UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Fluka Raytracer

Proyecto Final de Carrera

Ingeniería informatica

Autor: David Siñuela Pastor

Director: Francisco Javier Jaén-Martínez

February, 2013

Contents

Acknowledgements

I would like to acknowledge all the people involved in the development of this master thesis: my supervisor at CERN, Vasilis Vlachoudis, who has taught me an uncontable number of things, my supervisor at the UPV university, Francisco Javier Jaen-Martinez, for his direction and patience and Regina Kwee for her insightful revisions whithout which this document would have never been finished.

I would like to express my gratitude to my colleagues and friends who always were there during the never ending process of writing this document.

Last but not less important I would like to thank my parents Jose Antonio and Mari Carmen, my sister Lorena and my girlfriend Paula for their endless support on everything I do.

1. Introduction

The project described in this document was developed as the work for a technical student position at CERN in Geneva, Switzerland. CERN is the European Organization for Nuclear Research and is one of the leading physics laboratories across the world.

The development of the current project was held by the section EN-STI-EET (Engineering; Sources Targets and Interactions; Emerging Energy Technologies), which in collaboration with INFN Institute, Italy [8] is in charge of developing and maintaining the Monte Carlo generator FLUKA [2].

FLUKA is a general purpose tool for calculations of particle transport and interactions with matter, covering an extended range of applications spanning from proton and electron accelerator shielding to target design, calorimetry, activation, dosimetry, detector design, Accelerator Driven Systems, cosmic rays, neutrino physics, radiotherapy etc.

1.1. CERN

CERN, the European Organization for Nuclear Research, is one of the world's largest and most respected centers for scientific research. It is fundamentally focused on physics, finding out what the Universe is made of and how it works. At CERN, the world's largest and most complex scientific instruments are used to study the basic constituents of matter: the fundamental particles. By studying what happens when these particles collide, physicists learn about the laws of Nature.

The instruments used at CERN are particle accelerators and detectors. Accelerators boost beams of particles to high energies before they are made to collide with each other or with stationary targets. Detectors observe and record the results of these collisions. The analysis of those records leads to the confirmation or refutal of physical theories.

Founded in 1954, the CERN Laboratory is located at the Franco-Swiss border near Geneva. It was one of Europe's first joint ventures and now has 20 Member States.

1.2. FLUKA

Fluka simulates the interaction of a beam of particles with matter, it takes particles sampled from a beam definition and transports them through the geometry. The transport

of the beam throught the matter that composes the geometry produces several kind of interactions and secondary particles.

Fluka receives a description of the geometry along with their material properties, a beam definition with the type of particles, distribution type, energy and the physics settings desired for the simulation.

There are two main beam-matter interaction simulators used at CERN: FLUKA [9] and GEANT4 [7]. These two Monte Carlo codes are developed by two collaborations involving people from several countries across the world. Both codes can be used for the same purposes but each takes a different approach, GEANT is meant to be a toolkit where physicists can plug in the physical processes they are interested into while Fluka is meant to be a fully integrated package ready to be used.

1.3. Goals

Fluka is configured from the Fluka input file: a plain text file that contains all the directives needed to configure a simulation scenario. It includes also the description of the geometry in terms of bodies and composition operations. This file is written by the person (usually a physicist) who needs to elaborate a new study. Preparing the input file is usually a complex task especially when geometries grow in size.

The purpose of the raytracer is to help visualizing the geometry given as input to Fluka by generating photo-realistic three dimensional images. Before the development of the Fluka raytracer any kind of representation of the Fluka geometry involved exporting the geometry to a format understandable by a external tool and using that tool independently. The aim is also to shorten the feedback cycle when building a geometry, helping in the development or debugging of geometries.

1.4. Document structure

There is a deeper introduction to Fluka in the next chapter (chapter 2) where the purpose and design of Fluka is discussed. Chapter 3 describes the analysis phase of the project, illustrating why the raytracer approach was chosen. It follows, in chapter 4, an explanation of the computer graphics concepts needed to develop the rest of the thesis. In chapter 5, the design of the raytracer is discussed and follows, in chapter 6, an in-depth description of the implementation. Chapter 7 includes a detailed study of the performance of the raytracer along with some examples. A parallelization proposal is elaborated in chapter 8, and to finalize the conclusions are in chapter 9.

There are 2 appendices. Appendix A.1 includes the list of the proposed additions to the Fluka input file. Appendix A.2 contains the fortran data definitions used in the code.

2. Fluka

FLUKA is a fully integrated particle physics Monte Carlo simulation package. It has many applications in high energy experimental physics and engineering, shielding, detector and telescope design, cosmic ray studies, dosimetry, medical physics and radio-biology.

The highest priority in the design and development of FLUKA has always been the implementation and improvement of sound and modern physical models. Microscopic models are adopted whenever possible, consistency among all the reaction steps and/or reaction types is ensured, conservation laws are enforced at each step and results are checked against experimental data at single interaction level. As a result, final predictions are obtained with a minimal set of free parameters fixed for energy, target and projectile combinations. Results in complex cases, as well as properties and scaling laws, arise naturally from the underlying physical models, predictivity is provided where no experimental data are directly available, and correlations within interactions and among shower components are preserved.

Some example Fluka simulations can be found in [4].

FLUKA can simulate with high accuracy the interaction and propagation in matter of about 60 different particles, including photons and electrons from 1 keV to thousands of TeV, neutrinos, muons of any energy, hadrons of energies up to 20 TeV (up to 10 PeV by linking FLUKA with the DPMJET code) and all the corresponding antiparticles, neutrons down to thermal energies and heavy ions. The program can also transport polarised photons (e.g., synchrotron radiation) and optical photons. Time evolution and tracking of emitted radiation from unstable residual nuclei can be performed online.

FLUKA can handle very complex geometries, using a Combinatorial Geometry (CG) package. The FLUKA CG has been designed to track correctly also charged particles (even in the presence of magnetic or electric fields).

For most applications, no programming is required from the user. However, a number of user interface routines (in Fortran 77) are available for users with special requirements.

More information about the FLUKA physical models can be found in several journal and conference papers referenced in [6].

On the technical side, the stress has been put on four apparently conflicting requirements, namely efficiency, accuracy, consistency and flexibility. Efficiency has been achieved by table look-up sampling and a systematic use of double precision has had a great impact on overall accuracy. To attain a reasonable flexibility while minimising the need

for user-written code, the program has been provided with a large number of options available to the user.

The rest of the section gives an overview of the structure of Fluka, emphasizing on geometry aspects. The first section describes the execution phases of Fluka, follows an explaination on how the geometries are built and to finalize we discuss the extension points (or user routines) provided by Fluka.

2.1. Execution phases

Most of the configuration for the simulation is given to Fluka through an input file. This file contains all the instructions needed to set up the simulation and determine which physic processes have to be simulated and which outputs are desired. The Fluka input file is structured as follows:

1. Physics settings
2. Beam definition
3. Geometry description (bodies, regions, material assignments, lattices)
4. Material properties
5. Scoring definitions / Output configuration
6. Random number initialization
7. START
8. STOP

In the *physics settings* section the user can define the accuracy and the kind of events that she is interested in. The *beam definition* sets the properties of the beam as their position, the kind of particles, direction and energy. The *geometry definition* is divided into several subsections: the definition of the primitive bodies, the region definitions given as logic compositions of bodies, the material assignments and lattice (or region replicas) declarations. If needed, a section defining the properties of the materials follows.[1] The configuration of the output is given in the *scoring definition* section. After, the *random number generation* policy is set. The simulation starts when Fluka finds the *START* command in the input file. When the simulation is completed the *STOP* card is found and the process is finished.

From a higher level, the execution flow can be roughly divided into three parts:

1. Reading of the input

[1] Fluka comes with a material database, nonetheless extra materials and compounds can be defined in the *material properties* section.

2. Tracking of particles and writing scorings (when START is found)

3. Shutdown: closing files showing summary, etc.

2.2. Scorings

The outputs of Fluka are known as scorings. They are the areas of interest of the geometry in which the user wants to study some effects. While the transport of the particles takes place the values of the scorings are updated. Given that these kind of simulations can produce gigabytes of output data the output files are writen during the tracking.

2.3. Building geometries

The fluka geometries are built from bodies that are combined to define regions. The body primitives define half-spaces: they define which part of the space is inside, and consequently, what is outside. For example, given a sphere described by its position and radius we can check if a point in space is inside or outside. All body descriptions are defined by their parametric equations. A list of the most common body primitives follows:

Sphere A sphere can be described as a point in space and a radius

Ellipsoid Unlike the sphere the ellipsoid has a different radius for each of its axis. Hence to define an ellipsoid three axis are needed (their orientations) and the length of the ellipsoid along each of them. Its center is defined by a position in space.

Plane A plane is described as a point on the plane and a vector perpendicular to its surface (the normal). Planes are infinite.

Cylinder A cylinder is described by a vector defining the axis of the cylinder and its radius.

Elliptical cylinder An elliptical cylinder is a cylinder with an elliptical section, therefore it has two radii: one for each axis of the ellipse. It is defined by an axis defining the orientation of the cylinder and two extra axes defining the orientation of the elliptical section.

Cone A cone is described as a vector whose direction is the axis of the cone, the radius of the cone at the end of this vector and a point in space where the cone radius is 0.

Quad Bodies can also be defined through their general quadratic formula.

2.3.1. Solid geometry composition

More complex objects can be described by combining primitive objects, namely these operations are available: intersection, difference and union. All the geometries representable in Fluka can be defined by a tree of these operations and the primitive bodies. This compositing technique is called Constructive Solid Geometry (CSG).

The composition operations are explained below. The pictures illustrate the result of the operations, for which two primitive bodies are used, a cube A and a sphere B.

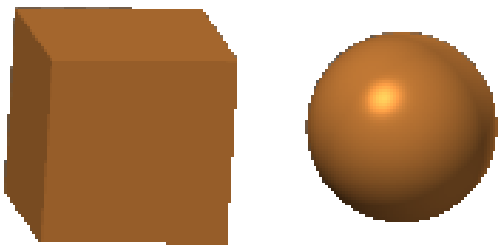


Figure 2.1.: A and B primitives

Intersection

The points in space belonging to the intersection of $A \cap B$ are those points that belong to A and B.

Figure 2.2.: Intersection of bodies A and B

Difference

The points of space in the difference $A \setminus B$ are the points that belong to A and do not belong to B.

(a) $B \setminus A$ (b) $A \setminus B$

Figure 2.3.: Difference of two bodies

Union

The points in the union $A \cup B$ are the points that belong to A or belong to B.

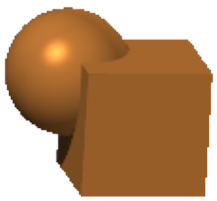


Figure 2.4.: Union of bodies A and B

2.3.2. A note on FLUKA geometries

It is worth pointing out that Fluka geometries must always define the whole space. The geometric object in Fluka is the region, a region is defined as a CSG expression over bodies and it defines a portion of space. A regions has a specific materials and physical properties.

When the tracking is being performed, Fluka requires that the particles are always located within the defined space for which it knows the material and properties. This implies that the whole space must be defined in the Fluka geometry, and every point in space must belong to one and only one region. If this condition is not satisfied the tracking results will be meaningless.

The tracking of a particle stops when the particle reaches a region with blackhole material or when other artificial limits have been reached (number of iterations, energy threshold, number of secondaries generated, etc.).

Defining this kind of geometries requires a significant effort by the physicists using the software but pays off in simulation speed and accuracy. A tool for visualizing the geometry can thus be an essential part for developing and validating the desired geometry.

2.4. Extension points

Fluka is designed to be extensible by allowing the users to define what are called *user routines*. These routines are executed at pre-defined execution points. For example at initialization, shutdown or when a certain event occurs during the tracking. Fluka provides default implementations for these routines that can be overriden if needed. A selection of user routines that can be redefined and that are relevant for this thesis can be found below. For a complete reference see [5].

- SOURCE This routine can be used as the source of particles to inject in the Fluka process for tracking instead of the standard BEAM and BEAMPOS definitions.
- FLUSCW If enabled, it will be invoked when a specific region of the geometry is reached by a particle.
- USRGLO Called before the Fluka intitialization occurs.
- USRINI Called at initialization phase.
- USREOU Called at the end of each event.

The user routines provided by the user will replace the default user routines for these events, if these are provided they must be compiled and linked to the Fluka executable. This process, performed with the utility `fff`, generates a customized Fluka executable for a specific set of user routines.

3. Analysis

There are other tools that can elaborate 3-dimensional representations of the FLUKA geometry. For example, SimpleGeo [13] is a Windows-only application that uses Direct3D to perform the drawing. Direct3D is, like OpenGL, an API based on the rasterization of triangles. For this reason SimpleGeo must convert FLUKA solid geometries to meshes. Also POVRay, an open source raytracer [11], can be used to render the geometry using the POVRay exporter in Flair [3]. This exporter converts the geometry to the POVRay format also involving some complex transformations.

The goal set for this project is to be more accurate than the previous solutions, all these solutions involve the conversion of geometry formats, a process prone to precision errors. Not only these conversions are affected by precision issues, they are separate products from Fluka and developed at a different pace. They also lack of advanced features of Fluka like the lattice or user provided functions.

A few alternative implementations were considered to decide the best way to proceed.

OpenGL viewport in Flair

One of the things that could improve Flair is a geometry viewer and editor. For example a 3D viewport using OpenGL to show an interactive representation of the geometry. An OpenGL view could leverage the power of modern GPUs to produce a fast interactive visualization of the geometry.

The main problem with this approach is the difficult process of converting the geometry that can lead to inaccuracies and/or incomplete models. After the conversion is done the visualization can be performed on real time. If this is later extended into a geometry editor it would require to update the meshes after each modification of the geometry.

Regarding dependencies, it will require the OpenGL libraries and CPU drivers on the host operating system.

External raytracer with POVRay

The conversion of the geometry is less problematic in this case because POVRay also handles solid geometry descriptions with CSG operations, but nonetheless special fluka features as lattices have to be carefully handled. After the conversion, the rendering will be slow because POVRay is based on the ray-tracing technique. It won't be interactive or real time. On the other hand the POVRay renderer is likely to be well optimized as it has been in development during many years and it has a supporting community.

Internal FLUKA Raytracer

Another option is possible: use the fluka geometry engine and write a ray-tracer on top of it. The ray-tracer would be integrated in the Fluka source code and therefore would be shipped with the standard Fluka distribution. By using the Fluka geometry engine it will support all the geometry features of Fluka including future ones. The accuracy problems would be delegated to the Fluka engine which has been proven to be robust in that regard. It would require nonetheless implementing all the image-generation logic and scene parsing.

As any ray-tracer the speed won't be one of the strongest points and probably it will be the slowest of the alternatives here presented.

To sum up the following table was collected:

1. **OpenGL [10] viewport in Flair**

   Speed

   Initial conversion and synchronization this solution needs an initial conversion from the solid geometry description of FLUKA to meshes. In addition every time anything changes in the geometry the meshes have to be updated.
   Real-time Visualization after the conversion the visualization is done in real time.

   Correction

   Many inaccuracies the conversion process will always have inaccuracies, the perfect description of solid objects can't be translated to meshes easily.

   Dependencies

   OpenGL libraries the viewport would need some libraries installed in the system: mesa and optionally GPU drivers.
   External math. libraries the geometry conversion will likely use external libraries to help with the mathematics.
   Hardware? the real advantage of an OpenGL viewport are the real-time graphics, usually rendered with the help of powerfull graphics cards.

   Features

   Real-time visualization
   Interactive allowing the user to navigate through the geometry.
   User friendly interactive visualizations are easier to understand.

2. **External Raytracer with PovRay**

   Speed

   Slow as any ray-tracing solution.

   Correction

Some inaccuracies the geometry must be converted to PovRay representation, involving some transformations, etc. Lattices will require some thought.

Dependencies
PovRay

Features
PovRay is a well-known raytracer.
Optimized after years of development.
Community the community can give support.

3. **Internal FLUKA Raytracer**

Speed
Slow as any ray-tracing solution.

Correction
Same geometry as the simulation interpreted by the FLUKA engine.

Dependencies
No external needs * except image conversion tools.

Features
Accuracy the same geometry as simulated
Lattices support of user defined functions, lattices, etc.
Built-in it is shipped with the simulator, this means it will be always compatible with FLUKA changes and enhancements.

We decided to implement a solution based on the third proposal described above, aiming for high quality results and full synchonization with all the features of the Fluka geometry engine.

3.1. Non-Functional requirements

Having taken the decision to implement a raytracer in Fluka there are a few important things to be taken into account.

All the input information needed by the raytracer must be entered by the user through the Fluka input file. Therefore new cards have to be designed in order to input the information needed by the raytracer.

Fluka is writen in Fortran 77 and that implies a certain number of constraints in the design and kind of algorithms that can be implemented, namely:

- There are no data structures. Only a few data types are available plus arrays made of those. One common way to define a structure is using "parallel" structures. For example if one needs to define the id and color of a material one way to do it is to

define an array of ids and an array of colors, then use the same index in the arrays to define the properties of a material. In this case `color[i]` and `id[i]` would be respectively the color and the id of the i_{th} material.

- Recursion is not allowed. The Fluka 77 standard does not allow functions that call themselves, directly or indirectly. The only way recursion can be implemented is through queues and/or stacks.
- There is no dynamic memory allocation. All the memory used by the program must be allocated at the initialization phase (with size defined at compilation time). This provides low flexibility to implement the kind of structures needed to implement recursion as pointed before.

  The size of these strucures, determined at compile time, limits the size of the problem that the program can solve. If the problem being solved does not fit in those structures the program must be aborted.
- Pointers do not exist in fortran 77 and function arguments are always passed by reference, therefore modifying input parameters will modify their values at the previous context. This has to be taken into account when coding the algorithms. One way to overcome the lack of pointers is to use the index on the array structure with the record of interest.
- Limited input/output. The input and output in Fortran 77 standard must follow some formatting rules. ASCII output can have almost any format but binary output always dumps the size of the field before the contents of the field. This will be a problem for writing the images to disk.

3.2. Validation approach

The author of the current document did not have any previous experience with Fortran and never had the oportunity to write a raytracer. The task was therefore two-fold: learn fortran and learn how to write a raytracer. It was decided to split the work in two phases: first the raytracer would be written in C – one of the author's preferred languages – and later it would be converted to Fortran.

It was convenient to know Fortran before writing the raytracer in another programming language, therefore, some time was spent in learning Fortran, its style and shortcomings. Having gained some experience with Fortran 77, the C code was written in a way that allowed an easy translation to Fortran. As described in the previous section neither recursion or dynamic memory allocation was available. That lead to the final design of the algorithms and data structures as explained in later chapters.

Once the raytracer written in C was working the code was translated to Fortran and a verification was performed. Later, some bug corrections and improvements have been

applied to the Fortran code so the two code bases have diverged.

4. Previous Concepts

There are many concepts that need to be known before implementing a raytracer. This chapter gives an introduction to the topic of computer generated graphics putting special emphasis on the techniques used in ray-tracing.

In section 4.1 we introduce the major technologies used currently for producing realistic 3-dimensional images. Section 4.2 explains the optical and physical effects observed in reality and describes some techniques to emulate them in a computer program. Section 4.3 introduces the reader to ray-scene optimization techniques used to accelerate ray-tracing.

4.1. Rendering technologies

We can discern two main techniques for producing computer graphics: rasterization and raytracing. Rasterization is a fast rendering technique that is used in virtually all the games and software in the market that requires of interactive 3D graphics. Raytraycing produces images that are closer to the physical reality at a higher computational cost and is mainly used for off-line rendering of movies or advertisements.

4.1.1. Rasterization

Rasterization is the technique of projecting objects into an image plane "the raster". These objects are usually triangles and they are drawn on the part of the image that they cover. The color of the pixels is determined by looking at the properties of the object material, lighting conditions and several other effects. This technique was first described in [19].

In the early days of interactive graphics the only reasonable option to produce interactive 3D images was using this technique. Soon specialized pieces of hardware that supported rasterization appeared on the market: the Graphical Processing Units (GPUs). GPUs and the consolidation of standard APIs as OpenGL or Direct3D led to the further development of the technique.

GPUs are pieces of hardware highly specialized on projecting and filling triangles onto the screen plane. However, interactive graphic applications are quite sophisticaded and do more than projecting triangles. Most of the work they perform is highly paralelizable and this is the kind of task where GPUs excel. GPUs are able to compute the position of

triangles in the scene and the color of each of the pixels of the screen applying complex models.

Indeed modern GPUs are programmable at two levels, the *vertex and pixel shaders*. In GPU terms, a shader is a small program that is performed over a set of data in parallel. The *vertex shaders* are in charge of positioning vertices in the space. The *pixel shaders* compute the exact color of each pixel on a surface. Postprocessing techniques also contribute to the realism of the output, convincing our eyes that the image is real. Although it may not be as physically accurate as the raytracing technique the images produced are very compelling for interactive graphics.

Rasterization requires geometries defined as meshes of triangles. FLUKA geometries are described as mathematical formulas, therefore, to use this technique the geometry would have to be converted to meshes.

4.1.2. Raytracing

The ray-tracing technique [15, 24, 20, 16] is more accurate in the sense that it tries to simulate the behaviour of light in the real world. In reality our eyes form an image out of the light that bounces into them coming from our surroundings. That light was produced by a light emitter, as for example the sun or a lightbulb. The rays of light bounce on the objects around us and reach our eyes. That light reaching our eyes produces the image of the world in our retina.

Due to technical limitations it is impossible to simulate the rays of light emmited by all the light sources in a virtual world, simulate all the bounces these rays perform on the objects and only collect only those that hit the eye (or the virtual camera for our purposes).

Therefore in ray-tracing the opposite aproach is taken: view rays are shot from the position of the camera towards the geometry. One ray is shot for each pixel of the final image, passing through the center of a pixel in the imaginary plane where the scene is projected. The color "seen" by this ray (the color of the first intersection indeed) is the color for that pixel. The computation of the color of the intersection point can be relatively simple or extremely complicated depending on the kind of realism desired. To obtain the highest realism the color of that point should be computed by taking into account all the light that bounces on it towards the camera. This requires intensive computational work and it is usually approximated with various techniques.

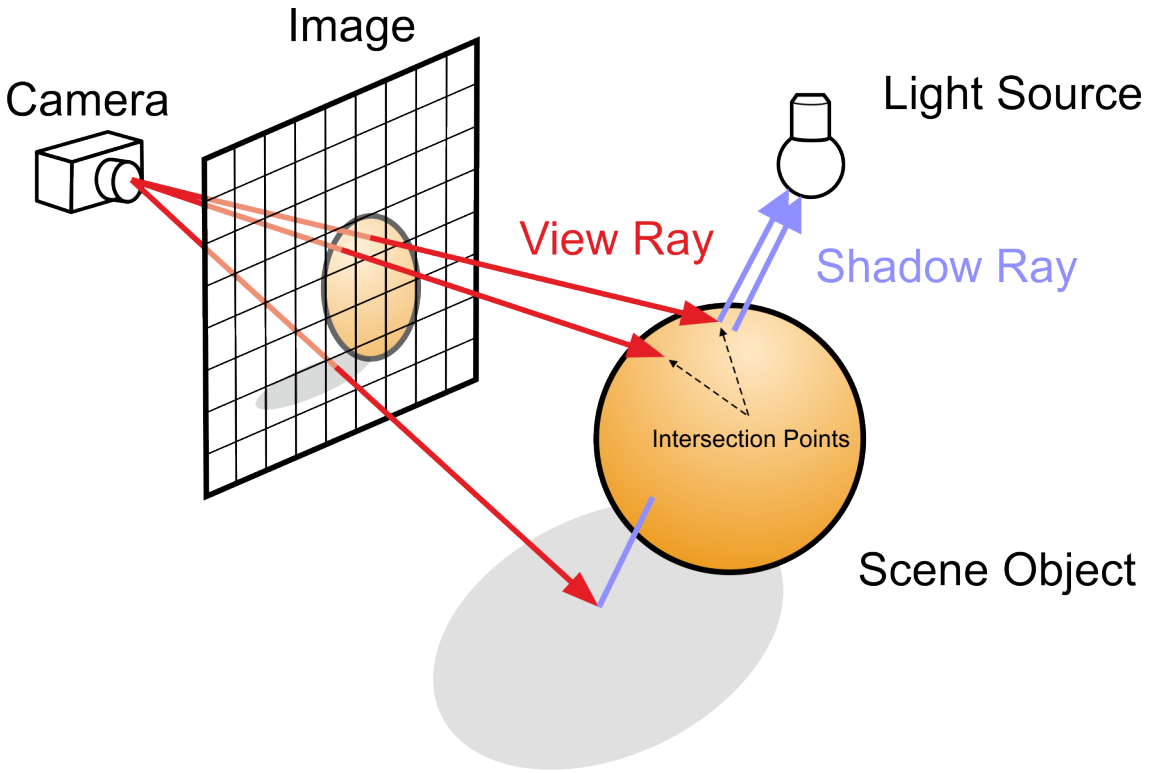


Figure 4.1.: Ray-tracing [12].

The raytracer developed within this thesis only considers direct lighting (light that comes directly from the light sources), or what is the same: the light bouncing in other objects is not computed. Checking for direct illumination from each of the lights is not complicated. One ray is shot from the the intersection point towards the position of the light source. If the ray intersects an opaque object before reaching the light this object is blocking the visibility of the light and the light has no effect on the color of the intersection point. An estimation of the color of the intersection is computed as a function of the material properties of the object, angles and distance to the light. For extra realism, reflected or refracted rays are computed.

The ray-tracing technique needs to perform a big number of computations as each ray is completely independent from the others and may hit different objects with different lighting conditions. Nonetheless, current computers are powerful enough to produce several frames per minute depending on the complexity of the scene and the quality desired. Raytracing wouldn't be considered for close to real-time graphics without the amount of effort that has been put on improving the algorithms behind it.

Currently the industry of interactive graphics is moving towards hybrid approaches using rasterization for geometry projection and path-tracing for lighting and adding realistic effects to the final image [23].

4.2. Rendering concepts

This section discusses the ideas needed to understand how images are generated and the data needed by the render engine to produce them.

4.2.1. Cameras

In order to recreate reality we need to define a camera model through which we can see the virtual world. A virtual camera is usually defined as the point where the spectator is located, a point in the space where it is looking at, an angle of view (or apperture) and a vector marking the up direction needed to know the roll of the camera.

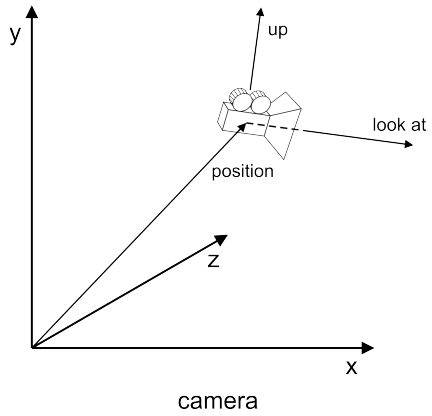


Figure 4.2.: Camera

Apperture angle

The apperture angle of a camera defines the angle between its sides, i.e. it defines a portion of the world that is visible by the camera. It can be measured horizontally, vertically or diagonally as shown in image 4.3.

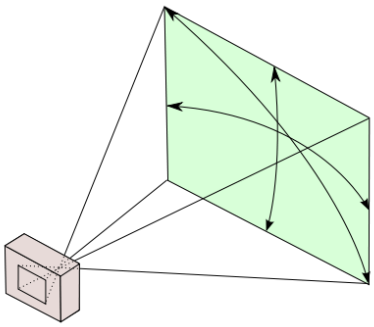


Figure 4.3.: Apperture of the camera

Perspective projection

Cameras with perspective projection produce images where the more distant objects appear smaller in the image. A perspective projection camera is the type of camera that we are most used to as it is the kind of camera used in photography, movies, etc. It is also the way we see the real world.

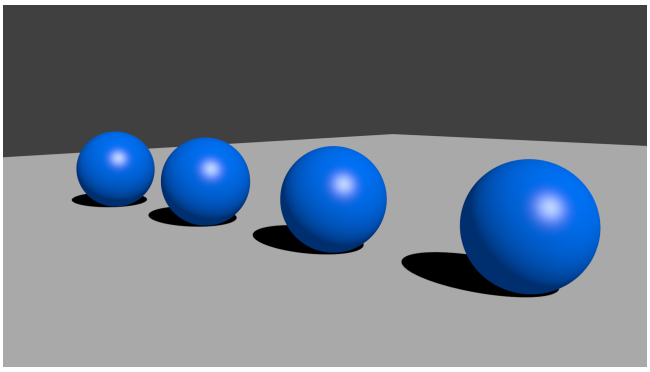


Figure 4.4.: Perspective projection

Orthogonal projection

Cameras with orthogonal or orthographic projection produce images on which the size of the objects is the same independently of the distance from the object to the camera. This effect is achieved by cameras that produce parallel view rays.

