4. Assignment: $\mu$ray

Markus Höferlin  Gregor Mückl

Institut für Visualisierung und Interaktive Systeme

1 Topics

- Ray Tracing
- Mathematical Description of 3D Objects
- Lighting and Shading Models

2 Introduction

The task of programming a Ray Tracer can be split into multiple independent parts. It includes e.g. loading the scene description, creating rays, calculating the intersection of the rays with the objects of the scene, calculating reflected or refracted rays, and determining the color at an intersection point.

A good object oriented design for a ray tracer can be alleviated by a top-down approach. At first, the entities interacting with each other and the nature of these mutual interactions have to be identified. Using the knowledge gained from analyzing the problem appropriate objects and methods can be designed and a rough draft of the basic algorithm can be realized on the base of these methods. Finally, the actual implementation of the methods follows.

For this assignment only a few building blocks of the ray tracing algorithm are given. These are e.g. three and four dimensional vectors, $4 \times 4$ matrices, and a base class for generic geometric entities. Update your SVN directory to retrieve the source skeleton. It will be your task to implement a fully functional ray tracer on the basis of the skeleton.

Alternatively, if you are not happy with the object-oriented design of our skeleton, you are free to design your own ray tracer from scratch. However, the following requirements have to be met:

- The ray tracer has to be implemented completely in C++. The object oriented approach is an important point of this assignment. Thus, it is e.g. not allowed to simply encapsulate C programs in C++ classes.
- Your implementation must be able to correctly load and render the reference scenes given in the data directory. Correct means that your ray tracing results match the reference images stored in the handout directory.

Compiling the program skeleton with make already results in an executable program. In addition to the src directory there is a data directory which contains various scene descriptions for the ray tracer including the reference scenes mentioned above.

When working in command-line mode the filename of the scene description as well as the filename of the output ppm-image is expected by the program as parameters. In this mode the rendering result is stored as an image in the Portable Pixmap format (ppm). Ppm is a platform independent image format that can be displayed directly or converted into other image formats with tools like xv, display (Image Magick), qiv, eeyes or eog.

Alternatively, the ray tracer can be started in GUI-mode which allows to directly display the rendering results.

3 Description

It will be your task to extend the given source skeleton of the ray tracer or as an alternative develop your own ray tracer featuring the same functionality. So far, the version of $\mu$ray we provide as source skeleton is able to process spheres only. Moreover, only a very simple lighting model is implemented. The src directory contains the following files:

- murray.dtd: This file specifies the grammar of the XML-based scene description language. Implementing a parser for this language is not your task. For loading and parsing the description of the scene we use QT's XML parser based on DOM.
- camera.cpp: Encapsulates the camera settings.
- color.h: A simple class for working with RGB color values.
- entity.h, object.h: Definition of entities and objects. In contrast to objects entities don’t have an intersect method.
- light.h: A lighting class. Three different light types are defined: GLOBAL, POSITIONAL and SPOT.
- material.h: Stores the material properties of Objects.
- muRayGUI.qt.h: A graphical user interface for the ray tracer implemented in QT 4 (so it's not necessary to set QTDIR).
- parseXML.h: Parses the XML file containing the scene description and creates a Camera and a Scene object.
- ray.h: A simple ray class. This class calculates reflected and refracted rays. It has no further functionality.
- rendererr: This module contains the skeleton of the Render-class which does the actual ray tracing. It needs a reference to a Scene and a Camera object.
- scene.h: A tree-like data structure which stores the scene description and which can be traversed similar to a scene graph.
- sphere.h: A complete implementation of a sphere class.
- v3d.h, v4d.h, m4x4: Helper classes for 3D/4D vectors and matrices. You have to differentiate between position vectors (points in space) and direction vectors when using these classes. The distinction is not achieved by separate classes for each vector type but through the homogeneous coordinate of a 4D vector. Position vectors get a homogeneous coordinate of 1 and direction vectors a value of 0. The helper function point2v4d can be of interest in this context. It creates a 4D position vector of type v3d out of a 3D vector of type v3d.

