Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Bachelor's Thesis

Ray Tracing Algorithm For Interactive Applications

Tomas Nikodym

Supervisor: Ing. Vlastimil Havran. Ph.D.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science

May 20, 2010

Acknowledgements

I would like to thank to the supervisor of my thesis, Ing. Vlastimil Havran, Ph.D. for his advices and for pointing me in the right direction. To my parents for their support and to everyone who positively influenced my life.

Declaration

Abstract

Rendering is a process of generating a 2D image from a mathematical description of a 3D scene. There are many ways to approach this task, driven by demands of a particular application. From animated movies, where the visual quality is the primary goal, to interactive applications (e.g. computer games), where the time available to render one frame is not much more than a few milliseconds.

The ray tracing algorithm has long been used for offline rendering, and the rendering times were far from interactive. But as the performance of rendering hardware increases, ray tracing becomes an option for interactive applications.

In this thesis, I will focus on the two different areas of the use of the ray tracing algorithm. I will attempt to implement an interactive application for the synthesis of images based on the ray tracing algorithm, with the option to render a particular view in much higher visual quality, though not in real-time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ray tracing is a very flexible rendering technique because of its ability to simulate optical effects. Unfortunately, it requires much more computational resources than rendering techniques based on rasterisation. It used to be a method suitable only for offline rendering, but with the ever increasing computational power of personal computers, time will soon come when ray tracing is used in real-time applications, executed on standard personal computers.

In this thesis, I will introduce the most important aspects of implementing the ray tracing algorithm for interactive applications, discuss possible solutions to each such aspect and present design and implementation of an interactive ray tracer.

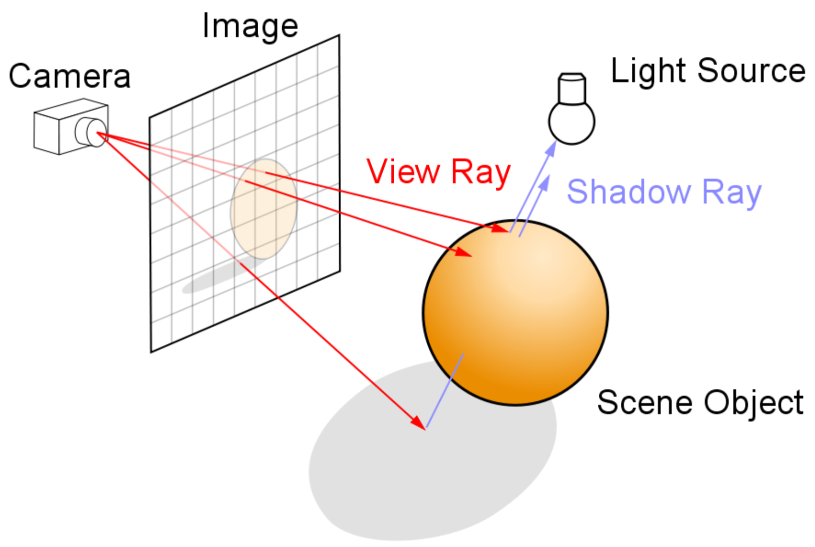


Figure 1.1: The ray tracing algorithm renders an image by casting rays into the scene. Diagram adopted from wikipedia.org under the terms of GNU Free Documentation License.

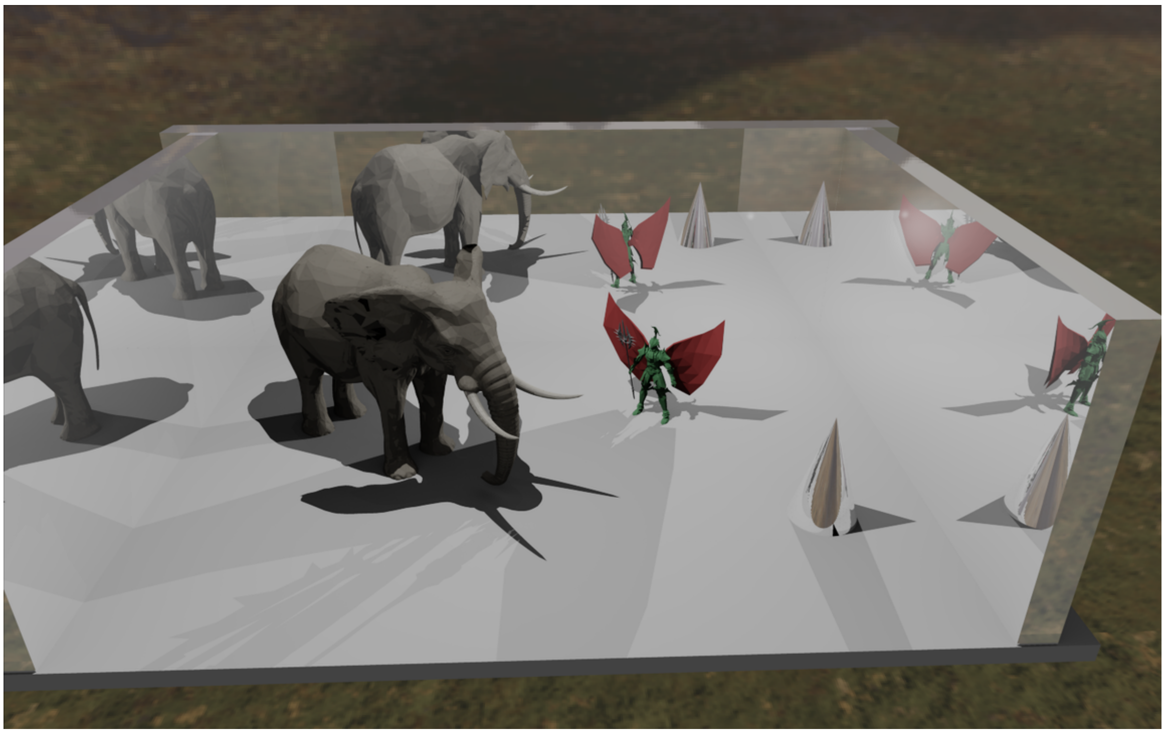


Figure 1.2: Ray tracing can achieve a very high degree of photorealism, as in this picture rendered with my implementation of the interactive ray tracing algorithm.

1.1 Principle

The ray tracing algorithm renders an image by casting rays into the scene. A ray is cast through every pixel of the image, tracing the light coming from that direction. When the ray hits an object, the light incident of that point is evaluated to compute the amount of light reflected back to the camera, casting more rays into the scene from this point. Sampling the scene with many rays from each point of interest would be way to costly to achieve interactive frame rates, so only the most important directions are sampled.

For diffuse surfaces, most reflected light will usually come directly from light sources, thus one “shadow ray” is cast towards each light source, to find whether the source is visible or not. If it is not shadowed, the light incident is used to compute the shading of the point of interest, according to the surface properties.

The steps described above would render an image with shadows and surface shading, but to obtain light reflection and eventually refraction, we need to introduce recursion into the algorithm. If the surface is reflective, another ray is cast in the direction of ideal reflection. This ray is traced in exactly the same way as the primary rays (rays cast from the camera). The recursion stops when a maximal depth is reached, or the contribution factor drops below certain threshold.

To reproduce light refraction at the surfaces of transparent materials, another ray is cast in the direction of ideal light refraction. The same rules apply as for the reflected rays.

The information obtained by casting these rays are then put together (modulated according to surface material properties - colour, shininess, ... and to the angles between the rays and the surface normal) to evaluate the amount of light that leaves this point in the direction of the camera.

See the pseudo code of this simplified algorithm in Figure 1.3, diagram illustrating the ray casting in Figure 1.1 and an example of an image obtained with this algorithm in Figure 1.2.

To improve the visual quality of the resulting images, various advanced rendering techniques are used. Most notably it is the global illumination (which takes into account diffuse-diffuse reflection), and several techniques simulating the interaction of the ray of light with the lens (light refraction causing decreased depth of field and light scattering causing bloom effect). Some of these techniques are described later in this thesis (Chapter 5).

1.2 Structure of the thesis

In chapter 2 we discuss several methods of scene representation, including acceleration structures and their construction. In chapter 3 we go over the pixel representation and introduce the concept of tone mapping. In chapter 4 we present the major optimization techniques, which help to achieve real-time performance. In chapter 5 we present some advanced rendering techniques. In chapter 6 we introduce the design of the rendering framework. In chapter 7 we give some more details about the implementation of the rendering framework. In chapter 8 we present the results of our work. And in chapter 9 we conclude what was achieved with respect to the goals set and give some pointers for further improvements of this project.

```
TraceRay(ray, depth)
{
  if(depth > maximal depth)
    return 0;

  find closest ray object/intersection;

  if(intersection exists)
  {
    for each light source in the scene
    {
      if(light source is visible)
      {
        illumination += light contribution;
      }
    }
    if(surface is reflective)
    {
      illumination += TraceRay(reflected ray, depth+1);
    }
    if(surface is transparent)
    {
      illumination += TraceRay(refracted ray, depth+1);
    }

    return illumination modulated according to the surface properties;
  }
  else return EnvironmentMap(ray);
}


for each pixel
{
  compute ray starting point and direction;
  illumination = TraceRay(ray, 0);
  pixel color = illumination tone mapped to displayable range;
}
```

Figure 1.3: Pseudo-code of a basic ray tracing algorithm.

Chapter 2

Scene representation

2.1 Acceleration structures

The ray tracing algorithm achieves high level of photorealism, and the fact that it is based on the physical properties of light simplifies the simulation of real world optical effects. Apart from that, using a proper space-partitioning technique, the asymptotic time complexity can reduce significantly. Indeed, a balanced space-partitioning data structure can yield worst-case time complexity $O(\sqrt[3]{n})$, and on average close to $O(log_2(n))$.

In this section, I will introduce some of the most popular space partitioning data structures, and discuss their advantages and disadvantages. The methods for the construction of such data structures will be discussed later in this chapter.

A more detailed comparison of common acceleration data structures can be found in [1, 2, 3] and implementation details in [4].

2.1.1 kD-Tree

The kD-Tree (short for k-Dimensional Tree), is a binary tree in which every node splits the space into two subspaces by a splitting plane perpendicular to one of the k axes (in the context of ray tracing, we are dealing with the 3-dimensional Euclidian space, so k=3) [5, 4, 3].

The advantage of kd-trees is primarily the fast traversal, making it probably the most popular acceleration structure for static scenes [1]. The main problem with kd-trees is that the rebuild/update time is significant, which limits its use for dynamic scenes, although special techniques for minimizing this overhead exist [1].

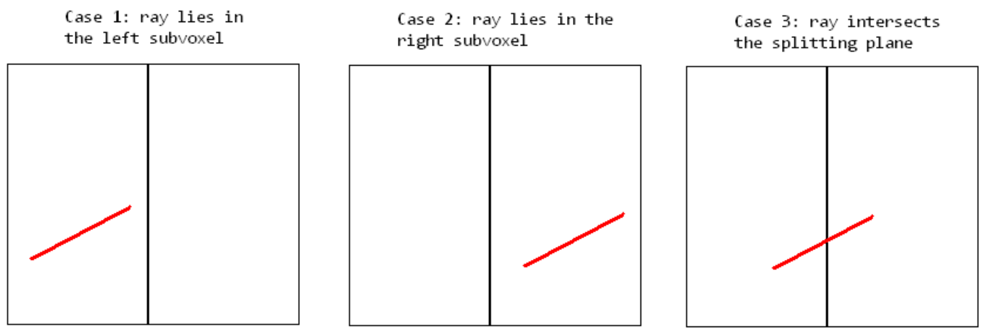

Case 1: ray lies in the left subvoxel
Case 2: ray lies in the right subvoxel
Case 3: ray intersects the splitting plane

Figure 2.1: kD-tree traversal

kD-tree traversal

When traversing a kd-tree data structure, there are two distinct cases that may occur [2]:

1) Both limiting points of the ray lie at the same side of the splitting plane.
In this case, we can omit traversal of the subspace which does not contain the ray, so that only the left/right child node needs to be examined.

2) The ray intersects the splitting plane.
It might be the case that the ray-object intersection lies in any of the two subspaces. Thus, we first need to traverse the subspace containing the end of the ray closer to the observer. If an intersection is found, the traversal ends. Otherwise, the other subspace has to be traversed.

See figure 2.1 for illustration.

Data representation

To represent this data structure in program memory, we need one floating point number to specify the coordinate of the splitting plane, one pointer to the child nodes (assuming the memory layout is organised so that the second child is located right after the first one), and a flag to tell whether this node is a leaf or not and eventually which one of the three axis it is perpendicular to (2 bits of information).

To further optimize this data structure, we can exploit the fact, that the data is aligned in memory in blocks of 32 bits (or more), so the lower 2bits of the pointer can accommodate the flags, which reduces the size of one node to only 8bytes (assuming 32 bits system). Masking the pointer to get the actual address or the flags introduces some extra overhead, but the advantage of having more compact data structure (so that cache can be better utilized) overweighs this overhead [4].

2.1.2 Bounding volumes hierarchy

A bounding volume for a set of objects is a closed volume that completely contains the union of the objects in the set [5]. If a ray does not intersect the bounding volume, it clearly cannot intersect any of the objects contained in the volume.

BVH

The efficiency is further improved by constructing a hierarchy of the bounding volumes, so that the actual geometry data is encapsulated in many small bounding volumes, which are than grouped together to created bigger bounding volumes containing them and so on until, at the top level, only a few bounding volumes exist, containing all the objects in the scene. By a simple recursive algorithm, one can then traverse through all the levels of this hierarchy to the geometry data and test the intersection with only a small fraction of the original number of primitives in the scene.

There are many different types of bounding volumes; bellow, there is a list of the most popular ones. Generally, the more complex the volume, the slower the intersection test, but the more closely it encapsulates the objects contained in it [5].