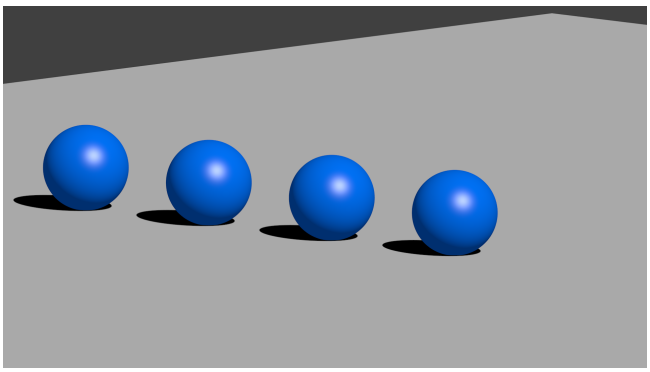


Figure 4.5.: Orthogonal projection

Depth of field

Our eyes have a focal point, i.e. there is a distance at which objects appear focussed, outside of a threshold objects begin to appear blurry.

Usually in computer graphics cameras are modelled as pin-hole cameras, i.e. all the rays originate at the same infinitely small point, the camera position. This type of cameras

produce images where all objects are in focus. Nevertheless, a range of techniques can be used to produce artificially the depth of field effect. The effect is shown in image 4.6.

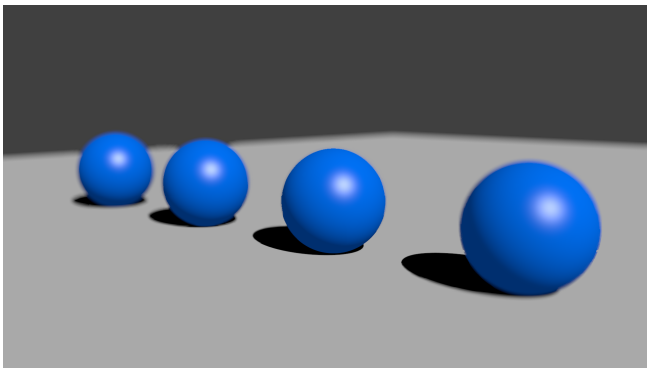


Figure 4.6.: Depth of field effect

4.2.2. Materials

Each material has different properties that affect the way the light interacts with them and therefore change the appereance of their surfaces. In computer graphics each material can define a different formula, the lighting model, that simulates this behaviour.

Lighting models, also known as Bidirectional Reflectance Distribution Functions (BRDF), model how the light bounces on surfaces and are used to compute the color of each point in a surface. Emulating the apereance of materials is not a simple task, advanced BRDF models are made of multiple layers and some of them even compute the light that goes beneath the surface and bounces back. For an in-depth description of some BRDFs refer to [17]. For a detailed validation of simple BRDF models against real data refer to [14].

For this thesis a simple BRDF is used: the Phong reflection model [22], it is fast and produces good results for visualizing geometries.

Phong reflection model

The Phong reflection model [22] defines the following equation to determine the illumination perceived by a viewer for each point on a surface:

$$I_p = \overbrace{k_a i_a}^{\text{ambient term}} + \sum_{m \in lights} \overbrace{(k_d(L_m \dot{N}) i_{m,d}}^{\text{diffuse term}} + \overbrace{k_s(R_m \dot{V})^{\alpha} i_{m,s})}^{\text{specular term}}$$

Where I is the intensity of the light bounced at a point in a surface p towards a specific point of view, the k terms are the blending ratio respect to other terms, the i terms are the intensity of each term for each of the lights. m, a, d and s indexes refer to the

ambient, diffuse and specular terms of the material. L is the vector going from the surface point to the light, N is the normal of the surface at the point, V is the reversed vector going from the viewer to the intersection point and R is the V vector reflected on the surface point. α is the specular power. All the vectors are normalized to simplify the computations.

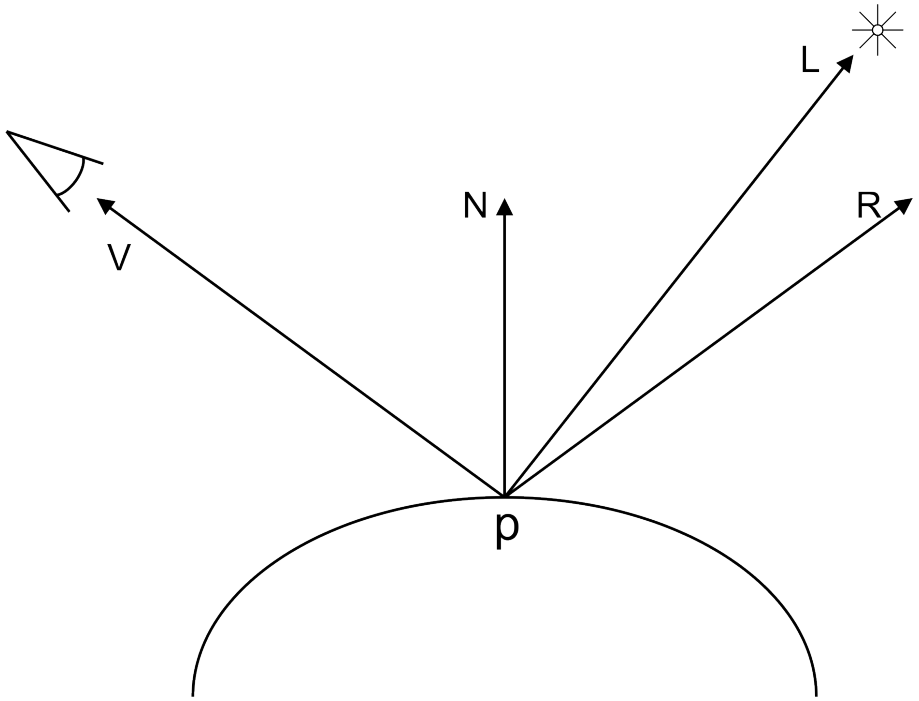


Figure 4.7.: Phong model vectors

Therefore, splitting this equation into parts: the output color of a point depends on a) a constant (*ambient light*), b) the *diffuse term* and c) the *specular term* contributed by each light. Each of these terms is described below.

This equation is applied to the three components of the colors (red, green and blue) producing the final intensity for each of the components of the final color.

The constant term: the ambient light

$$I_p = \overbrace{k_a i_a}^{\text{ambient term}} + \sum_{m \in lights} (\overbrace{k_d(L_m \dot{N}) i_{m,d}}^{\text{diffuse term}} + \overbrace{k_s(R_m \dot{V})^{\alpha} i_{m,s}}^{\text{specular term}})$$

Not simulating light bounces produces images that have higher contrast between illuminated and shaded parts. The ambient light is a trick used to paliate this effect. It is a constant light that is applied to every surface point in the image.

It produces images in which objects seem to have an internal source of light, appearing brighter than otherwise computed. When used with care and taste, it can improve the realism of the resulting images, applying too much of it generates flat looking images.

The diffuse term

$$I_p = \overbrace{k_a i_a}^{\text{ambient term}} + \sum_{m \in lights} (\overbrace{k_d(L_m \dot{N}) i_{m,d}}^{\text{diffuse term}} + \overbrace{k_s(R_m \dot{V})^{\alpha} i_{m,s}}^{\text{specular term}})$$

Most of the surfaces that surround us in our daily life scatter light in a very regular way. This kind of homogeneous light reflection is known as diffuse reflection. These surfaces bounce the light in all directions and produce soft-looking surfaces that add depth to the image.

A diagram of the light bouncing on a diffuse surface can be found in figure 4.8. As one can see the light is evenly reflected in all directions with varying intensity that depends on the cosine of the view angle respect to the normal.

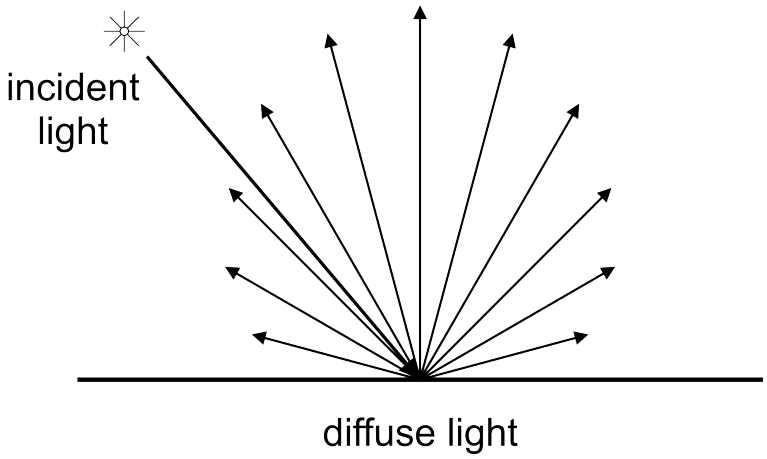


Figure 4.8.: Diffuse surface

The specular term

$$I_p = \overbrace{k_a i_a}^{\text{ambient term}} + \sum_{m \in lights} (\overbrace{k_d(L_m \dot{N}) i_{m,d}}^{\text{diffuse term}} + \overbrace{k_s(R_m \dot{V})^{\alpha} i_{m,s}}^{\text{specular term}})$$

The other term in the equation emulates the effect that can be apreciated in metals or other shiny surfaces. These kind of materials reflect the light in a focussed direction, producing shiny spots in their surfaces.

A diagram showing the rays of light bouncing on a specular surface is shown in figure 4.9.

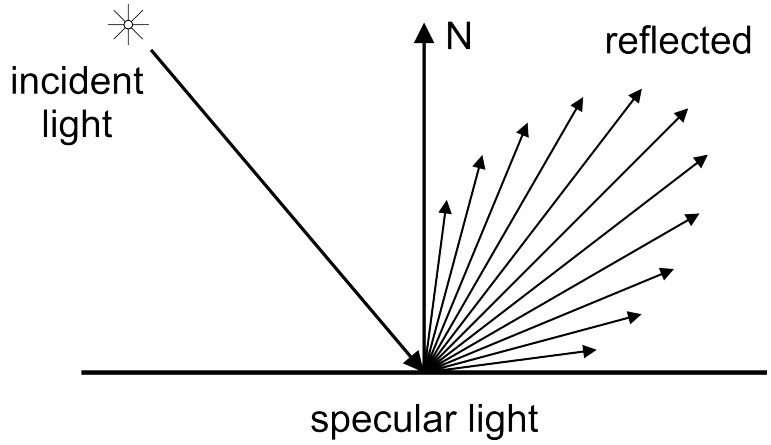


Figure 4.9.: Specular surface

The specular term is given by a power of the angle between the reflected view ray at the intersection point and the vector going to the light. The power applied, α in the model equation, is known as the specular power in the material properties. The higher the value of the power the sharper the shiny spot on the surface is.

Combination of the terms

The final effect is achieved by combining the three terms described, figure 4.10 shows the individual effects and the combined effect.

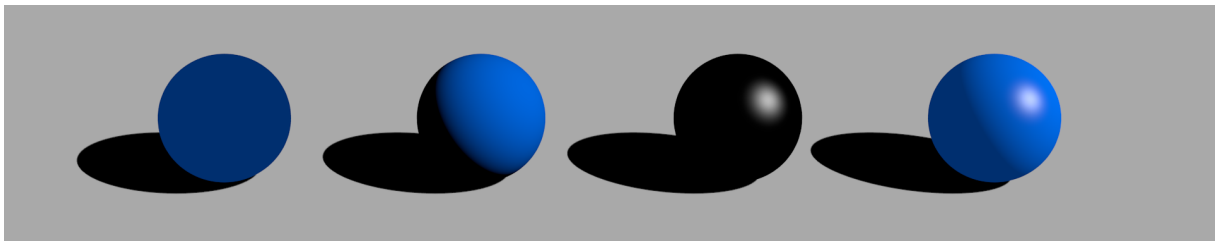


Figure 4.10.: Phong reflection model, from left to right: ambient term, diffuse term, specular term and the combined effect

Reflection

The reflection effect is produced when most of the light that a material receives is bounced unmodified. The incoming ray is reflected in a way that the outgoing ray makes the same angle respect to the surface normal at the intersection point. The incident, normal and reflected rays lie on the same plane.

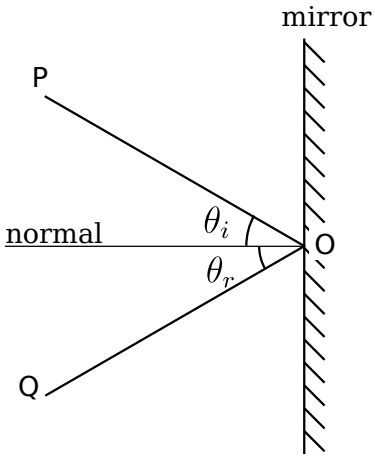


Figure 4.11.: Reflection

Refraction

Refraction occurs in materials that the light can traverse, for example with glass or water, producing the transparency effect. The light follows a straight line when traversing the object, but it undergoes a change in its direction due to the change of density at the boundary as stated by the Snell's law.

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{\lambda_1}{\lambda_2}$$

v is the velocity of light in the medium and λ is the refractive index specific to the material. A graphical representation can be seen in figure 4.12. For graphics only the ratio between materials is needed, refractive materials must therefore define the index of refraction.

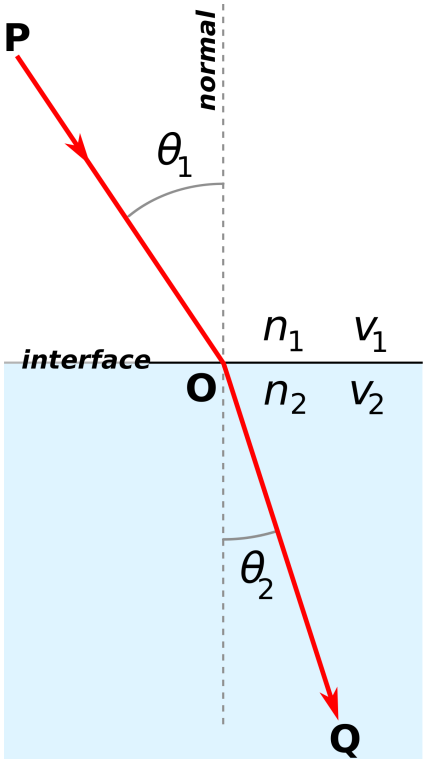


Figure 4.12.: Snell's law

A specific case is the total reflection. When light travels from a medium with a higher refractive index to one with a lower refractive index, this formula suggests that the sine of the angle of refraction is greater than one. As illustrated in figure 4.13, that is impossible, and in such cases the light is completely reflected, this phenomenon is known as total internal reflection.

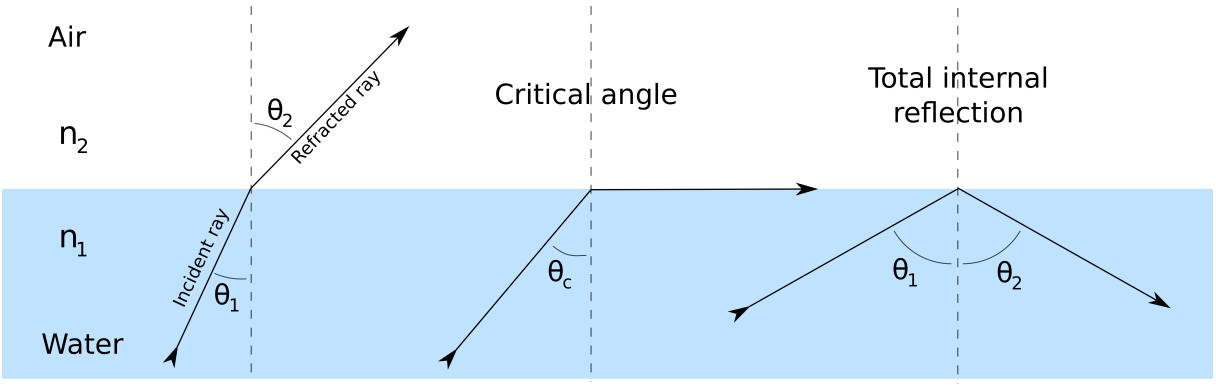


Figure 4.13.: Total reflection

Textures

Objects do not present an homogeneous color along their surfaces. Many properties of the material affect the looks of the surface. For example, objects are not made of pure materials, the concentration of each material at a specific point changes its appereance. The microscopic structure of the material has a great effect on the macroscopic appereance of a surface.

Image textures

The changes of color in a material can be simulated by applying images to their surfaces. Figure 4.14 shows a collection of textures applied to a set of spheres. There exist several techniques to specify how the images are applied to a surface. For example, using a 2D texture from bitmap image and projecting it on the surface of the object. Another option is loading a 3D grid specifying the color of each point (or cube) of the space. A third family of textures are those called procedural textures, they are specified by mathematical formulas that define the color of a point in space as a function of its coordinates.

Figure 4.14.: Texture collection

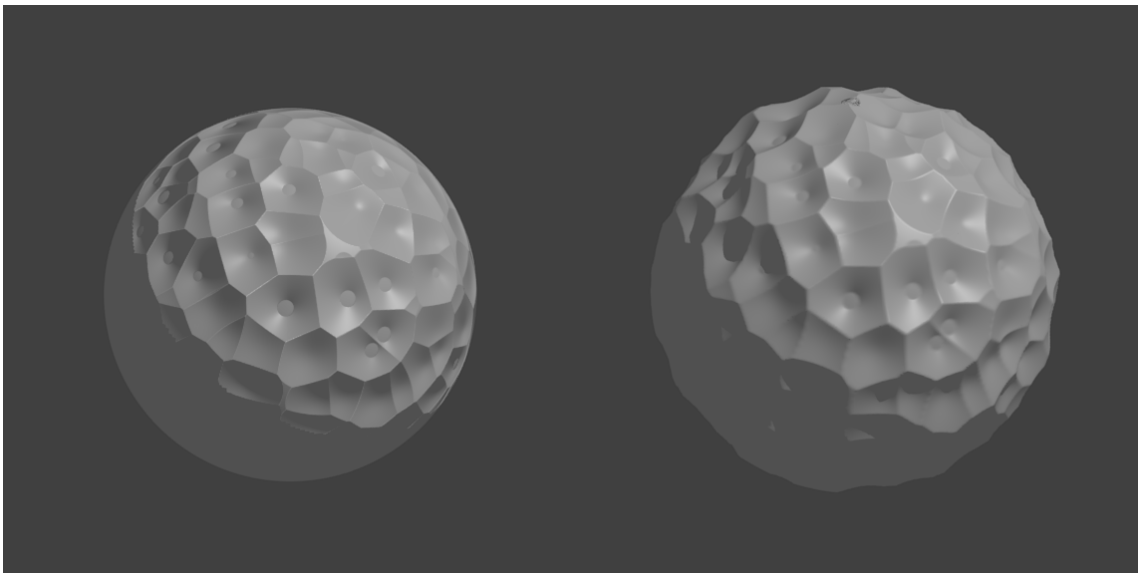
Bumpmapping and displacement mapping

There is a limit in the amount of detail in the geometry that a rendering engine can handle and, eventually, there will be details that are not worth modelling.

In such cases, and only for visualization purposes, another trick can be used to improve the realism of the image ([18]): a bumpmap. The bumpmap is a secondary texture used to modify the normals of each point on a surface. By doing so the geometry remains unmodified but it can represent small normal variations.

Bumpmapping does not change the shape an object and therefore it is only useful as a visual trick. For a more realistic effect displacement mapping can be used ([18]). This technique uses a secondary texture to displace vertices, effectively improving detail. Displacement mapping is only possible if the render engine is based on meshes.

An example of these two techniques can be found in in fig. 4.15.



(a) Bump mapping (b) Displacement mapping

Figure 4.15.: Surface modifiers

4.2.3. Lighs

This subsection studies the properties of light sources that are interesting for a visualization program. Color, intensity, falloff, light types, shadows and more are described below.

Color

Each source of light emits light at a specific wavelength or, what is the same, a color. The color of the emitted light affects the color of the objects it illuminates. For example, in figure 4.16 four blue balls are illuminated by 4 lights of different colors, each light varies the color of the ball that we can see.

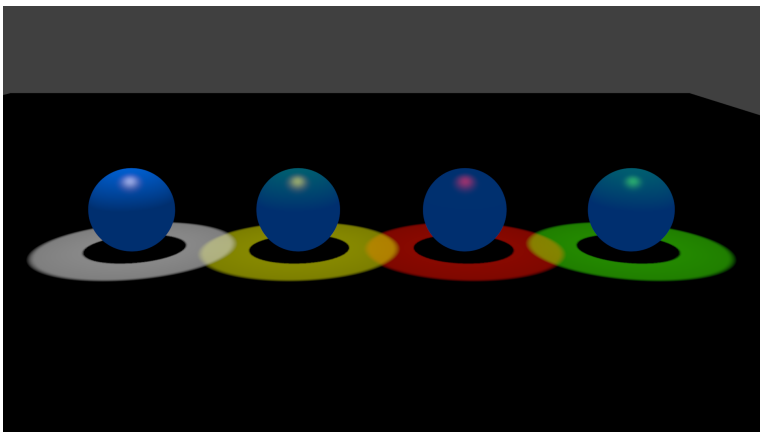


Figure 4.16.: Light color

Intensity

The intensity of a light source is the amount of rays of light that it emmits. More intense lights will produce brigther images.

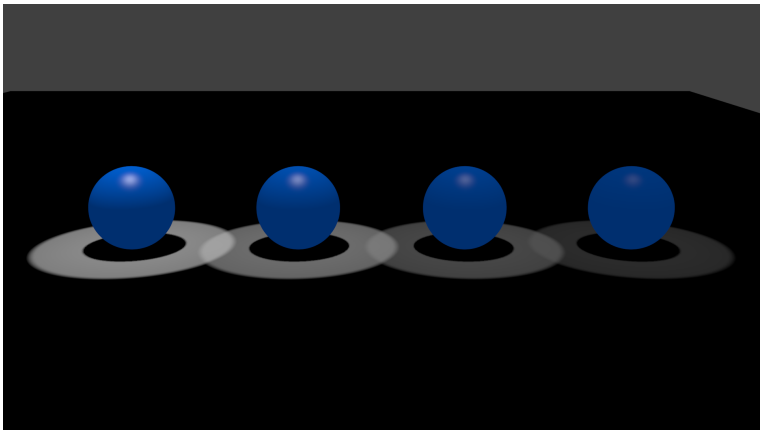
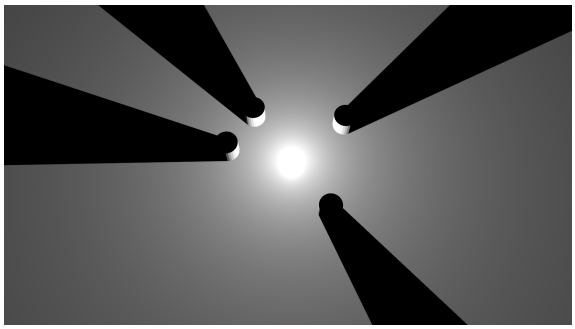


Figure 4.17.: Light intensity, from left to right: 1.0 intensity, 0.6 intensity, 0.25 intensity, 0.1 intensity

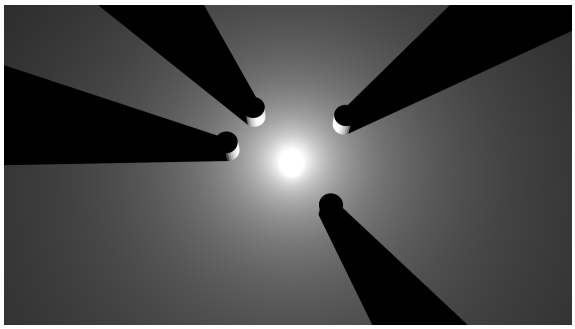
Falloff

Another effect that can be simulated by a trick is the loss of intensity as the light flies through a transparent medium. This is emulated by associating a falloff function to each light, producing an attenuation of the light intensity as a function of the distance to the emitter.

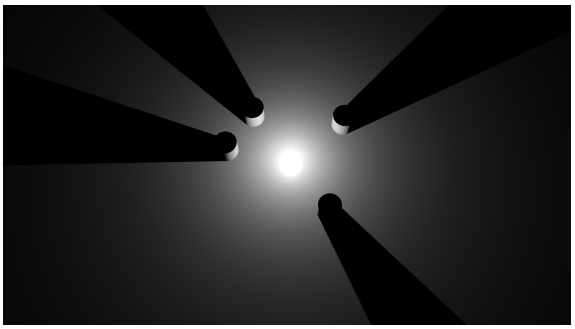
Image 4.18 shows different falloff functions: constant, linear and square of the distance.



(a) Constant falloff



(b) Linear falloff



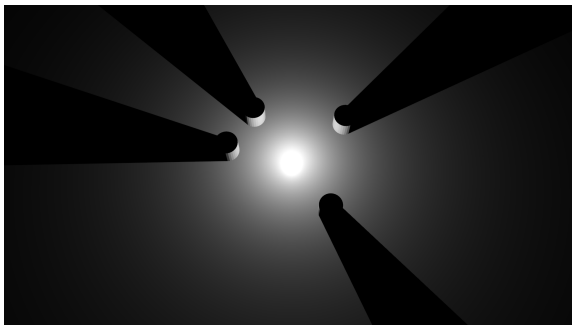
(c) Square falloff

Figure 4.18.: Light falloff settings

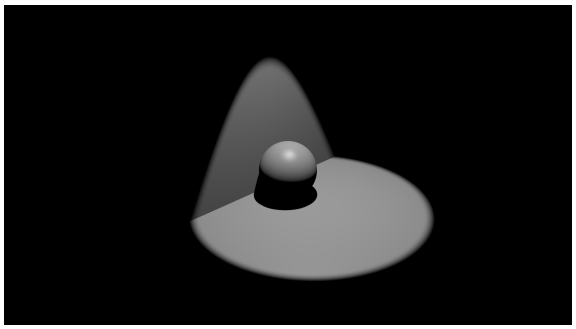
Light types

In the context of computer graphics the type of a light refers to the direction and scope of the light emitted.

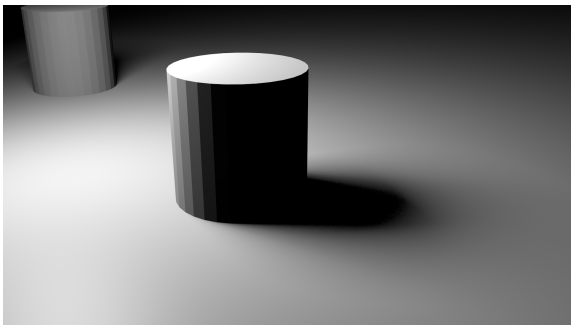
Onmidirectional lights emit light in all directions from their position (fig. 4.19a). Spot lights produce a cone of light, like the light emitted by a desktop light (fig. 4.19b). An area light is a surface that emits light, the light is not produced at a single point but along a surface (fig. 4.19c). This kind of light produce smoother illumination and shadow borders.



(a) Omnidirectional ligh



(b) Spot light



(c) Area light

Figure 4.19.: Light types

Shadows

In rasterized graphics a technique called stencil buffer shadow volumes is used to compute shadows [21]. This technique consists in projecting the silouete of the objects in the scene from the light position to the infinite. This yields a volume, each point of the geometry that falls outside of this volume is lit by the light. In order to apply this technique the scene is rendered several times, with and without illumination and the shadow volumes are used as a mask for the blending of the renders.

In raytracing the shadow effect is more natural, we can compute whether a light is illuminating a point of the geometry by shooting a ray from the point of interest towards the position of the light source. If an intersection with another object is found before reaching the position of the light the point is in a shadow.

For each point of the geometry for which the color is needed one ray is shot toward each of the lights of the scene in order to know if the light is affecting its color.

For the rest of this section the emphasis is put on the raytracing techniques to improve shadowing.

