Semester project spring 2000

Optics Tutorials in Java

Olivier Scherler, Olivier Ripoll (supervisor)

IMT Neuchâtel, Prof. R.Dändliker

July the 7th, 2000

Abstract

A number of problems in optics are not always very easy to to understand using only formulas and equations, hence the need of an alternate and more visual way of representing some of those problems. A good solution is to have a set of computer programs written to demonstrate interactively a given effect. This is why the motivation of this semester work was to provide a base for such programs in order to facilitate their creation. The work focuses on ray tracing because of the wide range of effects visible using it.

Contents

1 Introduction **1**

1.1 The goal 1

1.2 What is Java? 1

1.3 Why Java? 2

1.4 The limitations of Java 2

2 The basics of the project **3**

2.1 Ray tracing 3

2.1.1 Sequential ray tracing 3

2.1.2 Non-sequential ray tracing 3

2.2 General mechanisms 4

2.2.1 The coordinates system 4

2.2.2 Elements 4

2.2.3 Rays 4

2.2.4 Materials 4

2.3 OpticalElement 5

2.4 OpticalDevice 5

2.5 RayPoint 6

2.6 Ray 6

2.7 RayCaster 6

2.8 Material 7

2.9 Parameter 7

3 Details of the program's functionalities **8**

3.1 Elements and devices 8

3.2 Materials and parameters 9

3.3 Propagation 10

3.4 Drawing 10

3.5 Integration with AWT 11

4 Using the program **12**

4.1 The main objects 13

4.2 The main window 13

4.3 Construction of the OpticalDevice 14

4.4 The control pane 15

4.5 Interaction with the controls 16

4.6 Additional features 18

4.6.1 DeviceSwitcher 18

4.6.2 Nothing 18

5 The tutorials **19**

5.1 Field lens 19

5.2 Aspherical interface 19

5.3 Achromat 20

6 Outlook **24**

7 A few remarks **25**

8 Conclusion **25**

9 Acknowledgements **25**

A Equations used for refraction **26**

B The aspherical interface **27**

B.1 Equation of the surface . . . 27
B.2 Intersection of a RayPoint with the surface . . . 27
B.3 Normal vector to the surface at intersection . . . 28

C Refractive index formulas 28

C.1 Constant formula . . . 28
C.2 Schott formula . . . 28
C.3 Conrady formula . . . 29
C.4 Herzberger formula . . . 29
C.5 Sellmeier 1 formula . . . 29
C.6 Sellmeier 2 formula . . . 29
C.7 Sellmeier 3 formula . . . 30
C.8 Sellmeier 4 formula . . . 30
C.9 Handbook of Optics 1 formula . . . 30
C.10 Handbook of Optics 2 formula . . . 30

D Supporting new controls 31

List of Figures

1 The coordinates system . . . 4
2 Nesting of devices . . . 5
3 Types of `RayCasters` . . . 6
4 Propagation . . . 10
5 Drawing . . . 11
6 Moving elements . . . 17
7 The field lens applet . . . 19
8 Problem with an off-axis source . . . 20

9 Effect of the field lens 21
10 Spherical lens with aberrations 22
11 Hyperbolic lens without aberrations 22
12 Aberrations with a hyperbolic lens used off-axis 23
13 Achromat with geometrical aberrations correction 23
14 Refraction using Ewald spheres. 26
15 Aspherical surface. 27

List of Tables

1 `OpticalElement` properties 8
2 `OpticalElement` methods 8
3 `OpticalDevice` properties and methods 9
4 `Material` methods 9
5 `RespondToEvents` methods 16

1 Introduction

1.1 The goal

The goal of this semester work was to find a way to facilitate the creation of computer programs used to demonstrate some optics principles in an interactive way. Such tutorials are interesting because they provide an alternative way of understanding a problem to the usual 'formulas only' point of view.

For this project, we focused on ray tracing, as this method can be used to demonstrate a variety of problems, such as sphericity or chromaticity aberrations, using the same approach.

The requirements for the program were the following:

- The tutorials should be viewable on a variety of platforms, preferably through the internet, to ensure an easy access for everyone. Additionally, the size of a program targeted for the internet should be reasonably small;
- The code should be as reusable as possible, in order to minimize the amount of work to be done to create a new tutorial;
- It should be easy to improve the program by implementing new features, without having to revise the whole source code.

These requirements naturally led to the idea of using object oriented programming (OOP), and more particularly the Java language. The details of the reasons are explained in section 1.3 below.

1.2 What is Java?

Java is a cross-platform, object oriented programming language developed by Sun Microsystems, Inc. The particularity of this language is that it stands between a compiled and an interpreted language, since the compiler produces byte code for a virtual machine, instead of byte code for a given platform (i.e. the combination of the hardware and the operating system).

For each platform, an implementation of the virtual machine is written (usually either by the developer of the operating system or by the manufacturer of the hardware) which executes the byte code and 'translates' it into instructions understood by the underlying computer. In many ways, it is similar to an emulator, but for a machine that never physically existed, hence the name *virtual machine*.

The benefit of this approach is that only the virtual machine has to be written for a given platform, and provided it is installed on a computer, any Java program will run on it, without the need of a re-compilation or even a modification of the code, as usually needed for programs written in languages like Pascal or C++.

Another benefit is that a Java program can be either a standalone application, as any program usually executed on a computer, or a particular kind of application, called *applet*, which is intended to be executed inside of a web browser window. This solution is interesting, because it means that one can create a web page containing, in addition to the applet, text and graphics to provide explanations or background information.

On the drawbacks side, it is important to note that Java applications are slower than their platform specific counterparts, and that the virtual machines used in modern web browsers are several generations behind the current release (usually Java version 1.1.x), forcing the programmer who wants to create internet distributable applets to ignore the new features of the language and to only use the backward compatible classes.

1.3 Why Java?

As said earlier, Java seemed to be the natural choice for this project for several reasons:

- Java's syntax is similar to C++, but cleaner. As a matter of fact, C++ is more of an addition to C implementing object oriented programming than a real new language, whereas Java is a new language, intended to be object oriented from the beginning, and which got rid of all the non OOP features of C++. This means that it's easy to get started with Java. In fact, the harder part is to understand the basics of object oriented programming, not to learn the syntax of the language itself.
- Java is fully object oriented, which makes it possible to write reusable classes with the features logically distributed along the class tree. Furthermore, the behaviours are encapsulated into the classes, which allows someone else to use them without knowing the details of the internal mechanisms.
- The tutorials created should be available to a maximum of users, ideally through the internet, goal which is fully reached using Java applets.

1.4 The limitations of Java

The main drawback of Java is the speed issue. Java applications are run by the virtual machine, which adds a step in the execution, slowing everything down a bit. Furthermore, the different virtual machines are not equivalent in terms of speed, so the only safe way to write an application which works well everywhere is to avoid processor intensive functions as much as possible.

Also, as said earlier, the virtual machines implemented into web browsers are rarely the latest version, so the programmer should be careful not to use any Java 1.2 or older functions.

2 The basics of the project

2.1 Ray tracing

Ray tracing is often used to calculate the aberrations of an optical system. The system is composed of reflective or refractive surfaces, and a number of rays are propagated through the system. The results given by this approach are exact, but only for the rays that were traced, in opposition to aberration theory, that gives only an approximation but for the whole system. In ray tracing, aberrations are not calculated using a separate formula for each aberration type, but are a direct cause of the geometry of the system.

2.1.1 Sequential ray tracing

In sequential ray tracing, the order in which the rays travel through the system is known in advance. Every ray travels once through a given element and then through the next one, until it reaches the end of the system.

This is a good way to simplify the calculations, because the propagation of a ray through a series of elements limits itself to propagating the starting ray through the first element, then the resulting ray through the second element, and so on...

This way, a ray can be split up into elementary parts — one for each element in the system — each with its starting position and direction. The details will be discussed in section 2.2.

2.1.2 Non-sequential ray tracing

In non-sequential ray tracing, the system is described by a number of elements, and rays can travel freely through it. For example, they can enter the same element several times or never. They can even pass an element in a given direction and then being reflected and pass the same element in the opposite direction.