Common types of bounding volumes:

- Bounding sphere
- Bounding box - axes-aligned bounding box (AABB) or oriented bounding box (OBB)
- Discrete Oriented Polytope (DOP)
- Convex Hull

As mentioned in the previous section, bounding volumes hierarchy is most suitable for dynamic scenes [1], where many other data structures have problems because of the computationally expensive rebuild. If an object moves, the volumes encapsulating it move with it, making the update fairly simple.

2.2 Construction of acceleration structures

Because kd-trees are becoming the number one choice in data structures for the acceleration of ray tracing, outperforming alternative structures including BVHs (although they might be more suitable for dynamic scenes than kd-trees [1]) and grids, in this section, I will focus only on the aspects of building a kd-tree data structure.

More information about building of acceleration structures using heuristic methods can be found in [6, 7].

2.2.1 Spatial median splitting

In this naive method, the splitting plane is always positioned at the spatial median of the voxel, and the dimension in which to split the voxel is chosen is so called "round robin" fashion - that is, the first splitting plane is perpendicular to x, the next one to y, z, x again and so on [7].

The subdivision is usually performed until a maximal depth is reached, or the number of triangles in a node falls below a certain threshold.

The construction time of this trivial method is $T(n) = 2T(\frac{n}{2}) + O(n)$, which gives $O(n \times log(n))$, the theoretical lower bound for kd-tree construction. But the effectiveness of the resulting structure for traversal is several times less than if a heuristics based method was used [7].

2.2.2 Surface area heuristics

Surface area heuristics (SAH) is a method which attempts to estimate the cost of traversal of a random ray thought a voxel. It considers the geometry of the resulting child voxels as well as the number of triangles to each side of the plane, and the triangles intersecting the splitting plane.

The SAH method makes several assumptions to ease the process of evaluating the cost [7]:

- The rays intersecting the parent voxel are uniformly distributed infinite lines.
- The cost of intersecting n triangles is linear function of the number of triangles.
- The cost of triangle intersection test and traversal step are known.

The SAH cost function roots from the geometry theory, stating that the conditional probability that a ray hitting a convex voxel will hit its sub-voxel is proportional to the ratio of the surface area of the sub-voxel to the surface area of the voxel [6].

$$P(V_S|V) = \frac{SA(V_S)}{SA(V)}$$

V_S ... sub-voxel,

V ... voxel,

$SA(N)$ ... surface area of volume N,

P ... conditional probability,

This leads to a cost function of the following form [7]:

$$C_R = K_T + K_I \times \frac{1}{SA(N)} \times [n_l \times SA(n_l) + n_r \times SA(n_r)],$$

where:

C_R ... total cost,

K_T ... cost of traversal step,

K_I ... cost of intersection step,

$SA(N)$ ... surface area of volume N,

N_l, N_r ... left/right child sub-voxel,

n_r, n_l ... number of triangles intersecting left/right sub-voxel.

The goal of local greedy algorithm is to find a split with minimal cost.

Now that we have a tool to evaluate the cost of a split, we need to find out what splitting planes could be candidates for our analyses. Clearly, the minimum of the cost function must lie on one of the vertices of the triangles intersecting the voxel. There are more informed methods to avoid evaluating of the cost for vertices that cannot yield the minimum cost, but for our purpose of implementing an offline builder where only the quality of the resulting kd-tree matters, not the construction time, we can simply evaluate the cost for all three axis of each proposed vertex and choose the minimum.

2.2.3 Automated termination

For the methods not using any heuristics (e.g. spatial median splitting), the intersection usually ends when a certain depth is reached or when the number of triangles in the voxel drops below a threshold.

Given the means to evaluate the cost of traversing a voxel, the automated termination criterion can be easily modelled using these heuristics. Further subdivision is worth doing if the minimum cost is less than the cost of not splitting the voxel.

Although the above method works quite well, the problem is that it will stop subdividing the voxel in a local minimum, which does not necessarily have to be a global minimum. One possible solution to this would be to evaluate a further few steps even if the criterion tells us not to, to see whether a few more splits might yield any improvement [7].

2.3 Geometry representation

2.3.1 Constructive solid geometry

Constructive solid geometry (CSG) is a method of creating a complex surface or object by using Boolean operators to combine simpler objects [5]. The simplest solid objects used for the representation are called primitives. Typically they are objects of simple shape: cuboids, cylinders, prisms, pyramids, spheres, cones, etc.

Clearly, the advantage of using CSG in ray tracing is that the primitives are easy to test intersection with. On the other hand, it is quite restrictive, as most modelling software allows the user to model and export the objects only on polygonal bases.

2.3.2 Polygonal meshes

The advantage of using polygons to represent the geometry is that every object can be modelled in some level of details with sufficient number of polygons.

Another decision to be made is what types of polygons to support. Bellow I will discuss on what the pros and cons of each approach are.

Triangulation

Every polygon, no matter how complex, can be converted into a bunch of triangles [8]. But to achieve real-time rendering rates, we are most concerned about the performance, as the computational time and resources are very limited. Thus, the question is how to get the best efficiency.

The advantage of supporting complex polygons is that using a clever method to test the ray-polygon intersection can be faster than testing every of its triangles separately [9]. On the other hand, supporting more than one polygon type will introduce additional overhead in the form of branching required to identify the polygon type and to invoke the proper intersection test code [9].

Chapter 3

Pixel representation

The illuminance of typical scenes that can be observed in real life ranges from about $10^{-4}\ lx$, when only stars enlighten the scene, to that of direct sunlight, which is approximately $10^{5}\ lx$ [10]. The problem is that the luminance of a typical monitor is no more than around $10^{2}\ cd/m^{2}$.

To overcome this restriction, we will store the pixel intensity values as floating point numbers, and then, before the frame buffer can be displayed, a tone mapping function will be applied to map the values from $[0, \infty)$ to $[0, 1]$, so that the image can be displayed on a standard physical device.

3.1 High dynamic range

Rendering the scene in floating point numbers will not only preserve details in both bright and dark regions, but it also allows us to specify light sources far out of the range displayable on a physical device, and to do the computation with a physically based values, rather than just some approximation to make the result look realistic. Also, using a proper tone mapping operator gives us control over the exposure time, which is very useful for adaptive tone mapping [10, 11, 9].

3.2 Tone mapping

The main purpose of a tone mapping operator is to map the illuminance value to a range displayable on a standard display device.

The simplest method would be to clamp everything what we cannot display, but clearly, this method is not good enough to preserve the details and to make the image look natural and realistic. Another option is to map the values on logarithmic bases, which approximates much closer the human eye sensation of light than the above approach, but even this method causes many unnatural effects [10, 11, 9]. After testing several different tone mapping operators, I decided to use the sigmoid operator, proposed by Erik Reinhard (2007) in [12]:

“Sigmoids have several desirable properties. For very small luminance values the mapping is approximately linear, so that contrast is preserved in dark areas of the image. The function has an asymptote at one, which means that the output mapping is always bounded between 0 and 1. A further advantage of this function is that for intermediate values, the function affords an approximately logarithmic compression.”

(for more details, see [10, 12, 13, 11])

Sigmoid Curve

$$L = \frac{Y}{Y + c(t)},$$

where:

L . . . Relative pixel intensity $[0, 1]$,

Y . . . Illuminance $[0, \infty)$,

c(t) . . . Tone mapping constant (variable in time).

Another useful property of this operator is that the constant c represents the exposure time. This enables us to model the ability of a human eye to adapt to the current overall perceived light intensity, by changing this value according to the global (or local) average pixel intensity.

Chapter 4

Optimization techniques

Implementing proper space partitioning data structure to traverse the scene geometry and choosing sensible representation of the geometry data is not good enough to achieve real-time performance. In this section, I will describe some of the most important optimization techniques used to speed up the rendering process.

4.1 Cache coherence

On current computers, memory latency presents more of a bottleneck than the CPU [4], so in a real-time ray tracer, it is essential to minimize this overhead. To do so, there are several steps that need to be taken. First, to improve special locality, the tree representing the scene partitioning is stored in a continuous block of data, with the node data structure being as small as possible. For more details on this, read the "data representation" section of the chapter about kD-Trees (2.1.1. Similar technique is used to store the geometry data, where the geometry representation of a triangle stores only a pointer to shading data, instead of storing the data itself directly in the data structure, because a triangle geometry data structure is accessed on average 3 times more often than the shading data corresponding to the triangle.

4.2 MIMD

The computation of each pixel value is independent of any other pixel, which makes the algorithm easily parallelisable amongst multiple cores. Ideally I would implement the ray tracer in CUDA to exploit the computational power of the GPU, which is, for parallel task like this one, much higher than that of the CPU. But even a ray tracer implemented in C/C++, running on CPU, can benefit from the MIMD level of parallelism (short for Multiple Instruction Multiple Data), as most of current CPUs have more than one processing core. This can yield speed up factor almost linearly proportional to the number of cores [4].

4.3 SIMD

The SIMD (Single Instruction Multiple Data) level of parallelism is a bit harder to exploit. One possible way is to trace patches of rays [4] (most CPUs consist of 4 ALUs, meaning it can perform the operation on four 32b numbers at the same time, thus the patch would be of four rays). Ideally, this would speed up the traversal of the BSP tree by a factor of 4, but this technique introduces a significant overhead. Even though we choose the patch to consist of spatially coherent rays, it might be the case that while some of the rays in the patch does not intersect a splitting plane, others does, so we need to traverse both subspaces with the whole patch. According to Erik Reinhard (2002) - Practical parallel rendering [4], this technique can yield about 2 times faster traversal. The SIMD parallelism could be used in the ray-triangle intersection test as well, but coding the computation properly should allow the compiler to do most of the job for us.

4.4 Program structure

Function calls and recursion causes additional overhead, so it is sensible to omit those in the critical parts of the code. Ray traversal and intersection test is performed many times for each pixel, yielding in millions of times of execution per frame. This is clearly the section of the code we need to focus at most. Replacing function calls with inline functions or code pastes is fairly easy; the more tricky part is to get rid of the recursion. To implement the ray traversal using a loop, we need to use a stack to push the data on every time a ray intersects a splitting plane. Restructuring the program to minimize the overhead together with using some compiler dependent optimizations (e.g. branch hints) yielded about 1.3 times faster traversal in my implementation.

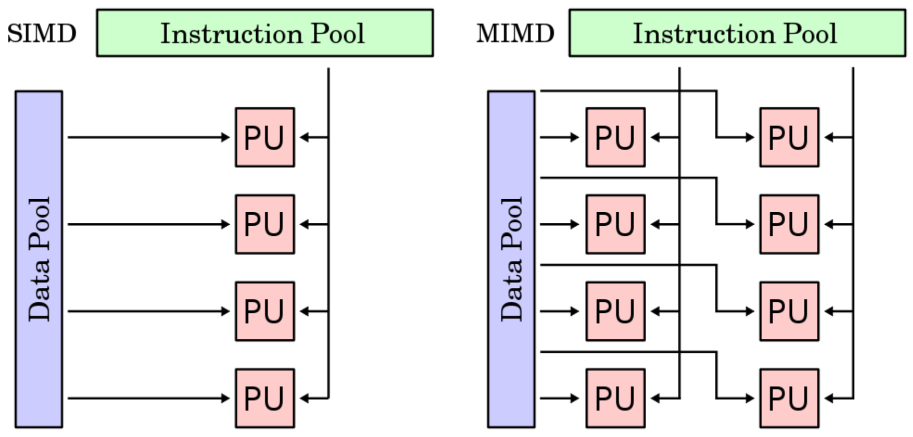


Figure 4.1: The difference between SIMD and MIMD level of parallelism. Diagram adopted from wikipedia.org under the terms of GNU Free Documentation License.

Chapter 5

Advanced rendering techniques

Most of the techniques described in this section are too costly to compute in real-time, but they can be used to increase the photorealism of the resulting image in offline rendering. A survey of most of these techniques can be found in [2, 14].

5.1 Anti-aliasing

Anti-aliasing is a technique of minimizing the artifacts known as aliasing when the signal is sampled at lower frequency than the actual resolution of the signal [5, 2]. What this means in ray tracing is that sampling the scene with only one ray per pixel can lead to visible artifacts decreasing the overall visual quality of the resulting image. Using a proper anti-aliasing method can make the resulting image appear softer and more realistic.

Supersampling

Supersampling is a technique to eliminate the aliasing of the image. Samples are taken at several instances inside each pixel, and the average colour is then calculated. It means that the image has to be rendered at higher resolution to be then downsampled to the desired size.

Types of supersampling

These methods basically just differ in where the extra samples are taken inside the pixel [2, 5]:

- Grid algorithm
- Random algorithm

- Poisson Disc algorithm
- Jitter algorithm
- Rotated Grid algorithm

See figure 5.1 for illustration.

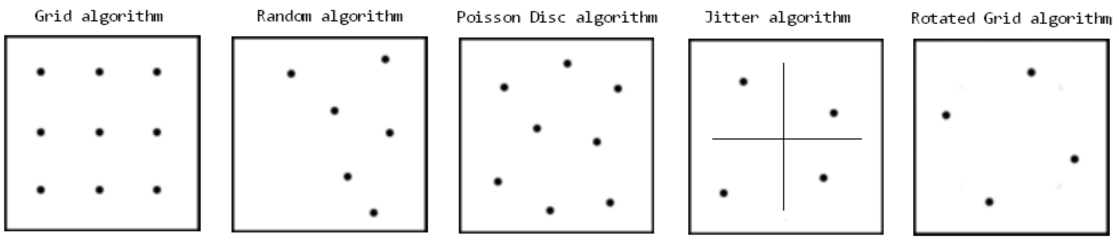


Figure 5.1: Sampling patterns for different supersampling methods.

5.2 Ambient occlusion

Ambient occlusion is a shading method which helps to add realism to local reflection models by taking into account attenuation of light due to occlusion. It attempts to approximate the way light radiates in real life, especially off what are normally considered non-reflective surfaces [5, 2]. Ambient occlusion is a global method (unlike Phong shading, which takes into account only light coming directly from light sources, not the scene geometry), but it is only an approximation of what would be achieved with a proper global illumination technique.