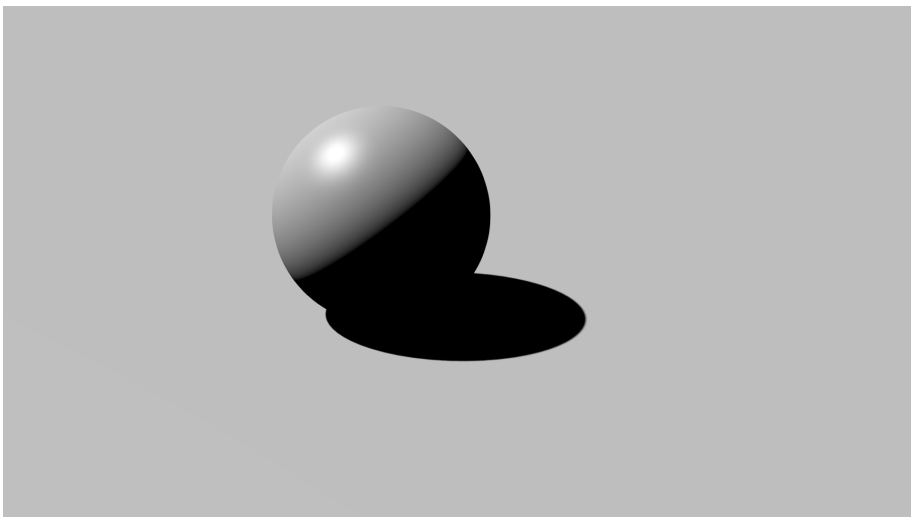


Figure 4.20.: Shadows

Transparent shadows

As explained before, there is one ray shot towards each light of the scene and the effect of the light is discarded if an object intersects the ray before reaching the light. Following this approach shadows are completely dark, even for those casted by semi-transparent objects. Transparent shadows take into account the material of the object traversed and let the light go through it partially if it is transparent. The transparent shadow effect is shown in image 4.21.

(a) Without transparent shadows (b) With transparent shadows

Figure 4.21.: Non-transparent and transparent shadows

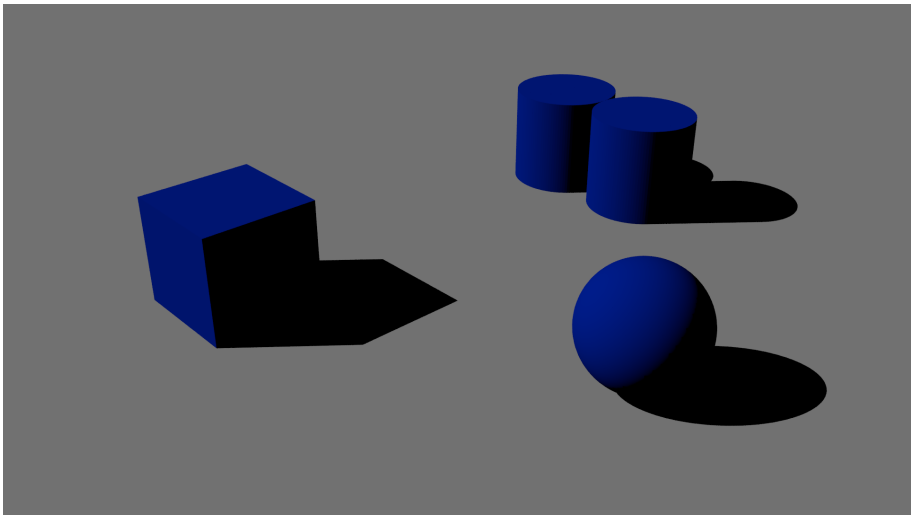
Global illumination

Until now we have discussed direct illumination only. But in reality the color of an object is not given only by the light that comes directly from the light sources. Instead the color

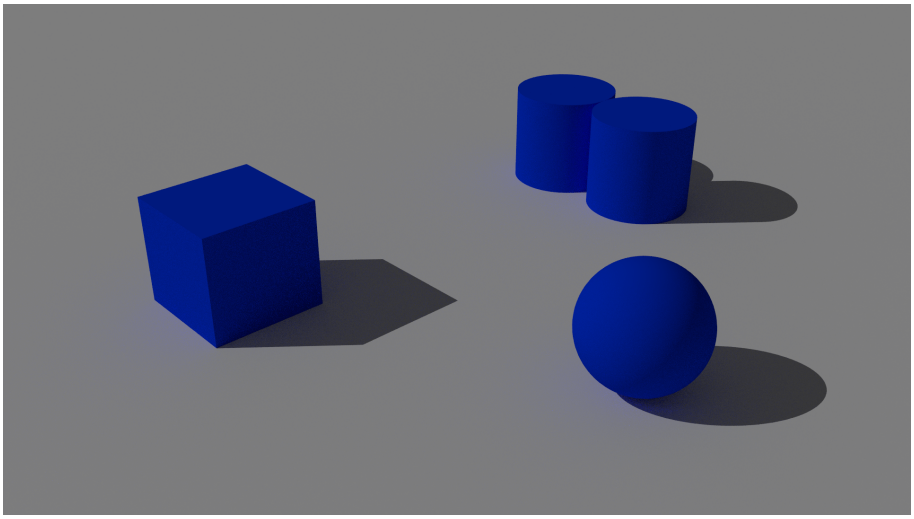
of a point is also influenced by the light that bounces on the objects surrounding it. The process of emulating those secondary sources of light is known as global illumination.

The more realistic techniques are the ones that try to simulate the behaviour of light in reality but have also higher computational costs. The number of samples that need to be taken to compute acceptable images can be reduced by using stochastic methods. This is an active research field which is leading to close to real-time rendering for simple scenes.

Image 4.22 shows an image computed using a global illumination approximation, it can be seen that the color of the objects affect the color of the ground and viceversa. Shadows are not homogeneous either.



(a) No global illumination



(b) Global illumination

Figure 4.22.: Global illumination

Ambient occlusion

This technique tries to fake global illumination by darkening areas of the geometry close to other objects or in concave regions. It computes the intersection distance of a point of the geometry to the rest of the geometry. The closer the other objects are, the darker the point is painted. This works by supposing that the bigger the "field of view" from the intersection point the more light can reach it, this assumption is illustrated in image 4.23. For example, if the point is close to a perpendicular wall less light would be able to reach it. Image 4.24 shows an image rendered without and with the ambient occlusion technique.

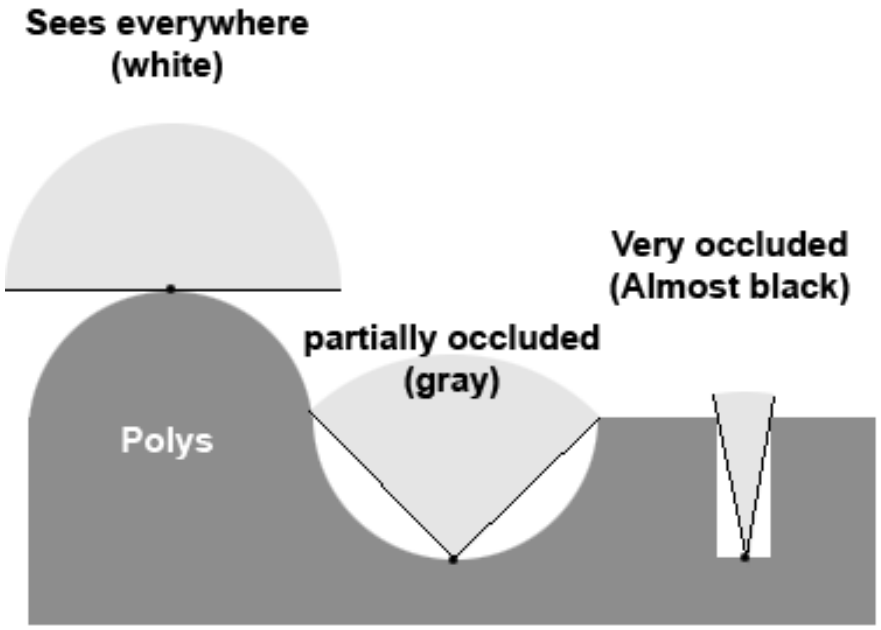


Figure 4.23.: Ambient occlusion principle

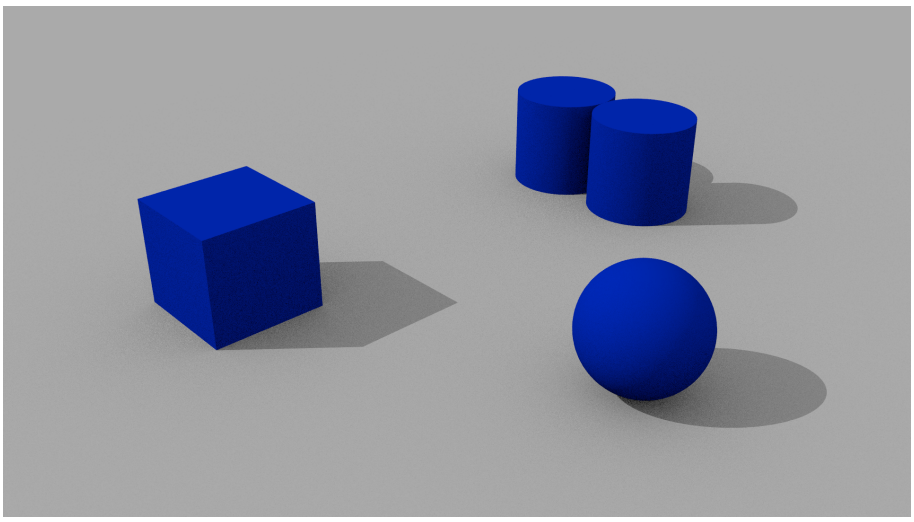


Figure 4.24.: Ambient occlusion

HDRI

HDRI stands for High Dynamic Range Images and is a technique used to emulate the environment of the scene that is not modelled. The HDR image is a spherical mapping of the light that reaches the geometry. That information is used to know the color and intensity of the light reaching the scene from all directions.

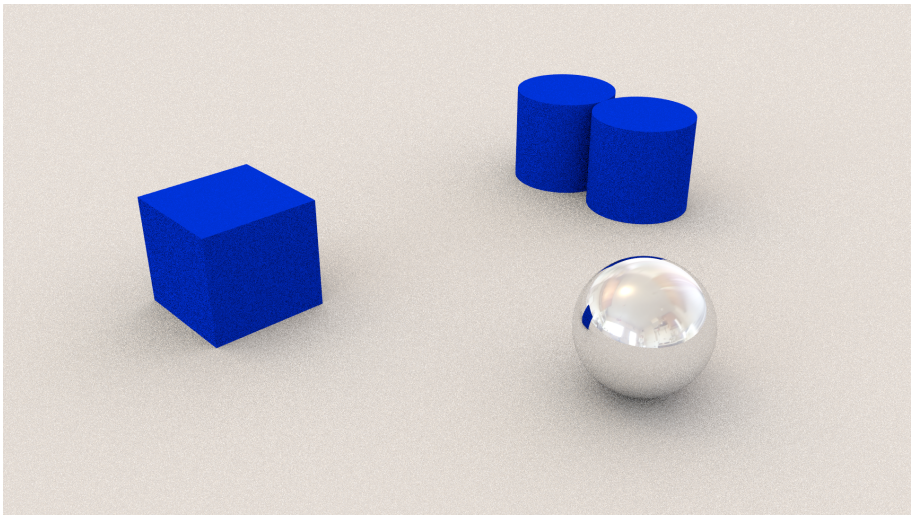


Figure 4.25.: HDRI

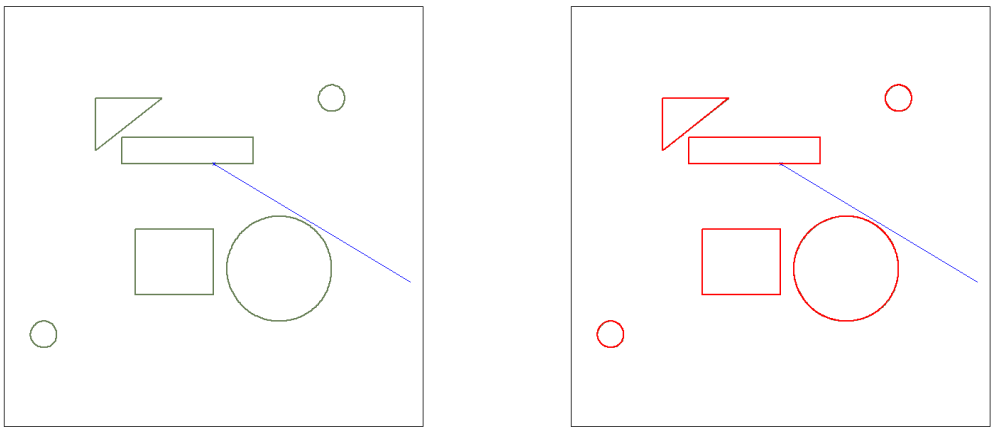
4.3. Ray-scene intersection optimization techniques

The ray-tracing technique is slow compared to rasterization because it requires intersecting a large number of rays with the geometry of the scene. The quantity of rays needed to obtain good quality images can be in the order of millions for scenes with few objects. Researchers, the entertainment and design industries have put extra effort in developing techniques that reduce the cost of computing the intersection of one ray with the geometry.

A naive way to compute the next intersection of a ray with the geometry is to compute the intersection point of the ray with all the objects in the scene, sort the intersections and use the closest to the ray origin.

All the optimization techniques introduced in this section try to reduce the number of bodies intersected to determine which is the closest intersection. For that purpose a pre-processing phase is applied to the geometry in which an acceleration structure is constructed. During the render, the ray-scene intersection algorithm uses those acceleration structures to reduce the number of checks.

Some of the most common acceleration structures are described below. We use diagrams to illustrate the optimization techniques and how the scene is traversed to find the closest intersection. The following convention is used: on the left the acceleration structure is shown along with the objects of the geometry. On the right the intersection operation (also known as traversal) is shown, the nodes of the acceleration structure that are visited are painted in a different color and the primitive objects that had to be intersected are painted in red.



(a) World (b) Traversal

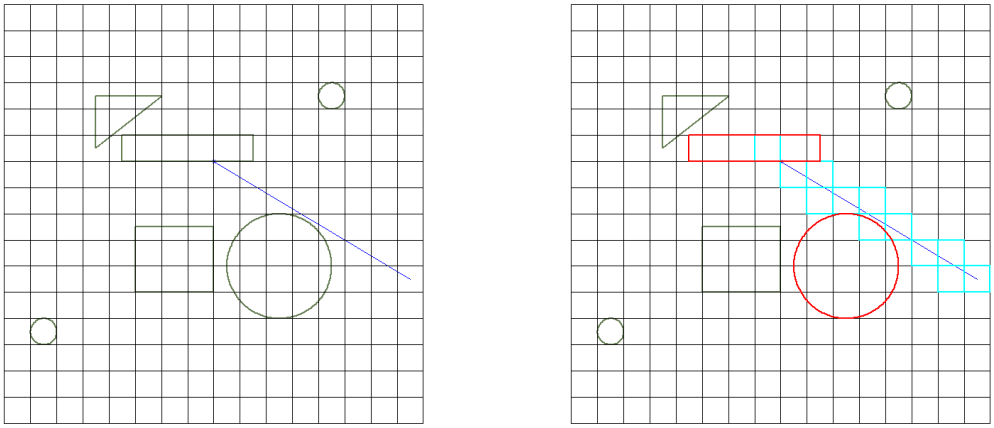
Figure 4.26.: Intersection without acceleration structures. On the left, the geometry and the ray are represented. On the right one can see that with this approach all objects have to be checked for intersection.

4.3.1. Spatial subdivision

Spatial subdivision techniques divide space into smaller volumes therefore if the ray traverses one of those smaller volumes there is a chance of intersecting the objects within it, otherwise the intersection is not possible. Some of these techniques are explained below: regular grids, octrees or kd-trees.

Regular grids, octrees

Regular grids divide the space in identical axis-aligned volumes. The traversal algorithm consists in enumerating which of these sub-volumes the ray intersects. Each sub-volume contains a list of objects that overlap with it. When a ray passes through one sub-volume the objects in its list are intersected, the closest intersection is returned as the next intersection. Regular grids are sketched in diagram 4.27.

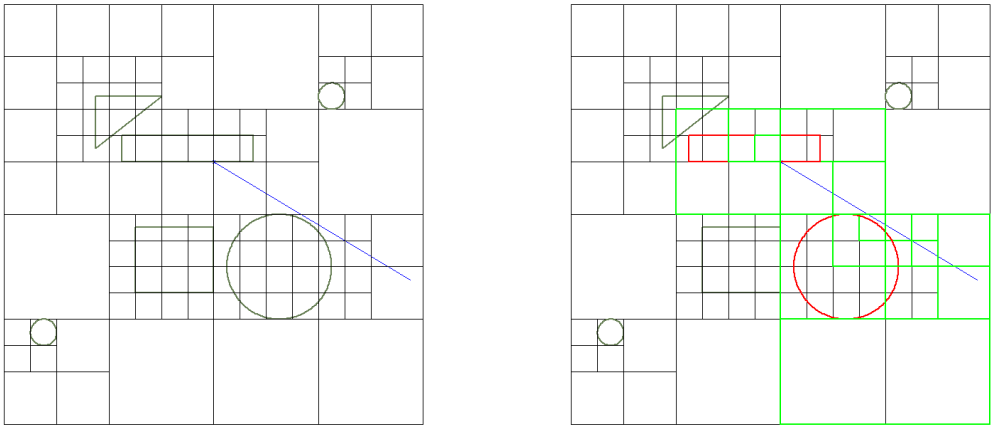


(a) Grid structure (b) Grid traversal

Figure 4.27.: As seen on the right only two objects had to be intersected during the traversal.

This approach suffers from a large memory footprint as the virtual world is evenly subdivided even when there is only empty space. Also, empty volumes are visited very often when computing the next intersection.

One alternative is to subdivide space hierarchically, octrees divide the space in eight equally sized subparts at each node. Hierarchicall structures save space and computing time by not subdividing empty volumes or where the detail is low. This is illustrated in figure 4.28.



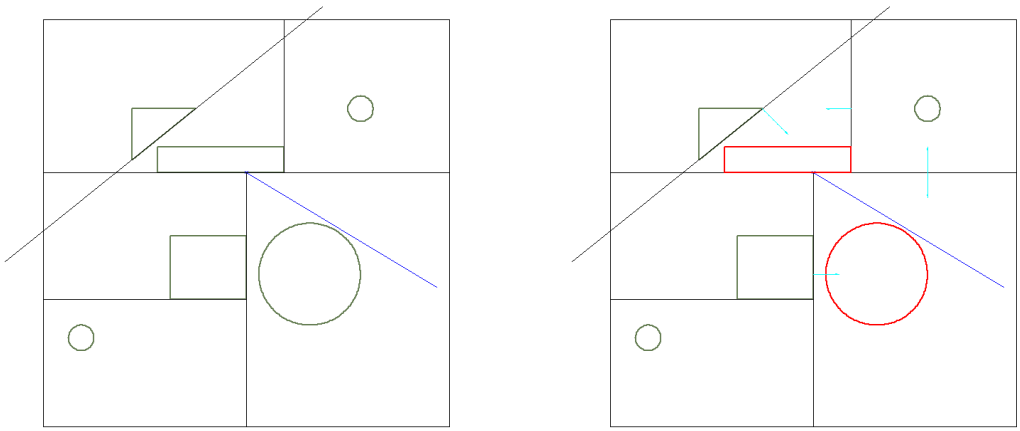
(a) OctTree structure (b) OctTree traversal

Figure 4.28.: Octrees require less nodes than regular grids to partition the space. Two objects were intersected during the traversal.

Binary space partitioning, Kd-trees

In these techniques space is subdivided hierarchicaly into two sub-spaces each step. This is achieved by choosing arbitrary planes as split objects. During the traversal, if the tree is well balanced, half of the objects will be discarded for intersection at each step.

Binary Space partitioning Trees (BSP) use planes that already exist in the geometry to subdivide space. Kd-trees restrict the partitioning planes to alternating planes that are perpendicular to the coordinate system axes. This allows the use of axis-aligned bounding boxes (often referred to as AABB) to construct the tree. In figure 4.29



(a) BSP structure (b) BSP traversal

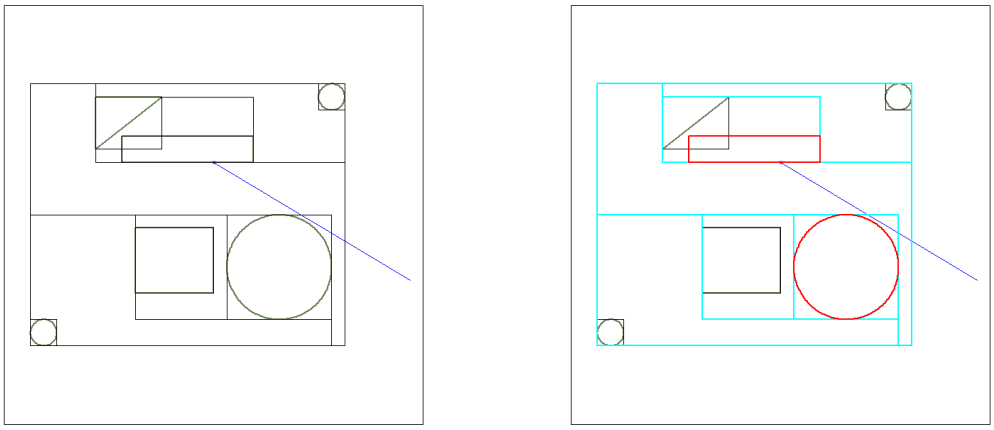
Figure 4.29.: BSP spatial subdivision. Two objects are intersected to find the closest intersection and 4 nodes visited

4.3.2. Partitioning object lists

These techniques divide the set of objects into disjoint groups. Each object is referenced at most once and check for intersection is computed only once.

BVH: Bounding Volume Hierarchy

It consists of a hierarchy of volumes enclosing one or more objects. When more than one object or volumes are stored in a node, the bounding box enclosing them becomes the parent bounding box. This volumes can overlap and therefore discarding one of the children nodes cannot be done easily at each step. If there is an overlap both children of the node have to be checked for intersection.



(a) BVH structure (b) BVH traversal

Figure 4.30.: BVH object division. Two objects are intersected and 6 nodes visited

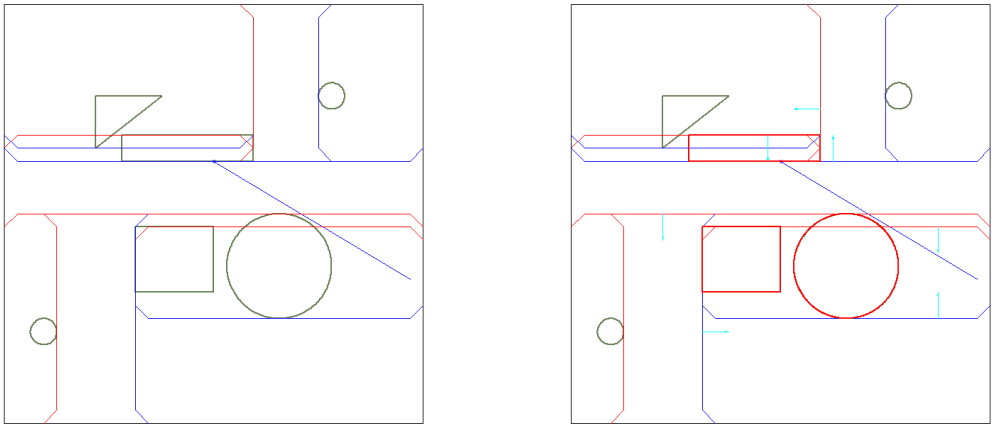
4.3.3. Hybrid aproaches

Bounding interval Hierarchy (BIH)

This technique can be classified as both space partitioning and object list partitioning. It is indeed a combination of kd-trees and bounding volume hierarchies.

While bounding volume hierarchies store a full axis aligned bounding box for each child, the idea of bounding interval hierarchies is to only store two parallel planes perpendicular to either one of x, y or z axis. Each of those planes defines the maximum (on the left side) and the minimum (on the right side) of two children volumes, therefore defining two bounding volumes and a gap in between.

Nodes are spatially ordered and thus the traveral algorithm is very similar to the kd-tree traversal. Also, it is possible to not intersect with any child if the ray traverses the empty space between two non-overlapping children.



(a) BIH structure (b) BIH traversal

Figure 4.31.: BIH space/object division. In this specific partitioning three objects are checked for intersection and 7 nodes are visited.

5. Design

An overview of the execution process of the raytracer is given in this chapter. We first explain the execution flow and how the raytracer code is inserted into Fluka. We continue by describing the interfaces with Fluka that the raytracer assumes. After we introduce the syntax for the configuration file, also called scene file. To finish we discuss the code organization with the help of a class diagram.

5.1. The off-line raytracer

The execution of the raytracer, similarly to the normal Fluka execution, follows these three stages: read the desired configuration, compute the image(s) and write the output(s). The code of the raytracer will eventually be merged with the main Fluka source code. The raytracer will be configured from the Fluka input file and its activation will be triggered by a card (a line) in the Fluka input.

As a first step towards the final integration with the Fluka source code we aimed at writing the raytracer using the normal extension mechanisms found in Fluka. Therefore the implementation that we describe here does not modify the Fluka source code, instead it is injected into the Fluka process by means of the Fluka user routines. The current implementation of the raytracer can be used to test its correctness and evaluate its performance.

We designed a strategy to insert the raytracer code into the Fluka executable. Two features were needed: an entry point for the raytracer and a mechanism to configure it.

The code can be injected into the Fluka process by means of the standard Fluka extension capabilities: the user routines. The USRICALL (as described in chapter 2) routine was chosen as the most appropriate for the needs of the raytracer.

Regarding the configuration of the raytracer, modifying the input file parser was discarded for the moment. Instead, the raytracer implements a parser that reads a separate configuration file containing the render settings. The content and semantics of this configuration file are described later.

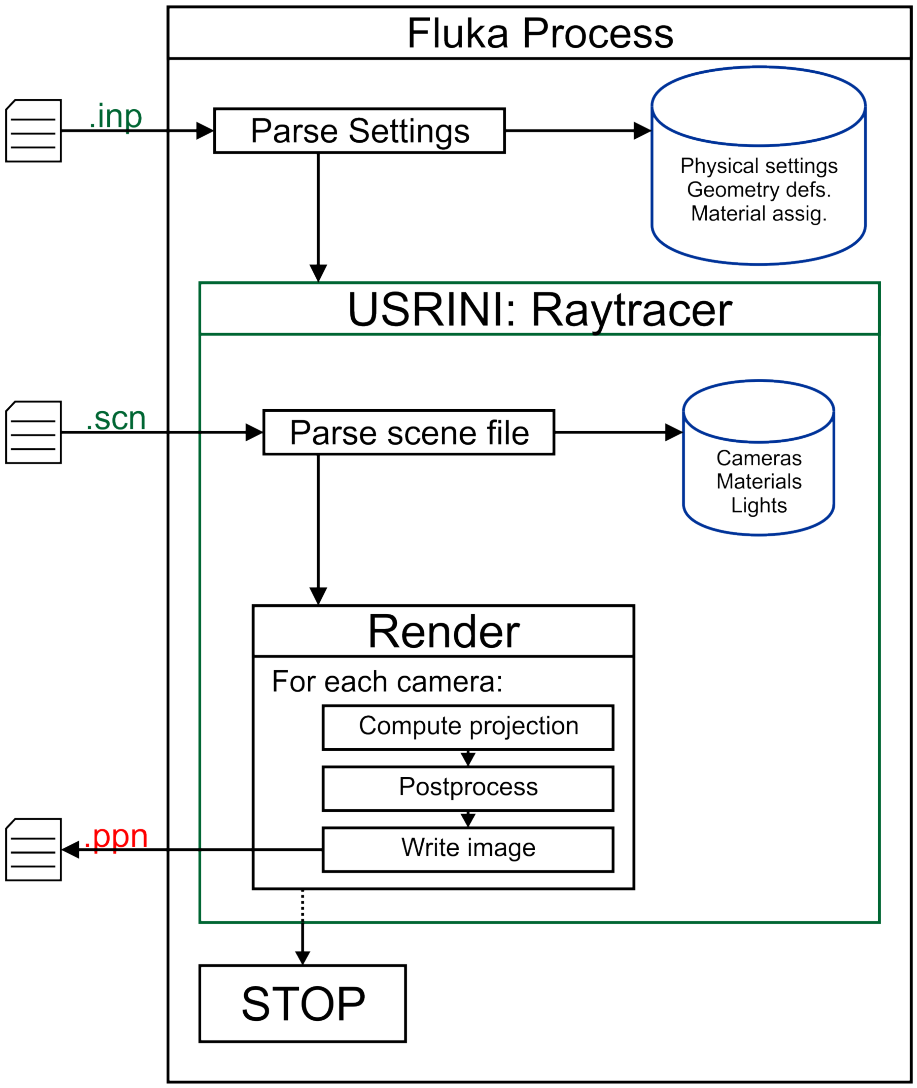


Figure 5.1.: Process execution

A diagram of the process execution is present in the flow diagram in figure 5.1. Fluka executes, reads the configuration settings and commands. Where it would normally find the START card it finds the USRICALL card that runs the raytracer. The raytracer in turn reads the "scene" configuration file, generates and writes to disk one image per camera and ends. Back to the regular fluka execution flow the STOP card is found in the input file instructing the Fluka process to end.

