

A New Computational Model
for
Optical Analysis

A Thesis Submitted for the Degree
of
Doctor of Philosophy of the University of London
by
Ronald Szumski

Optical Science Laboratory
Department of Physics and Astronomy
University College
University of London
1998



ProQuest Number: U643664

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest U643664

Published by ProQuest LLC(2016). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code.
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

The task of optical analysis and design has benefitted immensely from the power that the modern computer has brought to bear upon the highly numerical processes involved. Ever since the earliest optical analysis tools were designed, the underlying model adopted by the scientific programming fraternity has changed very little. This model is based upon the sub-division of a general optical system into a sequence of surfaces. In the intervening years, computer languages and the associated operating systems have become more and more sophisticated. During the last decade there has been an increasing emphasis placed upon object modelling, and several object-oriented programming languages have been developed in order to support this new view. This thesis attempts to describe how such modern theories can be employed to construct a radically different optical model that is in keeping with current computer modelling practices. The background of earlier optical models is explored, detailing their strong and weak points. Prior to a full description of the proposed model, the various facets of object-oriented programming are described. A new computational model is then developed, highlighting those features where model and reality closely approach one another. Finally, research work undertaken by the author into the field of immersed-echelles is reported.

Acknowledgements

I would like to take this opportunity to give recognition to those in my life who have been instrumental in my completing this doctorate. My greatest thanks go to my wife, who gave me much emotional and practical support, allowing me to step out of the family circle when necessary and into that solitary existence that all researchers are all too familiar with. My son Gregory, who is now 9 years old, thankfully proved to be the necessary counterweight, often reminding me that the real world has even greater rewards. My mother too was very supportive and gave me encouragement through the years. Unfortunately my father died just a few months before I commenced this work, but it was he who many years ago first set me on the road to science. I wouldn't have made it without him. On a practical note, my farsighted employer allowed me to take the necessary time off to complete this written work. Thankyou, Mats.

In undertaking any form of research, much depends upon the local environment in which it takes place. I consider myself fortunate to have been surrounded by so many young (some not so young) and keen minds, as are to be found within the Optical Systems Laboratory of University College and Sira Ltd. The many stimulating conversations I had with co-workers proved to be most beneficial, particularly when my spirit was low or flagging, or when I needed inspiration. Thanks guys.

Contents

1	Introduction	12
1.1	Evolution	16
1.2	Handling Complexity	18
1.3	A New Viewpoint	18
2	Lens Modelling	20
2.1	Requirements	21
2.2	The Surface-Based Model	23
2.3	The Component-Based Model	27
3	Object Oriented Programming	34
3.1	Encapsulation	36
3.2	Inheritance	39
3.3	Polymorphism	42
4	Implementation Details	46
4.1	Computer Platform	46
4.2	Programming Languages	48

4.2.1	C++	50
4.2.2	Visual Basic	53
4.2.3	Delphi/Object Pascal	57
4.2.4	Other Development Tools	61
4.2.5	Conclusion	61
5	Description of Model	63
5.1	The Toolbar	64
5.2	The System Form	66
5.3	The Lens Container	69
5.4	Methods and Properties of TLens	72
5.5	Editor Linkage	75
5.6	Support Modules	78
6	Lens Internals	84
6.1	Glasses	85
6.2	Surfaces	87
6.2.1	The Abstract Surface	87
6.2.2	The Plane Surface	90
6.2.3	The Conic Surface	95
7	Optical Ray Implementation	99
7.1	Paraxial Rays	100
7.2	Finite Rays	102

7.3	Ray Containers	106
7.4	The Component-Ray Interface	107
8	Detailed Component Descriptions	112
8.1	The Source	113
8.2	The Spacer	117
8.3	The Thick Lens	118
8.4	The RefAlign	121
8.5	The Prism	124
8.6	The Double-Pass Components	125
9	Optical System Processes	129
9.1	Validating the System	132
9.2	The Lens Sequencing Algorithm	136
9.3	Pupil Location	140
9.4	Other System Processes	144
10	Evaluation of Model	145
10.1	Raytracing Through Simple Lenses	148
10.2	Configuring the Prism	151
10.3	Decollimating and Focussing of the Output Beam	153
10.4	Raytracing a Littrow Spectrograph	156
10.5	Conclusion	160
11	Future Trends	162