Non-sequential ray tracing is much more complicated to implement, because at each step of the propagation, one must figure out which element is the first one encountered, which means calculating the intersection of the ray with every element in the system, in opposition to finding the intersection with the current element in sequential ray tracing.

For further details about ray tracing, see [1] and [2]

2.2 General mechanisms

2.2.1 The coordinates system

Since we are going to draw on a computer screen, it is easier to stick with the usual coordinates system on such a device, i.e. x horizontally and towards the right and y vertically and down. Using x instead of z for the optical axis seems natural in our case. However, it would be a good idea to prepare everything to work in three dimensions — even if it's not used in the first version — to make it easier to implement later. That's why we added a z coordinate, horizontally and away from the user (figure 1).

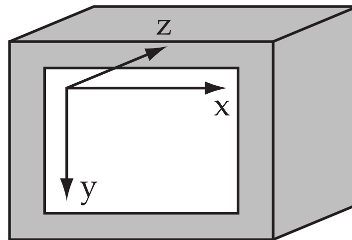

Figure 1: The coordinates system

2.2.2 Elements

To implement sequential ray tracing, it was first necessary to decide of the basic properties of a system. We found that the more convenient way was to define a system as a sequence of elements, which can either be thin or have a finite width, and with no space between them. Therefore, we build a complete system by creating the elements in the correct order, and each new element gets automatically positioned at the end of the system.

2.2.3 Rays

Since we are using sequential ray tracing, we can easily split a ray traveling through the system into elementary parts we called RayPoint. A ray has the same number of raypoints as there are elements in the system. Each element takes the last raypoint in the ray, calculates its corresponding outgoing raypoint and adds it to the end of the ray. The details of this mechanism are given in section 3.3.

2.2.4 Materials

In order to be able to demonstrate effects such as chromatic aberrations, it was necessary to implement the dispersion of the refractive index depending on the wavelength. For this purpose, we created a set of classes that deal with optical materials. It is possible to define a material using the

formulas used in Zemax (and summarized in appendix C) to calculate the refractive index corresponding to a given wavelength.

2.3 OpticalElement

It was clear from the beginning that it would be convenient to have a base class, parent of every element type, that would define all the standard attributes of an element, such as its position and dimensions, as well as the 'external behavior', i.e. the functions common to every element that take care of drawing the element, and propagating the rays through it. This class was named `OpticalElement`.

Of course, for each new element type, the programmer should override these functions to implement the element's actual behavior. Since the base class should not be used as an actual element, but only as a canvas, it was natural to declare it as abstract, forcing the child classes to define the core functions necessary to the program.

2.4 OpticalDevice

The `OpticalDevice` class was written to allow the creation of complex elements that behave like any other element. It is no more than a container of `OpticalElements`, with modified functions for drawing and propagation that passes the correct parameters to the corresponding function of each of its components.

The interesting point with this class is that it is a child of `OpticalElement`, which means that, depending on the context, it can be considered as an `OpticalElement`. This opens a number of possibilities because it implies that an `OpticalDevice` can be nested into another `OpticalDevice` as if it were a normal element, allowing us to build very complex structures using a number of simple 'blocks' (figure 2).

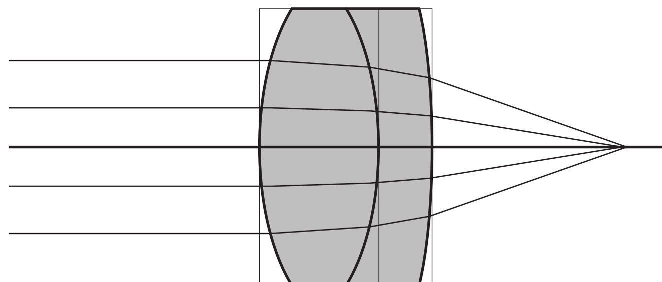

Figure 2: Nesting of devices: Both of the lenses are `OpticalDevices` composed of two spherical interfaces.

Another advantage is that by subclassing `OpticalDevice`, it is easy to create a new kind of element that is defined by a combination of already existing elements (e.g. a thick lens is made of two spherical interfaces with glass between them), without having to define its behavior. The only thing to add

is the code that builds the element list at the creation of the device. Everything else is taken care of by the components. The class `ThickLens` is an example of this feature.

2.5 RayPoint

Class `RayPoint` implements the elementary part of a ray. A list of `RayPoints` is stored in an instance of class `Ray`. Each `RayPoint` defines its position, wave vector $\vec{k}$ and wavelength, along with other properties already implemented but not yet used such as amplitude, phase or polarization.

Every time an element is reached a new `RayPoint` is added to the `Ray`, containing its new properties such as direction or existence. Each element uses the last `RayPoint` in the `Ray` to calculate how its properties will be affected.

2.6 Ray

Class `Ray` is a very simple one. It is basically a dynamic array of `RayPoints`, with methods to access them and to set global properties. The interesting properties of rays are defined in class `RayPoint`, discussed previously.

2.7 RayCaster

A `RayCaster` is a source of light. Its purpose is to create a number of rays to be propagated through the system. `RayCaster` must be subclassed to define a given kind of source, such as a number of parallel rays or rays that all start from the same point.

Four classes were already written to provide useful sources, but it is easy to create new ones by subclassing `RayCaster`.

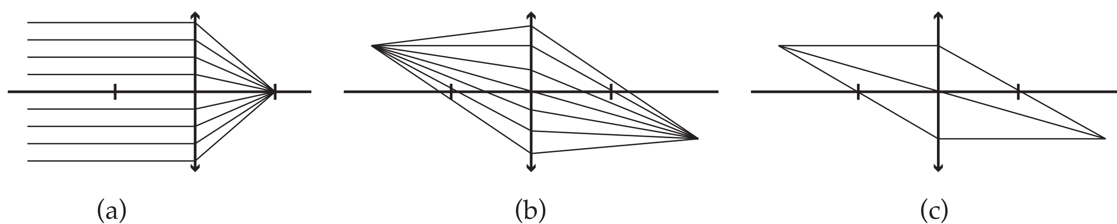

Figure 3: Three types of `RayCasters`: (a) ParallelRays, (b) PointSource, (c) ThreeRays.

ParallelRays (figure 3-a) Creates n parallel rays distributed along a given width. The direction can be chosen freely.

PointSource (figure 3-b) Given a starting point and two directions $\vec{k}_1$ and $\vec{k}_2$, this source creates n rays with directions distributed between $\vec{k}_1$ and $\vec{k}_2$.

ThreeRays (figure 3-c) Creates three rays starting from the same point. One is horizontal and the other two go in the direction of two given points. It can be used to create the three basic rays going through a lens, a horizontal one, one going through the center of the lens and one through the focal point.

RayCastersCollection An array of `RayCasters`. Can be used to create more than one source to propagate through the same system, such as two sets of rays with different wavelengths.

2.8 Material

Class `Material` makes it possible to create a material and assign one or more sets of parameters, depending on the formulas used for index calculation. The available formulas are listed in appendix C.

2.9 Parameter

Class `Parameter` is an abstract class providing the interface needed to communicate with class `Material`. Classes like `ConstantParameter` or `SchottParameters` are child classes of `Parameter`. Such classes implement the formula used to calculate the wavelength, as well as the necessary parameters.

Typically, an instance of a `Parameter` child class is created to store the parameters for the desired formula. Then the `Material` object is created, using the `Parameter` object, and can be used in the creation of an element such as an aspherical interface or a thick lens.

3 Details of the program's functionalities

3.1 Elements and devices

The `OpticalElement` class defines the basic properties and behaviors shared between all the elements. The properties are mainly the position and size of the element. Due to the particular way of building a system by aligning each element on the optical axis and next to the previous one[1], these properties are defined in a slightly unusual way. Table 1 shows them along with a short description.

x	The position in direction `x`. It is usually automatically adjusted to the end of the previous element.
axis_y, axis_z	The position of the optical axis. Usually the same for each element in a system.
offaxis_y, offaxis_z	The position of the element relative to the optical axis. Both default to zero, meaning the element is centered on the axis. These variables are used to move an element off-axis.
width	The size in direction `x` of the element. Determines where the next element will be placed.