5.2. The USRICALL routine

The user routine employed to insert the raytracer code must be a routine that is executed only once and, if possible, one that can be enabled and disabled from the Fluka input file. One user routine that satisfies these requirements is the USRINI subroutine which is associated to the card USRICALL. This routine is executed each time the keyword USRICALL appears in the Fluka input file and can be disabled by simply commenting it out or removing the line. Replacing the START card by the USRICALL card alternates between a normal Fluka simulation and a raytracer execution.

5.3. Interfaces with FLUKA

The raytracer examines the Fluka geometry by means of the Fluka geometry engine. [1] In this section we review the routines available in Fluka to inspect the geometry.

The raytracer algorithm "tracks" a ray as it traverses the geometry, a process similar to tracking a particle created in a physical process. Therefore, Fluka already implements the geometry routines needed by the raytracer. The raytracer needs the following information about the geometry:

- *Next region crossing* Find the position in space (or the distance to the ray origin) where the ray crosses a region boundary.
- *Normal at the intersection point* The normal at the intersection point is needed to compute illumination
- *Current region of the ray* The visual properties of an object are given by the material of the region

There are three routines in Fluka that can be used to obtain that information: GEOFAR, GEOREG and GEONOR:

GEOREG is used to know to which region one point in space belongs

[1] The Fluka Geometry engine are the set of routines in Fluka that are used to navigate or inspect the geometry

GEOFAR is used every time it is needed the next intersection between a ray and the scene geometry.

GEONOR is used to obtain the normal vector at a given intersection.

The information needed by the raytracer can be computed by combining those three subroutines. The routine `GEOREG` can be used in order to know the region where a ray is originated, with that information the next region crossing can be computed by calling repeatedly to `GEOFAR` until the region where the ray moves changes. Once a region crossing is found (the intersection point), the normal can be requested by calling `GEONOR` with the intersection information.

5.4. The scene file

The scene file contains the specification of the images to be generated. This information cannot be placed in the Fluka input file and therefore the existence of this extra configuration file. The scene file must have the same name as the input file but instead of ending with `.inp` it must end with `.scn`. This file contains information about the visible properties of materials, the cameras requested to render and the position and properties of the lights.

Scene file Conventions

The scene file is the configuration file for the raytracer, it has been designed so that it is easy to write and understand. It is a plain text file that can be edited with any text editor. Three kind of blocks can be defined: camera, material and light. These blocks have to be separated by one or more empty lines.

The syntax of the scene file is described in this section, these conventions are used:

- Lines between square brackets `[]` are optional.
- Values between angles `<>` must be replaced by real values, `<one\two>|` means that either the string `one` or the string `two` can be used.
- Lines starting with a `#` are comments

Cameras

Each camera renders a different image of the geometry, they have different quality settings, resolution, position, etcetera. The following listing defines the properties of a camera and their default values.

```
camera
[file <filename.ppm>] // Default: render.ppm
```

```
    position <x.x,y.y,z.z>
    look_at <x.x,y.y,z.z> // Default 0.0,0.0,0.0
    [up <x.x,y.u,z.z>] // Up vector, default 0.0,1.0,0.0
    [projection_type <orthogonal|perspective>] // Default perspective
    [sampling_mode <linear|adaptative>] // Default linear
    [samples <s>] // Samples per pixel side (samples per pixel = s*s), default 1
    [background_color <r.r,g.g,b.b>] // Default 0.0,0.0,0.0
    [angle <a.a>] // Aperture angle, default 45.0
    [width <w>] // Width in pixels of the final image, default 640
    [height <h>] // Height in pixels of the final image, default 480
    [max_depth <m>] // Max ray depth, default 7
    [no_shadows <true|false>] // Default false
    [no_specular <true|false>] // Default false
    [no_fuzziness <true|false>] // Default false
    [min_density <d.d>] // Default 0.1
    [print_materials <true|false>] // Default false
    [postprocess <true|false>] // Default false
```

The name and extension of the file does not change the format of the output image: it is always a ppm image. Nevertheless, it can be easily converted to other formats by using the ImageMagick command `convert`, for example `convert render.ppm render.png`.

The `max_depth` setting limits the number of secondary rays that can be shot per sample, for a 0 no reflection or reflaction effects will be computed.

Lights

The lights define the sources of light of the scene. If there are no lights in the configuration file the scene will be in complete darkness.

The complete list of settings for the lights follows:

```
    light
[type <point|spot|ambient>] // Default point (spot are not implemented)
    [intensity <x.x>] // Default 1.0
    position <x.x,y.y,z.z>
    [color <r.r,g.g,b.b>] // Default 1.0,1.0,1.0
    [falloff <f.f>] // Default 0.85
    [no_shadows <true|false>] // Default false
    [no_specular <true|false>] // Default false
```

Materials

As explained in chapter 4, some properties of the materials,only used by the raytracer, have to be set. When the properties of a material are not defined a default light gray

material is used. The names of the materials must be the same as in the ASSIGNMA cards.

```
material
name <name> // The same name as in the input file
[diffuse_color <r.r,g.g,b.b>] // Default 0.9,0.9,0.9
[specular_color <r.r,g.g,b.b>] // Default 1.0,1.0,1.0
[ambient_color <r.r,g.g,b.b>] // Default 0.0,0.0,0.0
[transparency <t.t>] // Default 0.0
[reflectivity <r.r>] // Default 0.0
[specular_power <p.p>] // Default 20.0
[specular_value <v.v>] // Default 1.0
[ior <i.i>] // Index of refraction, default 1.0
[fuzziness <f.f>] // Fuzziness of the surface, default 0.0
[attenuation <a.a>] // Attenuation when light traversing the material
```

5.5. Fluka input file cards

An alternative configuration proposal has been developed and defined in section A.1. This is the syntax proposed for the Fluka input file, following the Fluka input file conventions.

5.6. Code organization

The raytracer is structured as several logical units: cameras, lights, materials, the image file, statistics and the raytracer algorithm. The code is written in an object-oriented style, each of the logical units is responsible of its own state or data.

In figure 5.2 a class diagram with the static structure of the program is shown. At the bottom some data definitions can be found, and there is a diagram of the interaction with the Fluka geometry engine on the right.

5.7. Interactive viewer

An interactive viewer was also implemented. A minimalistic Python graphical user interface was designed and implemented as a proof-of-concept.

The viewer is a Python program that runs as a separate process. This process has to send commands to the raytracer process in order to change the settings of the render as camera position, quality, etcetera. A different fluka executable is produced for the

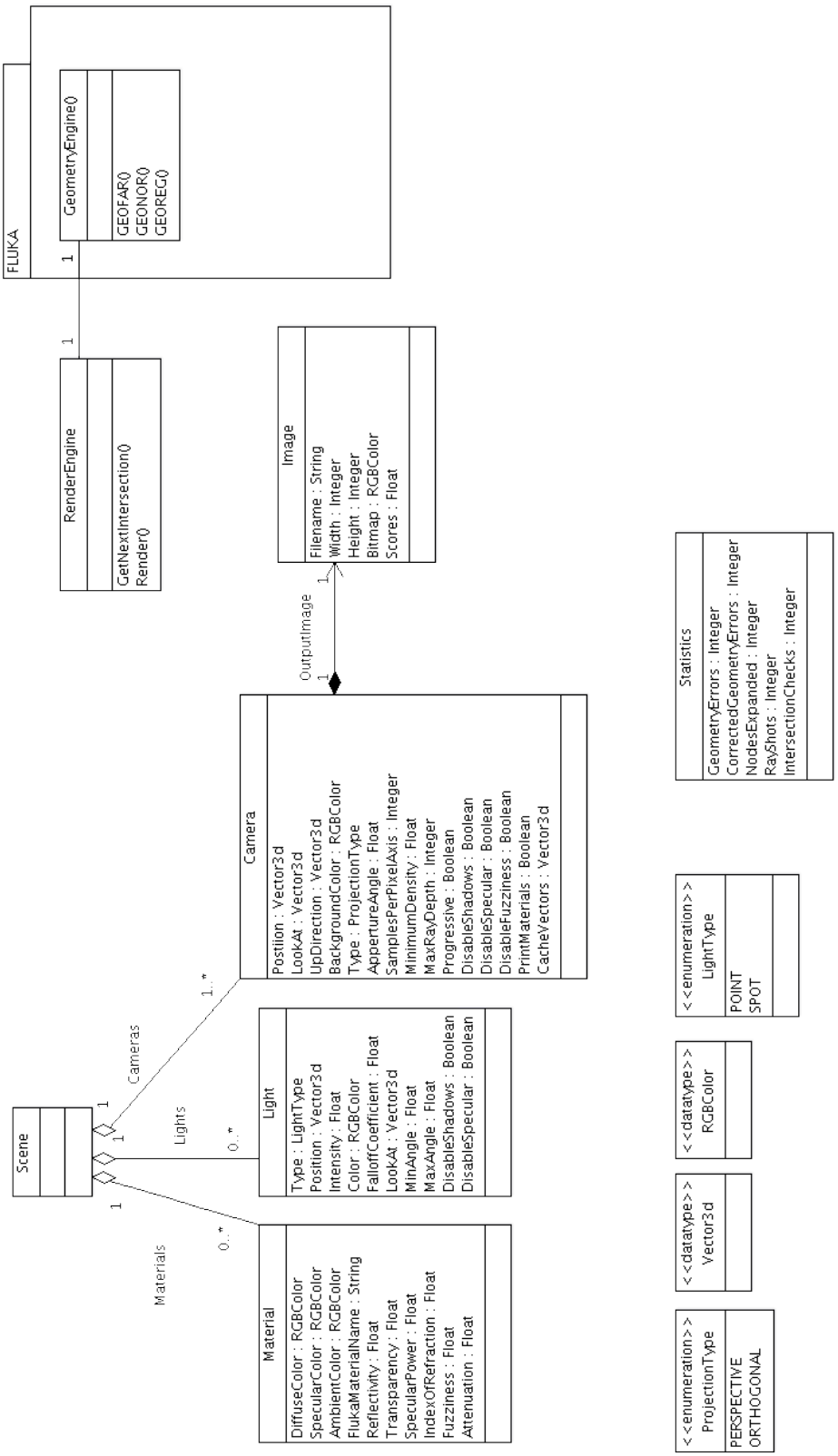


Figure 5.2.: Project Structure

real-time rendering, `flukartrt`, it handles the incoming commands from the Python process and sends back the partial image as it is rendered.

The two process need to exchange some data: the render settings and the output image. Two named pipes are used for this purpose, one of them (`rt_control`) is used to send commands from the Python GUI to the renderer process and the second (`rt_image`) is used to send to the viewer the partially rendered image. A diagram showing these interactions is shown in figure 5.3.

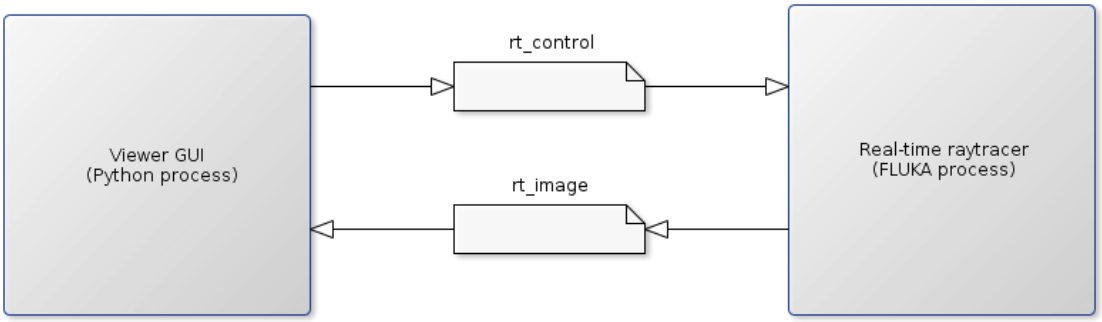


Figure 5.3.: Component diagram

The Fluka process receives commands from the Python process by means of the `rt_control` pipe. The commands conform a small protocol that both processes understand. The next listing contains the description of this protocol:

```
start
pause - Pause the render, still can be resumed *
resume - Resume a paused render *

image_size <width>,<height> - Set the output image size desired *
block_size <bside> - Set the side of the first block in pixels *

position <x>,<y>,<z> - Move the camera to that point
look_at <x>,<y>,<z> - Change the look at position of the camera

shadows <on/off> - Enable/disable shadows
specular <on/off> - Enable/disable specular lighting
fuzzyness <on/off> - Enable/disable surface fuzziness
maxdepth <num> - Sets the maximum ray depth of secondary rays
```

Figure 5.4.: rt_control specifications. Note that the commands marked with asterisks are not implemented yet

This small list of commands are the commands needed to navigate the geometry interactively. The Python program keeps track of the position of the camera and updates its position after the user command. The new position is sent to the fluka process through `rt_control`. Every time the Fluka process receives a message it stops the current render

and handles the message sent by the viewer. When the new settings have been sent to the renderer the Python process sends a "start" command and the renderer starts processing the image. The image is first rendered at low quality and next iterations improve its quality. These intermediate images are sent to the Python process throught the `rt_image` as soon as they are computed. Figure 5.5 shows a diagram showing the interaction of the raytracer and the GUI.

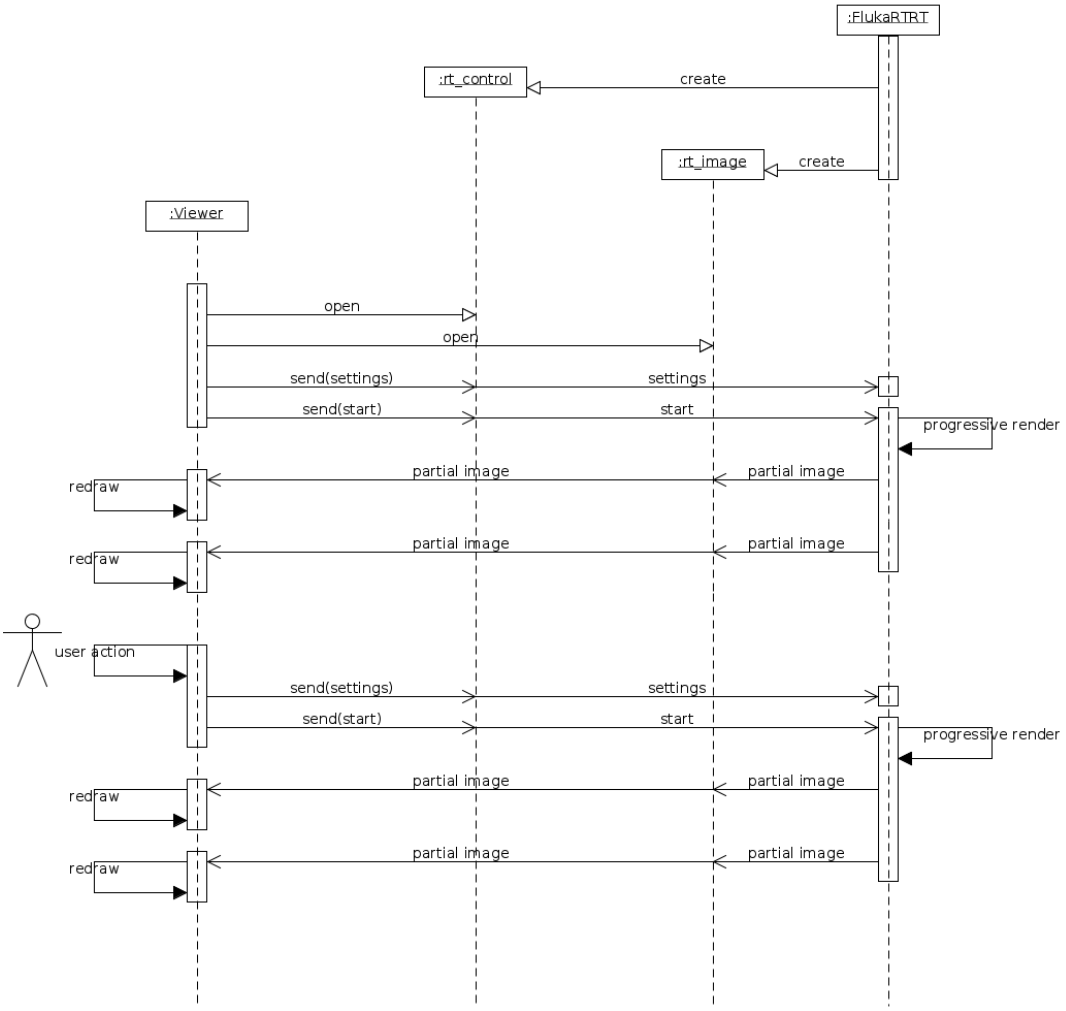


Figure 5.5.: Process execution

6. Implementation

This section contains a description of the implementation of the raytracer and shows the process followed to design the algorithms. The algorithms are presented in pseudocode which should be descriptive enough to explain them without the burden of using real code. A very simple implementation of a raytracer is constructed at the beginning and it is iteratively improved until it is very close to the final implementation.

In the second part of this chapter we discuss about the implementation details of the interactive viewer along with the new algorithms developed specifically.

6.1. The ray-tracer

As described in the previous chapter, the raytracer is part of the Fluka process. When Fluka starts, loads the configuration and calls the raytracer code. The raytracer in turn loads its configuration into memory and procedes to the rendering of the images.

Each camera in the configuration generates an image of the world as it appears from its position. The camera definition is used to describe an imaginary plane where the geometry is to be projected. This is illustrated in figure 6.1. The extents of each pixel are computed by subdividing the projection plane by the requested vertical and horizontal resolution of the image. In order to compute the projection of the geometry on this imaginary plane one or several rays are shot passing throught the dimensions of each pixel. Each of these rays sample the geometry: they are used to compute the color of that pixel. The simplest way to sample the image is to shoot one ray passing through the center of each of the pixels of the image.

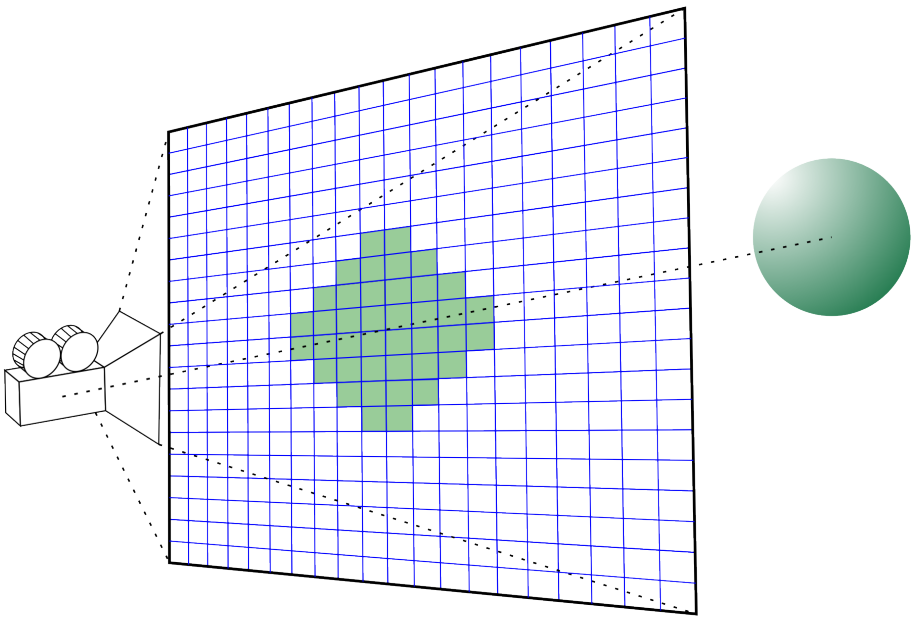


Figure 6.1.: Geometry projection

In cameras with perspective projection the rays are originated at the camera position and and pass through the pixels on the projection plane. In orthographic cameras the rays are originated at the center of each pixel and all have the same direction which is the look at vector of the camera. More details about the types of cameras can be found in chapter 4. Algorithm 1 shows how the primary rays for each pixel are generated. Note that the job of generating the rays is delegated to the camera as it is the camera that knows how to generate the rays for each pixel depending on its type and settings.

Algorithm 1: Basic Raytracer Algorithm

Input: screen_width, screen_height, samples
Result: A matrix (file) with all the screen colors
foreach *pixel* $p_{x,y} \in$ *[1,screen_width]*×*[1,screen_height]* **do**
    ray ← Camera.rayTo(x, y);
    pixelColor[x,y] ← Shoot(ray);
end

The Shoot routine is used to compute the output color of a ray. A basic version that does not implement reflection or refraction is shown below. This routine computes the first intersection of a ray with the geometry and calls ColorCompute. ColorCompute will compute the specular and diffuse colors of the ray for the intersection point. ColorBlend blends the two colors and the result is returned.

Algorithm 2: Shoot(ray)

Input: ray
Result: The color seen by the ray

I ← get_next_intersection(ray);
diffuse_color, specular_color ← ColorCompute(ray, I);
current_color ← ColorBlend(diffuse_color, specular_color, ratio);
return current_color;

The ColorCompute routine computes the Phong shading function (as explained in 4.2.2) for a given ray and intersection. In order to compute if one light affects the color of a point, one ray is shot from the point towards the position of the light. If the ray does not intersect any opaque object before reaching the light its effect is added to the color of the point. Discarding lights that are not visible from the intersection point produces the shadows in the final image.

A typical example is depicted in figure 6.2 where two light sources illuminate the scene. Light A reaches the intersection point and therefore its effect is added to the point. On the other hand light B doesn't reach the intersection point and doesn't change the color of the intersection.

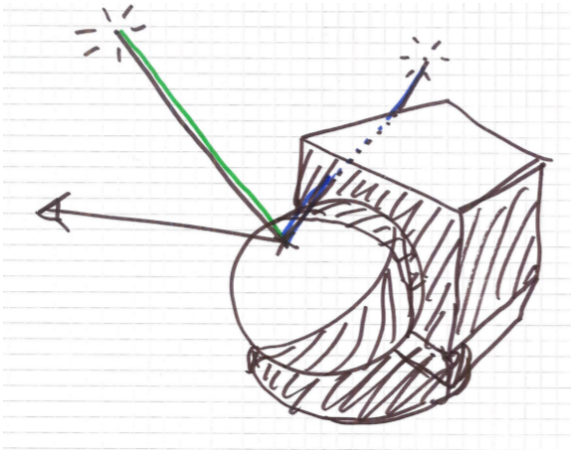


Figure 6.2.: Primary ray color

Algorithm 3: ColorCompute(ray, I): Compute the color of the current ray and intersection

```
Input: ray, intersection
Result: The diffuse and specular colors
if ray didn't intersect anything then
    diffuse_color ← Camera.BackgroundColor(ray);
else
    diffuse_color ← ray.new_material.ambient_color;
    specular_color ← BLACK;
    foreach light in the Scene do
        if the light is illuminating the surface (there is no object between the
        intersection point and the light source) then
            diffuse_color ← diffuse_color + DiffuseColor(light, intersection, normal,
            angle, attenuation...);
            specular_color ← specular_color + SpecularColor(light, intersection,
            normal, angle, attenuation...);
        end
    end
    Avoid oversaturation of diffuse and specular colors;
end
return diffuse_color, specular_color;
```

These simple algorithms already constitute a ray-tracer that can render a scene with simple materials. In the rest of the chapter we improve the raytracer by adding more features to these algorithms. One of the next steps is to smoothen the resultin image by super-sampling the scene. We continue improving these algorithms by introducing reflection and refraction.

Sampling strategies

Computing only one sample per pixel produces images with sharp borders, sharp shadows and artifatcs for patterns that are smaller than one pixel. To reduce these undesired effects multiple samples per pixel have to be taken and averaged. Figure 6.3 shows these kind of artifacts.

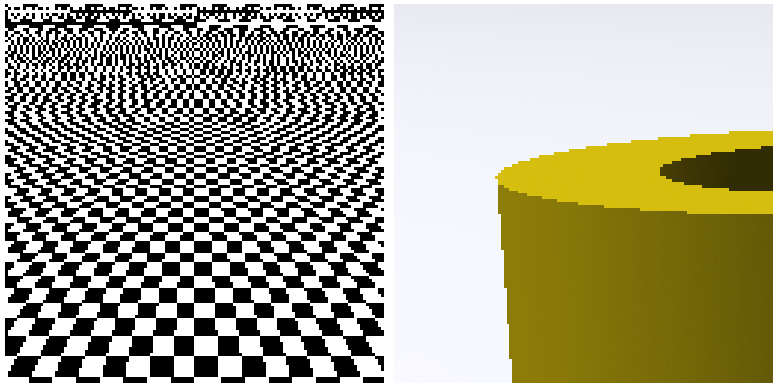
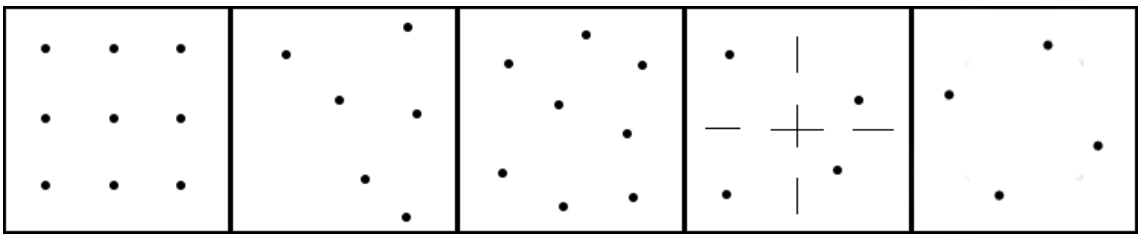


Figure 6.3.: Sampling artifacts

Several strategies can be chosen in order to decide where and how many samples of a pixel should be taken. For example, pixels can be subdivided in equally sized sub-pixels and rays shot passing through the center of these. Another strategy is to shoot rays throught randomly distributed points within the pixel until the statistical error is small enough.



(a) Grid (b) Random (c) Poisson disk (d) Jittered (e) Rotated grid

Figure 6.4.: Sampling strategies [1]

Algorithm 4 shows a solution that divides the original pixel in equal subpixels.

Algorithm 4: Linear Sampling Algorithm: Computes the color of a point averaging the color of several samples

Input: screen_width, screen_height, samples
Result: A matrix (file) with all the screen colors
foreach *pixel* $p_{x,y} \in$ *[1,screen_ width]*×*[1,screen_ height]* **do**
    pixelColor[x,y] ← BLACK;
    foreach $u \in$ *[1, samples],* $v \in$ *[1, samples]* **do**
        ray ← Camera.rayTo(x + u/samples, y + v/samples);
        color ← Shoot(ray);
        pixelColor[x,y] ← pixelColor[x,y] + color/samples^2;
    end
end

Subdividing each pixel into $n \times n$ subpixels has a direct impact on the performance of the raytracer: the image is rendered $n \times n$ times. It is interesting to approximate which pixels of the image require more samples and which are accurate enough by taking less samples. For this reason an adaptive algorithm was developed, shown in Algorithm **??**.

An adaptive technique, as the one in Algorithm 5 requires two phases: the first assigns a score to each of the pixels and the second computes a varying number of samples based on the score difference between a pixel and its neighbours. A simple score function was developed as shown in Algorithm 6

Algorithm 5: Adaptative Sampling Algorithm: Computes the color of a point supersampling it only if the neighbour pixels are different enough

Input: screen_width, screen_height, samples, threshold (minimum score difference needed to shoot more rays, for example 20)
Result: A bitmap with the resulting image

```
// First pass, assigns a score to each pixel on the screen plus one
   extra pixel on each edge
```
foreach *pixel* $p_{x,y} \in$ *[0,screen_width+1]×[0,screen_height+1]* **do**
  pixelScore[x,y] ← Score(ray);
end

```
// Estimate how many samples are needed for each pixel and shoot them
```
foreach *pixel* $p_{x,y} \in$ *[1,screen_width]×[1,screen_height]* **do**
  `// Compute differences with neighbours`
  points ← $\sum_{u=-1}^{1} \sum_{v=-1}^{1}$ |pixelScore[x,y] - pixelScore[x+u,y+v]|;
  if *points > threshold* **then**
    pixelColor[x,y] ← BLACK;
    `// Split the pixel in samples × samples sub-pixels, sample and average`
    foreach *u ∈ [1, samples], v ∈ [1, samples]* **do**
      `// Find ray passing through the center of the subpixel`
      ray ← Camera.rayTo(x + u / samples, y + v / samples);
      color ← Shoot(ray);
      `// Average color`
      pixelColor[x,y] ← pixelColor[x,y] + color / (samples^2);
    end
  else
    ray ← Camera.rayTo(x, y);
    pixelColor[x,y] ← Shoot(ray);
  end
end

Algorithm 6: Score(ray)

Input: ray
Result: An score value for the ray: Estimation of the object/detail found
score ← 0;
foreach *node expanded* **do**
  score ← score + 2;
end
foreach *light illuminating the object* **do**
  score ← score + 100 ∗ light_number;
  score ← score + 10 ∗ angle ∗ light_shade;
end
score ← score + 10000 ∗ first region_number traversed;
return *score*

Reflection and Refraction

The realism of the images can be improved by computing the effect of reflective and refractive surfaces. In that case if the material of the object is reflective or refractive the output color of the ray should include the effect of the reflected and/or reracted rays.