Calculation of ambient occlusion

Ambient occlusion is usually calculated by casting rays in every direction from the surface through the hemisphere, counting the fraction of rays which hit the background (sky) [2]. The higher this ratio is, the brighter the surface appears. This technique can generate for example dark corners, which would not be achieved with a local shading method [2, 15].

See figure 5.2 for illustration. Several images rendered with ambient occlusion are presented in 8.2. Most notably it is 8.3 and 8.2, which are rendered without direct illumination.

5.3 Bloom effect

Bloom effect is used to reproduce an artifact of real-world cameras (and human eye as well). When there is a very bright object in the scene, the light appears to bleed into surrounding objects. This effect is caused by scattering of the light in the lens [5, 2].

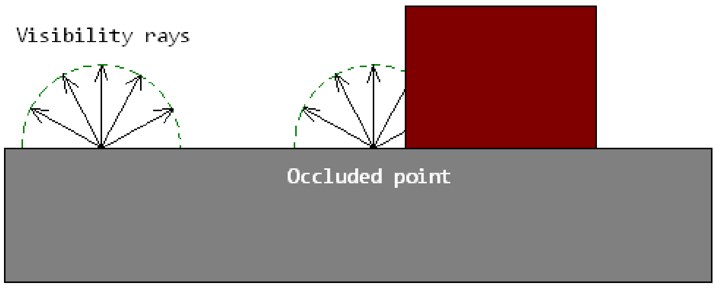


Figure 5.2: Calculation of ambient occlusion. Visibility rays are cast from the point of interest through the hemisphere - the more rays hit a surface, the more occluded the point is.

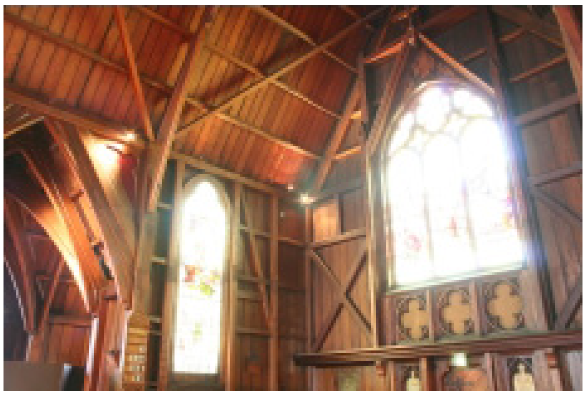


Figure 5.3: An example of bloom on a photo taken with a real-world camera. Notice the light on the window bleeding on the darker areas around it. This picture was published under the terms of GNU Free Documentation Licence.

Reproduction of the bloom

When the image is represented in high dynamic range (HDR), this effect can be reproduced by convolving the image with a kernel of an Airy disc (to simulate very good lenses), or by applying the Gaussian blur (for less perfect lenses), before converting the image into fixed-range (LDR) [2].

See figures 5.3 and 8.6 for illustration.

5.4 Depth of field

Depth of field (DOF) is the portion of a scene that appears acceptably sharp in the image. When a real-world lens is focused to one distance, the sharpness decreases gradually on each side of the focused distance [5, 2, 16]. Although sometimes it might be desirable to have a very deep DOF, in most cases it looks more realistic to have a smaller DOF, and it is also useful to emphasise the subject of interest while de-emphasising the rest of the scene.

Theory and implementation of DOF

In the simplest implementation of a ray tracer, the rays are casted from the camera through a point at the viewing plane corresponding to the pixel being rendered. This method will result in the entire image being sharp (infinite DOF). In real-world lenses, the light is refracted at the surface and due to the shape of the lens, it leaves under a different angle than the ray of light travelling through the centre of the lens. This physical property of light is the reason for the effect of decreased sharpness of images further from the focus point. Because the ray tracing is a simulation of the behaviour of light, this effect can be easily simulated.

This optical system has two basic properties, which can be used to reproduce the effect of finite DOF. First, rays of light passing through the optical centre of the lens are not refracted, they leave the lens at the same direction as they came [16]. Second, the rays hitting the "sharp plane" at the same point will project to the same point on the "projection plane" [16]. This idea is illustrated on figure 5.4.

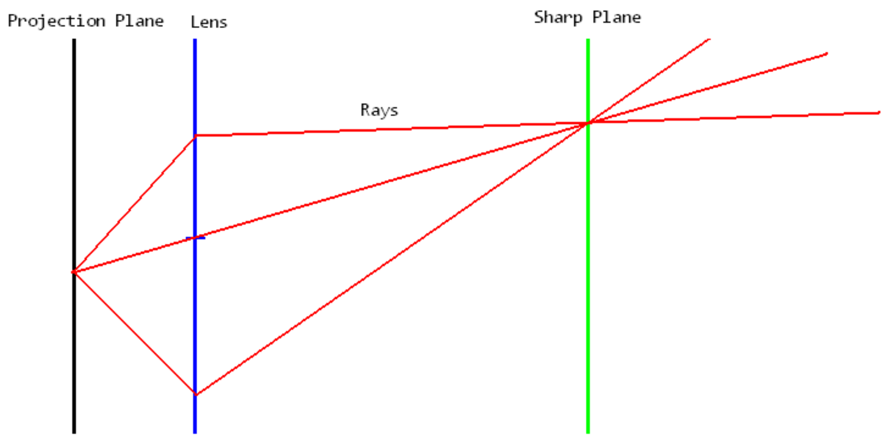


Figure 5.4: Depth of field.

Before casting a ray, intersection of a line from the camera to a pixel being rendered, and the surface of the imaginary lens (plane perpendicular to the viewing direction), has to be computed. This point is then used as a starting point of the ray to be traced, and the direction of the ray is derived from the point of perfect sharpness (every ray has to pass through this point). This ray is then traced in the same way as would be the original ray in the infinite DOF algorithm [2, 16].

Example

The effect of DOF can be best illustrated with a real-world camera photograph. Figure 5.5 is a photograph of a kitty taken with focal ration f/2.8. The face of the cat is in focus, making the background appear blurred and isolating the cat.

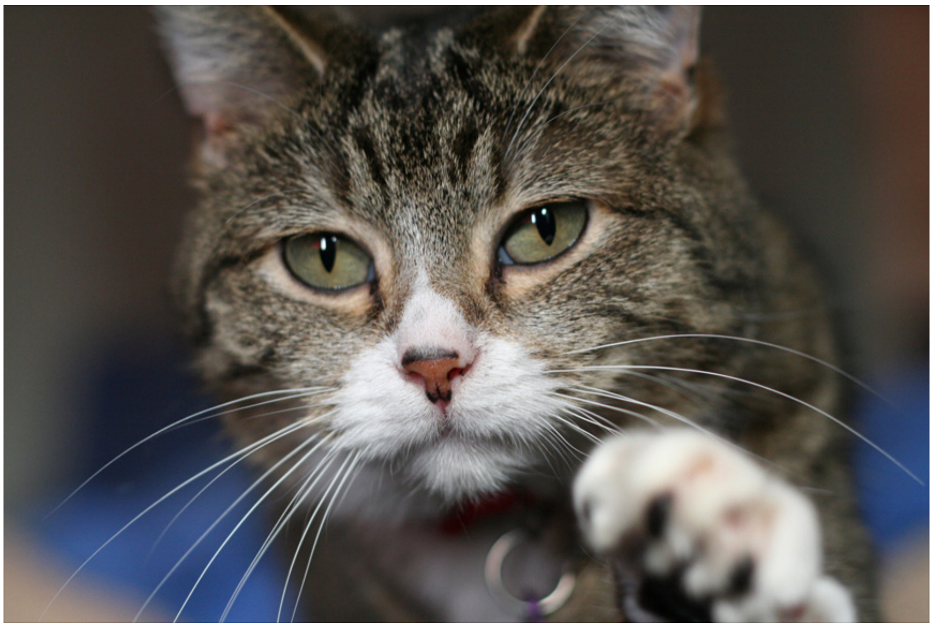


Figure 5.5: Photograph of a cat illustrating DOF. Photographed by David Corby and published under the terms of GNU Free Documentation License.

5.5 Ray propagation

In the above discussion we assumed that the radiance is constant along a ray. This assumption holds only if the scene is placed in vacuum, and becomes very inaccurate if the rays pass through a fog, smoke, dust, etc [14]. There are three main ways in which the ray is affected by the participating media [14]:

1. **Absorption**
   particles present in the medium causes the energy of the light to convert to another form of energy, usually heat.
2. **Emission**
   luminous particles present in the environment add energy to the ray.

3. **Scattering**
   light scatters to other directions due to collisions with the particles.

Absorption

Absorption is described by a constant σ_a, the probability that light is absorbed in the medium per distance travelled in that medium.

The fraction of light absorbed can be described with the following equation [14]:

$$\alpha = e^{-\int_0^d \sigma_a(\mathbf{p}+l\omega)dl}$$

where:

d ... distance travelled in the medium,

$\mathbf{p}$ ... position,

ω ... direction,

σ_a ... probability function.

Note that the term above can be significantly simplified for special cases, e.g. homogenous fog: $e^{-\sigma d}$

Scattering

The scattering effect includes two different components:

1. **Out-scattering**
   Out-scattering is when the light is scattered from the direction of the ray being traced out in other directions. It results in light attenuation and can be modelled with the same equation as absorption.

2. **In-scattering**
   In-scattering is when the light is scattered from other directions towards the direction of the ray being traced. It adds energy to the ray.

Chapter 6

Design of Rendering Framework

In the previous chapters, I discussed the theoretical aspects of the ray tracing algorithm. In this chapter, I will introduce the design patterns I decided to use in my implementation of the interactive ray tracer.

6.1 Application overview

The project consists of 5 independent applications, which can interact with each other through a user interface (UI) application (see 6.1).

6.2 Modularity

Ray tracing server

The Ray Tracing Server (RTS) is designed to run in the background, accepting commands from the user via shared memory and a message queue (through user interface), and storing the results into a block of shared memory, so that other applications running on the same machine (UIs) can access it.

The main task of the RTS is to render given scene interactively; the camera can be controlled by a user via the UI, and various options (image quality, etc.) can be set via the UI as well. Two modes of camera control are supported; one in which the camera can move freely throughout the scene, and second in which collisions are detected, so that the user cannot go through solid objects, and can walk through the terrain, jump, crawl, climb, etc.

The user can also export current camera settings into a file, so that it can be imported later by the RTS, or it can be used by the offline renderer to render the view in higher visual quality. Not only single images can be exported, but the user can also start/stop recording to export a full video sequence.

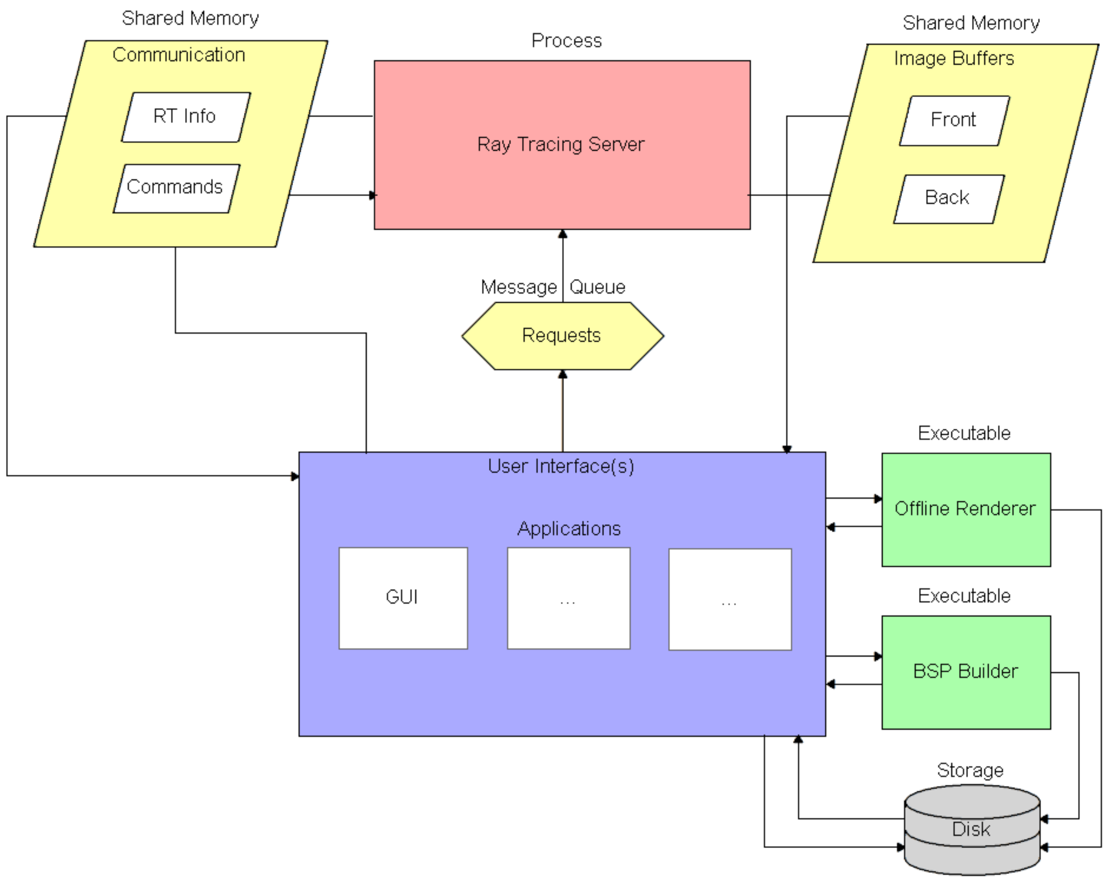


Figure 6.1: Application overview.

Offline renderer

The Offline Renderer is a console application (but can be run from the UI as well), which accepts a scene description file (specifying rendering quality, references to the camera description file, geometry file, output file name and format, environment maps, and more), to produce a synthesised image of the scene, taken from the specified camera. The Offline Renderer achieves higher level of photorealism than the interactive RTS, by enabling computationally expensive tasks such as ambient occlusion, bloom effect, and by rendering the scene in higher resolution with anti-aliasing.