Table 1: `OpticalElement` properties

Note: One can see that only the width of the element is defined in class `OpticalElement`. It is because size in other directions is not always needed. For example a lens will have an aperture radius, whereas an homogeneous medium will have no defined dimension in directions `y` and `z`.

Table 2 shows the methods implementing the basic behaviors of an element.

MoveAxis, MoveOnAxis	Used to set the `x`, `axis_y` and `axis_z` properties of the element. Because of the automatic placement of the elements next to each other and on the optical axis, these methods should not be used outside of the class and should be declared `protected` or `private` in a next version of the program.
MoveOffAxis	Used to set the `offaxis_y` and `offaxis_z` properties of the element. This method should be used instead of `MoveAxis` to move the center of an element off axis.
Propagate, PropagateRayPoint, PropagateRayPointSelf	Used to propagate a Ray or a RayPoint through the element. These methods are discussed in details in section 3.3.
Draw, DrawSelf, DrawRay	Used to draw the element and the rays traveling through it on the screen. Discussed in details in section 3.4

Table 2: `OpticalElement` methods

[1]This is not a limitation of the propagation algorithm but a choice made to facilitate the construction of a system. In a future version of the program, the way of building a system can be modified, without having to rewrite the propagation related code.

There are also a variety of accessor functions used to get the value of different properties. They are not described here, see the source code for reference.

The `OpticalDevice` class is a child of `OpticalElement`, and therefore inherits its properties and behaviours. However, in order to be useful, it must add new features. They are listed in table 3.

elements (Vector)	The list of all the components of the device. A `Vector` is a dynamic array that holds any kind of objects. Since every object we store is a child of `OpticalElement` which declares all the methods we need, we don't need to know the real class of each element of this vector. Java takes care of this at runtime.
Draw, DrawRay, Propagate	These methods are overridden by class `OpticalDevice`. The three of them work using the same mechanism: they call the method of the same name for each of the components of the device.
Append	This method adds an `OpticalElement` at the end of the components list. It is used to build the device, element by element.
Rearrange	This method is used to arrange all the components in the `OpticalDevice` in order to align them on the same axis and to put them next to each other in direction `x`. When a device is built, all the elements are first added to the components list using `Append`, and then a call to `Rearrange` puts everything in place. The mechanism behind `Rearrange` is recursive, which means that if a device is included into another device, it gets rearranged too.

Table 3: `OpticalDevice` properties and methods

3.2 Materials and parameters

Table 4 lists the most useful methods of class `Material`. For a list of the `Parameter` classes, see appendix C.

AddParameterSet	Adds an instance of a child class of `Parameter` to use as a formula for the material.
SetDefaultParameter	Used to choose the default formula to be used when calculating the refraction index.
IndexAtWavelength	Returns the refraction index from the given wavelength, using the default formula or the formula given as an optional parameter.

Table 4: `Material` methods

3.3 Propagation

This section describes the propagation of a ray through the system. First, an instance of a `RayCaster` child class creates the rays, each of which contains only one `RayPoint` giving the ray's initial position, direction, etc. Then the `Propagate` method is called successively for each element in the system. The three steps of propagation are described below:

1. `Propagate` takes the last `RayPoint` in the `Ray`, passes it to `PropagateRayPoint` and appends the result to the ray.
2. `PropagateRayPoint` duplicates the `RayPoint` and translates it relative to the current element. It then calls `PropagateRayPointSelf` and returns the result.
3. `PropagateRayPointSelf` calculates the path of the `RayPoint`. It moves it to the last location where any of its properties were modified, and returns it to `PropagateRayPoint`.

At any time, an element can decide that the ray will be stopped — for example when the ray falls outside the element, or in case of total reflection — by calling `Invalidate`. The ray will still exist, but won't be propagated further.

The translation of the coordinates system performed by `PropagateRayPoint` moves the center of the element to the origin. The center is an arbitrary point where it is convenient to have the origin for the calculations. The default location is at the far left of the element on the optical axis, but it can be redefined as needed by overriding method `GetCenter`. Figure 4 illustrates the propagation mechanism.

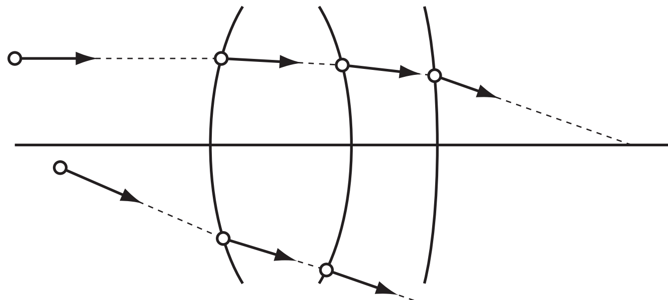

Figure 4: Each circle is the starting point of a `RayPoint`. The arrow shows its direction. Each element creates a new `RayPoint` when it changes any property of the ray. The bottom ray is invalidated by the third element and doesn't propagate further.

3.4 Drawing

Drawing takes place into two steps: displaying the element symbol and tracing the rays. The symbol is displayed using the `Draw` method, which translates the coordinates system — as for propagation — and calls `DrawSelf`. Elements override `DrawSelf` to define how their symbol will be plotted.

Tracing the rays is a bit different. Each element draws the `RayPoint` it propagated, i.e. from the previous element's last modification to the current element's last modification. Because of the ability to nest devices, a numeric marker is passed as a parameter to the `DrawRay` method, in order to keep track of which `RayPoint` should be traced. Each element increments the marker by 1, meaning that when a device draws a part of a ray, the marker is incremented by the number of elements in the device.

There is no limit on how a `RayPoint` is traced. It is usually just a straight line, but for example a graded index fiber would plot a sine, which is one `RayPoint` and not a succession of straight `RayPoints` used to approximate the sine. Figure 5 illustrates the drawing mechanism.

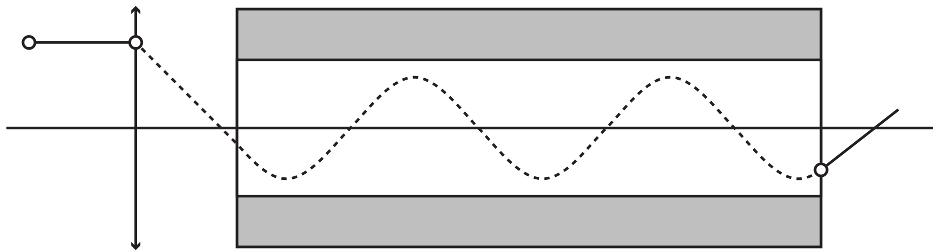

Figure 5: The fiber traces the ray from the lens to its output (dotted line), where the next raypoint is created. The sinusoidal path is defined by the drawing method of the fiber, not by a large number of raypoints approximating it.

3.5 Integration with AWT

The Abstract Window Toolkit (AWT) is a set of standard Java classes used to provide graphical user interface capabilities to an application. It contains definitions for the usual controls, as well as more basic classes that can be extended to draw into a window, such as `Graphics` or `Canvas`.

We wrote a set of classes based on AWT to create a bridge between it and our program. These classes are `OpticalCanvas`, `OpticalControl` and `RespondToEvents`. `OpticalCanvas` is a child of AWT class `Canvas` and is used to actually draw the system in the window, and `OpticalControl` is a child of `Panel` that collects events sent by the controls it contains and sends them to the main application. `RespondToEvents` is an interface class used in `OpticalApp` that declares the callback functions that respond to events sent by `OpticalControl`.

4 Using the program

The program is divided into a number of Java packages, which are listed below:

- **Optical**
  - **Elements:** `OpticalElement`, `OpticalDevice` and all the elements already defined.
  - **Gui:** Graphical User Interface. A number of classes used to integrate a system into a AWT application.
  - **Materials:** `Material`, `Parameter` and all the `Parameter` child classes.
  - **Rays:** `Ray`, `RayPoint`, `RayCaster` and all the `RayCaster` child classes.
- **Utils:** different utility classes.