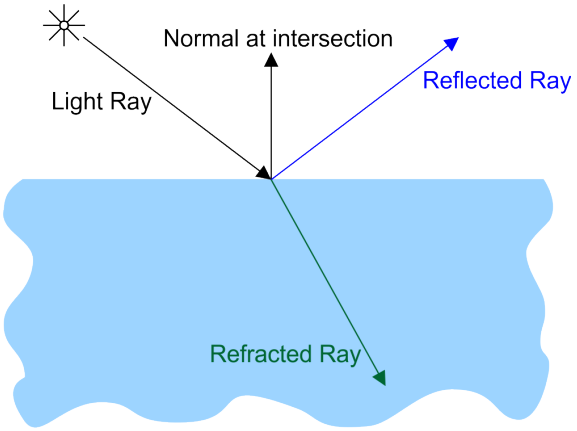


Figure 6.5.: Secondary rays

Rasterization based software has to usually implement a handful of tricks in order to add reflection and/or refraction to the rendered image. In the case of ray-tracing, reflection and refraction are natural effects that follow from the general idea of shooting rays. The output color of a ray will depend on this case also on the color of the reflected and refracted rays.

A reviewed version of the shoot routine is provided in Algorithm 7. This routine, apart from computing the Phong shading, computes the reflected and refracted rays at the intersection point and recursively calls itself to obtain the color seen by those rays. When the color of those rays is computed, it is blend into the final color of the current ray.

Algorithm 7: Shoot(ray)

Input: ray
Result: The computed pixel color

```
I ← get_next_intersection(ray);
diffuse_color, specular_color ← ColorCompute(I);
if intersection material is reflective then
    diffuse_color ← ColorBlend(diffuse_color, Shoot(reflect_ray(ray, I)), ratio);
end
if intersection material is refractive then
    diffuse_color ← ColorBlend(diffuse_color, Shoot(refract_ray(ray, I)), ratio);
end
current_color ← ColorBlend(diffuse_color, specular_color, ratio);
return current_color;
```

The companion ColorCompute routine is shown in algorithm 8. This update of the function includes the computation of the reflected and refracted colors.

Algorithm 8: ColorCompute(ray): Compute the color for the current ray given all the secondary rays already computed

```
Input: ray
Result: The output color of the ray
if ray didn't intersect anything then
    ray.color ← BackgroundColor(ray);
else
    diffuse_color ← rayintersection.ambient_color;
    specular_color ← BLACK;
    foreach light in the Scene do
        if the light is illuminating the surface (there is no object between the
        intersection point and the light source) then
            diffuse_color ← diffuse_color + DiffuseColor(light, intersection, normal,
            angle, attenuation...);
            specular_color ← specular_color + SpecularColor(light, intersection,
            normal, angle, attenuation...);
        end
    end
    Avoid oversaturation of diffuse and specular colors;
    color ← BlendColors(diffuse, specular, reflected_ray.color (if exists),
    refracted_ray.color (if exists));
end
return ray.color;
```

These secondary rays can also require the expansion of more rays that will in turn contribute to the color of the original ray. This process is sketched in figure 6.6. The height of this tree can be infinite, but at a certain level computing more nodes barely affects the quality of the final image. Two values can be used to decide when to stop generating secondary rays: the height of the tree or the overall effect of the ray on the final color. The impact of the node on the final color can be computed following the blending chain, if the first reflected ray accounts for 20% of the color of the primary ray and the reflected ray from the reflected ray accounts for 20% of the reflected ray color it adds only a 4% to the original color. In the code, if any of these values is out of certain range the computation is stopped.

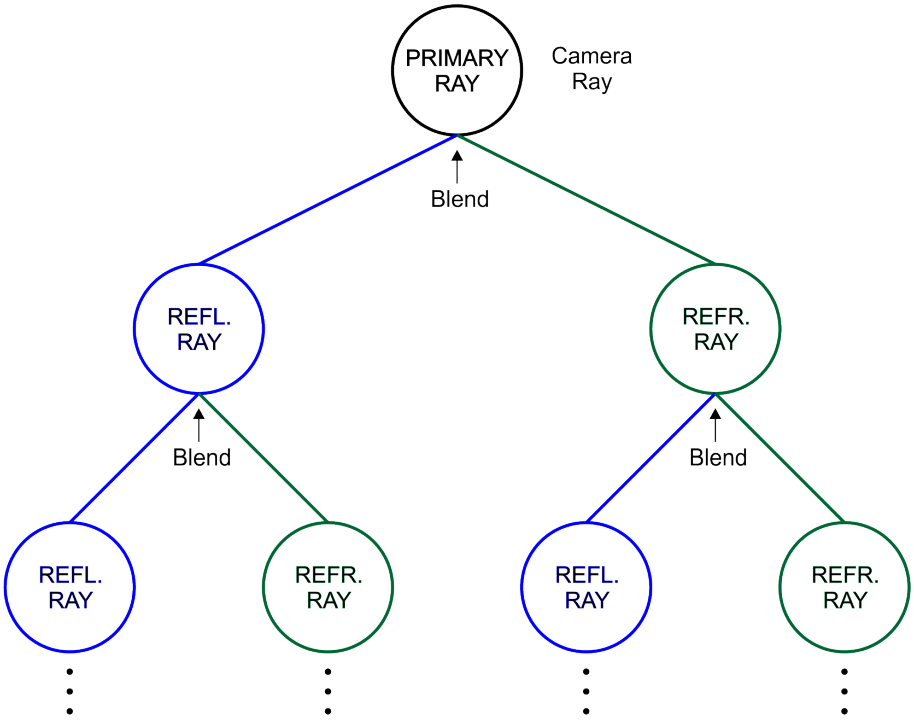


Figure 6.6.: Hierarchy of rays, reflected rays are painted in blue and refracted rays in green

Note that this function cannot be implemented in fortran 77 because recursion is not supported. Usually the recursive execution of a function is supported by the process stack. Each time a function is called the arguments and the return address are pushed onto the stack and the program execution jumps to the code of the function. When the function ends, the local variables, the arguments and the return address are unstacked and the program execution continues from the return address.

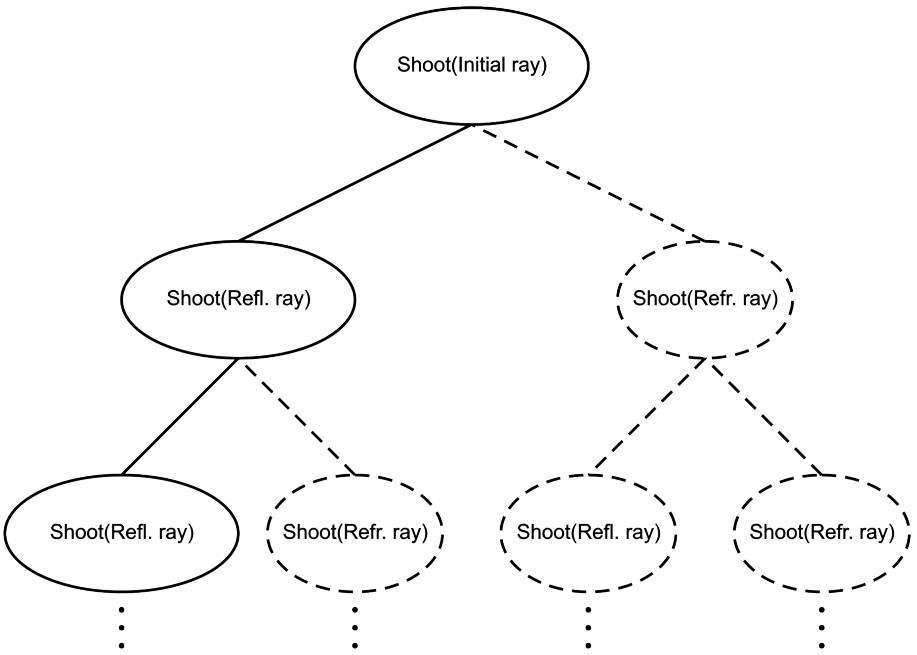


Figure 6.7.: Recursive execution: The execution of this algorithm spawns a tree

All recursive functions can be converted to iterative by using a stack handled by the program. As shown in Diagram 6.7 shoot will spawn at most two calls to itself. A binary tree can be used to support the iterative version of the shoot routine.

By using a binary tree the primary ray is stored in the root node, the secondary rays needed to compute the color of the primary ray are stored in children nodes in the tree. This process continues (depth-first) evaluating children nodes and creating more children as needed. The color of a node can only be computed when the color of all its children are computed. Eventually the secondary rays will stop hitting reflective or refractive surfaces that spawn new nodes and the algorithm will start processing the parent nodes. If that does not happen the artificial limits will apply.

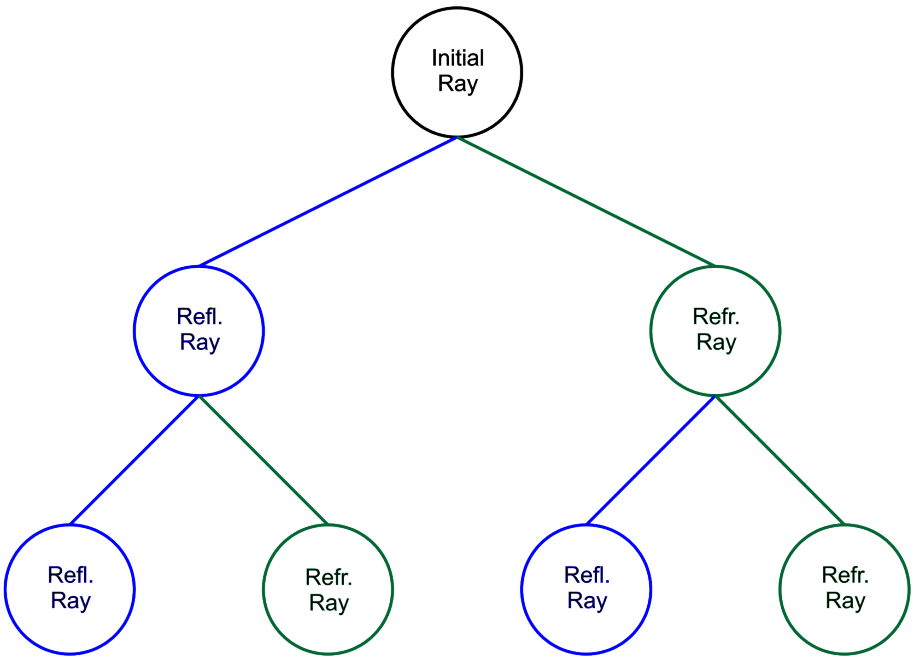


Figure 6.8.: Ray tree

The Shoot algorithm described in Algorithm 9 implements the iterative version of the shoot routine by using a binary tree.

Algorithm 9: Shoot(ray): Iterative Shoot algorithm for Raytracing

```
Input: ray
Result: The computed pixel color
current_node ← create_node(ray);
S ← {current_node};
repeat
    // Do while there are more than 1 node in the stack/tree
    if current_ node was not expanded then
        mark_expanded(current_node);
        I ← get_next_intersection(ray);
        if I ≠ ∅ then
            if material of intersection I is reflective then
                // Create reflected ray + node
                newray ← reflect_ray(current_node ray, I);
                reflected_node ← create_node(newray);
                S ← S ∪ {reflected_node};
            end
            if material of intersection I is refractive then
                // Create refracted ray + node
                newray ← refract_ray(current_node ray, I);
                refracted_node ← create_node(newray);
                S ← S ∪ {refracted_node};
            end
        end
        if Reflection or Refraction nodes created then
            // Iterate over the last created node
            current_node ← last node created;
        else
            // Do nothing:  Iterate again over the current_node
        end
    else
        // Children colors are already computed, use their color and
           remove them from the tree
        current_node.color ← ColorCompute(current_node);
        S ← S − { current_node.reflected_child, current_node.refracted_child };
        if current_ node has left sibling then
            current_node ← current_node.left_sibling;
        else
            current_node ← current_node.parent;
        end
    end
    // If only the primary node is left, the color is already computed
until size(S) = 1;
return current_node.color;
```

Algorithm 10: ColorCompute(node): Compute the color for the current node given all the subnodes already computed

```
Input: node
Result: The output color of the ray
if node.ray didn't intersect anything then
    node.color ← BackgroundColor(node.ray);
else
    diffuse_color ← node.intersection.ambient_color;
    specular_color ← BLACK;
    foreach light in the Scene do
        if the light is illuminating the surface (there is no object between the
        intersection point and the light source) then
            diffuse_color ← diffuse_color + DiffuseColor(light, intersection, normal,
            angle, attenuation...);
            specular_color ← specular_color + SpecularColor(light, intersection,
            normal, angle, attenuation...);
        end
    end
    Avoid oversaturation of diffuse and specular colors;
    node.color ← BlendColors(diffuse, specular, reflected_node.color (if exists),
    refracted_node.color (if exists));
end
return node.color;
```

The binary tree used to support the computation is called raystack in the source code. As there is no dynamic memory allocation in Fortran 77, the raystack must have a fixed maximum size that is allocated when the process starts executing. The raystack is actually stored in an array and indexes are used to point to the children nodes. The memory layout of the raystack structure looks like shown in image 6.9.

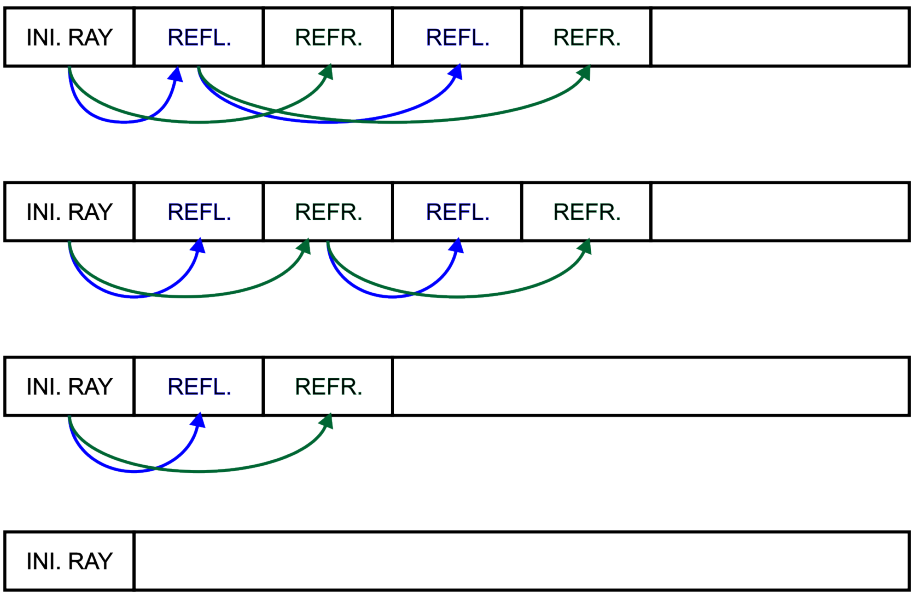


Figure 6.9.: State of the raystack at different points of execution of the raytarcer. Reflected nodes are painted green and refracted nodes blue

Cache usage

The execution of performance analysis tools showed that the processor cache memory was not being used optimally: many memory accesses lead to cache misses [1].

For example, the diffuse color of a material was defined as `real*8 matdiff(MATMAX, 3)` in a common file, this defines a matrix of `MATMAX*3` real variables. To evaluate the implications of the data definitions one has to look at the way these data are used. The three components of each color are usually accessed sequentially. Considering this, these variables can be placed together in order to improve the data locality and reduce the number of cache misses.

Unlike in C programming language matrices in FORTRAN are defined in a column major order. In the case of the diffuse color of the material all the red components are placed together followed by all green colors and then all blue colors. If the size of MATMAX is big enough each of the components will fall into different pages of memory, and accessing the three components of a color sequentially will cause up to three L2 cache misses: one for each component. Accessing these values is not only slower than it should be but also can cause pushing other important data off the cache.

The performance was improved by swapping the order of the components of matrices, for example the diffuse color of the materials is now defined as `real*8 matdiff(3,`

[1] A cache miss occurs when the data needed by a program is not present in the CPU's internal cache memory and has to be loaded from RAM. CPU cache memory is much faster thant RAM memory but much expensive and smaller. Processors use smart algorithms to make memory access as fast as possible by storing replicas of data used frequently in the cache. When data is not present in the cache the data is loaded together with a block surrounding it.

MATMAX), reducing the amount of L2 cache misses to one third for the typical access pattern. If the data are not in the cache it will probably load the three components at once, avoiding the two other cache failures.

6.2. Interactive viewer

The interactive viewer, already introduced in section 5.7 is composed of two programs: the python interactive GUI and a special Fluka executable called `flukartrt`. The algorithms developed for the customized Fluka executable are described in this section.

The real time raytracer has to respond to incoming messages as sent by the Python GUI. The control flow is different as it has to check if there are incoming commands and act accordingly. The two processes must be able to send messages to each other, for which unix named pipes are used. They act as FIFO queues: the messages are dispatched at the same order that they were sent. Unix named pipes exist in the filesystem like regular files, the API for reading and writing them is exactly the same as for regular files, only the creation process is different.

The real-time Fluka process creates two FIFO files at initialization phase: `rt_control` and `rt_image`. `rt_control` is used to pass the control messages to the Fluka process and is only written by the viewer. `rt_image` is used to send the rendered images to the viewer and is only written by the raytracer.

The raytracer must continuously check if there are incoming commands in the control file. Two points in the execution read from the control file: when the raytracer is idle waiting for new commands and while it is rendering. In the first case the process is blocked until incoming commands are ready, while in the second case it only needs to check for the presence of commands to abort the current operation. If the render is taking place and an incoming command is detected the render is stopped and the control flow goes back to the main loop which will handle the message.

FORTRAN 77 does not provide functions to deal with FIFO files nor the kind of control needed for the communication protocol, for this reason a small C library was written. The library is distributed in the source code under the directory `rt_control`. Two important functions are provided: `rt_control_input_wait()` and `rt_control_input()`, the former provides the blocking read and the second the non-blocking read.

The viewer

A few features were implemented in the Python viewer. The point of view can be rotated around the `look_at` point of the camera, panned (translating the camera parallel to the projection plane) and the zoom level can be adjusted. A few quality settings can be modified using the function keys. A list of shortcuts follows:

drag with Left Mouse Button Rotate the camera around the look_at point

drag up/down with Right Mouse Button Zoom in/out

F1 Enable/disable shadows

F2 Enable/disable specular reflections

F3 Enable/disable fuzziness of materials

F5 Increase maximum depth of secondary rays

F6 Decrease maximum depth of secondary rays

Algorithms

In this section we discuss the implementation of the algorithms implemented for the real-time raytracer.

The execution flow is now different, the renderer has to wait for incoming commands and only start rendering once it receives the command "start". The outer loop of the raytracer was modified to handle incoming messagesas sketched in Algorithm 11.

Algorithm 11: Progressive raytracer

```
Initialize/open control fifo file;
done ← false;
repeat
    // Check if there are new input commands (blocking)
    if rt_control_wait_input() then
        command ← rt_control_get_command();
        switch command do
            case "start"
                // Re-initialize camera projection and render
                Camera.init_projection();
                progrender();
            endsw
            case "exit"
                done ← true;
            endsw
            otherwise
                // Parse the input command and act properly:  move camera,
                   change quality settings, etc.
                parse_line(command);
            endsw
        endsw
    end
until done;
Close control fifo file;
```

The rendering operation is performed by the function ProgRender, which is in charge of generating the different iterations improving the resolution of the render. Each of these iterations will render the image at increasing resolution. The image is rendered in macro pixels, that is, it samples one pixel of a block of $n \times n$ pixels and fills the block with that color. In the next iterations the number of pixels on the side of the block is divided in half, taking four samples for the same block instead. In fig. 6.10 we show the iterations needed for rendering one scene starting from blocks of 32×32 pixels to blocks of 1×1.

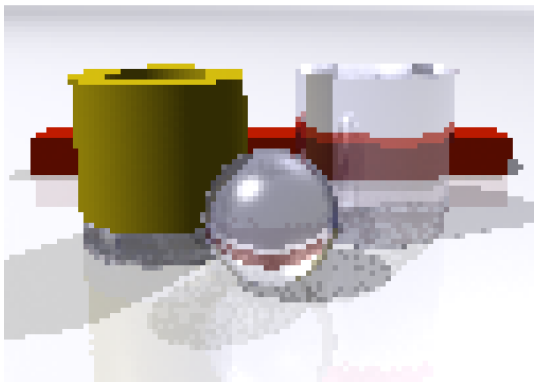
Algorithm 12 shows how the size of the block is updated at each iteration. Note that this algorithm also tries to read from the control pipe, but in this case it uses the non-blocking read, checking for the presence of new commands.

Algorithm 12: Progrender(): Progressive rendering

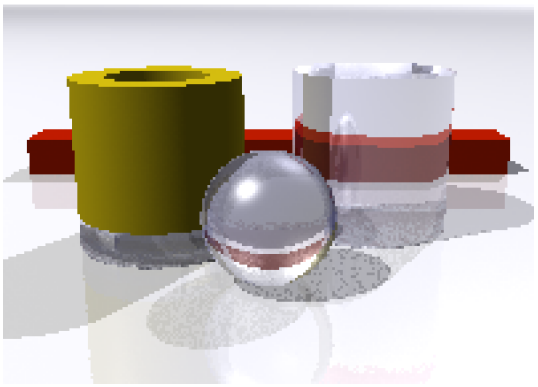
```
Result: The bitmap image
// Initial size in pixels of the side of a block
blockside ← 32;
// Until blocks of 1 × 1 pixels are computed
while blockside <> 0 do
    SpiralRender(blockside);
    Write image;
    if rt_control_input() then
        // Abort current computation and give control to upper function
        break;
    end
    blockside ← blockside / 2;
end
```

(a) Block of 32x32

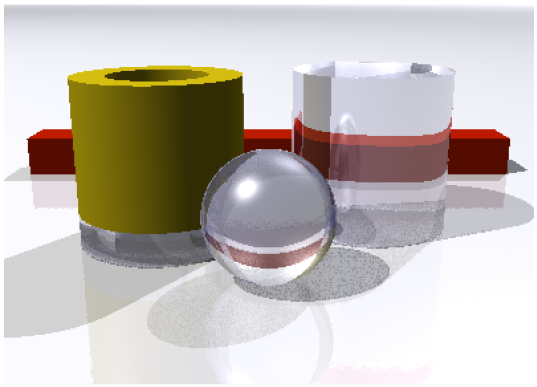
(b) Block of 16x16



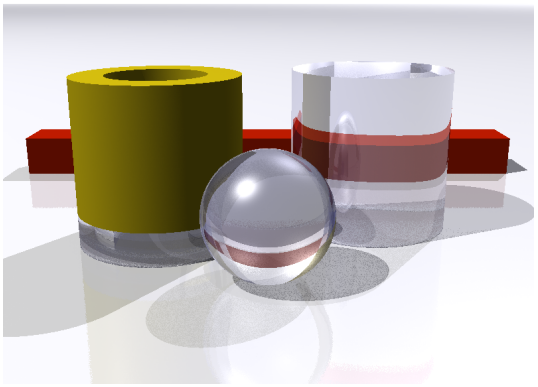
(c) Block of 8x8



(d) Block of 4x4



(e) Block of 2x2



(f) Block of 1x1

Figure 6.10.: Partial renders

The previous algorithm calls the function SpiralRender, this routine replaces the first algorithm described in this chapter (Algorithm 1), modifying the order in which the pixels (blocks) of the image are rendered. It renders the image from the center to the borders following a spiral shape (therefore the name *SpiralRender* algorithm 13). This

change was made in order to improve the perceived speed of the interactive viewer: the most important part of the image is rendered first, supposedly the center.

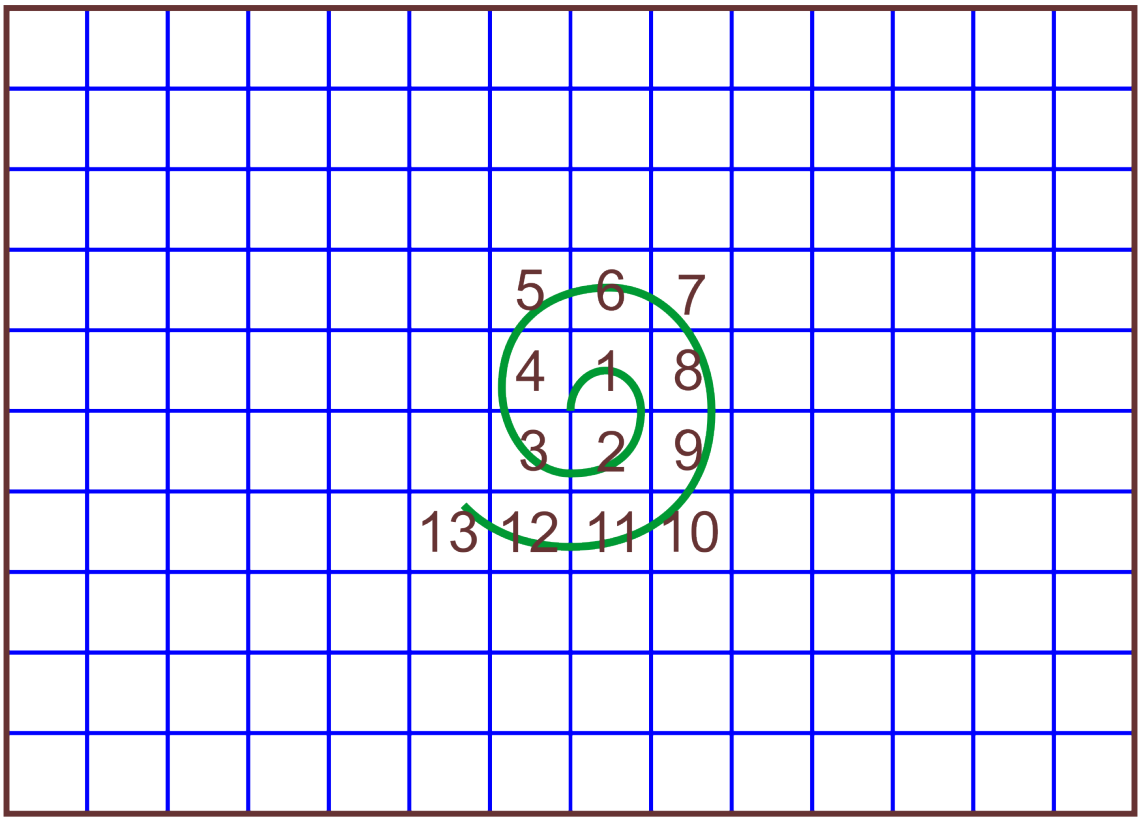


Figure 6.11.: Order of blocks rendered by SpiralRender

Algorithm 13: SpiralRender(blockside): Render the image from the center to the borders

Input: blockside: side in pixels of the block being rendered
Result: A matrix with all the screen colors

```
iterations ← 0;
lx ← ceil(current_camera.width / blockside);
ly ← ceil(current_camera.height / blockside);
if lx and ly are odd values then
    BlockShoot(Center block x from, Center block x to, Center block y from, Center
    block y to);
end
while not image completed do
    foreach block ∈ right side of current center block do
        BlockShoot(block.left, block.right, block.top, block.bottom);
        if rt_control_input() then exit from this function;
    end
    foreach block ∈ bottom side of current center block do
        BlockShoot(block.left, block.right, block.top, block.bottom);
        if rt_control_input() then exit from this function;
    end
    foreach block ∈ left side of current center block do
        BlockShoot(block.left, block.right, block.top, block.bottom);
        if rt_control_input() then exit from this function;
    end
    foreach block ∈ top side of current center block do
        BlockShoot(block.left, block.right, block.top, block.bottom);
        if rt_control_input() then exit from this function;
    end
    // Write the image file every 10 iterations
    if iterations >= 10 then
        iterations ← 0;
        image_write();
    else
        iterations ← iterations + 1;
    end
    if rt_control_input() then exit from this function;
    Grow center block in 1 block size on each border
end
```

This algorithm in turn relies on BlockShoot 14, which shoots one ray per block of pixels and paints them with the output color.