11.1 The Object Model	163
11.2 Improving the Picture	164
11.3 Intelligence and Optimisation	165
11.4 Classical Optimisation	168
12 Conclusion	171
A Glossary	176

List of Figures

2.1	The Doublet-Lens Icon	28
2.2	The Lens-System Container	29
5.1	The Toolbox	64
5.2	The OptikWerks ‘Worksheet’	67
5.3	IRIS Configuration Editor	68
5.4	The Delphi Object Inspector	76
5.5	The IRIS Lens Editor Inspector	77
7.1	Paraxial Ray Types	101
8.1	Source in Lens Editor	114
8.2	Thick Lens in Lens Editor	119
8.3	The action of TRefAlign upon Reference Rays and Ray-Sets	121
8.4	A Simple Spectrograph	123
8.5	The Spectrograph Form	123
8.6	Prism in Lens Editor	125
8.7	A Double-Pass Sequence of Lenses	127

9.1	Hierarchical structure of the IRIS program	130
9.2	The Source Pop-Up menu	132
9.3	An Example of Component Interfacing	134
10.1	The IRIS Program, showing analysis of Littrow Mount	147
10.2	Arrangement of Components for Test #1	148
10.3	Arrangement of Components for Test #2	152
10.4	Arrangement of Components for Test #3	154
10.5	The Zemax Program, showing analysis of Littrow Mount	157
10.6	The Littrow system as represented by IRIS	159
10.7	Full-field spot diagrams of the Littrow Mount as obtained by Zemax (LHS) and IRIS (RHS).	160

Preface

When I first embarked upon this research program I had little idea where it would take me. Initially, the intention was to write a computer program that would facilitate the analysis of complex optical systems, particularly systems such as astronomical spectrographs. One of the key features was to be the calculation of spectral throughput, taking into consideration glass absorption and multi-layer interference coatings.

One of the first problems that I encountered was the selection of a suitable computer language in which to write the program. I had decided early on that the target operating system was to be Microsoft Windows v3.11, the most current version at the time. Suitable languages for this platform were C++ and Turbo/Borland Pascal, and it was these that I explored in the initial stages. Shortly afterwards Visual Basic (VB) appeared. The writing of a Windows program was much facilitated by this new arrival, and it rapidly gained in popularity throughout the business world.

Eventually, it too began to show its limitations as my requirements slowly shifted with time. The lack of true Object-Oriented Programming features meant that the rather complex ideas I was trying to develop could only be implemented with the greatest of effort. I had been aware some time earlier of a product known menacingly as VB Killer, or VBK. This was being developed by Borland International as a true object-oriented visual programming language. Its basis was a variant of Pascal known as Object Pascal. I instinctively knew that this was to be my salvation, but would it arrive in time? I shall leave the reader to ascertain the details further in this thesis.

The main reason I mention this is so that the reader who is unfamiliar with modern computing trends can appreciate the shifting sands upon which this work has been based. Computer operating systems and application development languages have been of the greatest concern to programmers and developers since the invention of the modern computer. The one feature that has remained constant in this cyber-world is the ever-increasing pace of change. Standards, once introduced in order to maintain some degree of stability, are

regularly superseded by more advanced standards, more closely reflecting current and future technologies.

If you think that the poor pedestrian would-be computer user of today has a difficult time choosing which hardware/software packages to invest in, then spare a thought for the programmer who is at the leading edge of his/her field. It is only through the actions of long-sighted companies, such as Microsoft and Borland, that the ever increasing complexity of the computer world can be made manageable to lesser mortals like ourselves. In doing so, such endeavours as these will allow those of us who seek to create new worlds the perfect and timely opportunity to do just that.

In a world where change is the norm, and complexity is the vehicle of change, I sometimes worry as to where it will all end. I think this because I cannot imagine a world where change, at the pace we are used to today, can continue onwards indefinitely. Either science, the world's resources or our own imaginations, will eventually, I believe, falter. That said, the reader may conclude that I am one of the world's many pessimists, and who am I to argue.