To prepare a new project, both the Optical and Utils packages should be included. Then a new class should be created, child of `OpticalApp`. The following piece of code shows the structure of the file:

```
import java.awt.*;

import Optical.Elements.*;
import Optical.Rays.*;
import Optical.Materials.*;
import Optical.Gui.*;
import Utils.*;

public class MyApp extends OpticalApp
{
 ...
}
```

The development of a typical application can be separated into five steps, each of which will be described here with a simple example. We are going to write an applet with a point source on the optical axis, two thin lenses we can move and a screen to end the system. The five steps are the following:

1. Declaration of the main objects;
2. Building of the main window layout;
3. Building of the `OpticalDevice` object describing the system;
4. Building of the control pane;
5. Definition of the interactions between the controls and the device.

4.1 The main objects

The application class should declare the main object to be used in the tutorial:

- An `OpticalCanvas`, where the system will be displayed;
- An `OpticalControl`, to provide the user interface;
- An `OpticalDevice` to build the system;
- A `RayCaster` child, to provide the rays to trace.

Once declared, these objects are ready to be created in the next steps.

4.2 The main window

The code below shows an example of the `init` function of an applet. Functions `BuildDevice` and `BuildControls` are discussed in the next sections. We can see here that a new `BorderLayout` is created, and that the canvas and the controls are added to it, the canvas to the center, and the controls below.

```
public class MyApp extends OpticalApp
{
    OpticalCanvas       canvas;
    OpticalControl      controls;
    OpticalDevice       device;
    RayCaster           rays;

    public void init()
    {
        setLayout( new BorderLayout() );

        canvas = new OpticalCanvas();
        BuildDevice();
        canvas.SetDevice( device );

        rays = new ParallelRays(
            new FPoint( 20.0, 80.0, 0.0 ),
            new FPoint( 1.0, 0.0, 0.0 ),
            5,
            100.0,
            0.6328,
            Vaccum );
        canvas.SetRayCaster( rays );
```

```
        add( "Center", canvas );
        controls = new OpticalControl( this );
        BuildControls();
        add( "South", controls );
    }
 ...
}
```

The application class is given as a parameter for the `OpticalControl` constructor, because we want the application to respond to the events generated by the controls inside the `OpticalControl` object.

4.3 Construction of the OpticalDevice

To build the device, we create the `OpticalDevice` object and place it at the right place on the screen. Then we build the elements and add them to the device. Finally, we call `Rearrange` and pass the device to the canvas previously created using `SetDevice`.

```
double  lens1Position = 100.0, lens1FocalLength = 100.0,
        lensDistance = 100.0, lens2FocalLength = 100.0,
        lens1Aperture = 80.0, lens2Aperture = 80.0,
        screenDistance = 200.0, screensize = 100.0,
        totalDistance = lens1Position + lensDistance + screenDistance;

public void BuildDevice()
{
    Material    vacuum;

    vacuum = new Material( "Vacuum", new ConstantParameter( 1.0 ) );

    device = new OpticalDevice();
    device.MoveAxis( 150, 0 );

    // definition of the elements
    h1 = new Homogeneous( lens1Position, vacuum );
    lens1 = new SimpleLens( lens1FocalLength, lens1Aperture );
    h2 = new Homogeneous( lensDistance, vacuum );
    lens2 = new SimpleLens( lens2FocalLength, lens2Aperture );
    h3 = new Homogeneous( screenDistance, vacuum );
    screen = new Screen( screensize, screensize );

    // appending them to the system
    device.Append( h1 );
```

```
    device.Append( lens1 );
    device.Append( h2 );
    device.Append( lens2 );
    device.Append( h3 );
    device.Append( screen );

    device.Rearrange();
}
```

Comments on the above example:

- The elements are declared in the class body, to allow us to access them from the other methods in order to be able to modify the device in answer to user input.
- After creation, the device is placed where we want it on the screen using `MoveAxis`.
- Material vacuum is created using a `ConstantParameter` that gives a refractive index of 1.0 regardless of the wavelength.
- Elements of class `Homogeneous` define the distance and medium between the other elements. The constructor takes the width and the material for the element.
- Class `SimpleLens` implements a thin, paraxial lens of a given focal length and aperture.
- `Append` is called for each element in the order we want them in the device.
- Class `Screen` is used to finish the device. If we forget it, rays will be traced to the beginning of the last element (`h3`) and will stop there, at the location of the second lens.
- Never forget to call `Rearrange` after building the device, or you'll end up with a strange placement of the elements.

4.4 The control pane

The controls are created as usual with AWT, the only difference being that they are added to an `OpticalControl` object, which is also the listener for all the controls. When an event is sent, the `OpticalControl` object forwards it to the application class. We will see in next section how to react to these events.

```
public void BuildControls()
{
    controls.setLayout( null );
    controls.setSize( 250, 80 );

    sb11 = new Scrollbar(
        Scrollbar.HORIZONTAL,
```

```
        (int)lens1Position,  // initial
        1,  // thumb
        0,  // min
        (int)(lens1Position + lensDistance) // max
        );

    sb11.setName( "lens1Pos" );
    sb11.addAdjustmentListener( controls );
    controls.add( sb11 );
    sb11.setBounds( 0, 0, 200, 20 );

    sb12 = new Scrollbar(
        Scrollbar.HORIZONTAL,
        (int)lensDistance,  // initial
        1,  // thumb
        0,  // min
        (int)(lensDistance + screenDistance) // max
        );

    sb12.setName( "lens2Pos" );
    sb12.addAdjustmentListener( controls );
    controls.add( sb12 );
    sb12.setBounds( 0, 40, 200, 20 );
}
```

4.5 Interaction with the controls

Class `OpticalApp` implements `RespondToEvents`, which is an interface class that declares the methods where the event processing will be performed. In `OpticalApp`, these methods are empty. Therefore the child class should override them as needed. Table 5 lists the existing functions. Supporting other controls is fairly easy and is described in appendix D.

Scrollbar	Called when a scrollbar changes value. Parameters are a `String` containing the name of the scrollbar for identification, and an `int` with the new value.
Button	Called when a button is clicked, with the label of the button as parameter.
Checkbox	Called when a checkbox changes its value. Parameters are the name and the new value (as `boolean`) of the checkbox.

Table 5: `RespondToEvents` methods

Basically, the procedure is to override the methods corresponding to the controls used in the application, check the name of the control that fired the event, and act on the device accordingly. Below is an example with the `Scrollbar` method:

```
public void Scrollbar( String name, int value )
{
    if( name.equals( "lens1Pos" ) )
    {
        h1.SetWidth( value );
        h2.SetWidth( totalDistance - (int)h1.GetWidth()
            - (int)h3.GetWidth() );
        sb12.setMaximum( (int)h2.GetWidth() + (int)h3.GetWidth() );
        sb12.setValue( (int)h2.GetWidth() );
        dev.Rearrange();
    }
    else if( name.equals( "lens2Pos" ) )
    {
        h2.SetWidth( value );
        h3.SetWidth( totalDistance - (int)h1.GetWidth()
            - (int)h2.GetWidth() );
        sb11.setMaximum( (int)h1.GetWidth() + (int)h2.GetWidth() );
        sb11.setValue( (int)h1.GetWidth() );
        dev.Rearrange();
    }
    canvas.ForceRedraw();
}
```

There is an important thing to note here. To move elements along the axis, we cannot call their `MoveOnAxis` method, because elements are always supposed to touch each other. So if we move an element, we must adjust the width and position of the other elements as well. However, when we call `Rearrange`, the position of every element is adjusted automatically to ensure they are touching each other, so in order to move an object, we should change the width of other elements instead.

