it (returns a negative number if it doesn't hit it).

checkInnerRay(ray:Ray&):lu checks where a ray hits the back wall (ignoring the first surface) the same way as checkRay.

getNormal(position:Vect&):Vect returns a unit vector pointing out of the object. Always called for a point on the surface of the object.

wraps(position:Vect&):bool returns *true* if the position is inside the object.

onSurfice(position:Vect&):bool returns *true* if the position is on the surface of the object.

wraps and onSurfice arn't ever true for the same position. For this a small Δ constant is used since doubles aren't very exact.

3.6.1 Union

In union combination the order of the tow objects doesn't matter. The functions listed above work as follows for unions.

checkRay calls checkRay for both and returns the smaller positive one (returns a negative number if both ar enegative).

checkInnerRay chacks checkInnerRay for both. Returns the positive one if only one is positive (a negative number if both are positive). If both are positive it checks if the smaller ones point is inside the other object (wraps), and if so returns the grater one, else the smaller.

getNormal checks on which objects surface the position is (onSurface) an returns the normal (getNormal) from that object and point.

wraps returns *true* if wraps is true for either object.

onSurfice returns *true* if true fore one of the objects and not inside (wraps) the other object.

3.6.2 Intersection

Intersection is the space occupied by both objects.

checkRay if positive for both it returns the smaller one.

checkInnerRay if positive for both it returns the greater on (with some tweaks for non convex objects).

getNormal checks on which objects surface the position is (onSurface) an returns the normal (getNormal) from that object and point.

wraps returns *true* if true for both objects.

onSurfice returns *true* if true fore one of the objects and inside (wraps) the other object.

3.6.3 Complement Union

Intersection is the space occupied by the first (left) object, but not the second (right).

checkRay if positive for left it checks if that point is inside right. If so it returns rights checkInnerRay else lefts checkRay. If negative for left if returns a negative number. Except when the ray starts inside left, then rights inner ray is tested and if that point is inside left as well it returns rights inner ray.

checkInnerRay if positive for left it checks if that point is inside right. If so it returns rights checkRay else lefts chaeckInnerRay. If negative for left if returns a negative number. Except when the ray starts inside left, then lefts inner ray is tested and if that point is not inside right it returns lefts inner ray.

getNormal checks on which objects surface the position is (onSurface) an returns the normal from left if it is left and -normal from right else.

wraps returns *true* if true left and false for right.

onSurfice returns *true* if true fore left and not inside right or true for right and inside left.

4 Known bugs

- The ray tracer isn't counting reflections, so there is a possibility of infinite reflections (e.g. a CSG of a sphere and a smaller complement inside it). This is not hard to fix, has just not been done.
- The ThingPlane (created with `plane` in the `special_thing` environment) is onesided. So it works poorly as a transparent surface. (Not actually a bug.)

5 Tasks sharing and schedule

The group communicated via e-mail and weekly meetings. The meetings were convenient for keeping everybody up to date on the latest progress and possible changes to how things were done.

After the plan was presented to the assistant in the beginning of the project, a more detailed schedule was made. Milestones were set up for every week and the work was divided and shared with more detail. By November 17th we wanted to be able to create a picture with just spheres, a week later spheres and some other shapes with lights as well, and then by the 30th all the basic functionalities were to work. A shared document was made on Google Docs that we were supposed to tick off sections in every time we finished something.

In practice the project advanced in a different order and for the most part ahead of time. Very quickly we had pictures with spheres, lights, constructive solid geometry, reflections and translusans. The freely movable camera was

finished around the time scheduled, while the file reading function was worked on later than we had thought.

Most of the workload was done by Petter, who handled general things such as creating the header files and the file dependencies, but who also did most of the work in World, Light, Thing, Sphere, CSG... Anni handled the camera and its movements and writing the data out to a file. Hilkka made part of Material, Colour and Vect classes and wrote the first version of the file reading function, which Petter then improved and added error handling to.

Testing of each part was done individually and so well that there were no bigger problems when fitting them together. The testing was not planned in detail in advance, but that didn't cause any problems.

Documentation was done together: each group member wrote more or less the same amount.

6 Differences to the original plan

The original plan turned out to be quite good and no mayor changes where needed. There were naturally quite a bit of stuffing added in the implementations stage, but the main idea stayed the same. Our clear original plan helped us to understand each other and what we wanted from the project and made it easier to work towards a common goal.

References

[1] Raimo Seppänan et al. *MAOLs Tabeller*. Schildts Förlag Ab, Esbo, sjunde, reviderade upplagan edition, 2000.

C++ : Reference *online* `http://www.cplusplus.com/reference/`

A Detailed Instructions for Writing a Scene File

All the parameters of the picture are defined in a so-called scene file. That means that all information about the objects, the lights, the camera and the picture size should be found there, structured in a certain way.

The scene file is divided into main sections, called `thing`, `special_thing`, `light`, `camera` and `picture_size`, containing information about the respective subjects. All sections are defined using curly brackets { }. The brackets must not touch the other words meaning there must always be at least a white space between the other words and the brackets. The order of the sections or the order of the information within the sections are not important.