Algorithm 14: BlockShoot(left, right, top, bottom): Compute and store the color of a block of pixels

Input: ray

```
xdisp ← ( left + right ) / 2;
ydisp ← ( top + bottom ) / 2;
ray ← Camera.rayTo( xdisp, ydisp );
color ← Shoot(ray);
foreach x ∈ [left, right], y ∈ [top, bottom] do
  | pixelColor[x,y] ← color;
end
```

Algorithm 14 computes some of the pixels more than once. Figure 6.12 shows a picture where each pixel encodes with color the number of times that it is computed. An implementation avoiding the redundant sampling is shown in Algorithm 15. The improved version stores a flag for each pixel indicating if it has been computed or not. This change reduces the number of rays shot by a 25%.

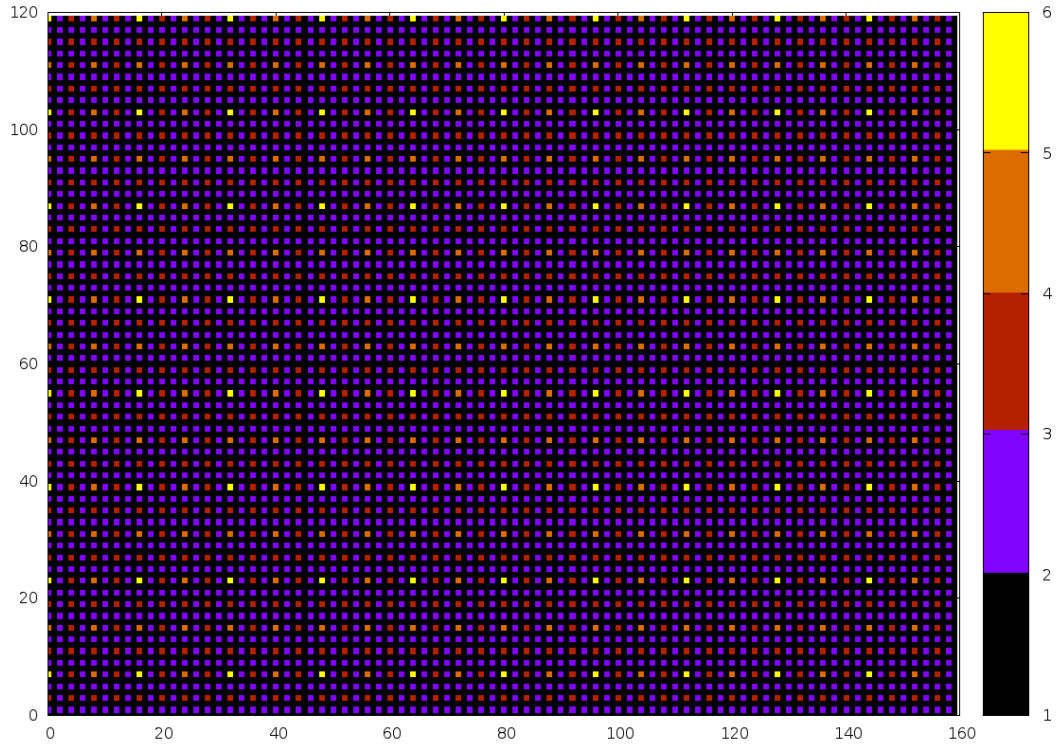


Figure 6.12.: This image shows the number of times that the same pixel is computed using Algorithm 14 with an image of 240×120 pixels and an initial blockside of 32 pixels. The colorband encodes the number of samples taken per pixel. For this image, 25600 samples would be computed instead of the 19200 strictly necessary.

Algorithm 15: BlockShoot(left, right, top, bottom): Compute and store the color of a block of pixels

```
Input: ray
found ← false;
// If any of the pixels colors of the block was already computed take
   it as the color of the whole block
foreach x ∈ [left, right], y ∈ [top, bottom] do
    if pixelCache[x,y] then
        color ← pixelColor[x,y];
        found ← true;
        break;
    end
end
if not found then
    // Compute the color of the nearest pixel on the image matrix, and
       mark it as computed
    xdisp ← ( left + right − 1 ) / 2;
    ydisp ← ( top + bottom − 1 ) / 2;
    x ← xdisp + 0.5;
    y ← ydisp + 0.5;
    ray ← Camera.rayTo( x, y );
    // Mark the x,y pixel as computed
    pixelCache[x, y] ← true;
    color ← Shoot(ray);
end
foreach x ∈ [left, right], y ∈ [top, bottom] do
    // Paint all the pixels of the block
    pixelColor[x,y] ← color;
end
```

7. Results

This chapter contains in section 7.1 the performance analysis of the raytracer describing several benchmark tests and their outcome. In addition, a gallery of sample renders is given in section 7.2.

7.1. Benchmark tests

A simple geometry was defined to test the different features of the raytracer and provide comparable benchmarking results. In order improve the accuracy of the timings all the images of the benchmark were rendered five times and the shortest render time was taken. The shortest time is taken in order to approximate the time that the raytracer needs to perform its computations and reduce the possible impact of other running processes on the timings.

7.1.1. The Fluka input file

For the purposes of the benchmark tests, only the geometry description and the material assignments are relevant and described in this section.

The geometry definition starts with the *bodies* used later to define the *regions* as explained in chapter 2. At the top of the file eleven bodies are defined by their parametric attributes.

In the region definitions the primitive body objects are combined to define regions of space by using CSG operations. Nine regions are defined: the ground, a sphere, a hollow cylinder divided in three parts, a second cylinder, the air surrounding the visible geometry and, as all Fluka geometries, the geometry is surrounded by a blackhole object to force finishing the tracking.

Various materials were chosen to highlight the different visualization of each material: the sphere and the two lower parts of the hollow cylinder are assigned water material, the upper part of the hollow cylinder and the second cylinder are assigned copper, the ground is made of aluminum and the material of the box is “rediron”.

The following block of code defines the geometry:

```
GEOBEGIN                                                              COMBNAME
*    0    0             Cylindrical Target
```

```
*
SPH BLK         0.0 0.0 0.0 10000.0
*  Void
RPP VOI         -1000.0 1000.0 -1000.0 1000.0 -1000.0 1000.0
* First cylinder
YCC TARG        -3.0 6.5 5.0
YCC TARGR       -3.0 6.5 3.0
*  planes limiting the target
XZP ZTlow       0.0
XZP ZThigh      10.0
*  planes segmenting the target
XZP T1seg       1.0
XZP T2seg       2.0
* Second cylinder
YCC TARGB       0.0 -6.5 5.0
* Ball
SPH BOLA        0.0 3.5 -9.5 3.5
* Box
RPP BOX         -20.0 20.0 0.0 3.0 12.0 16.0
*
* === End of bodies
END
*
* === Region definitions
BLKHOLE       5  +BLK -VOI
TARGS1        5  +TARG -ZTlow +T1seg -TARGR
TARGS2        5  +TARG -T1seg +T2seg -TARGR
TARGS3        5  +TARG -T2seg +ZThigh -TARGR
TARGS4        5  +TARGB -ZTlow +ZThigh
TARGS5        5  +VOI +ZTlow
TARGS6        5  +BOLA
TARGS7        5  +BOX
* Air around geometry
INAIR         5  +VOI -( +TARG -ZTlow +ZThigh -TARGR )
                 -( +TARGB -ZTlow +ZThigh )
                 -ZTlow -BOLA -BOX
END
GEOEND
```

And assigns the materials:

```
* === material assignments
ASSIGNMA    BLCKHOLE   BLKHOLE
ASSIGNMA       WATER    TARGS1                              1.
ASSIGNMA       WATER    TARGS2                              1.
```

```
ASSIGNMA      COPPER    TARGS3                          1.
ASSIGNMA       WATER    TARGS4                          1.
ASSIGNMA    ALUMINUM    TARGS5                          1.
ASSIGNMA       WATER    TARGS6                          1.
ASSIGNMA     REDIRON    TARGS7                          1.
ASSIGNMA         AIR     INAIR
```

7.1.2. The scene file

The configuration of the raytracer is specified in the scene file, as explained in chapter 5. This example defines only one camera. Each camera produces one image and defines the quality settings desired. It follows a list of lights and a list of materials with their properties.

The full content of the scene file is:

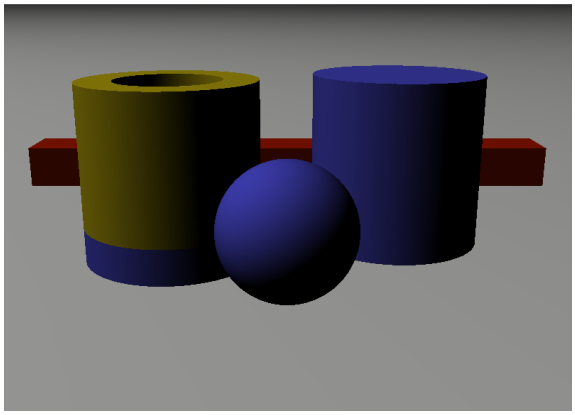
```
camera
file sample00.ppm
projection_type perspective
position 0.0,14.5,-42.0
look_at 0.0,2.5,0.0
up 0.0,1.0,0.0
background_color 1.0,1.0,1.0
angle 45.0
sampling_mode linear
samples 1
width 800
height 600
min_density 0.1
max_depth 1
no_shadows true
no_specular true
#postprocess true

#light
#type point
#color 0.9,0.9,1.0
#intensity 0.0
#falloff 0.89
#position -400.0,800.0,600.0

light
type point
color 1.0,1.0,0.95
```

```
intensity 0.8
falloff 0.89
position 200.0,300.0,-100.0

#light
#type ambient
#color 1.0,1.0,1.0
#intensity 0.15
#intensity 0.0
#position 0.0,0.0,0.0

material
name ALUMINUM
diffuse_color 0.906,0.906,0.945
specular_color 0.99,0.99,1.0
#reflectivity 0.15

material
name AIR
diffuse_color 0.3,0.3,0.3
#transparency 1.0
#ior 1.00

material
name WATER
diffuse_color 0.3,0.3,0.9
specular_color 0.9,0.9,0.9

material
name COPPER
diffuse_color 0.784,0.698,0.056
specular_color 0.99,0.90,0.85

material
name REDIRON
diffuse_color 0.7,0.1,0.0
specular_color 0.99,0.199,0.01
```

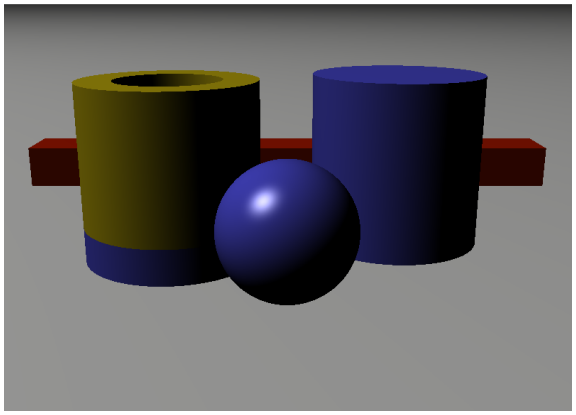
7.1.3. Light settings

The first benchmark test focuses on how the the quality settings for the lights affect to the final image and the render times. Specifically, these samples show the impact of enabling and disabling the specular reflection and shadows, controlled by the `no_specular` and

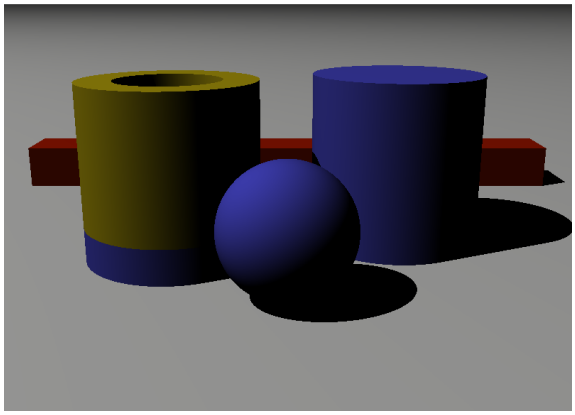
`no_shawows` settings. The first four samples (fig. 7.1) define only one light, the last four (fig. 7.2) define a lighting system with 3 points of light set up to show the features of the geometry.



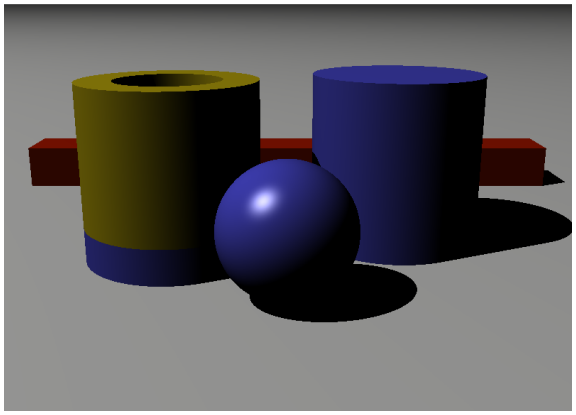
(a) 1 light, specular off, shadows off



(b) 1 light, specular on, shadows off

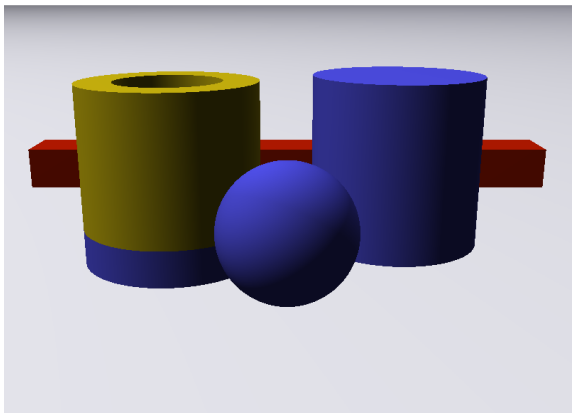


(c) 1 light, specular off, shadows on

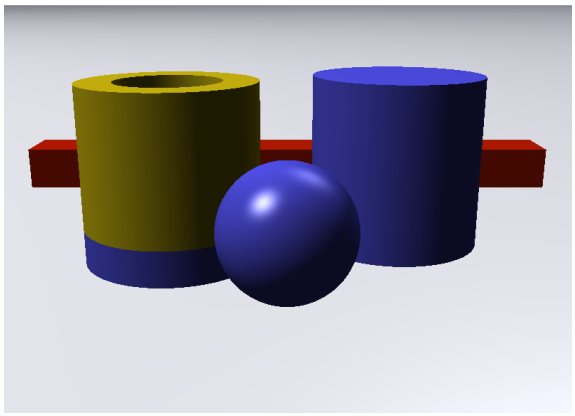


(d) 1 light, specular on, shadows on

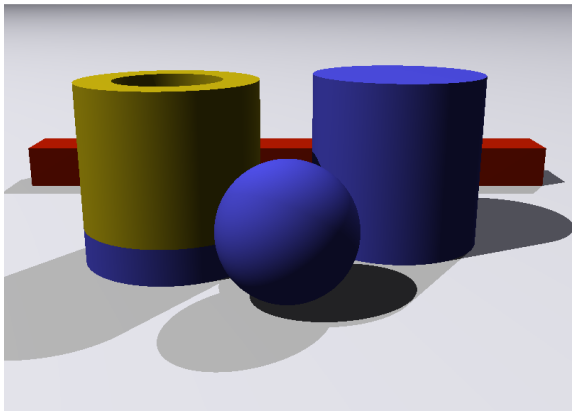
Figure 7.1.: Renders with different light settings (1 light).



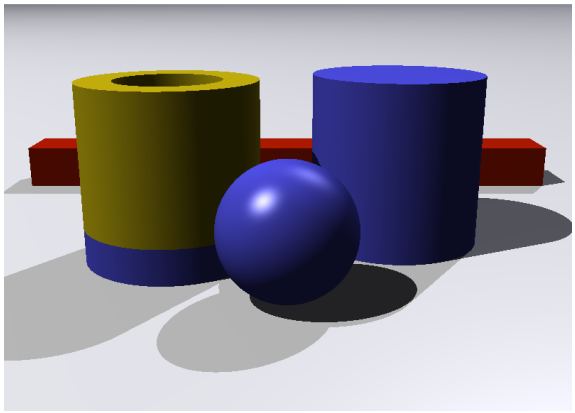
(a) 3 lights, specular off, shadows off



(b) 3 lights, specular on, shadows off



(c) 3 lights, specular off, shadows on



(d) 3 lights, specular on, shadows on

Figure 7.2.: Renders with different light settings (3 lights).

As one can see from the benchmark graph 7.3 specular highlights add a negligible amount of time to the render (1.3 seconds compared to 1.27), the shadow computation instead takes longer. This is to be expected as the specular highlight only requires one extra exponential operation and one addition for the computation of the color of an intersection. On the other hand, the shadow computation requires shooting one ray towards each of the lights and computing its intersections with the geometry.

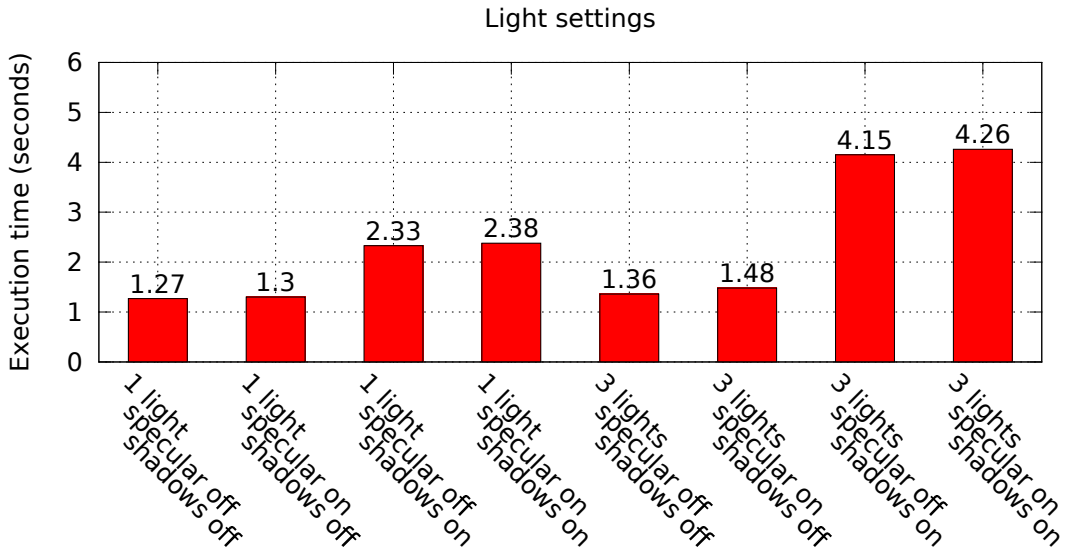


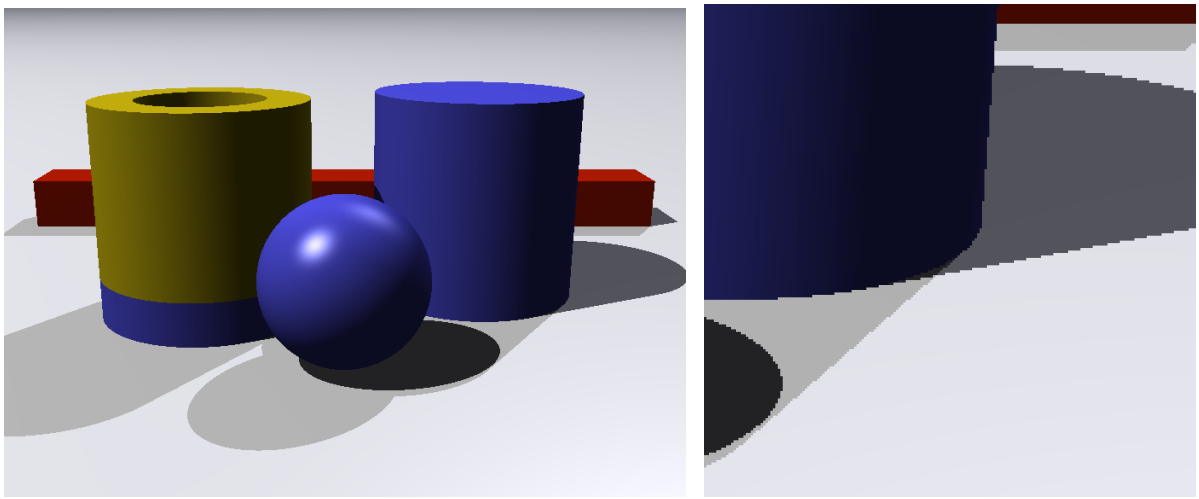
Figure 7.3.: Time benchmark for different light settings.

A few remarks can be made from this benchmark. The cost of computing both effects scale linearly with the number of lights. One can see that the time required to compute the specular term of the image is very low (0.03 s. from sample 1 to sample 2, 0.05 s. from sample 3 to sample 4, 0.12 s. from samples 5 to sample 6 and 0.11 seconds from sample 7 to sample 8). The time required by the computation of shadows is higher. For one light it is 83% slower from sample 1 to 3 and from sample 2 to 4. For three lights it is 205% slower from sample 5 to sample 7 and 187% slower from sample 6 to 8. From this graph one can see that the number of lights in the scene has a great impact on the final render time and is a therefore a limiting factor for quick renders.

7.1.4. Sampling settings

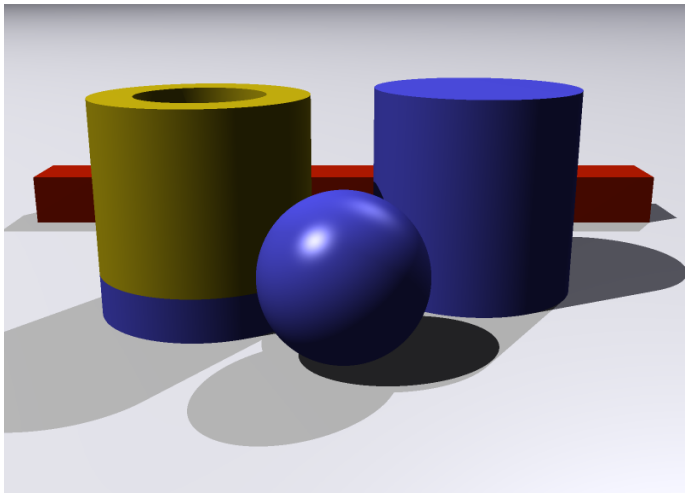
In a further test was is studied how supersampling the scene affects the quality of the image and the render time. Figure 7.4 (a) shows the scene rendered taking only one sample per pixel and fig. 7.4 (b) shows a detail of it . The six test images in fig. 7.5 were

computed using linear sampling and the six images in fig. 7.6 were computed using the adaptive sampling technique explained in section 6.1.

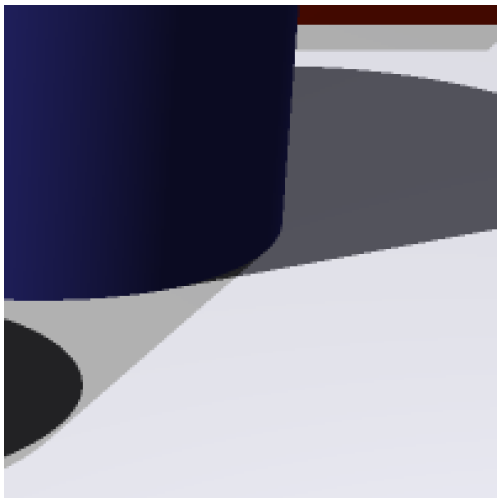


(a) linear 1x1 (b) Detail

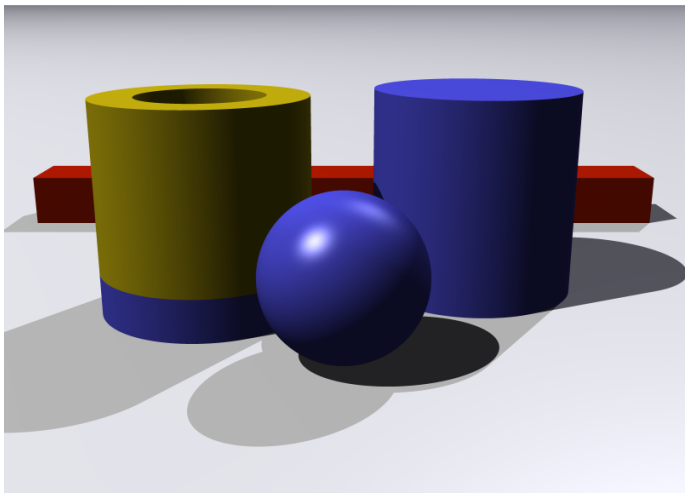
Figure 7.4.: Taking one sample per pixel produces sharp borders and shadows.



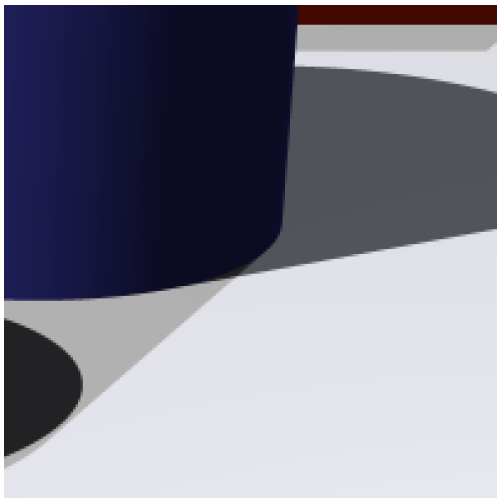
(a) linear 2x2



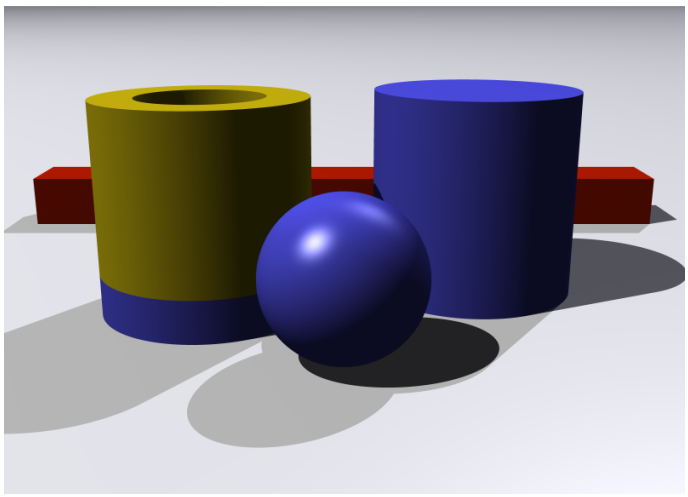
(b) Detail



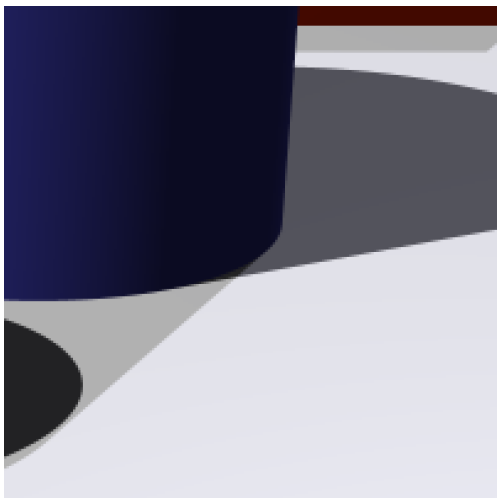
(c) linear 6x6



(d) Detail

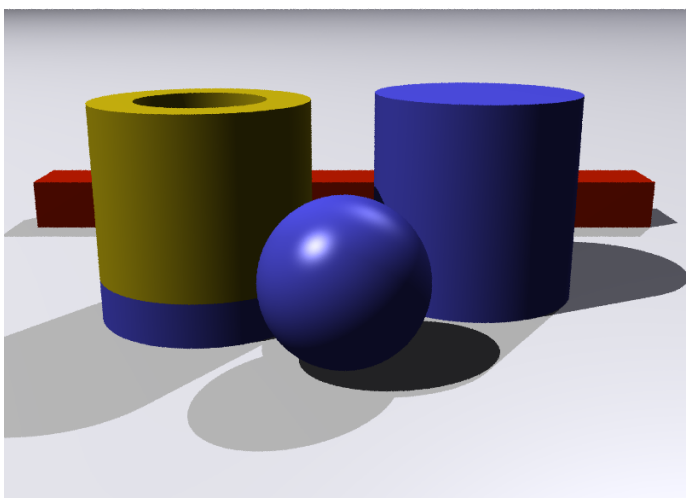


(e) linear 8x8

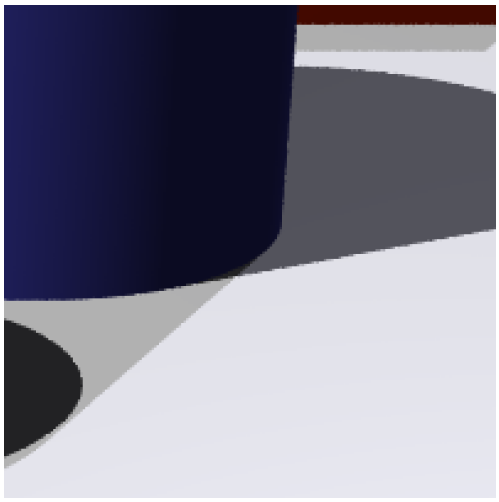


(f) Detail

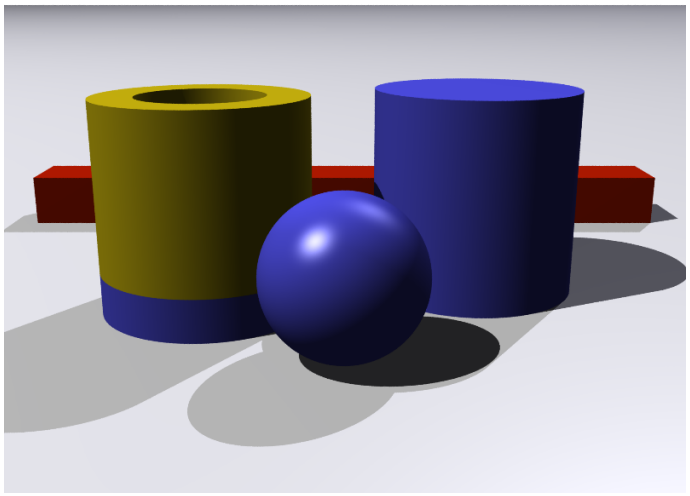
Figure 7.5.: Renders for linear sampling settings.