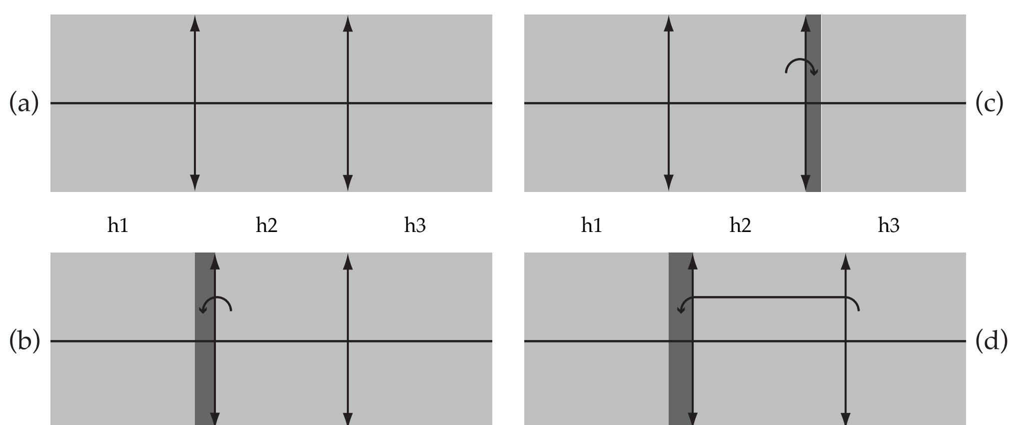

Figure 6: Moving elements

A real life example will illustrate the process more clearly. In the program we are building, we have two lenses and a screen, separated with three `Homogeneous` elements we called `h1`, `h2` and `h3`. `h1`

is before the first lens, h2 between the two lenses, and h3 between the second lens and the screen (figure 6-a).

Let's suppose we want to move the first lens 30 pixels[2] to the right. We can add 30 pixels to the width of h1 and remove them from the width of h2 and the desired effect will be achieved (figure 6-b). Now if we want to move the second lens 20 pixels to the left, we remove 20 pixels to the width of h2 and add them to the width of h3 (figure 6-c). Of course, we have to call Rearrange for the modification to take effect. Afterwards, we call ForceRedraw to refresh the display.

In both cases, the screen doesn't move, since the width we added somewhere was removed elsewhere, so we didn't change the total width of the system. To move both lenses together while keeping the same distance between them, all we have to do is resize h1 and h3 accordingly (figure 6-d).

Of course, we don't need to keep the screen where it is if we don't want to. If we change the width of h3 alone, we are able to move the screen only.

4.6 Additional features

4.6.1 DeviceSwitcher

DeviceSwitcher is a special kind of device, used to switch between two or more alternate configurations for a device. For example, let's imagine we want to demonstrate the differences between a paraxial lens and a real lens. We create both of them, but instead of adding them to the main device, we put them into a DeviceSwitcher which is added to the system. With a control, we can make the DeviceSwitcher display either of the lens in alternance, and if the lens share the same focal length, we have a very nice demonstration of geometrical aberrations.

DeviceSwitcher is very simple to use. Method AddDevice takes an OpticalElement and a string containing its name, and adds it to the list of the available elements to switch. SetCurrentDevice switches to the element of a given name. There is no limit to the number of different elements that a DeviceSwitcher can contain.

4.6.2 Nothing

Nothing is an element that does nothing. It has no width, and its only effect is to add a RayPoint to the rays going through it. It can be used for two purposes: To replace another thin element in a DeviceSwitcher — as seen in the field lens tutorial (section 5.1) where a thin lens can be switched with a Nothing element for comparison — and to take a 'slice' of the rays at a given location in the system, to have access to the RayPoints there.

[2]Although the distances in this version of the program are expressed in pixels, it does not mean that they have to be integer numbers. Distances are stored as doubles and are only rounded when the system is displayed. At the time being, there is a 1:1 correspondence between one unit of distance and a pixel on the screen, but it would be easy to extend the program to support a change of scale.

5 The tutorials

5.1 Field lens

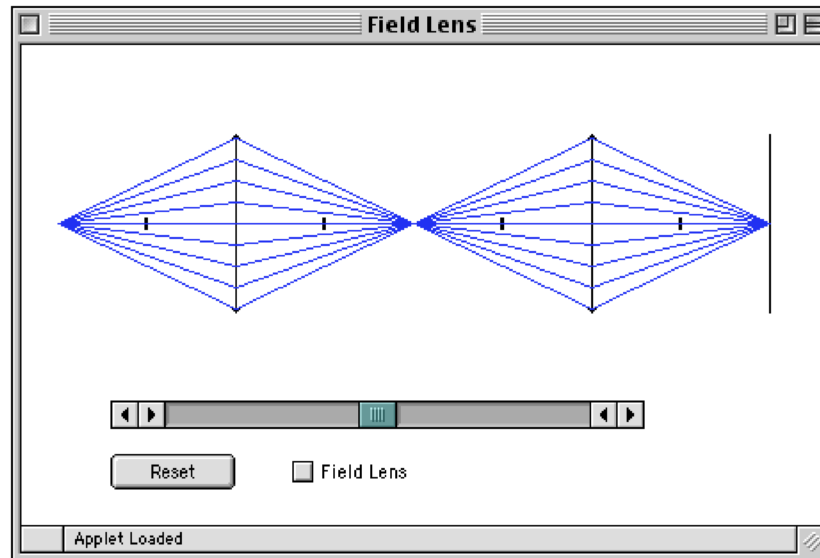

Figure 7: The field lens applet

The *Field lens* tutorial illustrates the effect of a field lens in a telecentric system. Both lenses have the same focal length f and are separated by a distance of $4f$. An object at a distance of $2f$ from the first lens is imaged $2f$ after the second lens (figure 7). However, since the lens has an aperture, if the object is not centered on the optical axis, some of the energy passing the first lens is lost when the second is encountered. This can be seen on figure 8, where some of the rays are lost outside of the second lens.

This is where the field lens is useful. Placed between the two lenses and with the same focal length f, it images the aperture of the first lens onto the aperture of the second one and all the light passing the first lens goes through the second one as well. This can be seen on figure 9, where all the rays pass the three lenses.

Such a setup can be used for photolithography (Microlens Projection Lithography), although with a few modifications. The lenses are replaced with microlens arrays, and two arrays are combined to be used as a field lens array [3].

5.2 Aspherical interface

It is well known that spherical lenses are not perfect. Sphericity aberrations begin to appear when a lens is used too far away from its axis. This effect is easily seen on figure 10: rays close to the axis focalize normally, but the other rays fall elsewhere, and we cannot speak of focalization anymore.

A good way of correcting these aberrations is to use an aspherical lens. The mathematical aspect of

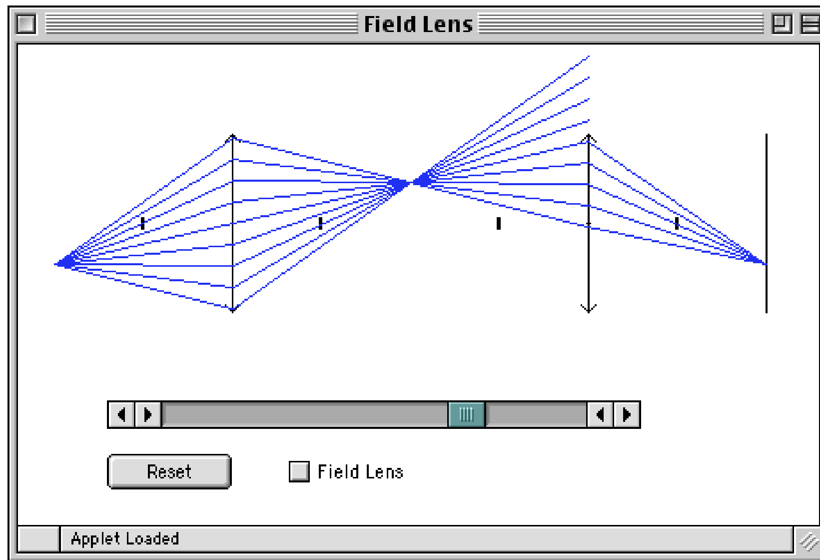

Figure 8: Problem with an off-axis source

the surface used is discussed in appendix B. Such a surface is defined with its curvature C and an asphericity factor K. The different values of K and the surface obtained are listed below:

$K > 0$	Ellipse
$K = 0$	Sphere
$-1 < K < 0$	Ellipse
$K = -1$	Parabole
$K < -1$	Hyperbole

In our case, a hyperbolic interface with $K = -(n_1/n_2)^2$ where n_1 and n_2 are the refractive indexes of the mediums before respectively after the interface give a perfect focalization (figure 11). However, the tutorial makes it possible to see that as soon as the incident rays are not parallel to the axis, the effect can be catastrophic (figure 12). This is one of the reasons why spherical lenses are still used, because their aberrations are almost constant, even when the direction of the incident rays change, the other reason being that aspherical lenses are expensive to manufacture.