BSP builder

The BSP Builder is responsible for the import of 3D models and construction of kd-trees. Once the kd-tree is computed, it is stored in a binary file in a form which can be loaded very quickly. This geometry file (.bsp) is then used by the RTS and the Offline Renderer. A large number of 3D file formats is supported (more than 25, including .3ds, .obj, .x). The BSP Builder uses a heuristic method, known as Surface Area Heuristics (SAH), to construct optimal kd-trees.

Graphical user interface

The Graphical User Interface (GUI) provides an interface between the user and all parts of the application (including RTS, Offline Renderer, and BSP Builder). Its main purpose is to send commands to the RTS via message queue and shared memory and to provide the user with the outcome of the computation done by the RTS. Although the Offline Renderer and BSP Builder are designed to work as standalone applications and can be executed from the command line, it might be preferable to use the GUI, as it provides more user-friendly access to all the features of these applications.

Console user interface

The Console User Interface (CUI) is a small application providing the user with access to RTS running on the same machine. It can pass commands from the user to a running RTS process or acquire and display information. The CUI can work concurrently with the GUI, connecting to the same RT server, and provides a subset of functionality provided by the GUI, in a command line manner.

Inter-process communication

For the purpose of connecting the RTS with the UIs, I wrote a small library giving the programmer access to the RTS without the need to know anything about the source code of the ray tracer, or even to possess it.

Thus, it would be fairly easy to extend the concept of modularity to applications communicating with the RT Server over network. One would just need to write a bridge application listening on a network port and passing the commands to the RTS through the IPC library, and possibly streaming the video output back to the client.

Chapter 7

Implementation

In this chapter, I will discuss the choices I made on how to implement the techniques and data structures described in the previous chapters, briefly explain their implementation, and present some diagrams scratching the modularity and program flow of the ray tracer.

7.1 Ray tracer

7.1.1 Acceleration structure

Traversal of the scene partitioning hierarchy is the key aspect of a fast ray tracing algorithm. I decided to use the kd-tree for this task. Its advantage is that the traversal step is very fast (basically just two floating point comparisons), and if the tree is suitably constructed and balanced, the number of triangle intersection tests to be performed per ray can be very low - only a few tests for scenes with millions of triangles. See chapters 2.1.1 and 4 for details on the implementation.

7.1.2 Geometry representation

The proposed ray tracer is intended to be an alternative of classical scanline algorithms, thus it should support file formats normally used in applications like computer games, that is polygonal meshes, rather than constructive solid geometry (CSG), which is rarely found in applications of this kind.

Next decision to be made is whether to support complex polygons or only triangles. Every polygon can be converted into several triangles in a pre-processing step, so this choice have no effect on what file formats will be supported. As mentioned above (section 2.3.2), testing ray-polygon intersection with a complex polygon can be faster that testing every of the triangular polygons derived from it, but to allow this feature, extra overhead is introduced in the form of branching, which is present in every intersection test, even if the tested polygon is triangular. I believe that the strategy of minimizing the time needed to compute a ray-triangle intersection will lead to a better overall performance that introducing various types of polygons, so in my implementation, every polygon is triangulated before the construction of a kd-tree and only triangles are supported in the rendering process. Chapter 2.3 deals with this matter in more details.

7.1.3 Pixel representation

As discussed in chapter 3, pixel intensities are computed in floating point numbers and then tone mapped onto the range displayable on a physical device using the sigmoid curve. For more details, see chapter 3.

7.1.4 Optimizations

All implemented optimization techniques are described in chapter 4. In this section, I will just briefly summaries the implementation of these techniques.

MIMD level of parallelism is exploited using OpenMP (Multi-processing API). Each processing unit is given severel lines of pixels to render, when its job is done, it gets more. The computation of every pixel is independent of the others, so there is almost no overhead caused by the communication between threads. The rendering performance scales almost linearly with the number of processing units.

The memory latency is minimized using a well design memory layout and cache aware programming techniques. More details can be found in 2.1.1 and 4.1.

The program structure is designed to minimize the overhead regarding the function calls and recursion in the most critical sections of execution, as is the ray traversal. To avoid the need for recursive function calls, a special stack is implemented to manage the recursive nature of the ray tracing algorithm.

The MMX/SSE extensions (SIMD level of parallelism) are not explicitly used in the traversal, although it would lead to quite significant performance improvement (see 4.3). Even though the rays are not traced in packets, the program still gains some performance improvement by doing the mathematical computation of the intersection test in parallel, using MMX/SSE.

7.1.5 Ambient occlusion

Three different methods for evaluation of the directions to which the sample rays are cast were tested. First, I used a regular grid, because randomly distributed samples do not seem as a suitable method for interactive scene viewing (the randomness creates noise which alters every frame, creating a strange shivering effects in the occluded areas). The regular sampling method did not give satisfactory results, creating visual artifacts. Random algorithm did not give good result either, as expected, especially for small numbers of samples.

The algorithm I used in the final version of my project is what I call “rotated grid”’ algorithm. It is based on the regular sampling method, but the distribution of the rays is then slightly altered in a way that is unique for each point being sampled. This solution suppressed the artifacts introduced in the regular sampling method completely, and created much less noise than the random algorithm. Even for as low as 9 samples per point the noise is almost unnoticeable, and when combined with anti-aliasing, even lower number of samples yields satisfactory results. See figure 8.12 for comparison of different sampling methods.

7.1.6 Bloom

As the image is rendered in high dynamic range, the computation of the bloom is fairly simply. The image is convolved with the kernel of a gaussian filter - producing gaussian blur. When it is tone mapped to low dynamic range, the light appears to bleed from very bright objects on the surrounding areas.

The size of the kernel of the gaussian filter can be set via a configuration file or through command line arguments, giving the user control over the strength of the bloom.

7.1.7 Anti-aliasing

The type of supersampling I decided to use is known as the grid algorithm. It is the simplest and fastest sampling method, where the samples are distributed regularly. The disadvantage is that the regular nature of this sampling method might prevent it from removing some of the aliasing artifacts, which would probably be removed with an irregular algorithm. On the other hand, it will remove most of the artifacts and apart from being fast and easy to implement, it enables us to render the scene in just the same way as if no anti-aliasing was used, and do the filtering in a postprocessing step. In cases when the anti-aliasing is disabled, the conditional branches would still cause a little overhead in the code that gets executed millions of times every frame, so it is an advantage to move this code to where it is executed only if it needs to be.

7.1.8 Environment mapping

The environment map is implemented as a cube map - the environment is mapped on a cube of infinite size. Because the camera moves only finite distances, it will be always positioned in the centre of this cube. Thus, the relative coordinates of an intersection of a ray and this cube are independent on the size of the cube. To simplify the computation, we will assume that the cube has unit size and is positioned at the centre of the coordinate system. From a direction of a ray, the relative coordinates of the intersection point can be derived analytically.

The equations obtained from the above assumptions are implemented in the cube map class, so that when a ray does not hit any of the objects in the scene, it is passed to the cube map object to evaluate the light coming from the environment.

Because the resolution of the environment map is finite, the ray does not always hit the centre of a pixel - it has to be interpolated. Depending on the preferences set in the configuration file, either bilinear or nearest-neighbour interpolation method is used.

7.1.9 Fog

Two types of fog are implemented in the proposed ray tracer - homogenous and exponential.

The homogenous fog assumes the particles causing the light absorption, emission and scattering (dust, smoke, drops of water, etc.) are distributed uniformly throughout the entire scene. Thus, the attenuation of the light depends only on the distance that the light travel through the environment.

The exponential fog takes into account the effects of gravity, causing higher density of the particles at lower altitudes. The attenuation factor is no longer constant along the ray, leading to a more complex equation describing the light attenuation, emission and scattering.

A general equation describing the absorption of the light as a function of absorptive particles density is given in section 5.5, as is its simplified form for homogenous distribution. The emission and scattering of light is described analogously and simplified equations for special cases can be derived from the equation 5.5.

Note that as the environment map is placed at an infinite distance from the camera, the ray propagation through either the homogenous or the exponential fog will prevent it from being viewed and the colour of distant objects will converge to the colour of the fog.

7.1.10 Construction of kD-trees

Because the proposed ray tracer is targeted primarily to static scenes, the construction of kd-trees can be done offline. Thus, the construction time is not much of an issue; the effectiveness of the resulting data structure for traversal is important. As discussed in section 2.2, using a heuristic method for the kd-tree construction can improve the traversal performance significantly. I used the SAH method to construct effective kd-trees which are then stored in a binary file. Thus, we do not need to recompute them every time the ray tracer is started.

7.2 User control

The previous section outlines the implementation of methods for generation of the image. In this section, I will focus on the ways in which the interactive mode can by controlled by the user.

7.2.1 Collision system

One can take the advantage of having implemented the ray traversal through the acceleration structure and use this methods not only for ray tracing, but for the collision detection as well.

To find the altitude of the terrain at the coordinates of the player, a ray is cast downwards from the camera, in the direction of the gravity acceleration. The distance from the camera to the nearest intersection is the height of the camera above the surface.

To prevent the player from penetrating the surface of solid objects, a ray (or more precisely a line segment), starting at the current camera position and ending at the position the camera should be in the next frame if no collision occurred, is traversed through the acceleration structure and if no intersection is found, the player can do the movement (no surface penetration will occur).

The information obtained in the previous two steps can be used to restrict the player in a way that can be found in most game engines, so that the player can walk on the surfaces of the solid objects, jump or free fall affected by the gravity, and lots more.

Two modes of camera control are implemented in the proposed interactive ray tracer:

Viewing mode - the camera can fly freely through the scene, not restricted by the geometry.

Walk-through mode - the collisions are detected and basic physical laws are applied to restrict the movements of the player.

7.2.2 Progressive rendering

When the camera stays static and the resulting image would look the same as the previous frame (exposure time is adapted to the illumination level etc.), the ray tracing server signals idle to the user interfaces (a special flag in the shared memory is set). The GUI responds with a request to export the current camera and scene states, followed by a call to the offline renderer, which renders the view in higher visual quality (anti-aliasing, bloom, eventually ambient occlusion, higher resolution, etc.). When the rendering is done, the GUI displays the resulting image. If the process is interrupted (user moves the camera, etc.), the offline renderer is stopped immediately, as the frame being rendered is already out-dated.

This feature can be disabled either on the UI side (RTS signals idle, but the UI ignores it - other UIs might still use it, and processing time is not wasted rendering the same frame again and again), or on the RTS side (UI send a request to disable the idle mode - RTS never stops rendering, until the idle mode is enabled again).

7.3 Class diagram

The diagram 7.1 illustrates the modularity and dependencies of both RTS and the Offline Renderer. The program is divided into several function blocks; the core of the ray tracer, which is responsible for the most essential tasks, and optional modules, which can be enabled to extend the functionality of the ray tracer.

Outline of the function blocks

Core: I/O, scene representation, static camera, ray traversal, pixel colour evaluation
Interactive: dynamic camera, scene update, user control, collision system
Inter-Process Communication: shared memory, message queue, image buffer management
Non-Essential: environment map, fog

7.4 Code portability

All parts of the application are coded in C++ and are platform independent to the extended supported by the 3rd party libraries used in this project. The project was tested on several versions of MS Windows and Linux systems, both 32b and 64b.

Table 7.1 summarizes the libraries employed in the project, their purpose and the parts of the project dependent on them.

Name	Description	Usage
Boost	IPC - shared memory, message queue	RTS, UI
ASSIMP	Import of 3D models	BSP Builder
OpenMP	Multiprocessing API	RTS, Offline Renderer - can be disabled
CImg	Image loading/saving	RTS, Offline Renderer
ImageMagick	Support for advanced image formats	RTS, Offline Renderer - non-essential
Qt	GUI framework	GUI

Table 7.1: 3rd party libraries

7.5 Program flow

The diagram 7.2 illustrates the program flow of the Ray Tracing Server. Once the scene is loaded and server initialized, the program enters an infinite loop, where it serves user requests. It updates the scene and either renders the scene or, if the outcome would be the same as the previous frame and the idle mode is enabled, it raises the idle flag and sleeps for a brief moment. This loop is repeated until an exit signal is sent (e.g. via GUI or CUI, see Chapter 6 for more details on inter-process communication).

7.6 Summary of features

- Simulation of optical effects including:
  - shadows,
  - light reflection,
  - Phong shading,
  - bloom effect,
  - ray propagation.
- Environment mapping.
- Ambient occlusion.
- Anti-aliasing.
- HDR rendering; tone mapping.
- KD-tree construction using heuristic methods.
- GUI and/or console control.
- Own file format for efficient storing of scenes (pre-computed kd-tree, geometry, lights, etc.).
- Real-time performance.
- Collision system.
- MIMD level of parallelism.
- Portability to all major operating systems.