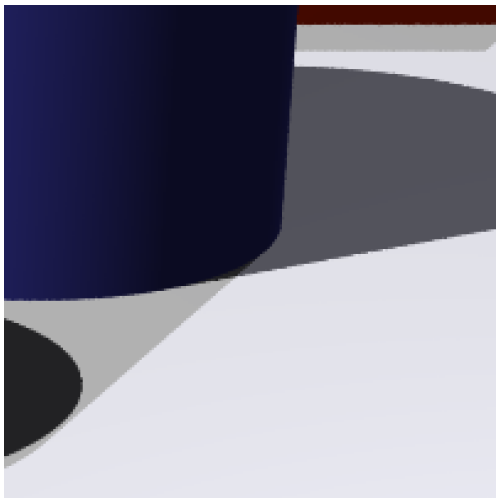
(a) adaptive 2x2



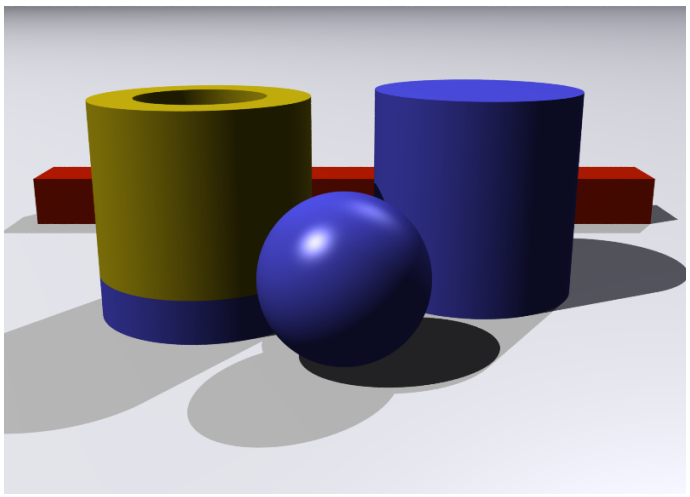
(b) Detail



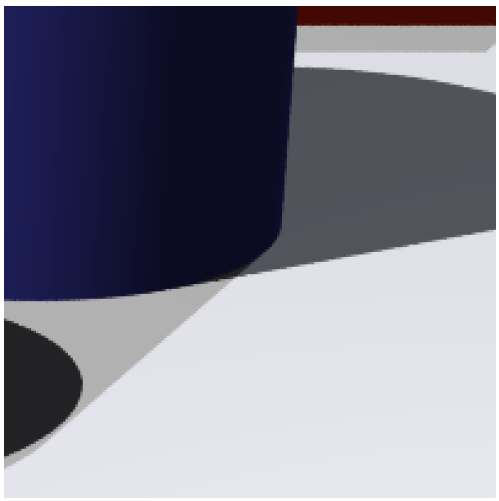
(c) adaptive 6x6



(d) Detail



(e) adaptive 8x8



(f) Detail

Figure 7.6.: Renders for adaptive sampling settings.

As shown in the graph 7.7 the cost of computing the image is multiplied in the case of linear sampling. The render time increases slower if the adaptive technique is used. As explained in chapter 6 the adaptive technique reduces the number of samples taken per pixel by approximating which ones require more samples. In the highest quality setting, using 8x8 samples, one can see that the adaptive technique reduces the render time to aproximately the 40%. The render quality remains very similar to that obtained with the linear sampling technique, therefore the adaptive sampling is well worth it.

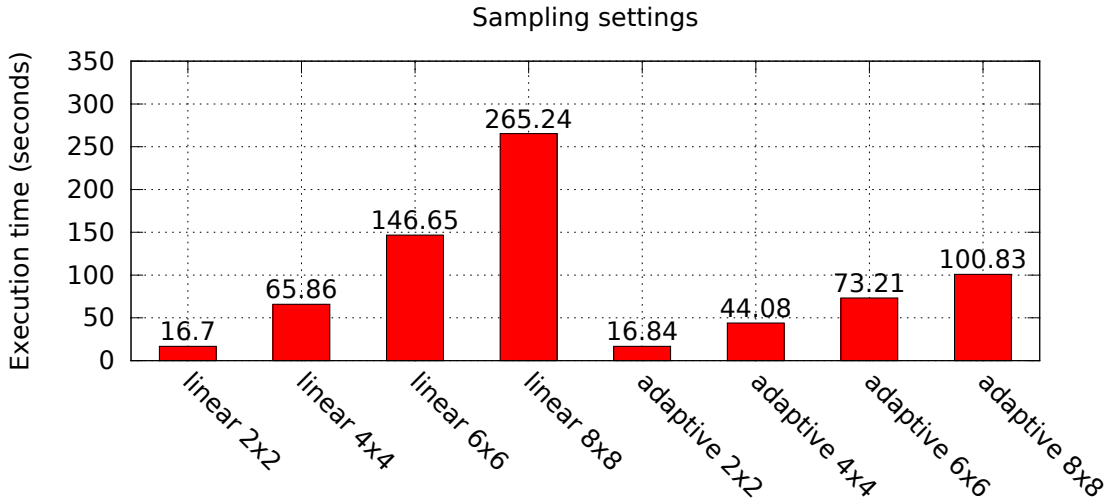


Figure 7.7.: Time benchmarks for sampling settings.

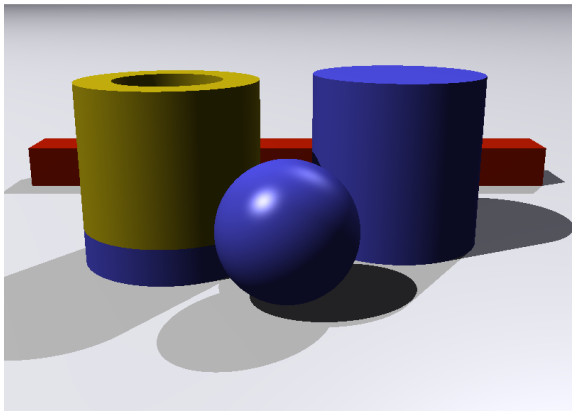
7.1.5. Maximum depth settings

Maximum depth refers to the maximum height of the tree storing the secondary rays as explained in chapter 6.

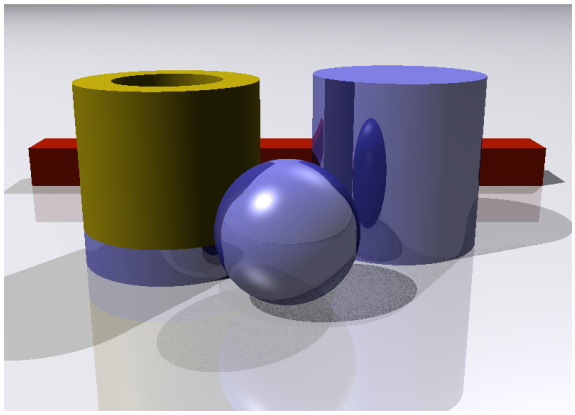
In this geometry, the bottom of the hollow cylinder, the solid cylinder and the sphere are made of water and computing the color of one point on their surfaces requires the computation of the refracted and reflected rays. The floor is assigned aluminum that is reflective and requires the computation of one secondary ray.

One can see how the `max_depth` setting affects to the output images in fig. 7.8. When only one ray bounce is allowed (`max_depth` 2) the color of the 'glass' material becomes blue and the objects behind cannot be seen. When the number of bounces is increased the image looks more natural. Notice how in the sample with a maximum depth of 4 the

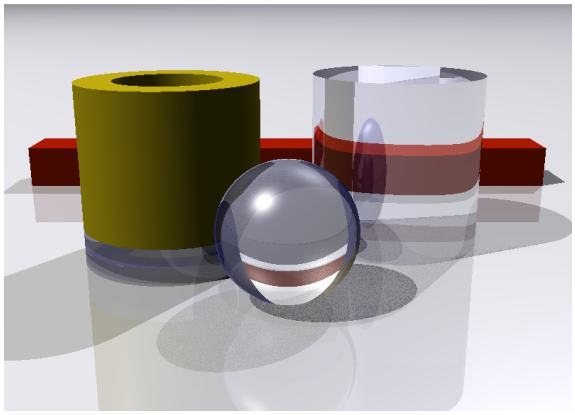
reflection of the ball in the cylinder still appears blue. The last image in the series shows a high quality render where `max_depth` is set to 12.



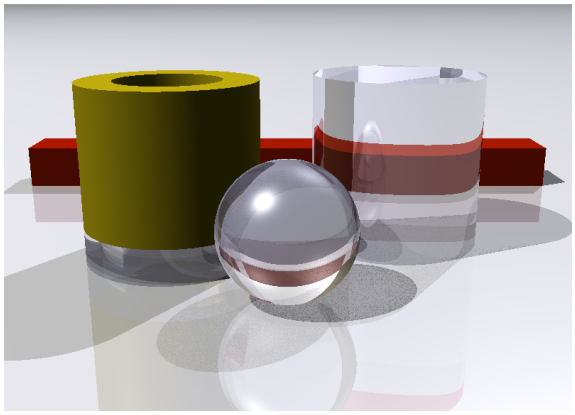
(a) max_depth 1



(b) max_depth 2



(c) max_depth 4



(d) max_depth 12

Figure 7.8.: Renders for different max_depth settings.

The render times for different `max_depth` settings are shown in fig. 7.9. Moving from maximum depth 1 to 2 (fig. 7.8 a and b) the render time increased by a 40%. This amount corresponds to the pixels in the image that are required shooting secondary rays and that with this setting expanded one extra level. In the two other quality settings (fig. 7.8 c and d), this set of pixels expanded many more rays causing the further increase in render time (162% from sample b to c and 232% from sample c to d).

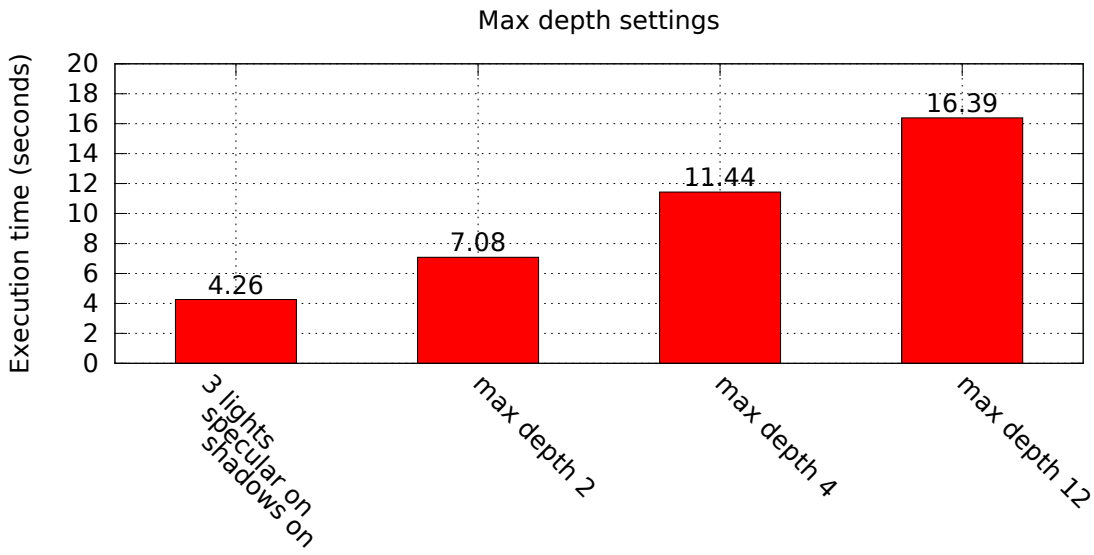
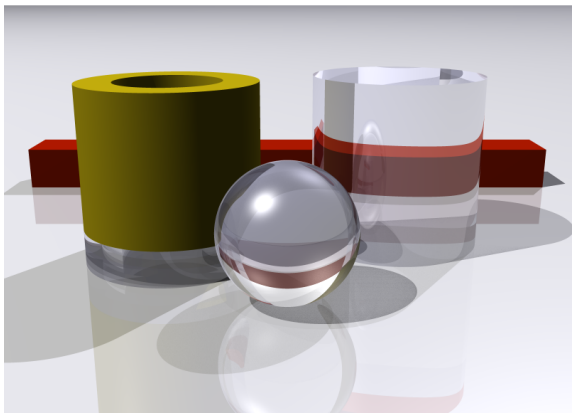


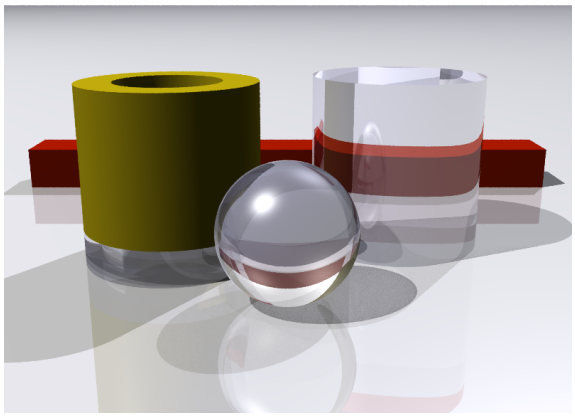
Figure 7.9.: Time benchmarks for max_depth settings.

7.1.6. Max depth settings with supersampling

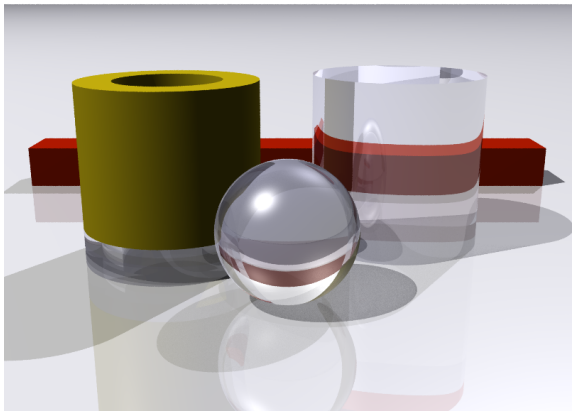
In another interesting high quality benchmark test, two setting for `max_depth` of 12 and supersampling were combined. Four samples were used, the first one using a 6x6 linear sampling technique, while the other three use the adaptive technique. The render results are shown in fig. 7.10.



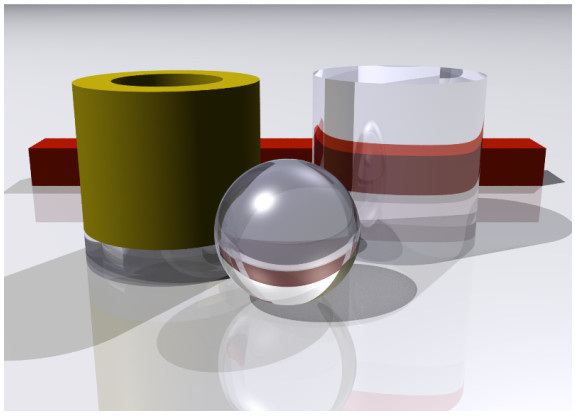
(a) Max depth 12, linear 6x6



(b) Max depth 12, adaptive 3x3



(c) Max depth 12, adaptive 4x4



(d) Max depth 12, adaptive 6x6

Figure 7.10.: Maximum depth 12 with supersampling.

One can observe (fig. 7.10 how using an adaptive sampling technique produces a smooth image without paying the higher cost of the linear sampling.

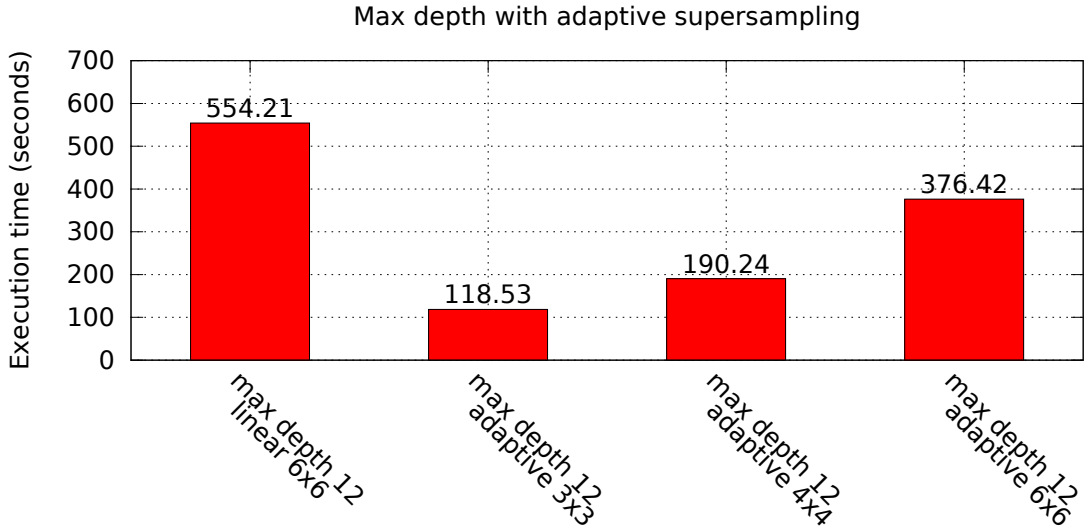
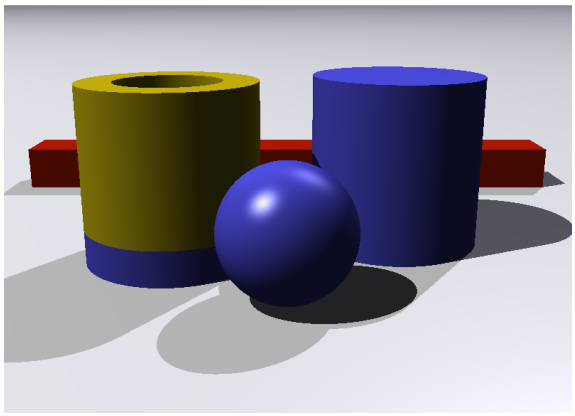


Figure 7.11.: Time benchmark for max_depth 12 with supersampling.

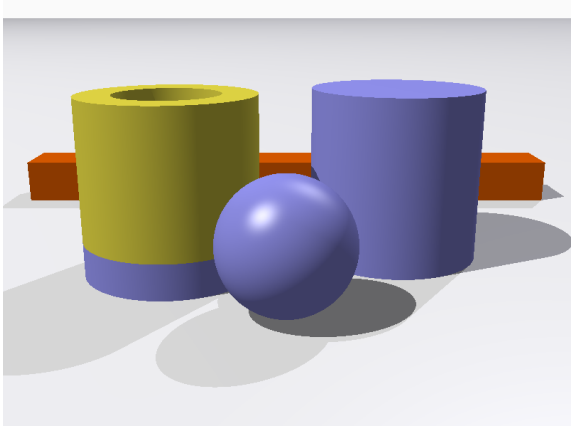
As found in the previous benchmarks the quality of the render is not sacrificed by using the adaptive sampling and the render time is much better (fig. 7.11.

7.1.7. Color postprocessing

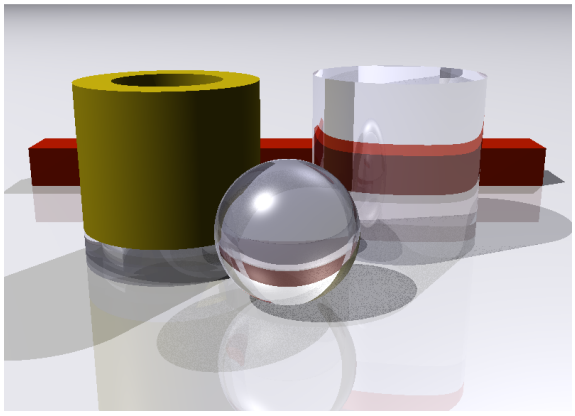
After computing the images as in the previous examples, they are post-processed in order to simulate the adaptation to light that our eyes produce by filtering the colors. To show the effect of this phase we use previously computed images to produce the images in fig. 7.12.



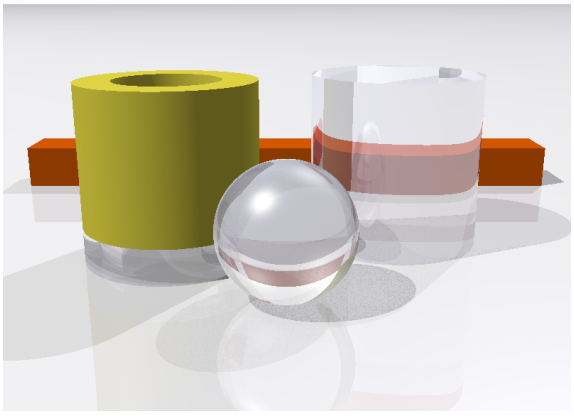
(a) max depth 1, w/o postprocessing



(b) max depth 1, w/ postprocessing



(c) max depth 12, w/o postprocessing



(d) max depth 12, w/ postprocessing

Figure 7.12.: Time of renders with and without postprocessing.

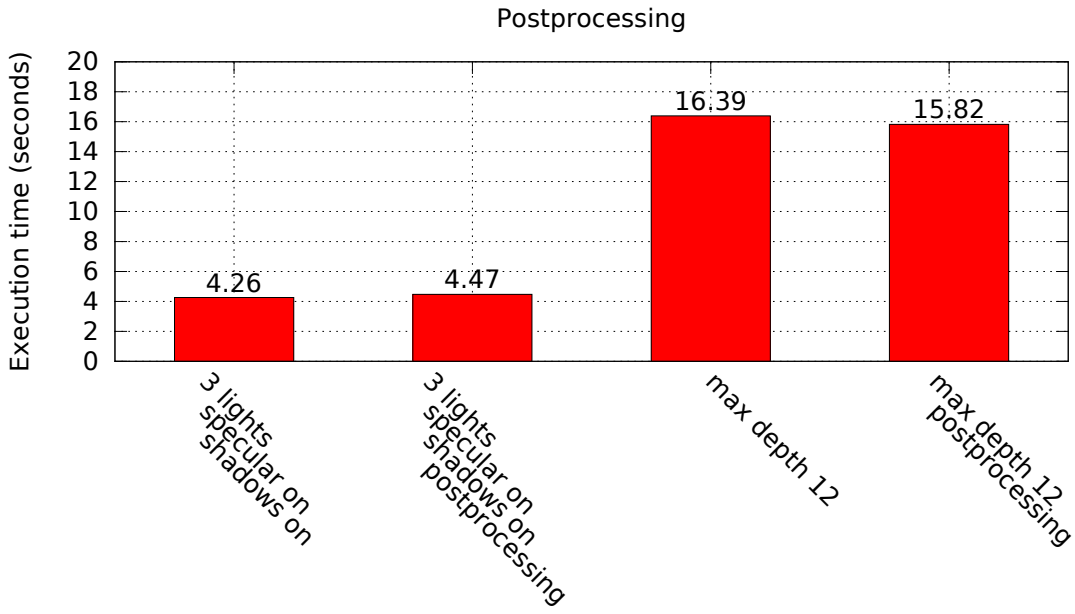


Figure 7.13.: Postprocessing settings.

As graph 7.13 shows, one can observe that the render time is barely affected by the postprocessing phase as this technique simply iterates over the color of each pixel of the scene and makes some adjustments.

7.1.8. Summary

These results show how the different settings affect the ouput image and the render time. While some features are crucial for visualizing the depth of the geometry as specular reflections and shadows, other settings can be kept at a lower setting during tests and only increase them for producing the final render. For example, the number of samples per pixel set to 1 generating a very effective preview of the final render (as in fig. 7.2 d). The `max_depth` setting can also be kept low to see the first secondary rays while maintaining a low render time.

As for the sampling strategies, the images obtained using the adaptive and linear techniques are almost identical. The time required to obtain the image is much lower using the adaptive technique, therefore, we consider that using adaptive sampling with a high number of samples is recommended for the final render (fig. 7.5 and fig. 7.6).

As shown, the postprocessing phase of the renderer can also improve the final quality of the render with a minimum cost (fig. 7.12).

7.2. Image gallery

7.2.1. Sample 1

A final high resolution render of the sample geometry follows. The most important settings are: sampling set to adaptive, number of samples 8, `max_depth` is 12 and the resolution is 1600x1200.

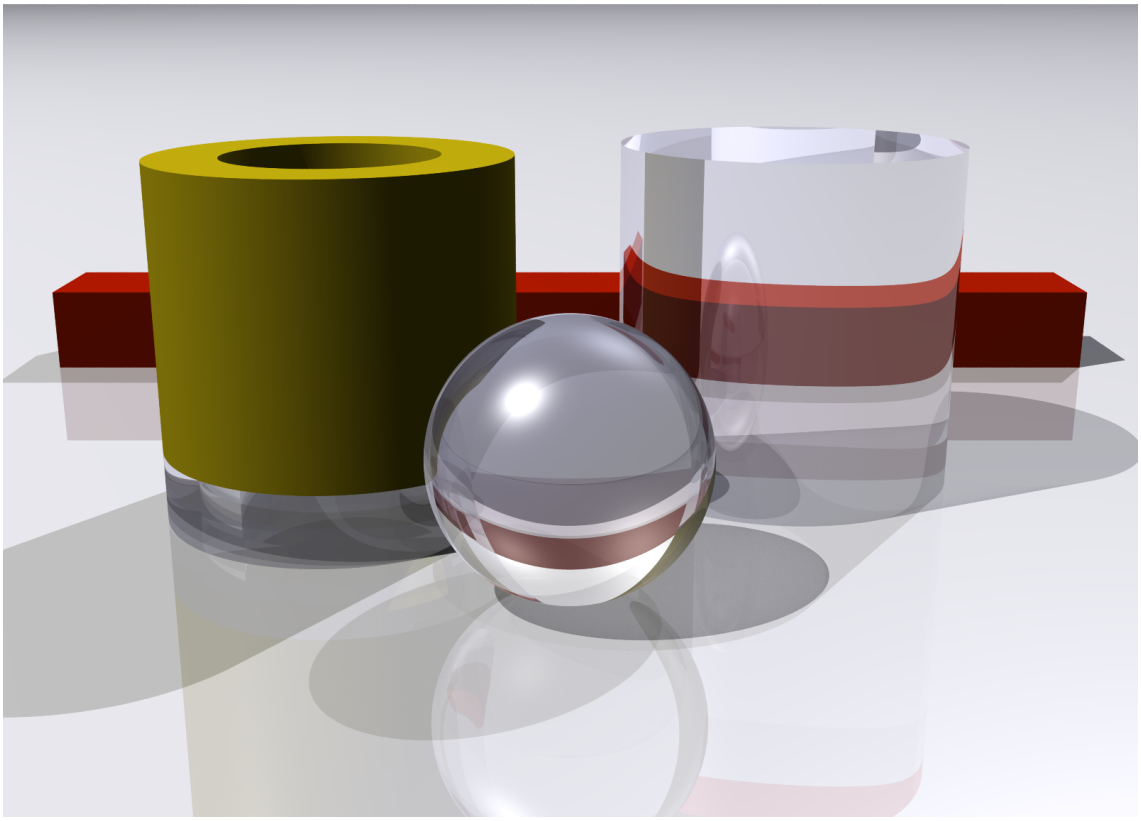


Figure 7.14.: The sample geometry rendered at high quality

7.2.2. ITER

ITER is an international project to design and build an experimental fusion reactor based on the "tokamak" concept. The ITER machine is a torus-shaped reactor in which matter is heated up to 150 degrees forming plasma.