5.3 Achromat

The achromat is a combination of two lenses used to correct chromaticity and sphericity aberrations. The first lens is made of a not very dispersive glass (such as BK7), and the other of a highly dispersive one (such as F2). When the curvature radii of the lenses are chosen correctly, the focal length of the achromat is the same for red and blue light, although rays take different paths inside of the lens. The equations used to obtain the radii are the following (see figure 13):

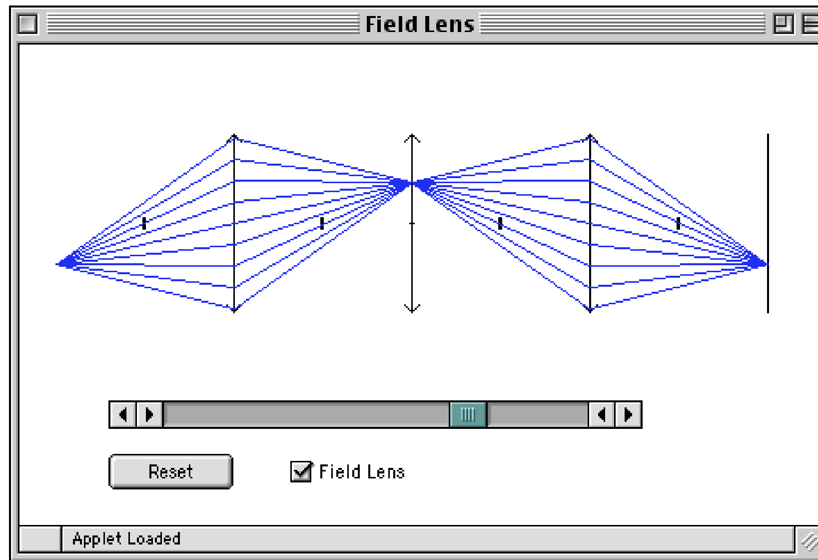

Figure 9: Effect of the field lens

$$\frac{1}{f_{1d}} = (n_{1d} - 1)\left(\frac{1}{r_1} - \frac{1}{r_2}\right) \quad (1)$$

$$\frac{1}{f_{2d}} = (n_{2d} - 1)\left(\frac{1}{r_2} - \frac{1}{r_3}\right) \quad (2)$$

where

f_{1d} : the focal length of the first lens for wavelength λ_d
f_{2d} : the focal length of the second lens for wavelength λ_d
n_{1d} : the refractive index of the first glass for wavelength λ_d
n_{2d} : the refractive index of the second glass for wavelength λ_d
$\lambda_d = 587.6\,\mathrm{nm}$

These equations give a relation between r_1, r_2 and r_3. We can choose, for example, r_3, and calculate r_1 and r_2. The tutorial shows the effect of the achromat, and allows the user to change r_3 (r_1 and r_2 are adjusted automatically in consequence) to try to correct spherical aberrations. It is visible that the best configuration is when the curvature is distributed evenly between the three interfaces, as shown on figure 13.

For the program, we used two materials that does not exist, but that we created using the Conrady parameters to exaggerate the effect of chromaticity aberrations, in order to make them visible in the program.