All modules are extensively documented via doxygen. Bear in mind to comment your extensions to the source code adequately. You should use the doxygen-style commenting. Executing make doc generates a html documentation of your program.
The final ray tracer must feature the following functionality:

- Data stored in a .xml scene file can be loaded with the help of a parser. The exact specification of the file format is given in section 3.1.
- The ray tracer should be able to render the described scene correctly. Which means in particular, that
  - various surface types can be processed.
  - the intersection with the nearest object is determined for every viewing ray. Additionally, secondary shadow rays are send from an intersection point to all light sources.
  - lighting calculations are done correctly for every visible light source.
  - the reflection ray is calculated and lighting calculations are performed for it as well.
  - transparent objects refract the ray according to snell’s law. Inside transparent objects a simplified model is applied that neglects caustics as well as the attenuation of the light along the ray. Thus, transparent objects cast black shadows like opaque objects.
  - the final image is written into a ppm image file.

- The ray tracer is extended. Either an additional feature that delivers more realistic images or an approach that improves the effectiveness of the rendering process can be implemented. Some suggestions are given later. All extensions to the ray tracer have to be described in a README-file. The user of the program should be able to toggle additional functionality through command line switches (on/off). It is also possible to extend the scene description language with your own parameters. However, your implementation should deliver the same results as our reference implementation when it is called with no additional command line options.

- A self-created scene exist and is stored as my_scene.xml in the data directory. Your scene has to be unique and should showcase the extensions you have implemented. The scene must contain at least five objects. It is not necessary to copy the resulting render image my_scene.ppm into the directory—we will generate it with your program on judgment day anyway.

Note: The specifications for the description language of the geometry and the surface properties—given below—are minimum requirements only. For your extensions of the ray tracer it may make sense to add further description tags for the geometry and modify the parser accordingly.

But: In any case, even after modifications, your program must be able to handle files given in the format specified below and deliver the expected images. This includes the reference scenes in particular.

### 3.1 Format of the Scene Description Files

This section specifies the format of the scene description file. It is based on XML. The complete Document Type Definition is given in muray.dtd. You are allowed to extend the format. However, the listed tokens must not be modified! In the following <int> stands for an integer, <float> represents a floating point number and <0-1> represents a restricted range of floats between 0 and 1. The most important elements are:

- **muray** Marks the beginning of a muray document. It may contain a camera and a scene element.
- **camera** Describes a camera. Its position, viewing direction and its Up vector are defined in world coordinates. The vertical viewing angle (cp. fovy) is given in degrees.
  
  \[
  \begin{align*}
  \text{pos} &\text{ x<float> y<float> z<float>} \\
  \text{dir} &\text{ x<float> y<float> z<float>)} \\
  \text{up} &\text{ x<float> y<float> z<float>)} \\
  \text{fovy} &\text{ angle<float>}
  \end{align*}
  \]

- **scene** Contains one background element at most and an arbitrary number of transformations t, geometric objects shape or light sources light.
  
  \[
  \begin{align*}
  \text{t} &\text{ Describes a transformation. The transformation can be composed of a translation (translation="x y z"), a rotation (rotation="x y z a") with rotation axis x y z and angle a, and a scaling (scaling="x y z"). In the case when all three kinds of transformations are specified for a transformation t, the matrix } T \times R \times S \text{ is applied from } \text{left onto geometric vectors specified in its scope. An element t can additionally contain an arbitrary number of elements of type t, shape or light.}
  \end{align*}
  \]

- **shape** Describes geometric elements and their material properties.