The machine was modelled in Fluka by Elias Lebbos in order to perform radiation protection and activation studies. The Fluka model is defined as a slice of the machine and replicated using the lattice command in Fluka. In the following renders one can see the detail obtained in the renders of one slice of the machine.

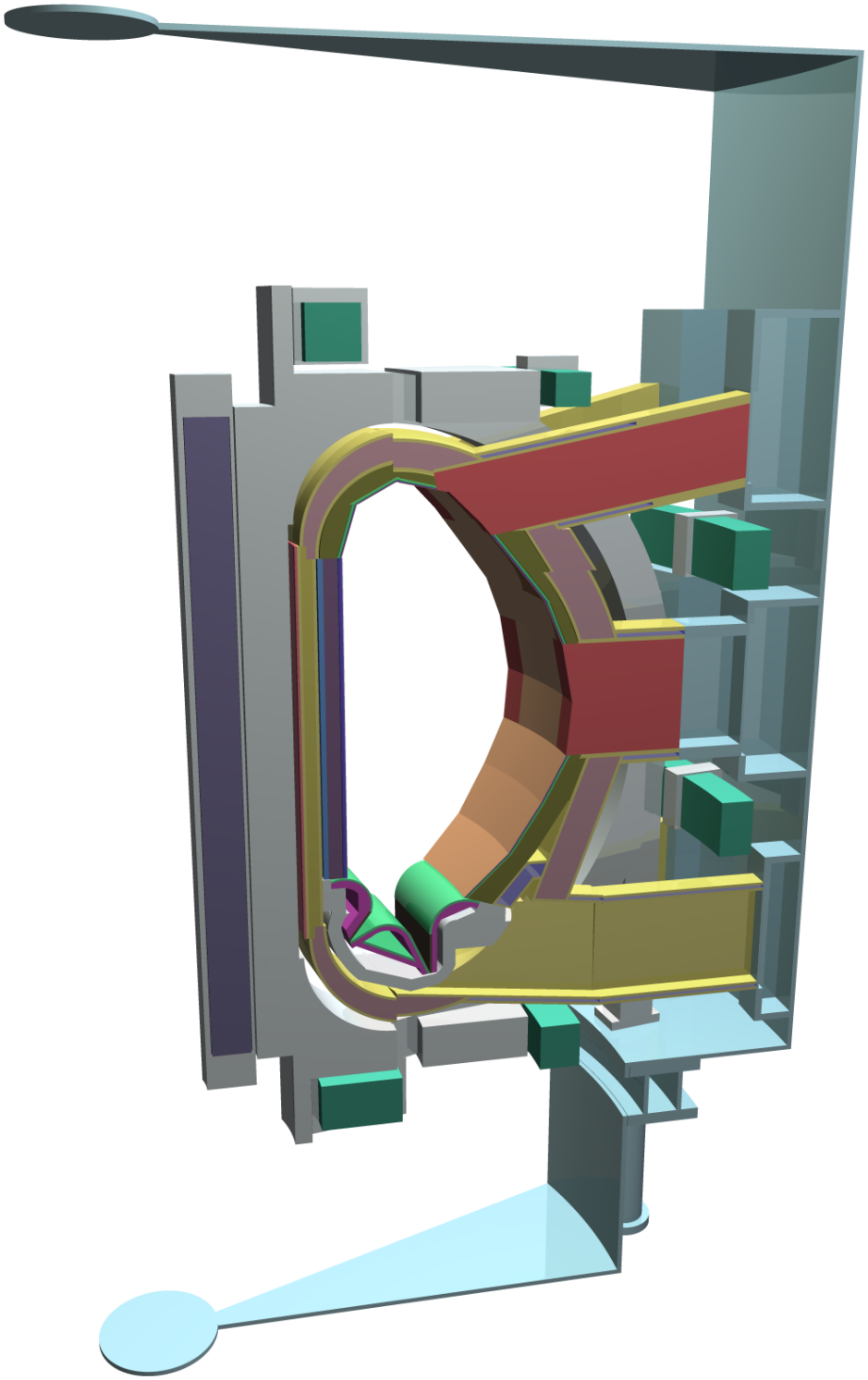


Figure 7.15.: A slice of ITER.

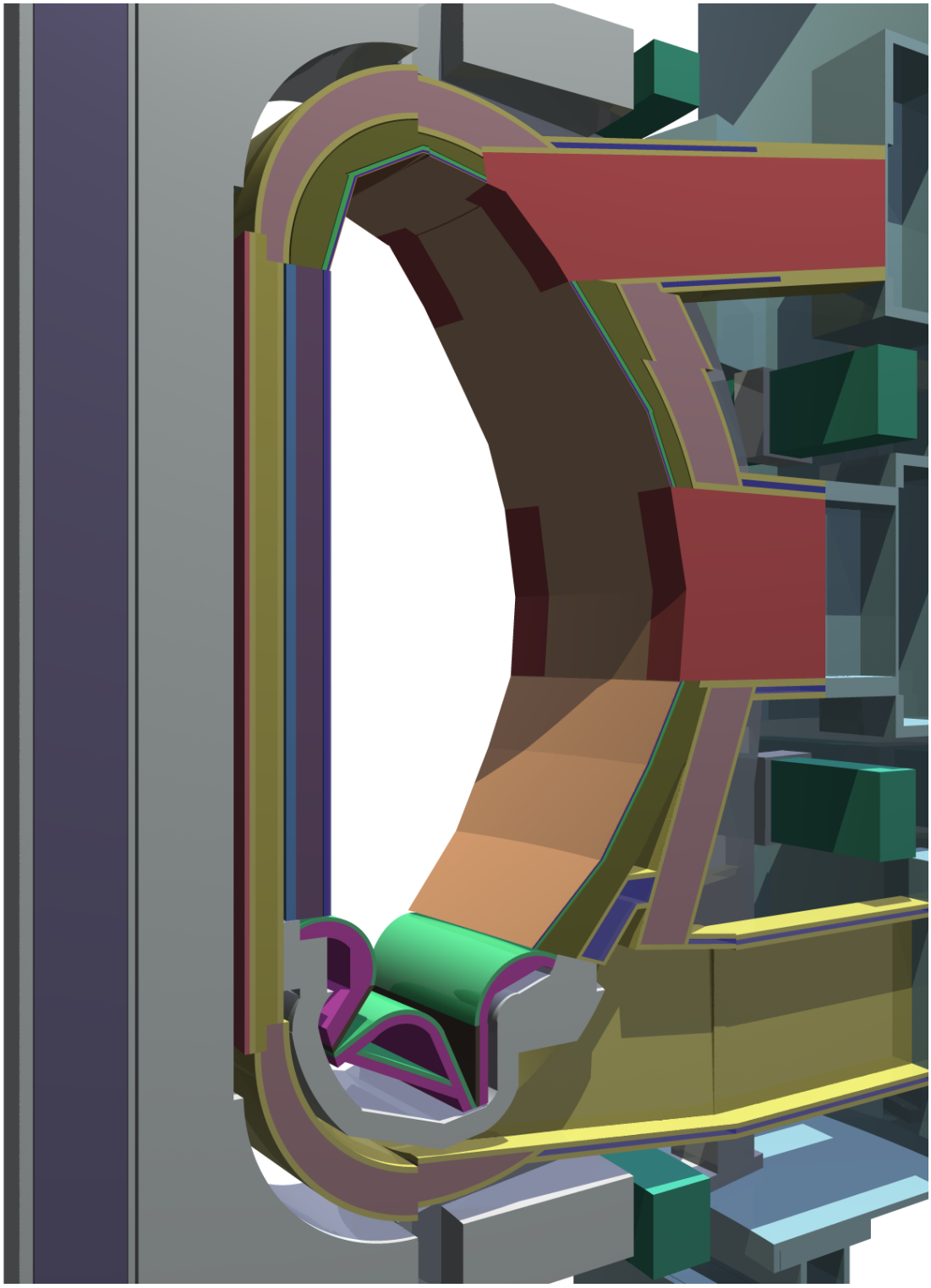


Figure 7.16.: A slice of ITER rendered with shadows.

8. Parallelization proposal

8.1. Motivation

In the past the power of computers had been growing at a constant pace. CPUs would double their speed every year and programs would run faster without making any change in the code. Since a few years processor manufacturers have not been able to keep increasing the clock speed of processors. Among the problems found are circuit interferences due to the minituarization and clock speed and increased heating.

Current advances in processors include: better instruction sets, better pipelining or smarter memory usage. Alternatively what we are seeing is the appereance of increasingly parallel CPUs. New CPUs are usually compounded of several cores, that is, several processing units that can work in parallel. Everyday computers have 4, 6 or 8 cores and soon we will see computers with 16, 32 or 48 cores. This change in architecture does not improve the performance of old software, instead it has to be designed to make use of parallel processing units.

The appereance of massively parallel processing units does not limit to CPUs, GPUs are in fact massively parallel by definition and in later revisions they are programable to the point that they can be used for general purpose computing.

The conversion of previous software to the parallel paradigm is not simple and usually requires full re-architecture of applications and algorithms. Many vendors are trying to solve this problem and they are developing tools, programming languages and frameworks to facilitate this task. Several mainstream possibilities are available, for example: OpenMP, OpenCL, Intel TBB, CUDA, etc.

The next section tries to define a strategy that could be used to parallelize the execution of the raytracer.

8.2. Strategy

Parallel programs must reduce the synchronization needed among the threads or they will spend most of the time waiting for other threads – effectively serializing the execution flow. Looking at the structure and data flow of the raytracer we realize that it is highly parallelizable: each ray is completele independed and there is no data exchange or

dependencies among them. The problem can be split in many parallel taks, each of them computing the color of one pixel (or block of pixels) of the final image.

The software has to be designed so that there is no global state or variables shared among threads. Global variables would require using synchronization mechanisms to protect their use. A way to avoid global variables is to have one replica of each of those resources or variables per thread.

By redefining the raytracer as a series of independent subproblems we can parallelize the algorithm effectively. As explained before the computation of each ray is completely independent of other rays. We can use each computing each of the pixels of the image as independent subproblems. By doing so several threads can be spawn by the raytracer process and run in parallel on the different cores of the computer.

At this point we have to consider all the functions that are called for the processing of the color of one pixel and eliminate all kind of global state sharing or isolate them. Right now the Fluka geometry engine does rely on global state and running several threads in parallel would interfere on the computations and produce wrong results. Even if we have control over the ray-tracer code and we could perform this task there is very little we can change of the Fluka geometry engine. The only solution is to revise the code of the geometry engine to make it more suitable for parallel computing. A difficult task that could affect all the Fluka code base. A re-structure of the Fluka geometry engine is far from the scope of this thesis and therefore is not considered as viable for the moment.

9. Conclusions

Within this document we have discussed how to implement a raytracer that uses the Fluka geometry engine to inspect the geometry used in the simulation. We described the implementation of the raytracer which in its final state is fully capable of generating high quality representations of Fluka geometries. The performance measurements show that it can be used to generate high quality images in a reasonable time.

Compared to the alternatives the quality of the images produced by the raytracer is of much higher quality. The addition of reflection and refraction effects greatly improves the quality of the image for high impact presentations.

In the results chapter we found that the adaptive sampling technique aleviates the poor performance caused by supersampling the image while maintainning a comparable level of quality. Another interesting feature of the raytracer is the way it is configured: it uses a configuration file which is extensible and easy to write. The work was completed with a proposal of new cards for Fluka to integrate the code of the raytracer into the Fluka standard distribution.

With the real-time viewer we have found that the performance is good enough to provide a semi-interactive viewer. The images are generated at a reasonable speed and rendering the center of the image first produces a higher sense of interactivity. This leads us to think that it would be possible to create a better interactive viewer having more control on the implementation and precision of the geometry engine. The Fluka geometry engine provides a reliable tracking of particles at high precision, which is appropriate for the purposes of Fluka but not needed for the visualization.

In the parallelization chapter we speculated about a possible adaptation of the raytracer to multi-core machines. We arrived at the conclusion that in order to parallelize the raytracer the Fluka geometry engine needs to be parallelized first, something out of the scope of this work.

A set of features that could not be implemented in time would make a follow-up feasible. Among those features are textures (in its bitmap or procedural form) which would enable also the projection of Fluka output values on top of the geometry surfaces. Global illumination as discussed in chapter 4 could help in the task of visualizing the geometry; simpler techniques as ambient occlusion would provide better images while keeping the render time within reasonable margins. It is still possible to improve the overall performance of the raytracer exploring alternative sampling techniques, better data locality or smarter reuse of the rays and intermediate values.

After the development of the Fluka raytracer Flair's geometry viewing and editing features have improved dramatically. A new geometry editor was implemented in C++, it includes a 3D view of the geometry which uses the ray-tracing technique. Allegedly, the work exposed here has proven the feasability of the ray-tracing technique for rendering Fluka geometries in real-time. The author of this document has also been a contributor to the geometry editor in Flair. Flair implements its own geometry engine and can therefore be adapted for the needs of the editor. Parallelization and further optimizations are scheduled for future development.

We consider that the current document and associated source code have successfully matched the main objective of providing high quality renders of the geometries as interpreted by Fluka. Furthermore, the development allowed to implement a proof-of-concept interactive viewer that allows the user to navigate the geometries interactively.

A. Appendices

A.1. Proposed new FLUKA cards

```
MATERIALS ==========================================================================

*
MAT-PROP   <MATNAME> <DIFFUSE_COL> <SPECULAR_COL> <AMBIENT_COL> - - COLOR

MATNAME:      STRING
DIFFUSE_COL:  #HEXCOLOR
SPECULAR_COL: #HEXCOLOR
AMBIENT_COL:  #HEXCOLOR

*
MAT-PROP   <MATNAME> [<SPEC_POWER>] [<FUZZINESS>] [<RELECTIVITY>] - - SURFACE

MATNAME:      STRING
SPEC_POWER:   FLOAT(0..):20
FUZZINESS:    FLOAT(0..1):0
REFLECTIVITY: FLOAT(0..1):0

*
MAT-PROP   <MATNAME> [<TRANSPARENCY>] [<IOR>] [<ATTENUATION>] - - TRANSPARENCY

MATNAME:      STRING
TRANSPARENCY: FLOAT(0..1):0
IOR:          FLOAT(0..1):0
ATTENUATION:  FLOAT(0..):0

*
ASSIGNMAT: Use the last field to specify if a region has to be rendered?


LIGHTS =============================================================================

*
LIGHT <LIGHTNAME> <X> <Y> <Z> [<INTENSITY>] [<ENABLE>] DEFINITION

LIGHTNAME: STRING
X, Y, Z:   FLOAT
```

```
INTENSITY: FLOAT(0..1):1
ENABLE:    LOGICAL:TRUE

*
LIGHT <LIGHTNAME> <COLOR> [<FALLOFF>] [<NOSHADOWS>] [<NOSPECULAR>] - COLOR

LIGHTNAME:  STRING
COLOR:      #HEXCOLOR
FALLOFF:    FLOAT(0..1):1
NOSHADOWS:  LOGICAL:FALSE - Disables shadows
NOSPECULAR: LOGICAL:FALSE - Disables specular reflections

*
LIGHT <LIGHTNAME> <LOOKX> <LOOKY> <LOOKZ> <MINANGLE> <MAXANGLE> SPOT
LIGHT <LIGHTNAME> <LOOKX> <LOOKY> <LOOKZ> <MINANGLE> <MAXANGLE> DIRECTIONAL

LIGHTNAME:             STRING
LOOKX, LOOKY, LOOKZ:   FLOAT
MINANGLE:              FLOAT
MAXANGLE:              FLOAT


CAMERAS ========================================================================

*
CAMERA <CAMNAME> [<FILE>] [<WIDTH>] [<HEIGHT>] [<ANGLE>] [<TYPE>] DEFINITION

CAMNAME: STRING
FILE:    STRING:CAMNAME
WIDTH:   INTEGER:320
HEIGHT:  INTEGER:240
ANGLE:   FLOAT(0..):45
TYPE:    PROJECTION/ORTHOGONAL:PROJECTION

*
CAMERA <CAMNAME> <X> <Y> <Z> - - POSITION
CAMERA <CAMNAME> <X> <Y> <Z> - - LOOK_AT
```

```
CAMERA <CAMNAME> <X> <Y> <Z> - - UP

CAMNAME: STRING
X, Y, Z: FLOAT

*
CAMERA <CAMNAME> [<BKG_COLOR>] [<MINDENS>] [<NOSHADOWS>] [<PRINTMAT>] - PROPERTIES

CAMNAME:   STRING
BKG_COLOR: #HEXCOLOR:#FFFFFF
MINDENS:   FLOAT:1
NOSHADOWS: LOGICAL:FALSE - Disables shadows
PRINTMAT:  LOGICAL:FALSE - Enables printing of not defined material names

*
CAMERA <CAMNAME> <SAMPLE_TYPE> <SAMPLES> - - - SAMPLING

CAMNAME:   STRING
SAMPLE_TYPE: LINEAR/ADAPTATIVE
SAMPLES:     INTEGER - Number of samples per pixel per angle, samples per pixel = samples * samples
```

A.2. FORTRAN data definitions

cameras.f

```
  *
  *=== Camera definitions ========================================================*
  *
  *
  *-------------------------------------------------------------------------------*
  *
*
  *     Created on 25-mar-09                        David Sinuela
*
  *
*
  *-------------------------------------------------------------------------------*
  *
  *

  *     Maximum amount of cameras
        integer CAMMAX
        parameter (CAMMAX = 3)

  *     Amount of cameras stored
        integer camcount

  *     Cameras ================================================================

  *     Projection type (1 - perspective, 2 - Orthogonal)
        integer camtype(CAMMAX)
  *     Filenames
        character camfile(CAMMAX)*256
  *     Position
        double precision campos(3, CAMMAX)
  *     Look at
        double precision camlook(3, CAMMAX)
  *     Up direction
        double precision camup(3, CAMMAX)

  *     Background color
        real*8 cambcolor(3, CAMMAX)

  *     Angle
        real*8 camangle(CAMMAX)
  *     Width (pixels)
        integer camwidth(CAMMAX)
  *     Height (pixels)
        integer camheight(CAMMAX)

  *     Sampling type
        integer camstype(CAMMAX)
```

```
*     Number of samples
      integer camsamples(CAMMAX)
*     Don't Render shadows?
      logical camnshad(CAMMAX)
*     Minimum density to render
      real*8 camminden(CAMMAX)

*     Maximum depth in ray reflections/refractions
      integer cammdepth(CAMMAX)
*     Don't render specular reflections
      logical camnspec(CAMMAX)
*     Don't render fuzziness in surfaces
      logical camnfuzz(CAMMAX)
*     Is the camera a progressive camera?
      logical camprog(CAMMAX)

*     Print materials? (Helps with the setup of the scene and
*     materials)
      logical campmat(CAMMAX)

*     Cache vectors
      double precision camdir(3, CAMMAX)
      double precision camdside(3, CAMMAX)
      double precision camdup(3, CAMMAX)

*     ================================================================

      common /cameras/ campos, camlook, camup, cambcolor,
     &                 camdir, camdside, camdup,
     &                 camwidth, camheight,
     &                 camtype, camstype, camsamples, camnshad,
     &                 camminden, campmat, camfile,
     &                 camcount, camangle, cammdepth, camprog,
     &                 camnspec, camnfuzz
      save /cameras/
*=====================================================================*
```

image.f

```
*
*=== Definition of image ===============================================*
*
*
*-----------------------------------------------------------------------*
*
*
*     Created on 25-mar-09                    David Sinuela
*
*
*
*-----------------------------------------------------------------------*
*
```

```
*
*     Maximum size of images
      integer MAXWIDTH, MAXHEIGHT
      parameter (MAXWIDTH = 1600)
      parameter (MAXHEIGHT = 1200)

*     Dimensions of the current image
      integer imgwidth, imgheight

      real*8 imgpixels(3, maxheight+2, maxwidth+2)
      logical imgcache(maxheight+2, maxwidth+2)
      real*8 imgscores(maxheight+2, maxwidth+2)

*     Unit number for the current image
      integer imgunit

      common /image/ imgpixels, imgscores, imgcache,
     &               imgunit, imgwidth, imgheight
      save /image/
```

lights.f

```
*
*=== Light definitions ================================================*
*
*
*----------------------------------------------------------------------*
*
*     Created on 25-mar-09                      David Sinuela
*
*
*----------------------------------------------------------------------*
*
*

*     Maximum amount of lights
      integer LIGHTMAX
      parameter (LIGHTMAX = 10)

*     Amount of lights stored
      integer lightcount

*     Lights ===============================================================

*     Type of light
      integer lighttype(LIGHTMAX)
*     Color
      real*8 lightcolor(3, LIGHTMAX)
```

```
*     Intensity
      real*8 lightint(LIGHTMAX)
*     Position
      double precision lightpos(3, LIGHTMAX)
*     Look at
      double precision lightlook(3, LIGHTMAX)
*     Falloff
      real*8 lightfall(LIGHTMAX)
*     No shadows
      logical lightnshad(LIGHTMAX)
*     No specular
      logical lightnspec(LIGHTMAX)
*     Min Angle
      real*8 lightmina(LIGHTMAX)
*     Max Angle
      real*8 lightmaxa(LIGHTMAX)

*     ================================================================

      common /lights/ lightcolor, lightint,
     &                lightpos, lightlook, lightfall,
     &                lightmina, lightmaxa, lightcount,
     &                lightnspec, lightnshad, lighttype
      save /lights/

*=====================================================================*
```

materials.f

```
*
*=== Camera definitions ==============================================*
*
*
*---------------------------------------------------------------------*
*
*     Created on 25-mar-09                    David Sinuela
*
*
*---------------------------------------------------------------------*
*
*

*     DEFAULT material number
      integer DEFMAT
      parameter(DEFMAT = 1)

*     Specular color
      real*8 matspec(3, MXXMDF)
*     Ambient color
      real*8 matamb(3, MXXMDF)
```

```
*     Diffuse color
      real*8 matdiff(3, MXXMDF)

*     Is material initialized?
      logical matinit(MXXMDF)

*     Reflection ratio
      real*8 matrefl(MXXMDF)
*     Transparency ratio
      real*8 mattran(MXXMDF)

*     Specular power
      real*8 matspow(MXXMDF)

*     IOR: Index Of Refraction
      real*8 matior(MXXMDF)

*     Surface fuzziness
      real*8 matfuzz(MXXMDF), matatt(MXXMDF)
*     ================================================================

      COMMON /MATERIALS/  matspec, matamb, matdiff,
     &                    matinit,
     &                    matrefl, mattran,
     &                    matspow, matior,
     &                    matfuzz, matatt

      SAVE /MATERIALS/
*=====================================================================*
```

raystack.f

```
*
*=== Camera definitions ==============================================*
*
*
*---------------------------------------------------------------------*
*
*
*     Created on 25-mar-09                    David Sinuela
*
*
*
*     Definition of RayStack
*
*     Implements a custom binary tree in an array
*
*
*
*---------------------------------------------------------------------*
*
*
```

```
*     Maximum depth of tree
      integer maxrsdepth
      parameter (maxrsdepth = 7)

*     real*8 size of the array storing the tree
      integer rssize
      parameter (rssize = maxrsdepth*2+1)

*     Number of nodes in the stack
      integer rscount

*     RayStack node ====================================================

*     Ray position and direction
      double precision rsrpos(3, rssize)
      double precision rsrdir(3, rssize)

*     Computed color
      real*8 rscolor(3, rssize)

*     Is expanded?
      logical rsexpanded(rssize)

*     Node type (0=root, 1=reflected, 2=refracted)
      integer rsntype(rssize)

*     Previous node, reflected and refracted node indexes
      integer rsprevnidx(rssize)
      integer rsreflnidx(rssize)
      integer rsrefrnidx(rssize)

*     RayStack node intersection ========================================

*     Has intersection?
      logical rshasint(rssize)

*     Intersection point
      double precision rsipoint(3, rssize)
*     Intersection normal
      double precision rsinormal(3, rssize)

*     Material index
      integer rsimatidx(rssize)

*     Is normal cached?
      logical rsinorm(rssize)

*     Region from
      integer rsiregfrom(rssize)

*     Region to
```

```
      integer rsiregto(rssize)

*     Intersection dot
      double precision rsidot(rssize)

*     End of RayStack node ============================================

      common /raystack/ rsrpos, rsrdir, rsidot, rscolor, rsntype,
     &                  rsexpanded, rsprevnidx, rsreflnidx, rsrefrnidx,
     &                  rscount, rsipoint, rsinormal, rsimatidx,
     &                  rshasint, rsinorm, rsiregfrom, rsiregto
      save /raystack/
```

scene.f

```
* ************************************************************************
*
* Definition of Scene information
*
* ************************************************************************

*     Current camera
      integer sccamidx

*     Current/last region
      integer screg

      common /scene/ sccamidx, screg
      save /scene/
```

stats.f

```
*
*=== Counters for statistics ============================================*
*
*
*------------------------------------------------------------------------*
*
*     Created on 25-mar-09                      David Sinuela
*
*
*------------------------------------------------------------------------*
*
*

      integer geoerrors
      integer correrrors
      integer totalnodes
      integer totalshoots
```

```
      integer totalintget
      integer ccachehits

*     ================================================================

      COMMON /STATS/  geoerrors, correrrors, totalnodes,
     &                totalshoots, totalintget, ccachehits

      SAVE /STATS/

*=====================================================================*
```

Bibliography

[1] Advanced computer graphics sampling. http://cg.informatik.uni-freiburg.de/course_notes/graphics2_04_sampling.pdf.

[2] Cern engineering department structure. http://en-dep.web.cern.ch/en-dep/structure/STI/.

[3] flair - FLUKA advanced interface. http://www.fluka.org/flair/index.html.

[4] Fluka examples in the official FLUKA site. http://www.fluka.org/fluka.php?id=examples&mm2=3.

[5] Fluka manual: User routines. http://www.fluka.org/fluka.php?id=man_onl&sub=99.

[6] Fluka physics models in the official FLUKA site. http://www.fluka.org/fluka.php?id=publications&mm2=3.

[7] Geant4 website. http://geant4.cern.ch/.

[8] Infn - the national institute of nuclear physics. http://www.infn.it.

[9] The official FLUKA site. http://www.fluka.org.

[10] OpenGL - the industry standard for high performance graphics. http://www.opengl.org/.

[11] POV-Ray. http://www.povray.org.

[12] Ray tracing (Graphics). http://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[13] SimpleGeo - solid modeling for particle transport monte carlo simulations. http://theis.web.cern.ch/theis/simplegeo/.

[14] Fredo Durand Addy Ngan and Wojciech Matusik. Experimental validation of analytical brdf models. http://people.csail.mit.edu/wojciech/BRDFValidation/index.html.

[15] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[16] James Arvo. Backward ray tracing. In *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, pages 259–263, 1986.

[17] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[18] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, August 1978.

[19] W. Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Commun. ACM*, 13(9):527–536, September 1970.

[20] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[21] Cass W. Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *CoRR*, cs.GR/0301002, 2003.

[22] B. T. Phong. Illumination for computer generated pictures. *Communications of ACM*, 18(6):311–317, 1975.

[23] Yusuke Tokuyoshi and Shinji Ogaki. Real-time bidirectional path tracing via rasterization. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 183–190, New York, NY, USA, 2012. ACM.

[24] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.