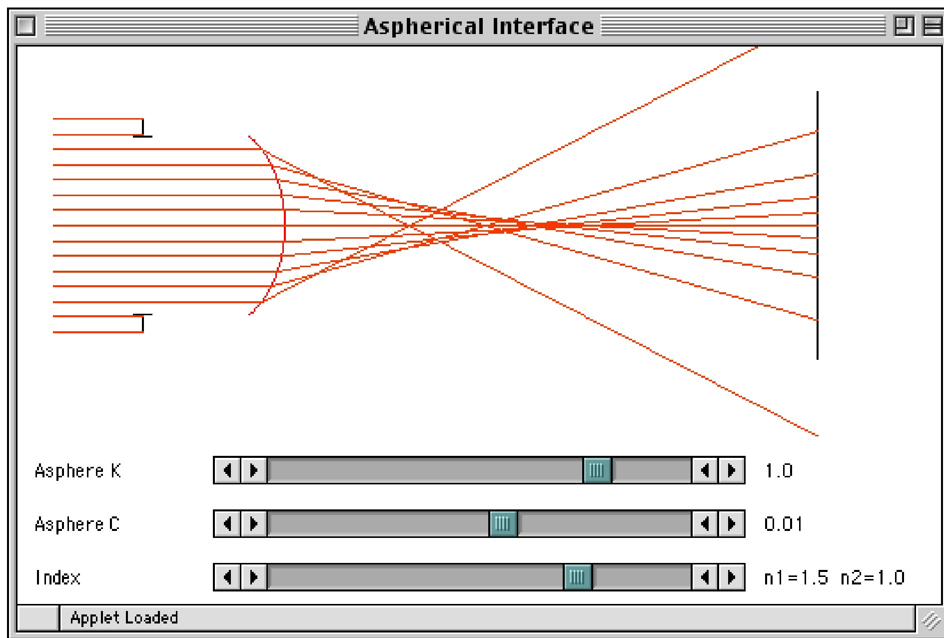

Figure 10: Spherical lens with aberrations

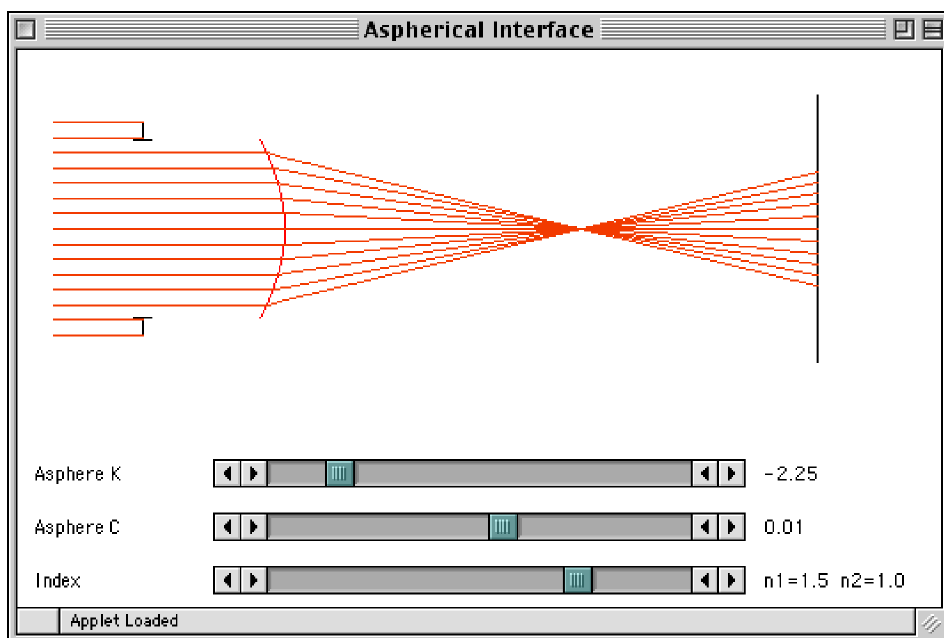

Figure 11: Hyperbolic lens without aberrations

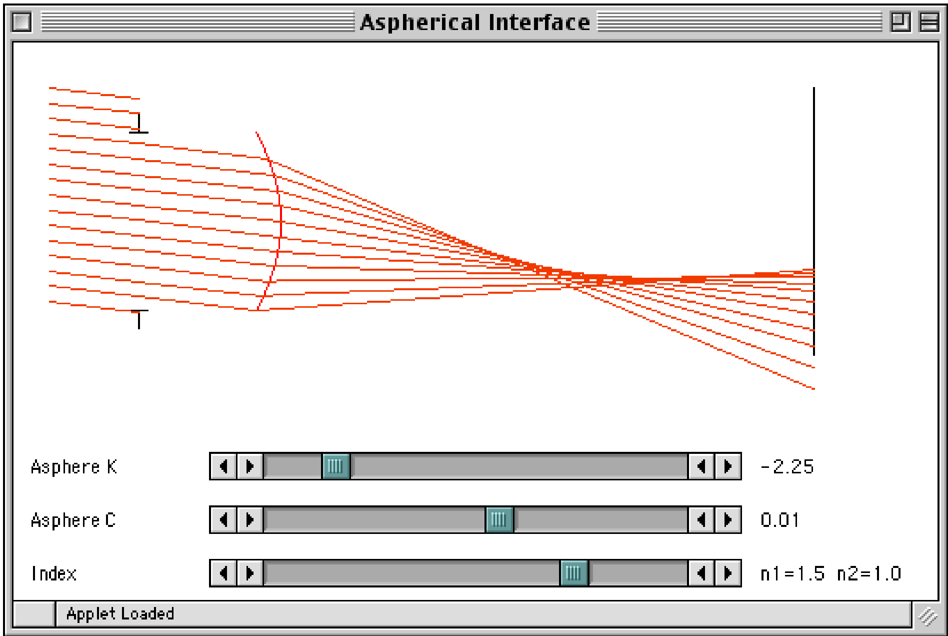

Figure 12: Aberrations with a hyperbolic lens used off-axis

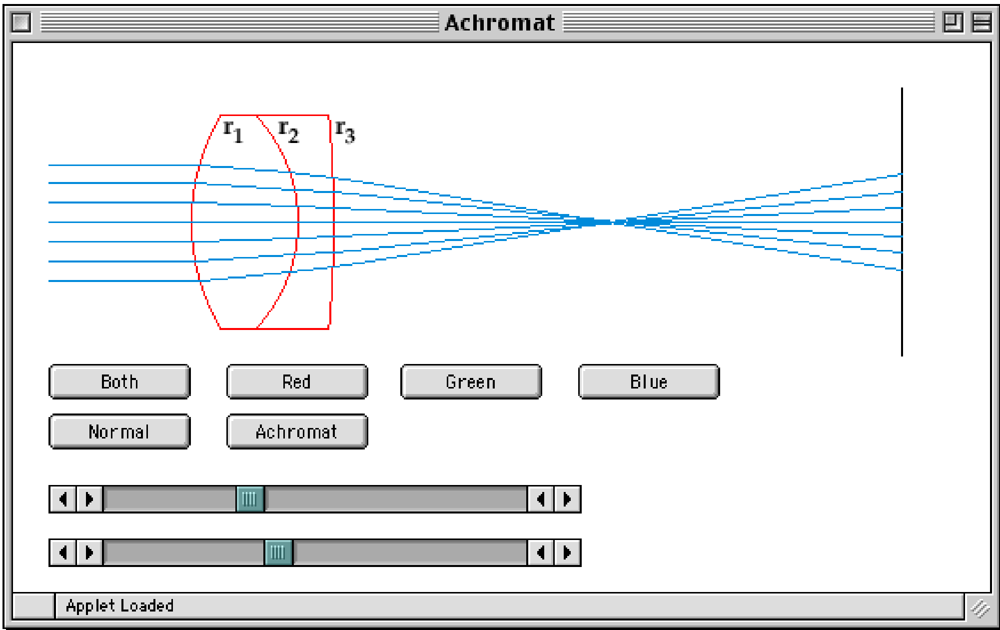

Figure 13: Achromat with geometrical aberrations correction

6 Outlook

While developing the program, we thought of a number of improvements that we decided not to implement right away. They are mentioned in this section to serve as a source of ideas for anyone wanting to work with this program.

- Although the display takes place only in two dimensions, the propagation algorithm has everything prepared for three dimensions. The properties are present in elements and rays and most of the propagation methods already take them into account. For the other, the z component was set to zero to avoid surprises but the equations should be correct.
- A number of properties could be added to rays, to make it possible to illustrate new problems and solutions. For example, amplitude and phase, or polarization.
- New elements could be added, either as combinations of existing elements or as completely new ones. For example, a special kind of device could be written to create matrices of other elements.
- At the time being, the program's internal coordinates system is tightly bound to the usual coordinates system on a computer screen. It would be interesting to completely separate both systems, to allow things such as 3D rotation or zooming.
- The program could display more than a side view of the system. For example a picture of the rays reaching the screen or a cut of the system.
- Devices, the way they are built, force their components to touch each other. However, it is not a limitation in the propagation algorithm. A good idea would be to redefine the way devices are constructed, to allow a more intuitive building of systems.
- It could be interesting if material and device definitions could be read from a file on the hard drive, instead of being 'hard coded' into the program. Such a feature would simplify the creation of a system, and could even lead to the development of an editor.
- A class providing mouse input functionalities, in order to be able to move elements directly by clicking on them would greatly reduce the number of controls needed for a given program.

7 A few remarks

- The program is released under the GNU General Public License (GPL). For more information about GPL, see [4].
- The amount of code written for this program can seem big. As a matter of fact, for a simple system, the code would be much more compact if it were written from scratch for this specific application. However, it would then be very difficult to make the program evolve (like by adding a second lens to a single lens system) without rewriting everything. In opposition, the classes written for this semester work allow the programmer to build a modular system without having to revise the whole code at each modification, and the code that must actually be written to make an application is very reduced considering the possibilities offered by what was already written.
- The algorithm used for ray tracing is not paraxial. However, paraxial elements can be implemented. An example of this is class `SimpleLens` which implements a lens using paraxial laws for propagation.
- It is possible to propagate rays without having to draw them. This can be used to obtain a spot function for example, by calculating the final position fo a great number of rays, without having to display them on screen. One can imagine a reduced set of rays used to display in the system, and another set used to calculate the spot function.

8 Conclusion

We wrote a set of Java classes allowing anyone with some programming knowledge to create interactive optics tutorials based on ray tracing. The classes can be easily extended to provide more functionalities, either at the user interface level or at the optical level.

This work combined learning of the Java programming language and learning of many optical subjects. It was very interesting to have to deal with both the optical and the programming problems at the same time, even if it was sometimes difficult to find a good compromise between correct functionality and program complexity. There are still lots of improvements that can be made but the base is laid and I think the goal is achieved.

9 Acknowledgements

I wish to thank my assistant Olivier Ripoll, whose support was invaluable. Without him, I would have missed many good ideas, and many problems in the conception of the program would have been discovered too late to be easily solved. His help with problems in optics was essential too. I also wish to thank my friends Stephan, Johann and Charlotte, who always supported me, bearing with my temper when everything went the wrong way and always listening to me even when talking of the most specific and boring detail, which helped me a lot to clarify my thoughts.

A Equations used for refraction

Figure 14 shows refraction calculated using the Ewald spheres. It is easy to see that between $\vec{k}_1$ and $\vec{k}_2$, only the component in the direction normal to the surface changes, and the amount of the change can be easily calculated using the Snell law.

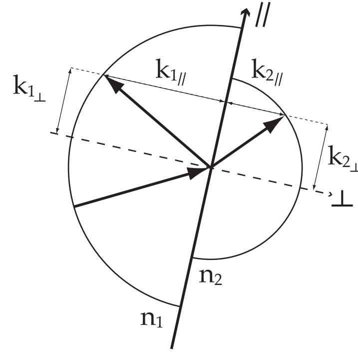

Figure 14: Refraction using Ewald spheres.

$$k_1 \sin\theta_1 = k_2 \sin\theta_2 \tag{3}$$

This leads to the conclusion that all we have to do in order to find the wave vector after refraction is to add a carefully chosen multiple of the normal vector to the incident ray, multiple which is calculated using the Snell law.

The following parameters are needed before we can calculate the direction of the refracted ray:

- the wave vector of the incident ray $\vec{k}_1$;
- the wavelength of the incident ray λ;
- the index of refraction $n_2(\lambda)$ of the second material, depending of the wavelength;
- the coordinates $\vec{r}_2$ of the intersection of the ray with the interface;
- the normal vector $\vec{n}$ to the surface at the location of the intersection.