Finally, I wish to exhort all scientists who are actively engaged in developing technical software applications to consider the new vistas that object oriented programming languages make possible. I have read in many computer-related journals and magazines of the difficulties faced by general programmers in adapting to this new approach, but my own experience, if typical, leads me to believe that scientists should find the transition to be much easier. I say this because I am aware that scientists are involved in developing system models that encapsulate how the elements of a model are constructed and how they interact with one another. In essence, we are constructing *objects* and describing their *behaviour*. These qualities exactly correspond to those we would develop and make visible in an object-oriented application. Such a close correspondence should not be ignored.

Chapter 1

Introduction

The following chapters go into great detail as they begin to unfold for the reader each of the premises upon which this thesis is based, but it is probably appropriate at this point if I try to summarise them in a manner that is synonymous with a historical record, while also imparting the degree of inherent innovation that I believe each to have.

The origins of this work lay in a perceived need to quantify the optical throughput of an optical system, particularly systems belonging to the class we know as ‘astronomical spectrographs’. This was to be achieved by the design and construction of an optical analysis program (IRIS), allowing configuration data to be entered by the designer (user) and subsequent analysis to be undertaken. As time passed the emphasis began to shift away from the physics-related algorithms and towards the design of the user interface. Through my own work in the past as an optical designer I knew that any successful lens analysis is dependent upon i) the analysis program being accurate and reliable, and ii) the description of the system as entered by the designer being a faithful one. My own experience has taught me that the majority of problems usually arise from a fault or error in the second part, while defects in the program code are much less common. In general, most optical design programs require the user to enter system data in a manner that is peculiar to the program, but for simple axisymmetric systems all programs seem to agree on a similar input format. In contrast, a generalised non-axisymmetric optical system, such as a spectrograph, poses far greater problems, not only for the lens designer but also for the systems programmer who might be responsible for the general *look-and-feel* (‘usability’) of the program.

If a lens designer is unable to understand every syntactical nuance of data entry, which results in incorrect data being input into the program, then regardless of what power lies under the 'hood' (*English: bonnet*), it can never be properly realised. This dilemma caused me to reconsider the direction my work should take. I realised shortly afterwards that the real problem stemmed from the disparity in models that were held by the program and by the user. It became clear to me that during the many years that software was being developed for the purpose of enhancing the capabilities of the optical designer, at no time had any programmer ever considered the internal or imaginary model that a lens designer held, or so it appeared. It would be conceited indeed to think that users should, for a period of at least 20 years, be required to abandon the physical model of a lens system in favour of a system based upon a sequentially ordered series of surfaces, but in fact this is exactly what has happened. Optical designers do not necessarily achieve a reputation based upon the quality of their designs or analyses alone, but also for their ability to understand the most complex workings of some of the most complex design programs. If the human-computer interface is so essential to the task of design and analysis, then why can't it be made simpler? This was the question that I posed myself, and the solution I found is recorded herein.

Probably the primary reason why the interface had remained so counter-intuitive for so long was because the necessary programming tools (and operating systems) could not support such a complex model. But that was in the past, and I realised that such a restriction might not apply to today's software technology, and with that in mind I proceeded to create a radically new model in software that would be the similitude of the user's model in the mind's eye, that same model that had to be abandoned each time a designer logged-on to a conventional design program. It was my belief that the time was right to attempt such a transition in models for the following two reasons: i) most operating systems (OSs) have now evolved to encompass a graphical user interface (GUI) that is itself a representation, or model of a desktop; since the OS supports such a capability for model abstraction, then it would be likely that any other model might also be simulated, within reason; and ii) language development tools have recently progressed in such a way as to facilitate, as never before, the creation of complex GUIs; along with the new object-oriented features (see Chapter 3) that these languages support, these GUIs may be coupled to advanced system models in much the same way as the operating system is coupled to the model of a desktop, i. e. a filing cabinet may be viewed on screen as a

filing cabinet icon, and its contents viewed as a list of documents.