- **light** Describes a light source. The type of the light is given by the parameter type. The light source can be defined as a positional, global or spot light. For all light types the light color is given by three parameters (r, g and b).

  \[
  \begin{align*}
  \text{pos} &\text{ x<float> y<float> z<float>} \\
  \text{dir} &\text{ x<float> y<float> z<float>)} \\
  \text{up} &\text{ x<float> y<float> z<float>)} \\
  \text{fovy} &\text{ angle<float>}
  \end{align*}
  \]

4 **Lighting model**

The lighting model we will implement in this assignment is a simplified version of the model proposed in the book „An introduction to raytracing“ (Andrew Glassner). The equation for the lighting calculations can be expressed as:

\[
I(\lambda) = k_{\text{diff}} \sum_i I_{\text{ai}}(\lambda) + k_{\text{dir}} \sum_j I_{\text{lj}}(\lambda) \cdot \left(N \cdot L_j\right) + k_{\text{st}} \left( \sum_j I_{\text{lj}}(\lambda) \left(N \cdot H_j\right)^n + I_{\text{st}}(\lambda) \right)
\]

\[
H' = \frac{\text{H} \cdot V}{\text{H} \cdot V}
\]

N is the normalized (!) normal, L the normalized light direction, \( V \) the normalized viewing direction, \( \beta \) the ratio of the refraction indices, \( \text{H} \) the so called halfway vector, and \( \text{H}' \) a helper vector for lighting transmissive objects. \( k_{\text{diff}}, k_{\text{st}} \) and \( k_{\text{dir}} \) are material dependent.
coefficients for diffuse reflection, specular reflection, and specular transmission. Secondary rays enter the equation as terms $I_{sr}(\lambda)$ for reflected and $I_{re}(\lambda)$ for refracted rays. For the calculation of the intensity $I(\lambda)$ the intensities $I_{sr}(\lambda)$ of all ambient light sources and the diffuse and specular parts of all other light sources $I_{j}$ are accumulated. Please observe that the intensity parameters of the light sources, as well as the material coefficients, are RGB vectors!

5 Extensibilities

Of course there is a set of possibilities to extend the ray tracer. Many methods on improving the algorithms like the intersection point calculation or methods regarding additional functionality have been published.

5.1 Desired Extensions

This paragraph lists possible extensions for the ray tracer. Further descriptions of the individual techniques can be found in the books by Glassner and Foley.

- **Different Surface Area Types:** You may support Tensor product surfaces or other higher order algebraic surfaces (e.g., Tori).
- **Different Object Types:** Fog, fractals, local light sources, emission (phosphorescence, fluorescence), volumes (with uniform or non-uniform density)
- **Different Optical Effects:** Refraction, dispersion (e.g., light being split into its spectral components by a prism), gravity effects for modeling water surfaces, glass, ice . . .
- **Correct Transparency:** Light attenuation with increasing path length, caustics instead of hard shadows for transparent solids.
- **Constructive Solid Geometry CSG:** Modeling complex geometric objects using boolean operations on simple objects. Subtracting a cylinder from a sphere yields a sphere with a hole for example.
- **Area Light Sources:** Point light sources do not model real world light sources adequately, thus area light sources, objects with a light emitting surface, are used.
- **Texture Mapping:** You may apply textures to your objects in order to model fine details like cloth, leather, or paper.
- **Bump Mapping/ Displacement Mapping:** A 2D image may be used as a height map in order to generate geometric detail on existing objects. This effect suits modeling fine surface structures like those of strawberries, citrus fruit, or teddy bears (fur) excellently.
- **Procedural Shaders:** A special function (turbulence, noise, . . .) procedurally defines the color of each surface point depending on its coordinate in space (“solid texture”).
- **Adaptive Super Sampling:** Depending on the image resolution ray tracing artifacts may occur due to the discretization showing as block artifacts. Reduction of such artifacts can be achieved by refinement and shooting of more viewing rays. This is a rather costly approach though. Adaptive super sampling aims at refinement in image regions where large intensity discrepancies between neighboring samples occur.
- **Stochastic Super Sampling:** The rays’ directions are slightly jittered using a random function. This reduces the artificial look of the image and allows better anisotropic effects.
- **Distributed Ray Tracing:** Besides the direction of the viewing ray, its origin, the direction of the shadow ray or the reflection ray may be jittered by a random distribution as well. Each method has its own effects.