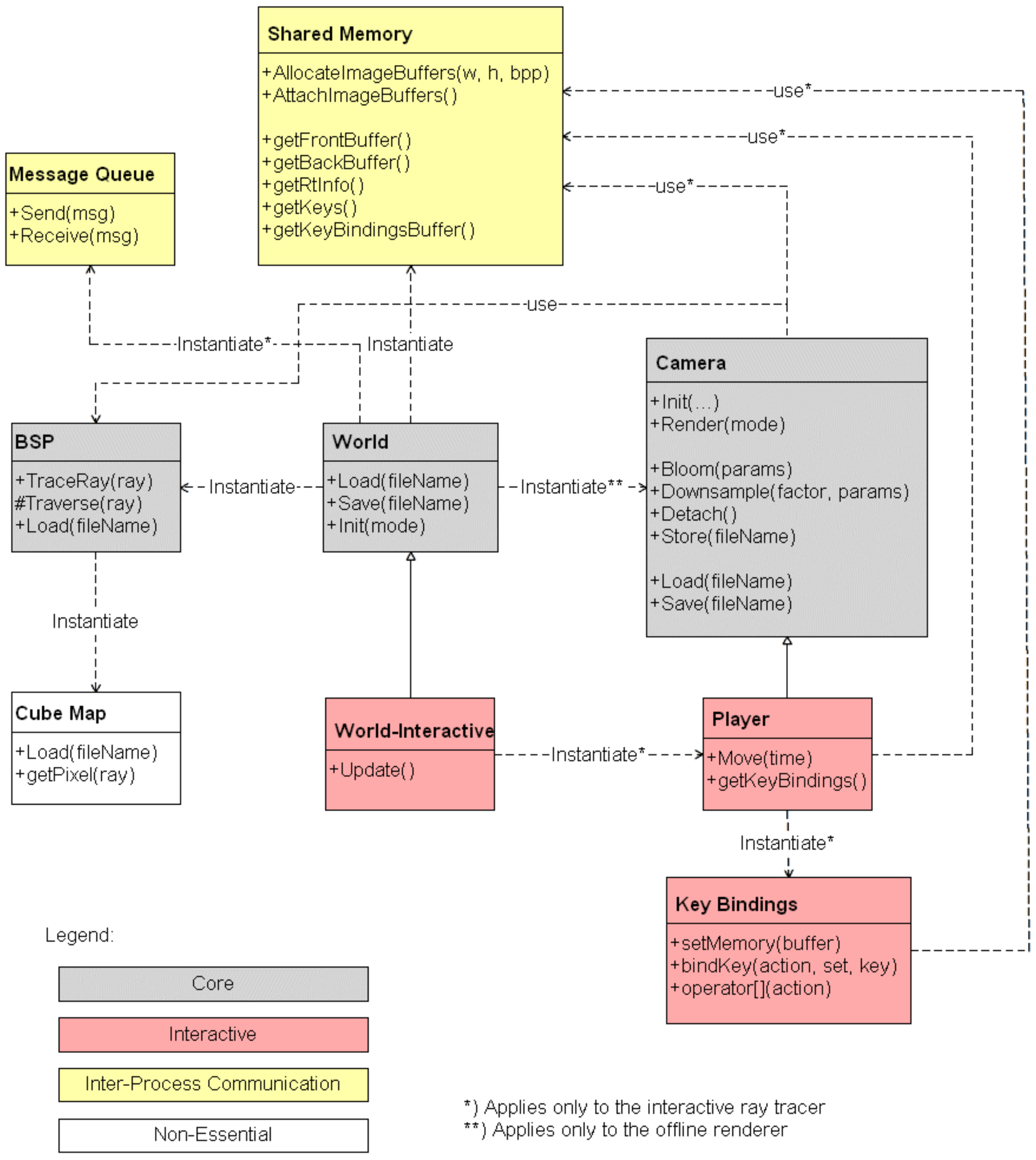


Figure 7.1: Class diagram of the ray tracer.

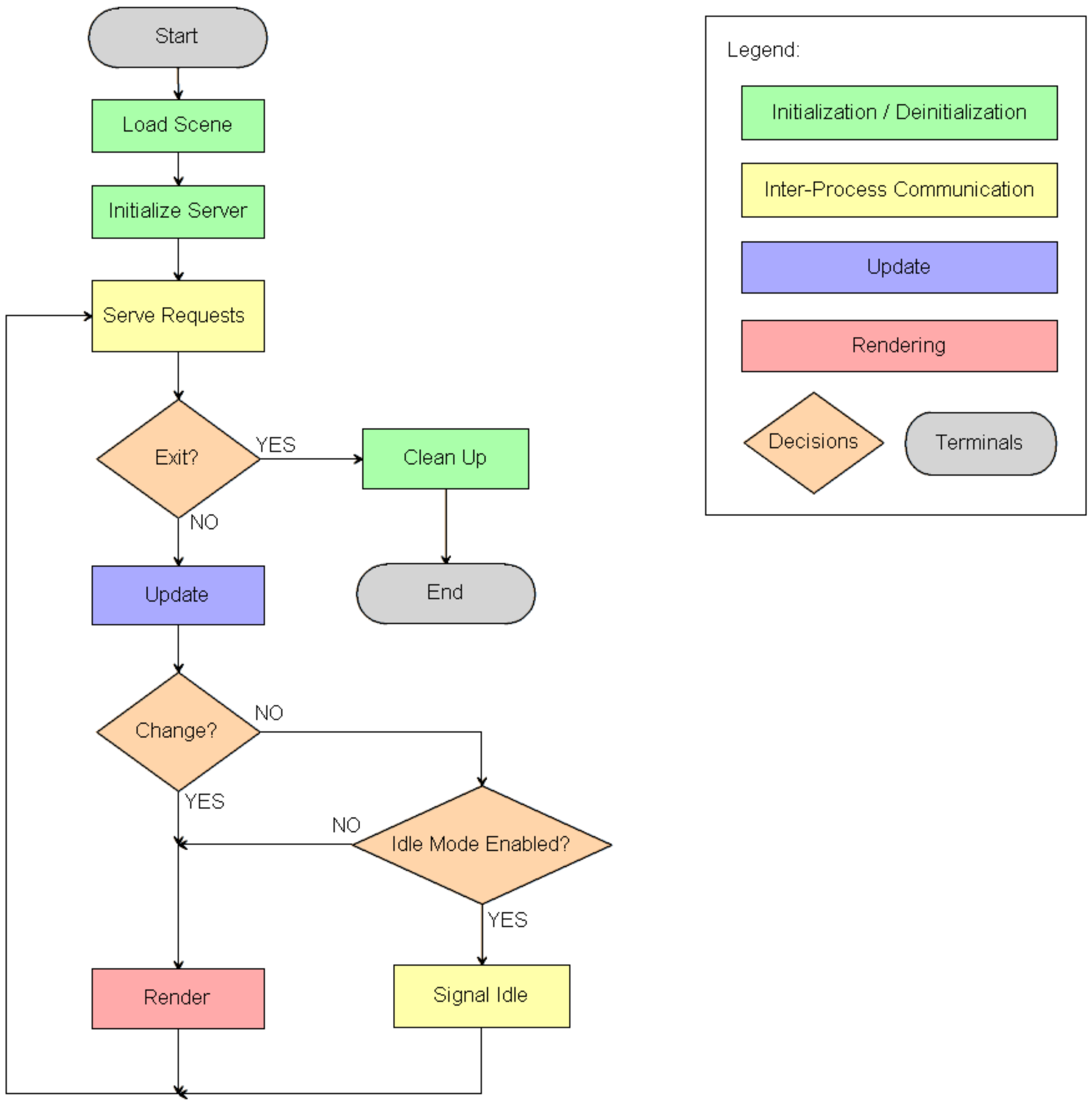


Figure 7.2: Flowchart of the Ray Tracing Server.

Chapter 8

Results

In this chapter we present results of the performance tests, and several sample images rendered with the proposed ray tracing algorithm.

8.1 Performance

In this section, we present a summary of the results of some performance test done on the presented ray tracer.

Note that the testing was performed on a standard personal computer (see Table 8.2). The tests can be easily reproduced by running the script $\%PATH\%/scripts/performance/run.sh$, which can be found on the attached DVD.

The results of the tests can be found in table 8.1, details about the hardware in table 8.2.

View	**Rendering time**	**FPS**	**Number of polygons**	**Scene**	**Resolution**
Looking at the girl	60 ms	16.7	156442	girl.obj	320 x 240
Inside the cathedral	100 ms	10	139225	cgr-c.3DS	320 x 240
Outside, looking at a mirror	130 ms	7.7	139225	cgr-c.3DS	320 x 240
In the centre of the scene	220 ms	4.5	281952	ladybirds_6.obj	320 x 240

Table 8.1: Performance test results.

Processor	Intel Core 2 Duo
Frequency	1.6 GHz
Main memory	2 GB, DDR2
Cache L1	2x32 KB, 8-way set associative
Cache L2	2 MB, 8-way set associative
OS	Linux Debian, 64 bit

Table 8.2: Hardware the tests were performed on.

8.2 Images

In this section, we present some images generated with the ray tracer described in previous sections. The most interesting points to notice are pointed out below each picture.

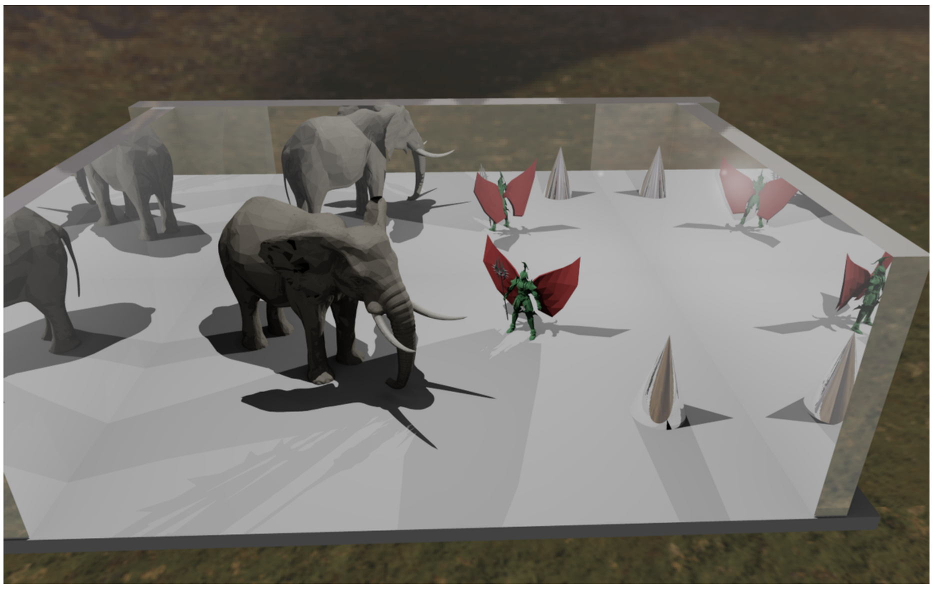


Figure 8.1: Rendered image of an elephant and a knight surrounded with mirrors.

This image demonstrates the ability of the ray tracer to render point-light shadows and light reflection. Notice the clear reflection of the scene in the mirrors and the deformed image on the surface of the cone, due to its convex shape.

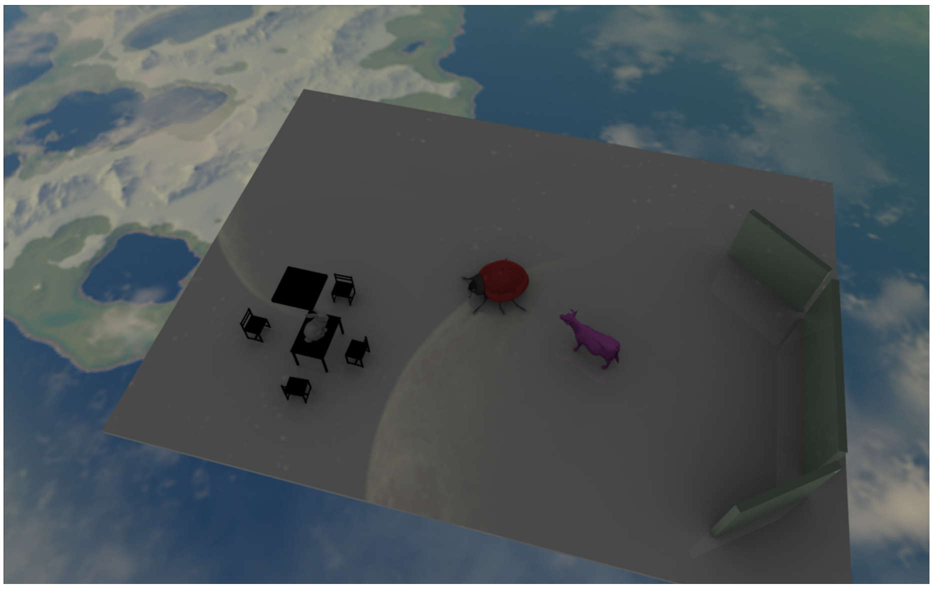


Figure 8.2: Environment map reflecting in the scene geometry.

Notice the dark corners at the bottom of the green wall and the ambient shadows under the table and chairs. Also, the reflection of the environment map and the geometry is clearly visible.

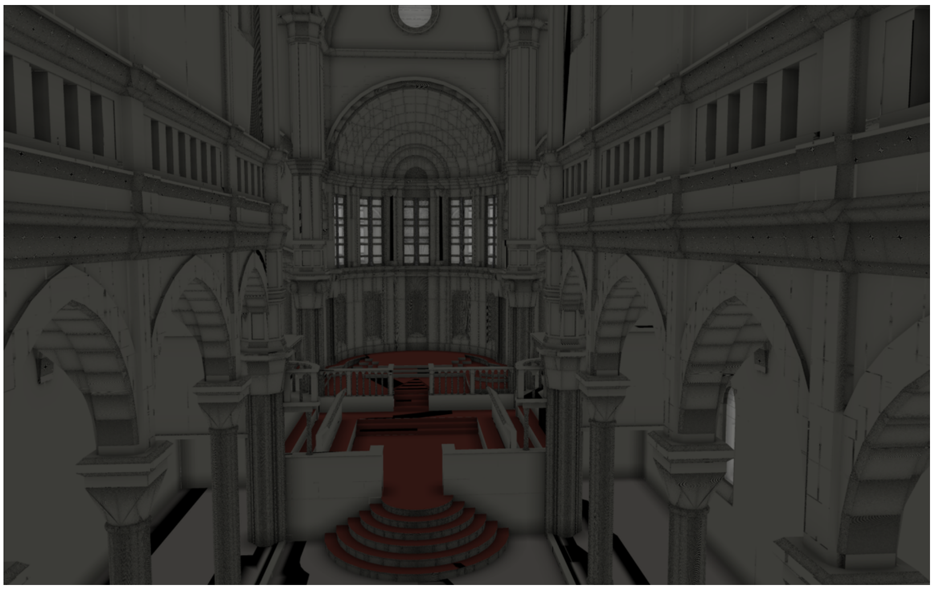


Figure 8.3: Interior of the cathedral in Sibenik, Croatia.