The first challenge was the development of an ideological model around which I could create a system that supported the normal physical attributes and characteristics of a real optical system (i. e. lens insertion, deletion, copying, moving, editing, etc.) at both the screen and code levels. Such a task in itself is quite a challenge, but fortunately there was one development tool (Delphi) that offered an environment that was object-oriented at both these levels, and so completion of the task was atleast feasible. The main aim at this stage was to arrive at a form of development editor that minimised the use of the keyboard, and this was acheived by providing greater functionality for mouse operations. This approach proved to be quite successful, if measured by the fact that virtually all but one of the above functions (insertion, deletion, etc.) are completed through simple mouse operations.

The second challenge was to create a graphical screen environment that would provide the user with sufficient functionality to undertake the required operations as briefly outlined in the preceding paragraph. A major step forward was the reduction of a generalised optical system into a much simpler form that conformed with normal design practices. The lens system as a whole was abstracted to resemble a standard window, while the lens elements were reduced to iconic form (components) that occupied distinct positions upon a rectilinear grid within the system window, much like a chess board populated with chess pieces. System editing functions, such as lens moving and copying, are facilitated by the computer mouse, much in the same way as these functions are accessed in Windows 3.X and other GUI operating systems. In addition, non-optical components such as axial tilts and decentrations may be represented in a similar manner; in fact, using this architecture it is possible to represent and correctly integrate almost any optical and non-optical component into the system environment. This powerful feature is used later on (see Chapter 8) to develop the notion of a multi-pass optical component, something that is extremely difficult, or even impossible to simulate in current commercial software offerings.

Dispensing completely with prior practices, I abandoned the more common spreadsheet editor in favour of a component editor, in keeping with the component centred model that I was developing. This approach is not of my own creation, but is a solution adopted by companies such as Borland and Microsoft and included in their Rapid Applications

Development (RAD) tools: component based programming environments such as Delphi and Visual Basic. The component editor has proved to be universally popular and extremely flexible, easily adapting to unforeseen components of the future. I could do a lot worse than follow in the same footsteps as these giants of the software industry!

In order to bring all these ideas together into a working model it proved necessary to design many subsidiary software modules, not least the components themselves, which comprised such familiar elements as the prism and the single lens. Two such items that are not usually associated with an optical analysis program at this stage of development are customised **Vector** and **Matrix** units. I had anticipated earlier on that since the tracing of polarised light rays might be an important feature of the program, then such modules would be of great benefit. In fact they are employed by virtually all algorithms that can possibly use them, with the exception of the routines responsible for ray transfer and ray refraction, which are coded for speed rather than elegance. Additionally, reflecting the versatility of the **vector** class, the all-important finite ray is actually a vector ‘masquerading’ as a ray. Both the **Vector** and **Matrix** units supply the class definitions that enable real entities of either type to be created at any point within the code. In order to support the many operations that such types are capable of entering into, both units also provide class methods (procedures) that enable a large variety of familiar operators (dot, cross, vector rotation, etc.). A particular problem that I encountered involved those class methods that ideally should return a new object, but which proved difficult in practice due to the dynamic nature of the **vector** and **matrix** entities (see §5.6 for a more complete description of the problem and its solution). The solution proved to be both effective and innovative, and is known to be applicable to other categories of this problem peculiar to Object Pascal and similar languages.

The final few chapters consider how the completed program and its underlying component-based architecture perform under actual operating conditions. Though the various tests applied to it may not appear overly difficult or stringent, they do however reveal the great potential of such a novel approach. Constructing a bank of prisms and then tracing a ray through them is normally a very difficult task under any raytrace program, but IRIS succeeds in performing the same task with simplicity combined with efficiency. The unique architecture also lends itself to other more sophisticated developments, such as ‘cooperative’ and ‘intelligent’ components (see Chapter 11, ‘Future Trends’), that are able to engage in semi-autonomous behaviour. Such enhancements could constitute the

first step toward, for example, a self-optimising system - the theory of which is yet to be fleshed out!