Then the following equations give the wave vector of the refracted ray $\vec{k}_2$:

$$\begin{aligned} k_2 &= \frac{2\pi n(\lambda)}{\lambda} \\ \vec{k}_2 &= \vec{k}_1 + \mu\vec{n} \\ a\mu^2 + b\mu + c &= 0 \end{aligned} \tag{4}$$

with

$$a = ||\vec{n}||^2$$
$$b = 2\vec{k}_1.\vec{n}$$
$$c = k_1^2 - k_2^2 \quad (5)$$

B The aspherical interface

B.1 Equation of the surface

The equation describing the aspherical interface — adjusted to our coordinates system as shown on figure 15 — is the following:

$$x = -\frac{C(y^2 + z^2)}{1 + \sqrt{1 - C^2(K+1)(y^2 + z^2)}} \quad (6)$$

Note: This is only one of the possible equations one can use to describe an aspherical surface.

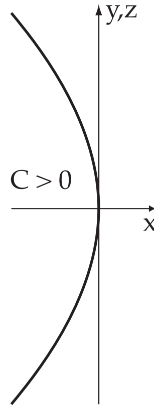

Figure 15: Aspherical surface.

B.2 Intersection of a RayPoint with the surface

The RayPoint is given by its position $\vec{r}_1$ and wave vector $\vec{k}_1$. The intersection is called $\vec{r}_2$.

$$\vec{r}_2 = \vec{r}_1 + \mu\vec{k}_1 \quad (7)$$
$$a\mu^2 + b\mu + c = 0$$

with

$$\begin{aligned} a &= -C\left((K+1)k_{1x}^2 + k_{1y}^2 + k_{1z}^2\right) \\ b &= -2C\left((K+1)k_{1x}r_{1x} + k_{1y}r_{1y} + k_{1z}r_{1z}\right) - 2k_{1x} \\ c &= -C\left((K+1)r_{1x}^2 + r_{1y}^2 + r_{1z}^2\right) - 2r_{1x} \end{aligned} \tag{8}$$

B.3 Normal vector to the surface at intersection

A normal vector to the surface is given by the following equations:

$$\begin{aligned} n_x &= -1 \\ n_y &= -\frac{Cr_{2y}}{\sqrt{1 - C^2(K+1)(r_{2y}^2 + r_{2z}^2)}} \\ n_z &= -\frac{Cr_{2z}}{\sqrt{1 - C^2(K+1)(r_{2y}^2 + r_{2z}^2)}} \end{aligned} \tag{9}$$

C Refractive index formulas

Following are the formulas used to calculate the refractive index of a material depending of the wavelength, taken from the Zemax manual [5] and the Handbook of Optics [6]. The name of the corresponding child class of `Parameter` is also given.

Important note: For these formulas to work with the parameters given in Zemax, the wavelength must be expressed in micrometers.

C.1 Constant formula

Name in `Material` table: Constant
Name of class: `ConstantParameter`
Parameters: n_0
Formula:

$$n = n_0$$

C.2 Schott formula

Name in `Material` table: Schott
Name of class: `SchottParameters`

Parameters: $a_0, a_1, a_2, a_3, a_4, a_5$
Formula:

$$n^2 = a_0 + a_1\lambda^2 + a_2\lambda^{-2} + a_3\lambda^{-4} + a_4\lambda^{-6} + a_5\lambda^{-8}$$

C.3 Conrady formula

Name in `Material` table: Conrady
Name of class: `ConradyParameters`
Parameters: n_0, A, B
Formula:

$$n = n_0 + \frac{A}{\lambda} + \frac{B}{\lambda^{3.5}}$$

C.4 Herzberger formula

Name in `Material` table: Herzberger
Name of class: `HerzbergerParameters`
Parameters: $\lambda_0, A, B, C, D, E, F$
Formula:

$$\begin{aligned} \lambda_0^2 &= 0.028 \\ L &= \frac{1}{\lambda^2 - \lambda_0} \\ n &= A + BL + CL^2 + D\lambda^2 + E\lambda^4 + F\lambda^6 \end{aligned}$$

C.5 Sellmeier 1 formula

Name in `Material` table: Sellmeier1
Name of class: `Sellmeier1Parameters`
Parameters: $K_1, L_1, K_2, L_2, K_3, L_3$
Formula:

$$n^2 - 1 = \frac{K_1\lambda^2}{\lambda^2 - L_1} + \frac{K_2\lambda^2}{\lambda^2 - L_2} + \frac{K_3\lambda^2}{\lambda^2 - L_3}$$

C.6 Sellmeier 2 formula

Name in `Material` table: Sellmeier2
Name of class: `Sellmeier2Parameters`

Parameters: $A, b_1, \lambda_1, b_2, \lambda_2$
Formula:

$$n^2 - 1 = A + \frac{b_1 \lambda^2}{\lambda^2 - \lambda_1^2} + \frac{b_2 \lambda^2}{\lambda^2 - \lambda_2^2}$$

C.7 Sellmeier 3 formula

Name in `Material` table: Sellmeier3
Name of class: `Sellmeier3Parameters`
Parameters: $K_1, L_1, K_2, L_2, K_3, L_3, K_4, L_4$
Formula:

$$n^2 - 1 = \frac{K_1 \lambda^2}{\lambda^2 - L_1} + \frac{K_2 \lambda^2}{\lambda^2 - L_2} + \frac{K_3 \lambda^2}{\lambda^2 - L_3} + \frac{K_4 \lambda^2}{\lambda^2 - L_4}$$

C.8 Sellmeier 4 formula

Name in `Material` table: Sellmeier4
Name of class: `Sellmeier4Parameters`
Parameters: A, B, C, D, E
Formula:

$$n^2 = A + \frac{B \lambda^2}{\lambda^2 - C} + \frac{D \lambda^2}{\lambda^2 - E}$$

C.9 Handbook of Optics 1 formula

Name in `Material` table: HoO1
Name of class: `HoO1Parameters`
Parameters: A, B, C, D
Formula:

$$n^2 = A + \frac{B}{\lambda^2 - C} - D\lambda^2$$

C.10 Handbook of Optics 2 formula

Name in `Material` table: HoO2
Name of class: `HoO2Parameters`
Parameters: A, B, C, D
Formula:

$$n^2 = A + \frac{B \lambda^2}{\lambda^2 - C} - D\lambda^2$$

D Supporting new controls

Extending classes `OpticalControl` and `RespondToEvents` in order to support other control types is fairly easy.

The first thing to do is to make `OpticalControl` a listener to the kind of events sent by the new control. This is done by adding the corresponding listener interface to the class' `implements` list. The following example shows how check box support would be added (although it is already present):

```
public class OpticalControl extends Panel
    implements AdjustmentListener, ActionListener, ItemListener
{
 ...
}
```

`ItemListener` is the interface class that listens to check box events. The listening method must be added:

```
public void itemStateChanged( newEvent e )
{
    if( e.getItemSelectable() instanceof Checkbox )
        resp.Checkbox( ((Checkbox)e.getItemSelectable()).getName(),
        (e.getStateChange() == ItemEvent.SELECTED) );
}
```

The `CheckBox` method must then be declared in class `RespondToEvents`:

```
public void Checkbox( String name, boolean value );
```

Finally, an empty `CheckBox` method must be defined in class `OpticalApp` or the user will be forced to define it in his own application class even if he does not use the new control:

```
public void Checkbox( String name, boolean value )
{
}
```

References

[1] J. M. Teijido, "Conception and design of illumination light pipes," Université de Neuchâtel, 2000.

[2] V. N. Mahajan, "Optical imaging and aberrations," SPIE Optical Engineering Press, 1998.

[3] R. Völkel, H.-P. Herzig, P. Nussbaum, R. Dändliker, W. B. Hugle, "Microlens array imaging system for photolithography," Opt. Eng. **35**(11) 3323–3330 (November 1996).

[4] "GNU General Public License", `http://www.gnu.org/copyleft/gpl.html`.

[5] "Zemax User's Guide, Version 8.0," Focus Software, Inc.

[6] M. Bass (ed.), Optical Society of America, "Handbook of Optics, second edition", McGraw-Hill, 1995.