This is an interior view of the cathedral in Sibenik, rendered without direct illumination. What you might observe here is the effect of ambient occlusion. Notice that without it, the walls would all have the same tone of gray colour and the relief would not be recognizable at all. Give a closer look to the stairs, to see the dark corners of the wall and stairs.

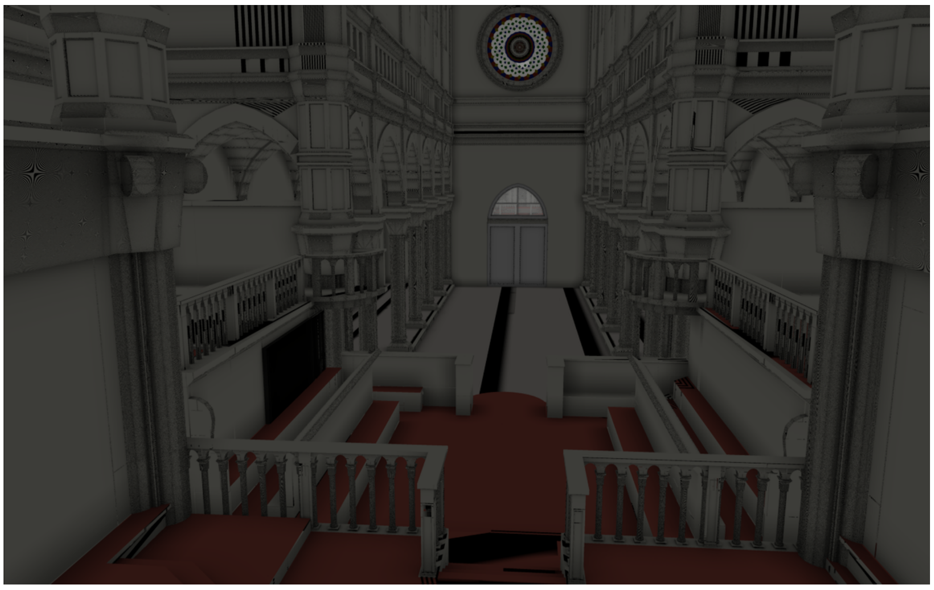


Figure 8.4: Interior of the cathedral in Sibenik, Croatia.

Another view of the cathedral, with no direct illumination. Examine the red carpet near on the floor to see the smooth shading. You might also notice the reflection of the scene in the glass window above the door.

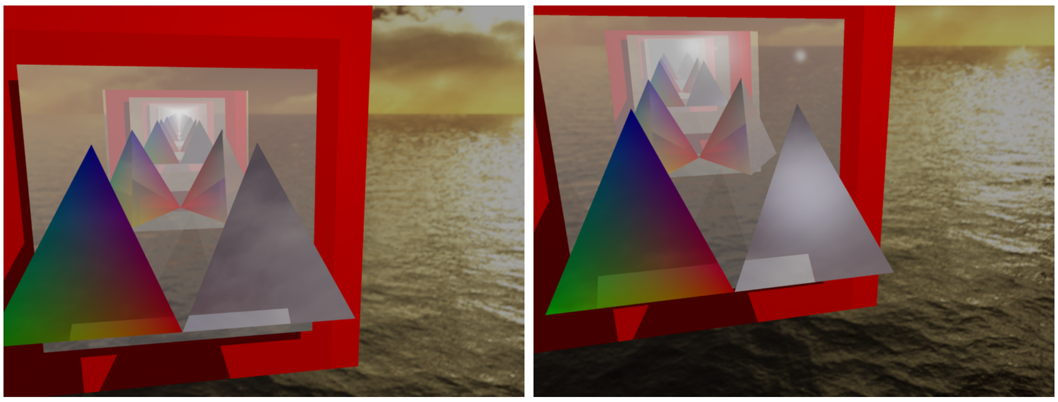


Figure 8.5: Scene with two opposite mirrors demonstrating the recursive nature of the ray tracing algorithm.

This image demonstrates recursive light reflection. The scene contained two mirrors with the camera placed in-between. In the picture on the right, the maximum depth of recursion was set to 5, while the picture on the left was generated with much higher value. You might also observe shadows from multiple light sources, colour interpolation, environment mapping, etc. Anti-aliasing was used to produce the smooth edges.

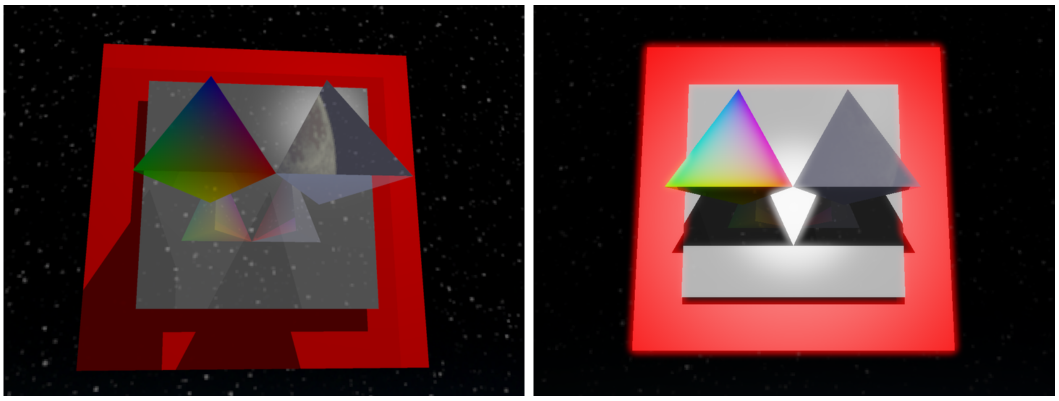


Figure 8.6: Render of a scene producing the effect of light scattering in the lens, known as a bloom.

The image on the left demonstrates per vertex colours. Notice the smooth colour interpolation between the red, green, and blue corners. The image on the right shows the bloom effect. Notice the blurred corners and colour bleeding into surrounding objects. Also, note that the light sources in this scene were 100 times more intense than those in the scene on the left. It is due to the tone mapping that all details and colours are preserved. You might also notice that the stars visible in the background of the image on the left are not visible in the image on the right. This is because the much higher intensity of light reflected off the geometry made the exposure time to adapt to this level of illumination, making the less bright objects barely noticeable - just as if you were watching starts during a sunny day.

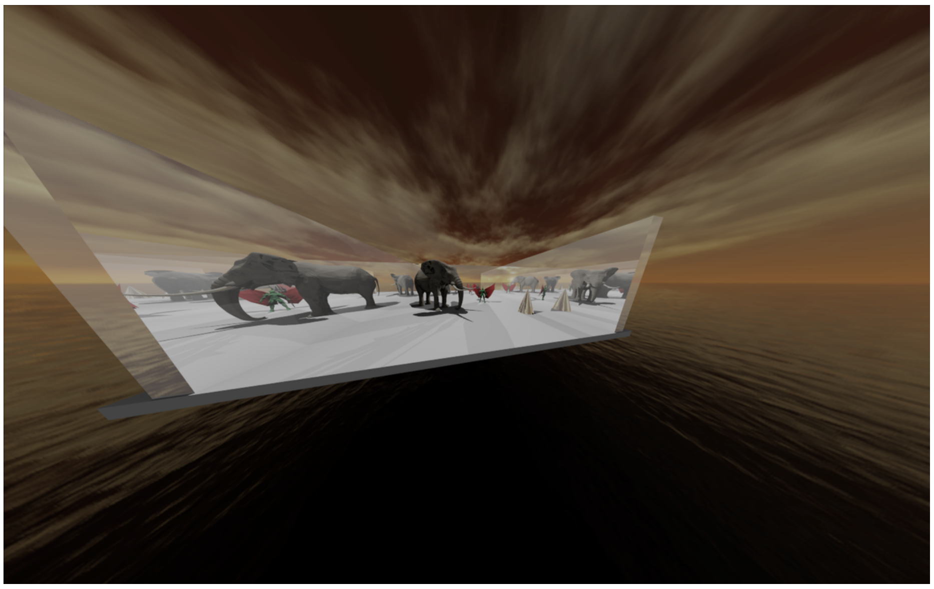


Figure 8.7: Fish eye view (FOV 150 degrees, rectangular).

This image was rendered with a very wide field of view (150 deg).

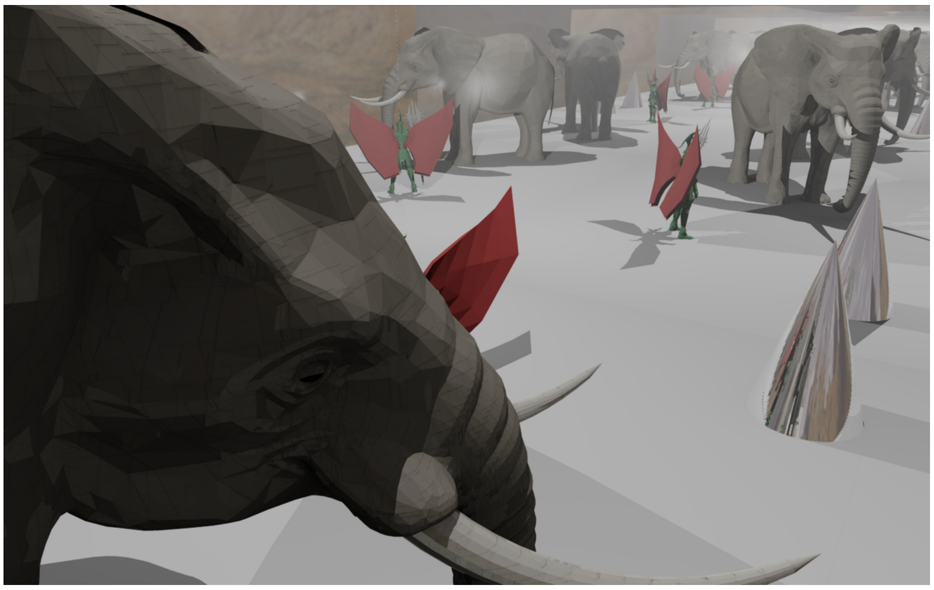


Figure 8.8: Elephant close up.

Close up of an elephant surrounded with mirrors. Notice the realistic reflection of the models in the mirrors.

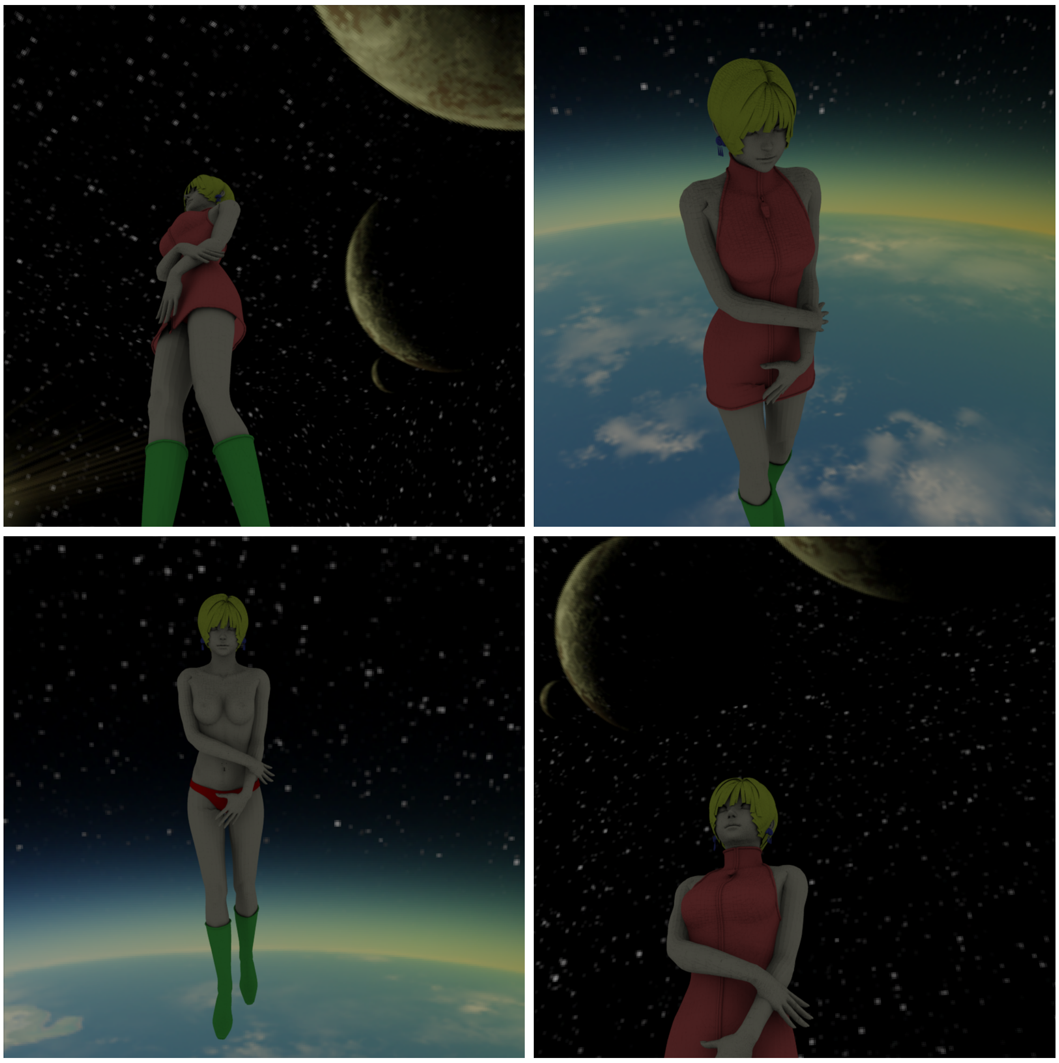


Figure 8.9: Picture of a girl in a red dress, levitating on the earth's orbit.

This set of pictures was rendered without any source of direct illumination. All shading is based on the ambient occlusion technique. Examine the occluded areas to see the ambient shadows and the shading.