1.1 Evolution

The idea of a lens data-model as we know it today probably came into being unbeknownst to the earliest pioneers of lens design. In laying down the fundamental principles of lens analysis, the data-model arose out of a need to organise data in a meaningful manner that would facilitate numerical calculation. There are two distinct periods in the evolution of the data-model, each of which is separated from the other by the birth of the modern computer. There is little evidence available today, in the way of source code (much of which was/is commercially confidential) or development reports, to suggest how the various models were constructed, but much can be inferred from the interfaces presented to the lens designer and the state of software technology at the time.

The most major problematic area during the dawn of lens analysis concerned the copious numerical calculations required to trace finite, or real rays through even the most simple of optical systems. The data-model used to represent the system was of comparatively lesser importance, but is worth describing even briefly, since it forms the basis for subsequent work, and it also illustrates the simplicity and efficiency embodied within it. The table below illustrates the current industry accepted format for presenting lens data, although no such standard actually exists. The rows are identified with the surfaces of a system, but for completeness the object, image and stop (sometimes called the iris) surfaces are often included. The first column gives the curvature (occasionally the inverse), the second is the inter-surface separation, the third is the refractive index or name of the inter-surface medium, and the final row is the semi-aperture or occasionally the full diameter. There are other variations of data-table designed to encompass the specific data entry requirements of any particular program, although the differences are quite minor.

During that time prior to computers it is very difficult to find any evidence of a computational data-model being supported, as obviously none was required. On the other hand, there is indirect evidence to suggest that analytical data-models were being used, and it is probably here that later models have their origin. The mathematical notation prevalent in optics uses simple variable names and associated subscripts. For example, the

Surface	Curvature	Sepr	Glass	Aperture/2
1	0.032800	4.00	SK3	11.50
2	-0.007577	3.47	Air	11.50
3	0.024000	1.80	LF1	10.50
4	0.035643	3.70	Air	10.00
5	0.000000	1.30	Air	9.86
6	-0.013500	1.80	KF8	10.50
7	0.032500	6.20	SK8	11.00
8	-0.032166	92.07	Air	48.00

Table 1.1: Example Lens Data-Table (Kingslake Tessar)

curvature of the n^{th} surface is given by c_n and the refractive index of the glass following the j^{th} surface is n_j . This notation corresponds very closely to that which one would intuitively expect from the spreadsheet data-table in Table 1.1.

Whilst the first electronic computers were programmed using machine language, that is simple instructions which were directed at a very basic level of the computer's central processor, the later development of FORTRAN[3] enabled instructions to be much larger in scope. Since FORTRAN was a language designed by scientists for scientists, it was a natural choice for scientific work from the 1950s and onwards. Even in its earliest form it was able to provide the scientific programmer with subscripted variables that would allow arrays to be constructed. FORTRAN was therefore used extensively in the field of optical analysis and design, where not only could the subscripted variables be put to good use, but other features, such as looping and especially subprogram support, enabled large and complex tasks to be broken down into more manageable modules. Thus we see a natural transition from an analytical model to a computational model, based upon a certain degree of commonality between the two. From the 1950s to the present day, both the independent programmers and larger commercial concerns who are engaged in optics programming are still largely using this model.

1.2 Handling Complexity

The emphasis that has so far been placed on the visual presentation of data and its correspondence to the internal model has, for the past 50 years at least, been of little consequence to most lens designers. After all, when the activity of lens design once involved numerous individuals employing slide-rules or electromechanical calculators in order to trace just a few skew rays, then any modern lens design program might be greeted with much joy and celebration, particularly in the knowledge that productivity could be increased a thousand-fold. But, in the intervening years optical systems have steadily grown in complexity, and the ability to create and manipulate these systems has been compromised by the shortcomings inherent in the human - computer interface.

Nowhere does this become more apparent than in the design of the high resolution spectrograph. The reason why this is so stems from the underlying lack of axial symmetry that these systems exhibit. Consider the optical train comprising three cross-dispersing prisms that are to be found in the UCLES[1] spectrograph. There are six surfaces, each tilted with respect to any other, and there is a complete absence of any optical axis; the latter point results in five of these surfaces also being assigned decentrations. The lens designer is faced with an enormous task just to set up this configuration. If the requirement dictated a double-pass through this sub-system, then the difficulty faced by the lens designer increases enormously.