Some ideas to reduce the run time of the ray tracer:

- **Hidden Surface Algorithm:** Intersection point calculation can be accelerated excluding surfaces hidden from the primary viewing ray reducing the amount of intersection tests.
- **Bounding Volumes:** Each object is enclosed by a convex region which can be inexpensively intersected. You may use axis aligned bounding boxes (AABB) or spheres. A viewing ray is first tested for intersection with the volume and depending on whether the volume is intersected, with the enclosed object itself.
- **Space Partitioning:** Each ray (primary, secondary or shadow ray) is only tested against a subregion of the entire scene and thus only against a subset of the entire set of primitives contained. The space subdivision may be uniform (uniform grids) or adaptive to the “local density” of the objects in the scene (octree, kd-tree).

Verify the correctness of your implementation:

- You find a set of example files named ref*.xml in the directory /proj/fapra/handout/A4/data. The correct results, named identically, can be found as .ppm-files. Render all scenes in the directory and compare your results to the reference images. This should reveal the most common mistakes.
- Further be concerned about the efficiency of your program. Determine the run time of your implementation for the individual reference scenes and compare the results to the reference implementation. Your implementation shouldn’t exceed the run time of the reference implementation considerably. For timing measurements you may use the shell command time.

5.2 Undesired Extensions

Following approaches are not approved and lead to deduction of points.

- **Hacks** to “accelerate” the existing code. We are aware that our program skeleton is not as efficient as it could be in all areas. But in favor of readability and simplicity it is implemented the way it is. Optimizing our code is the task of the compiler, not yours!
- **External Code** to achieve high performance, possibly provided in form of an ugly C-hack, Fortran library, or Assembler routine and other code not written in object oriented C++ is not allowed!
- **Unknown Behavior:** Strive for clean code and coding techniques, like dynamically allocating memory for arrays of unknown size instead of an obscure maximal size. Your implementation should behave efficient for different scenes with different object complexities. For example a rectangular bounding volume enclosing a cube looks any kind of sense.
6 Literature


An additional short overview and a well done introduction to ray tracing is available under:

- http://www.geocities.com/jamisbuck/raytracing.html

7 Rating Criteria

For this assignment 20 points can be achieved in total. Therefore the following criteria have to be fulfilled:

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Point</td>
<td>The entire ray tracer is written in object oriented ANSI C++ code. There are no global variables or any kind of C-code relics.</td>
</tr>
<tr>
<td>1 Point</td>
<td>The code is well and reasonable commented and compatible to doxygen.</td>
</tr>
<tr>
<td>1 Point</td>
<td>The parser is capable of reading and processing data in the format described.</td>
</tr>
<tr>
<td>3 Points</td>
<td>Code and data structures are extended to also handle cones and infinite planes.</td>
</tr>
<tr>
<td>1 Points</td>
<td>Objects are transformed and normals are calculated correctly.</td>
</tr>
<tr>
<td>2 Points</td>
<td>Primary rays are calculated correctly.</td>
</tr>
<tr>
<td>2 Points</td>
<td>Objects are visible in correct order, i.e., the nearest surface is displayed for each pixel (correct depth ordering).</td>
</tr>
<tr>
<td>1 Point</td>
<td>Diffuse lighting is done correctly.</td>
</tr>
<tr>
<td>2 Point</td>
<td>Specular reflection and refraction are handled correctly.</td>
</tr>
<tr>
<td>3 Points</td>
<td>The ray tracer has been improved, i.e., either extended or improved functionality has been implemented. Although the improvement may be trivial it has to be documented in the README file.</td>
</tr>
<tr>
<td>2 Points</td>
<td>The file data/my_scene.xml describing a scene with at least 5 objects has been created. The scene has to be different from all other FaPra participants and has to be correctly rendered with your own ray tracer.</td>
</tr>
<tr>
<td>1 Point</td>
<td>The ray tracer is not slower than the reference implementation (a program is slower than another if it takes at least twice as long for rendering). Crucial for grading are the reference scenes (ref*.xml).</td>
</tr>
</tbody>
</table>

Precondition for grading is the error-free compilation of the program on the computers of the VISGS pool. There will be deduction of points if the program compilation leads to warnings (Warnings of QT in the style of "...has virtual functions but non-virtual destructor" are excluded therefrom)! The annotation of the source code should be that extensive to provide enough information to the author to explain the functionality of the code to the supervisors.