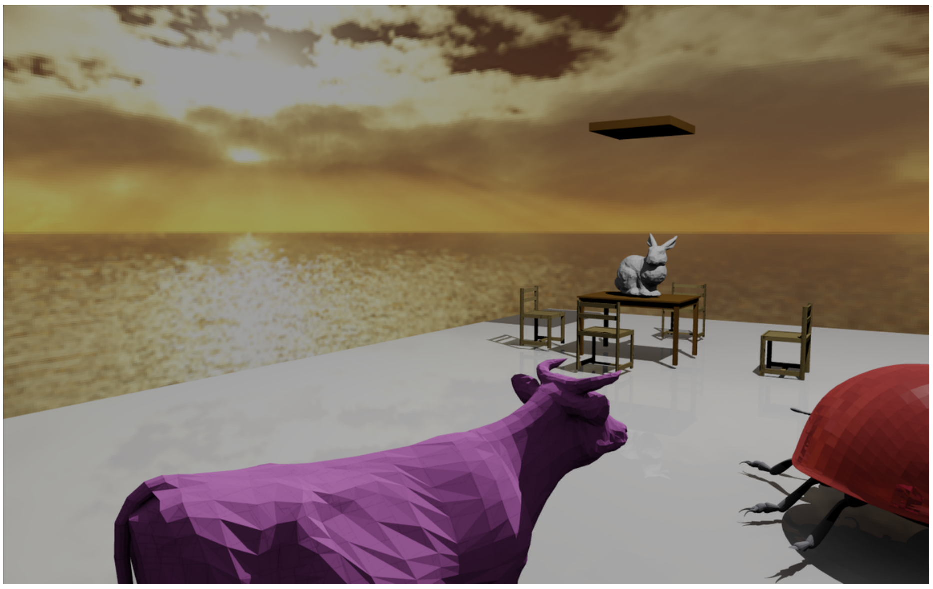


Figure 8.10: Picture of a cow watching the sunset.

On this image you might observe point-light shadows, Phong shading, environment mapping and after careful examination also light reflection and ambient shadows might be noticeable.

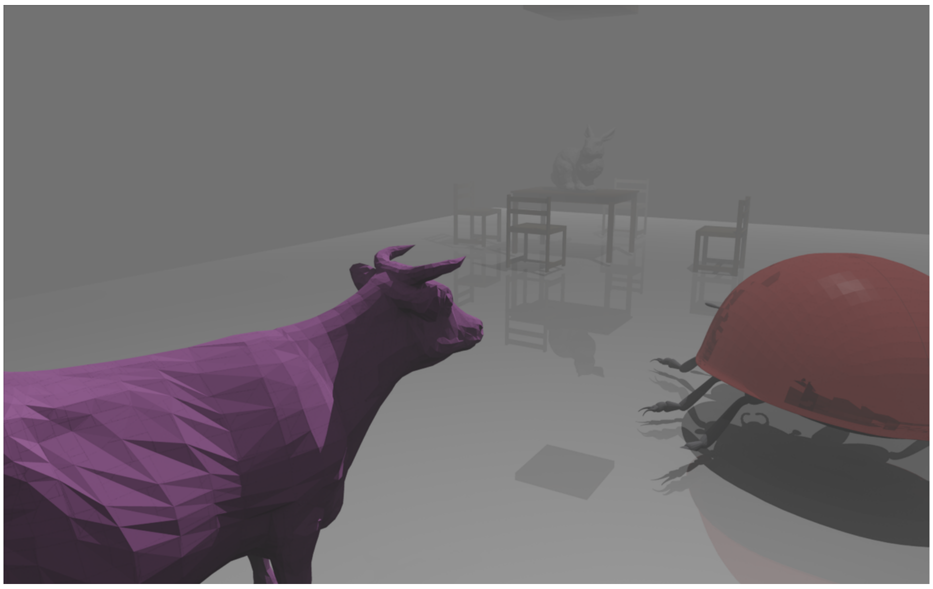


Figure 8.11: Image of a scene in a fogy environment.

This image demonstrates the effects of light propagation through absorptive environments. In this setting, it should reproduce the look and feel of scene placed in a fogy environment.

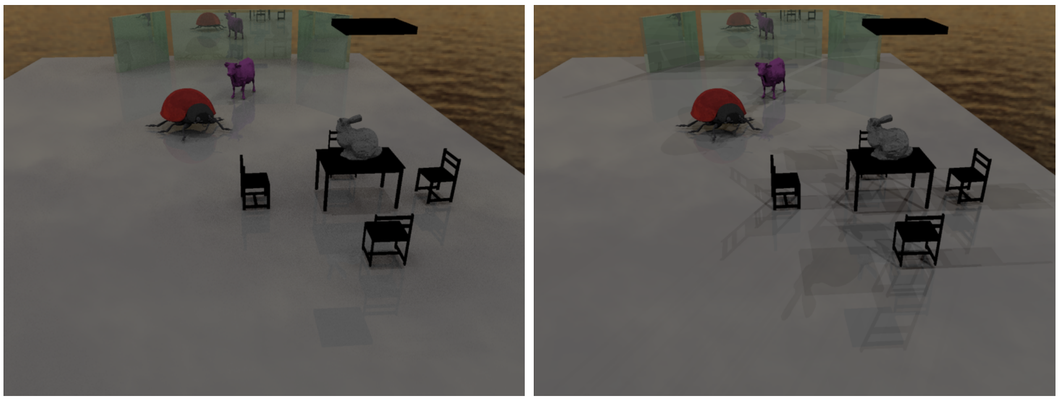


Figure 8.12: Comparison of two sampling methods used for the computation of ambient occlusion.

The two images were renderer with exactly the same camera settings; they only differ in the sampling algorithm used for the ambient occlusion computation. The image on the right shows the artifacts caused by regular sampling. In the one on the left, some noise is noticeable, but considering only 7 visibility rays were shoot per point, the noise is quite minimal.

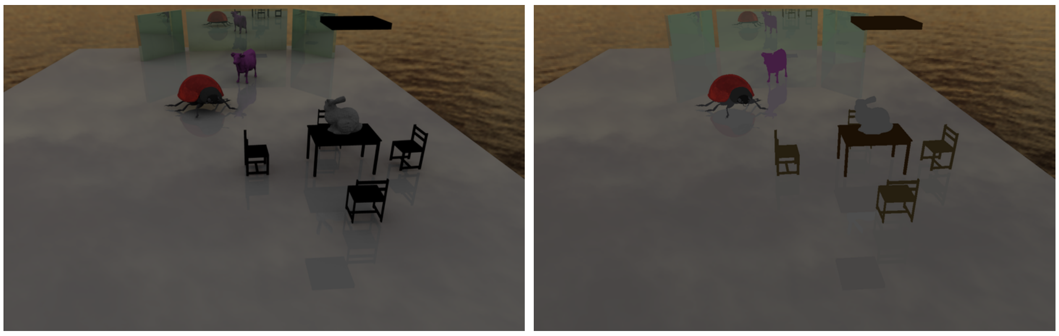


Figure 8.13: Comparison of images rendered with and without ambient occlusion.

These two images compare the results of using ambient occlusion and not using it. You might notice that the image on the right (no ambient occlusion) looks very flat. That is because no light source was present to shade the geometry. Even though the image on the left was rendered with exactly the same settings, including lighting conditions, it does not look so flat at all. Notice the dark corners, shading of the rabbit and ambient shadows under the ladybird, cow and chairs. Combined with the reflection, it gives interesting look to the scene even without lighting.

Chapter 9

Conclusion

We have proposed and implemented a program generating an image of a 3D scene using the ray tracing algorithm. The data is represented in memory in the form of kd-tree acceleration structure. The computation of pixel intensities is carried out in floating point numbers to be then tone-mapped to a range displayable on a standard physical device.

Two applicationaly different areas are targeted: interactive (real-time) and photorealistic (offline) rendering.

The interactive mode offers camera export and movement recording, the camera can be controlled using standard input devices. Two modes of camera control are supported: a viewing mode (no interaction with the scene geometry) and a walk-through mode (movement restricted by the scene geometry and gravitational force).

The photorealistic mode offers rendering of the scene in higher visual quality. Optical effects including ambient occlusion (approximation of global illumination) and bloom are simulated. The aliasing artifacts are suppressed using supersampling.

The two rendering modes described above can cooperate to offer what is called progressive rendering, or a view or video sequence can be exported in the interactive mode to be rendered offline.

The interactive ray tracer is designed as a server, communicating with the user interface using IPC techniques. The offline renderer and the program for construction of kd-trees are also designed as stand-alone applications, so that any part can be changed independently.

The algorithm is implemented in the C++ programming language, using Boost C++ libraries for platform independent implementation of inter-process communication. To support a wide range of 3D scene formats, ASSIMP library is used for the import of this data, which are then stored in a specialised format incorporating the constructed kd-tree. The graphical user interface is implemented in the Qt framework. The MIMD level of parallelism is achieved using OpenMP multiprocessing API. The resulting program is platform independent to the extend of these libraries, and was successfully tested on several versions of Linux and MS Windows.

During performance testing, real-time frame rates have been achieved, although only in a limited resolution.

9.1 Further work

At any state of a software development, there are always things to improve. Here are some suggestions for further work on this project.

To improve the visual quality of generated images, a proper global illumination technique could be implemented (currently, this is approximated with the ambient occlusion).

To extend the client-server design, a network interface could be implemented to accept commands through a socket, passing them to the ray tracing server, and streaming the video output back to the client.

Bibliography

[1] I. Wald, W.R. Mark, J. Gunther, S. Boulos, T. Ize, W. Hunt, S.G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. *Eurographics 2007 State of the Art Reports*, pages 89–116, 2007.

[2] K. Suffern. *Ray tracing from the ground up*. Mirage, 2007.

[3] Meinrad Recheis. Realtime Ray Tracing, 2008. Vienna University of Technology.

[4] A. Chalmers, T. Davis, and E. Reinhard. *Practical parallel rendering*. AK Peters, Ltd., 2002.

[5] Wikipedia contributors. Wikipedia - The Free Encyclopedia.
http://en.wikipedia.org/, 2010.

[6] V. Havran. Heuristic ray shooting algorithms. 2000. Construction of Kd-Trees.

[7] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O (N log N). In *IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, 2006.

[8] M. De Berg, M.D. Berg, O. Schwartskopf, M. Overmars, and M. Van Kreveld. *Computational geometry*. Springer Berlin, Germany, 2000.

[9] Erik Reinhard. Lectures on Computer Graphics, 2009.
Info: http://www.cs.bris.ac.uk/Teaching/, University of Bristol.

[10] E. Reinhard. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2006.

[11] Erik Reinhard and Garrett Johnson. Color, 2008. University of Bristol.

[12] Erik Reinhard et al. Image display algorithms for high and low dynamic range display devices, 2007. University of Bristol.

[13] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21(3):267–276, 2002.

[14] M. Pharr and G. Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2004.

[15] À. Méndez-Feliu and M. Sbert. From obscurances to ambient occlusion: A survey. *The Visual Computer*, 25(2):181–196, 2009.

[16] Gregory Massal. A raytracer in C++ - Part IV - Depth of field, Fresnel and blobs [online].
`http://www.codermind.com/articles/Raytracer-in-C+`
`+-Depth-of-field-Fresnel-blobs.html/`, 2008.

[17] I. Wald, A. Dietrich, C. Benthin, A. Efremov, T. Dahmen, J. Gunther, V. Havran, H.P. Seidel, and P. Slusallek. Applying ray tracing for virtual reality and industrial design. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 177–185, 2006.

[18] S. Boulos, I. Wald, and P. Shirley. Geometric and arithmetic culling methods for entire ray packets, 2006.

[19] V. Havran and J. Bittner. Stackless Ray Traversal for kD-Trees with Sparse Boxes. *Computer Graphics and Geometry*, 9(3):16–30, 2007.

Appendix A

Manual

A.1 Tutorial

In this section, I will explain the basics of working with the application in an easy-to-read style. A more thorough description of individual functions can be found in the subsequent sections.

The ray tracing server can be started either from the console,

```
RTServer scene.scn
```

or from the GUI (see figure A.3).

```
"Menu->Connection->Start new server" or Ctrl+S
```

If the server is up and running, we can proceed to connecting to it.

```
"Menu->Connection->Connect" or Ctrl+C
```

If the connection was successful (the server is running and the shared memory and message queue were successfully opened), the output of the renderer should appear on the screen (figure A.1).

Now, we can start controlling the camera using the keyboard (see A.3.2).

The camera can be exported (figure A.2)

```
"Menu->Action->Export camera"
```

to be used in the offline renderer or as initial settings next time the server is started.

To record a video sequence, go to

```
"Menu->Action->Recording"
```

and click start. From now on the server records camera position and exposure time for every frame, storing it in the specified path. To stop the recording, simply go to the recording menu again and click stop (figure A.2).

Clicking on `"Menu->Action->Statistics"` will display current frame rate, camera resolution, etc. (figure A.4).

Defaultly, progressive rendering is enabled, meaning that when the camera becomes stationary, the view is rendered in higher visual quality. This feature can be disabled through

```
"Menu->Settings->Progressive Rendering" or Ctrl+H.
```

If it is disabled and you still want the particular view to be rendered and displayed in higher quality, simply click

```
"Menu->Action->Quick Render" or hit Ctrl+R.
```

To stop the server, click `"Menu->Commands->Stop Server"` or hit Ctrl+K.

A.2 Console

The executables that can be controlled from the console:

RTServer Ray Tracing Server

Renderer Offline Renderer

BSPBuilder kD-Tree construction

CUI Console User Interface

Calling any of them without any arguments will display a brief help.

A.2.1 Ray tracing server

This command starts a new instance of the ray tracing server. Once started, it can be controlled from the console using *CUI* (A.2.4).