1.3 A New Viewpoint

At the heart of these problems lies the surface-based model, and particularly a sequential surface-based model. A sequential model requires that ray-tracing progresses from one surface to another, in the same way that one surface follows another surface in the lens data-table. Sequential ray-tracing may only proceed from one surface to another surface that occupies the next row lower down the data-table. Any other sequence is not permitted, and so surface groups that are in double-pass have to be recreated further down the table.

The component-based model is an attempt to simulate a lens system by emphasizing the various functional surface-groups, such as those that comprise lens singlets, doublets, prisms, etc. In doing so, the model becomes more meaningful to the designer, particularly

the novice or apprentice optical designer who cannot make the transition into the 3D world of the imagination, for it is the ability to view an optical system through the mind's eye that ultimately determines the degree of success one has in modelling it. Putting it another way, the surface-based model is akin to viewing this paragraph as a sequence of characters, while the component-based model puts emphasis on the character groups that we know as words. The former interpretation offers the computer word processor the greatest advantage when manipulating text, but it is the latter interpretation that is of the greatest benefit to the human reader. Taking this analogy one step further, it appears that current lens models have been designed to conform to machine interfaces and software practices that have been prevalent from the 1950s, and onwards; no attempt has been made to alter this model to make it conform to the most modern software practices of today, nor has the human-computer interface been satisfactorily improved upon. It is not a coincidence that at the time of writing, some modern computer languages have been radically updated to include theories of object oriented programming (OOP). It is this paradigm shift that makes the world of objects accessible, and we shall see in later chapters how it can be put to use in creating a practical and intuitive model of an optical system.

The following chapters and sections will attempt to derive in greater detail, as explained above, how the transition from a surface-centred to a component-centred model is achieved. In doing so it will be necessary to review in some detail the salient points that are embodied in object oriented programming. To flesh out these ideas, the author has developed a computer program that employs several important elements that prove crucial to supporting the proposed model. Emphasis will be placed upon explaining how these software elements correspond to meaningful entities which the lens designer will immediately recognise in the context of an optical laboratory. In order to illustrate how these ideas are put into practice, a simple spectrograph will be used as the basis of a model and raytracing results will be obtained and analysed. Finally, having torn down the 'false gods' of old in favour of the 'new gods', we shall speculate upon how the domain upon which these new gods walk, may be extended.

Chapter 2

Lens Modelling

Probably the most significant feature that separates human beings from the rest of the animal kingdom is the capacity and functionality of the brain. One of the higher level functions of a brain is the construction of a world model in which the organism, be it human or not, may operate in a manner that is both efficient and productive, where the latter implies that the organism will survive the trials of life and so reproduce. In so doing the organism will ensure the survival of its genes and that of its species. In man, the world model is more advanced and complex than that of any other creature. It has reached such a level of sophistication that it has allowed mankind to transfer consciousness into the surrounding environment. The manner in which we see this manifested is through the diverse ways in which order has been forced onto a world where none was evident before. Society and even civilisation itself has arisen from man's need to create order and to witness reflections of his own self. All these aspects of order may also be considered as world models that have stood the test of time, since it is the wide and popular acceptance of a model that ensures its long term survival. The success of any model ultimately depends upon how well it serves its purpose. Models exist to bind together individuals in a common purpose, to enable meaningful and sophisticated communication, or to provide a framework for abstract exploration.

A model, in the current context, is an attempt at a description of a real-world phenomenon or system. It is usually based upon a set of axioms or principles that are accepted to apply in such a circumstance, and is generally considered to represent one instance of a model amongst many others. A model may be used to prove or disprove

theories that relate to how such a system or phenomenon operates. Whether it succeeds in this task or not, it does not necessarily invalidate the original premises of that model. Either way, the model is shown to be robust or it is modified. In the long term, most models can be shown to undergo some degree of evolution. This is only natural and expected. Alternatively, a model may be used as a basis for calculation and extrapolation. Such models are prevalent throughout our society. To name just a few, we have economic models, population models, ecological models and galaxy formation models. In some of these cases the mechanisms involved are well understood, but in other cases there may be factors that are either ill-defined or unknown.