Indentation does not matter, but is advisable for clarity. Everything should be in lowercase, except the type parameter in constructive solid geometry. Commenting is possible, and it is done by writing `//` before the comment. As in C++, the rest of the line after the `//` will be ignored. There is a difference though; here it is compulsory to have a white space between the `//` and the text.

In the `thing` section, an unlimited amout of objects can be declared. Each new sections starts with the name of the type of object: `sphere`, `tetrahedron`, `cylinder` or `constructive_solid_geometry`. So far the thing section would look like this:

```
thing {
     sphere {
     }
     sphere {
     }
     tetrahedron {
     }
     cylinder {
     }
     constructive_solid_geometry {
     }
}
```

A sphere must have a position, a radius and a material section containing the material values and the colour. All values are defined by writing first the name of the value, then the value itself, e.g. `radius 3.5`. Position and colour, which have several values, are also defined on one line: position in the form `position x y z` and colour by `colour r g b`.

The material section should have the `reflection`, `refraction`, `scattering`, `translusans`, refractive index (written together: `refractive_index`) and the

`colour`. Note that in order the be realistic, the values for reflection, refraction and scattering need to add up to 1.0. Unless they are all given as 0, the program will make sure this law is fulfilled. All the values in the material section should be in the interval [0, 1]. The only exception are the colour values for lamps, which can be significantly bigger.

The `sphere` section will then look like this:

```
sphere {
     material {
         reflection 0.2
         refraction 0.6
         scattering 0.2
         translusans 0.5
         refractive_index 0.2
         colour 0.1 0.9 1
     }
     position 1 2 1
     radius 1.5
}
```

For tetrahedrons, the material is defined the same way as for the sphere, but the position and size are defined by giving the coordinates of the shape's corners in space. The `tetrahedron` section would thus look like this:

```
tetrahedron {
    material {
      ...
    }
    vect1 1 2 3
    vect2 2 3.5 3
    vect3 3 2 3.3
    vect4 4 2 3
}
```

Cylinders are made the same way. They are created with `cylinder` and their parameters are `end_1` and `end_1` for the ends and `radius`. eg:

```
cylinder {
    material {
      ...
    }
    end_1 14 -1 0
    end_2 0 1 0
    radius 1
}
```

Constructive solid geometry is a little more complicated. In this program, the CSGs are always made up of two SimpleThings, i.e. spheres, tetrahedrons or even other CSGs. The type of interaction between the Things is defined as union (`U`), intersection (`I`) or complement (`C`). CSGs also need a material. The Things are called `left` and `right`, but the names have no significance. Note: it is up to the user to make sure the two parts of the CSG intersect.

The `constructive_solid_geometry` section will have a `left` and a `right` section. These will include a Thing definition. Material definitions for left and right are not needed and can be left out. The `constructive_solid_geometry` section will then look like this:

```
constructive_solid_geometry {
    left {
        sphere {
            position 2 2 1
            radius 1
        }
    }
    right {
        constructive_solid_geometry {
            left {
                sphere {
                    position 1 2 1
                    radius 1.5
                }
            }
            right {
                sphere {
                    position 2 2 2
                    radius 1
                }
            }
            type U
        }
    }
    material {
        ...
    }
    type I
}
```

The `special_thing` environment have thing which can not be used in a `constructive_solid_geometry`.

`plane` is a infinite plane. It is constructed from three points in the plan and one (nr. 4) the shows which site is the top of the plane. The plane should always have its top towards the camera, so a good practice is the use the cameras position as the fourth point. Using transparent surfaces is not recommended, since the plane is only onesided. Example of plane construction:

```
special_thing {
     plane {
         mateial {
             ...
         }
         point 1 1 2 3
         point 2 3 2 1
         point 3 1 1 0
```

```
        point 4 0 0 0
    }
}
```

The `light` section works the same way as the Thing section; all the Light objects are defined there for each type (`lamp` or `spotlight`). As explained before, the difference between the two types of Light is that `spotlight` is a point light, it is only just the light source, whereas `lamp` has a sphere around the light source to give it a more natural look.

Spotlights are defined by simply giving the colour and the position. For lamps, the size of the sphere is also given. Remember that for lights the colour values can be much greater than 1. The `light` section will then look something like this:

```
light {
    lamp {
        colour 200 300 200
        position 3 4.5 6
        size 1
    }
    lamp {
        colour 180 30 150
        position 1 2.5 6
        size 3
    }
    spotlight {
        colour 100 200 250
        position 1 3.5 6
    }
}
```

Everything related to the camera is described in the `camera` section. The camera is defined by its position, which direction it shoots at, which direction is up, how wide is the angle of the shoot, and what type of projection is it. The angles are given in degrees in x and y directions, and the type of projection is given as a character: 1 means a polar coordinate system, anything else mean a cartesian coordinate system. The `camera` section does not have running numbers, and looks for instance like this:

```
camera {
    position 2.05 3.10 4.2
    centre 0.0000 2.1 4.5
    up 3.2000 9.5 3.3
    projection i
    angle_x 45.0
    angle_y 90
}
```

The last part of the scene file is the picture size. It is in its own section, like everything else, and is quite simple:

```
picture_size {
    x_max 1500
    y_max 1000
}
```

The scene file must end with the word `end`.

B World UML-diagram