Command: *RTServer*

```
Usage: <bin-name> <scene> [options]
Where:
<bin-name>name of the executable
<scene>    scene description file
[options]  optional arguments
```

A.2.2 Offline renderer

This command starts the offline renderer to render a particular view in the photorealistic mode. The quality of the generated image can be set via the configuration files (.cam, .set), or specified as command line arguments (the arguments override the settings in the configuration file).

Command: *Renderer*

```
Usage: <bin-name> <scene> [options]
Where:
<bin-name>name of the executable
<scene>    scene description file
[options]  optional arguments
 -o <path> output file
 -a <integer> anti-aliasing
 -b <integer> bloom effect
 -w <integer> image width (horizontal resolution)
 -h <integer> image height (vertical resolution)
 -l <path> video sequence (<path> = list of .cam files)
 -ao        enable ambient occlusion
 -nao       disable ambient occlusion
 -aon <integer> enable ambient occlusion (<int> = number of samples)
 -aok <float> enable ambient occlusion (<float> = sampling interval)
 -dbg <path>   export debug scene and exit
 -help      display this help and exit
```

A.2.3 BSP builder

This command constructs a kd-tree of a given scene and stores it in a specialised file. The output file is readable by the ray tracing server and the offline renderer.

Command: *BSPBuilder*

```
Usage: <bin-name> <input-file> <output-file> [options]
Where:
<bin-name>     name of the executable
<input-file>   input file name (.3DS, .x, .obj, ...)
<output-file>  output file name (.bsp)
[options]      optional arguments
 -f  fast build
 -v  verbose
 -vv very verbose
 -m <integer>kd-tree quality (applies only to fast build mode)
 -hist <integer>display histogram (<integer> = number of bands)
```

A.2.4 Console user interface

The console user interface passes commands from the console to the ray tracing server.

Command: *CUI*

```
Usage: <bin-name> <command> [options]
Supported commands:
 exit Tell the server to stop
 free Release shared memory
 fps  Print fps
 resolution <width> <height>Request change of resolution
 export <target>Export camera/video sequence/scene
 about  Display information about the program

Export options:
 export camera <path> <width> <height> <anti-aliasing> <bloom>
  <anti-aliasing> super sampling factor (integer)
  <bloom>         sigma (gaussian convolution coefficient)
 export video <start/stop> <path>
```

A.3 GUI

The graphical user interface is primarily targeted on operating the ray tracing server. How to use it is described in section A.1, in this section, a short description of the menu items and the key bindings is given.

A.3.1 Menu structure

File

- **Exit** Terminates the GUI application

Connection

- **Connect** Connects to a ray tracing server
- **Start new server** Starts a new ray tracing server

Action

- **Export Camera** Displays camera export dialog
- **Quick Render** Renders and displays current view in higher visual quality
- **Statistics** Displays statistics (FPS, resolution, ...)

Commands

- **Stop server** Stops the server to which the GUI is currently connected to
- **Pause server** Pauses the server (stops rendering but continues accepting requests)
- **Ping server** Ping the server

Display

- **Ignore Aspect Ration** If checked, the image is scaled according to the size of the window
- **Smooth Transformation** If checked and the window size does not match the size of the image, smooth filter is used

Settings

- **Progressive Rendering** If checked and the camera becomes stationary, the view is rendered in higher visual quality
- **Configure** Displays configuration dialog

Help

- **About** Displays information about the project

A.3.2 Key bindings

The codes of keys being pressed are stored in a block of shared memory, so that the ray tracing server can access and process it. The key bindings can be overridden from the GUI, which then informs the server about the new bindings. The default key bindings is listed below.

General:

WASDIO / **Arrows** Movement

CapsLock + **Arrows** Rotation

F Viewing/Walk-through mode

Numbers Lights on/off

M + Number Choose environment map

B Bloom effect on/off

Walk-through mode:

C Crouch

Ctrl Crawl

Space Jump

Shift Sprint

Shortcuts:

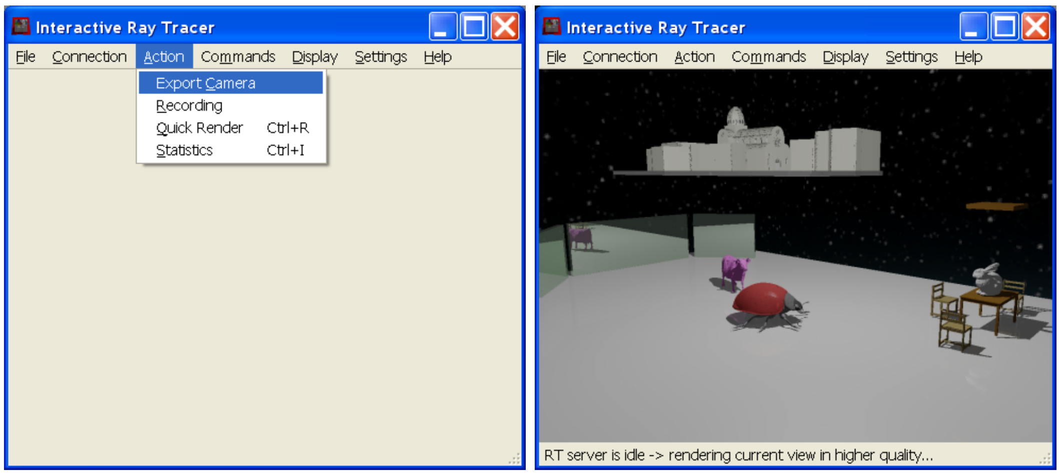
E Export current camera settings

H Export high quality camera settings

R Render current view in higher visual quality

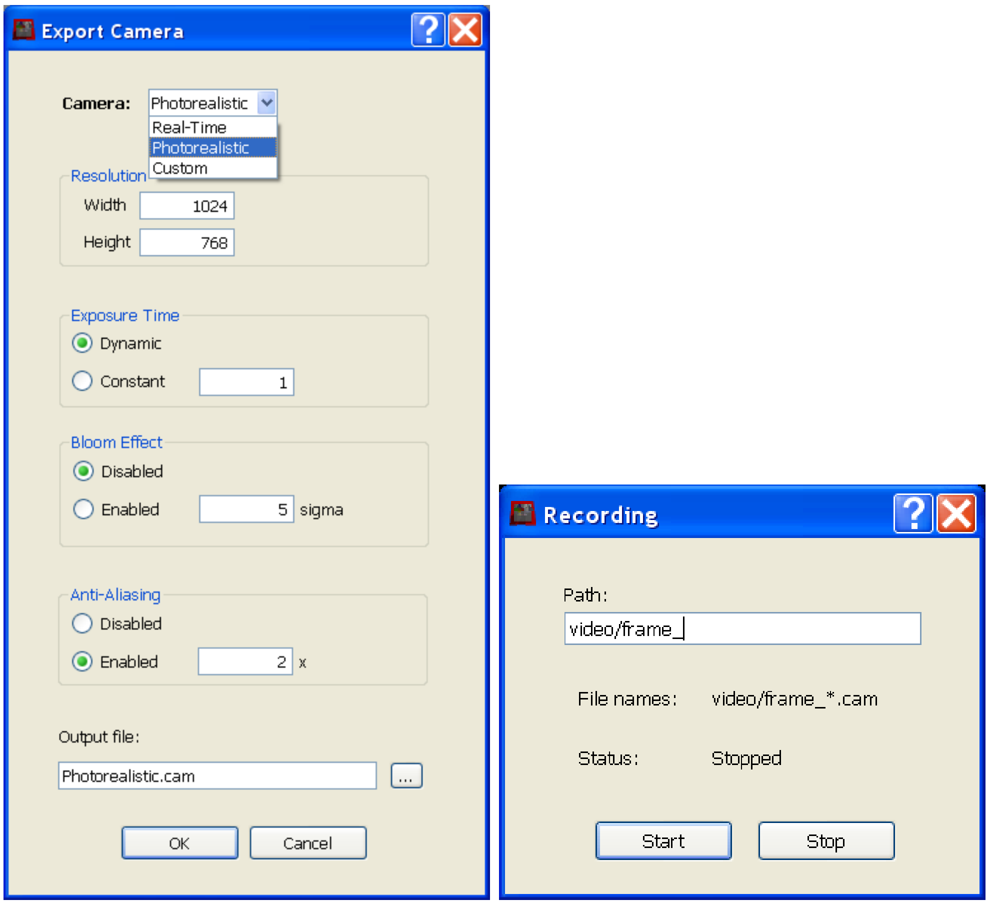
R+L Render current view in higher visual quality, enabling anti-aliasing

A.3.3 Screenshots


Interactive Ray Tracer
File Connection Action Commands Display Settings Help
Export Camera
Recording
Quick Render Ctrl+R
Statistics Ctrl+I
Interactive Ray Tracer
File Connection Action Commands Display Settings Help

RT server is idle -> rendering current view in higher quality...

Figure A.1: Main window


Export Camera

Camera: Photorealistic
- Real-Time
- Photorealistic
- Custom

Resolution
Width 1024
Height 768

Exposure Time
[✓ Dynamic]
Constant 1

Bloom Effect
[✓ Disabled]
Enabled 5 sigma

Anti-Aliasing
Disabled
[✓ Enabled] 2 x

Output file: Photorealistic.cam

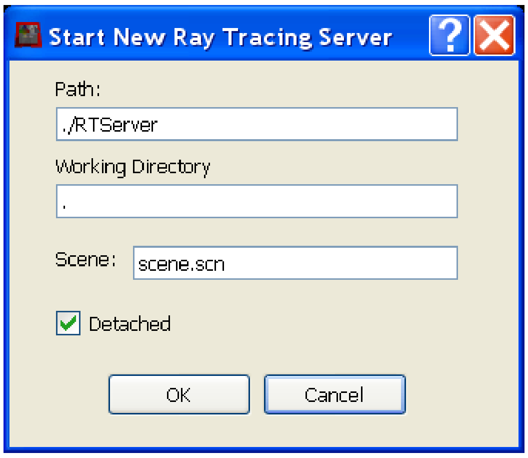
Recording

Path: video/frame_

File names: video/frame_*.cam

Status: Stopped

Figure A.2: “Export Camera” and “Recording” dialogs


Start New Ray Tracing Server

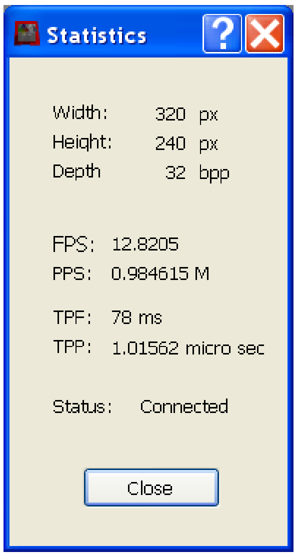
Path: ./RTServer

Working Directory .

Scene: scene.scn

[✓ Detached]

Figure A.3: “Start new server” dialog



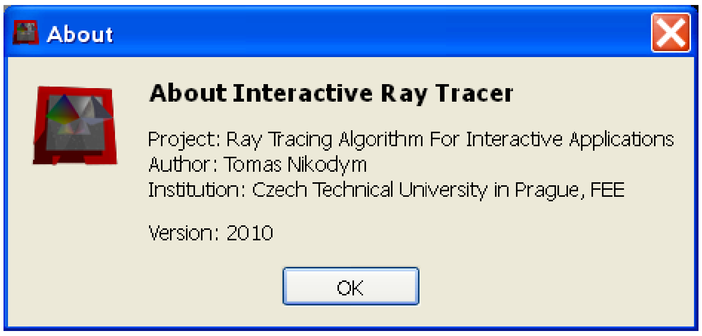
```
Width:      320  px
Height:     240  px
Depth        32  bpp

FPS:   12.8205
PPS:   0.984615 M

TPF:   78 ms
TPP:   1.01562 micro sec

Status:    Connected
```

Figure A.4: “Statistics” dialog



About Interactive Ray Tracer

Project: Ray Tracing Algorithm For Interactive Applications
Author: Tomas Nikodym
Institution: Czech Technical University in Prague, FEE

Version: 2010

Figure A.5: “About” dialog

Appendix B

Contents of the DVD

ConsoleUI	Console user interface (source code + project)
Data	
3DScenes	3Ds Max scene files
Environment Maps	Environment maps (png)
Icons	Application icons
Models	Models (3ds, obj)
Documentation	
diagrams	Diagrams (ddd)
doxygen	Doxygen settings
latex	Source code of this thesis in LaTeX
refman	Reference manual generated by doxygen
txt	Text files (README, etc.)
rtaia.pdf	This thesis (pdf)
Executable	
images	Images generated with the ray tracer
linux64	Application executables (64b Linux OS)
scripts	Bash and batch scripts (performance tests, sample images generation)
win32	Application executables (32b MS Windows OS)
GUI	
icons	GUI icon
src	Source code of the GUI
∗.pro	Project file (Qt Creator)
RayTracer	
include	Third party libraries (not including Boost and ASSIMP)
shared	Source code shared by the ray tracer and user interface (IPC)
src	Source code of the ray tracer (RTS, Renderer, BSPBuilder)
∗.cbp	Project files (Code::Blocks)
∗.workspace	Workspace files (Code::Blocks)

Appendix C

List of abbreviations

kD-Tree k-Dimensional Tree

BSP Binary Space Partitioning

SAH Surface Area Heuristics

AABB Axes Align Bounding Box

OBB Oriented Bounding Box

DOP Discrete Oriented Polytope

CSG Constructive Solid Geometry

HDR High Dynamic Range

LDR Low Dynamic Range

CPU Central Processing Unit

GPU Graphics Processing Unit

MIMD Multiple Instruction Multiple Data

SIMD Single Instruction Multiple Data

CUDA Compute Unified Device Architecture

OpenMP Open Multi-Processing API

DOF Depth Of Field

MPx Mega Pixel

DDR Double Data Rate

RAM Random Access Memory

SA Set Associative

FPS Frames Per Second

API Application Programming Interface

ASSIMP Open Asset Import Library

RTS Ray Tracing Server

UI User Interface

GUI Graphical User Interface

IPC Inter-process Communication