A computational model is a description of a system that is embedded in software, and as such is limited by the capabilities of the underlying operating system and the computer language employed. In addition, it may also be developed in such a way as to allow the model to be extended into unforeseen territories. The range and extensibility is ultimately determined by the starting point, which is the crux of this thesis. The following sections throw light on the two principal models that relate to an optical system, highlighting their capabilities, weak points and strong points.

2.1 Requirements

Any modern computer application has some basis in a model if it is to operate in a meaningful manner. At the simplest level, the very architecture of the operating system behaves as a surrogate model. Present-day operating systems such as those based on Microsoft Windows or IBM OS/2 endeavour to create a graphical environment that may be intuitively understood in terms of an office or personal desk-top. This environment was adopted in preference to any other because the original developers recognised that people who are involved in creative work are also office-bound or desk-bound individuals. The domain over which these people usually operate extends to their office furniture, and the function of the operating system is to create an environment similar to that which the computer user is already immersed in. Thus we can see several parallels between entities that exist in the office and those that exist on the computer screen, as Table 2.1 shows.

When Windows was originally developed during the late 1980s it was widely regarded as a graphical user interface (GUI) with object oriented pretensions. That is, the

Office		Computer
Desk	→	Computer screen
Filing cabinet	→	Disk drive
File	→	Disk file
Telephone	→	Modem/Terminal
Letter pad	→	Word Processor

Table 2.1: Office Parallels

organisation of the basic functional elements of the operating system conformed to an object model as described above. While Windows offered the user new and simpler ways to interact with software, it proved to be a difficult environment to program for. The reason why this should be so stems from the structure of the supporting code that Windows provided in the form of library routines that were made accessible to developers. Requiring many tens of man-years to develop, the Windows operating system was coded entirely in a language called ‘C’, which at the time had absolutely no facilities for object oriented programming. The result is that the simplest of tasks required many lines of code. Even today, programmers still refer to having had to write several hundred lines of ‘C’ code in order to open a window and write ‘Hello’ inside it. Amongst other things, what Windows lacked was the ability to encapsulate data and function, which is the first prerequisite for an object oriented language.

Rather than rewrite Windows in an OOP style, which would have been a terrific undertaking, Microsoft and Borland (two of the leading proponents of the object oriented approach) subsequently employed true OOP languages like C++ and Borland Pascal to encapsulate the original ‘C’ code and reformulate Windows into a new framework. Microsoft called this framework MFC (Microsoft Foundation Classes) and Borland called their framework OWL (Object Windows Library). Both models allowed developers to interact with the Windows operating system in a more intuitive and structured manner, and more importantly in an efficient manner. Thus the traditional ‘Hello’ program could now be written in around a dozen lines of code.

The above example concerning Microsoft Windows illustrates the importance of having a well constructed graphical user interface matched by an appropriately constructed

internal model. This leads us to a few key points one may make regarding the nature of a computer model:

1. the visual aspect of the model must reflect a familiar environment;
2. the visual interface must be in context;
3. the user must be allowed to interact meaningfully with the environment of the application;
4. the underlying object model must closely conform to the visual description;
5. the object model must reflect the behaviour of the real-world model, at the graphical level and preferably the data level also;
6. the object model must be capable of extension.

The first three points concern the nature of the user interface and its applicability to the task at hand, whilst the two points that follow are concerned with the accuracy and integrity of the model. The final point reminds us that a model may be a transitional one, and that at some time later it may require to be modified in order to conform with some new understanding or behaviour that is to be imparted to the model.

2.2 The Surface-Based Model

Largely due to its simplicity and efficiency, the surface-based model (or SBM) has dominated the world of optical analysis and design programs. Another reason why it has prevailed over the years is due to the limitations imposed by the various computer languages that have been employed to develop these applications. Languages such as FORTRAN, Basic, Pascal and 'C' provide only basic functionality for data abstraction, while combined data **and** function encapsulation is completely absent. That is not to say that it is impossible to achieve this goal using any of these procedural languages, but that attempting to do so puts a greater burden on the programmer, who then has to take care of the various administrative tasks, such as creating a consistent naming scheme and devising the functional interfaces.

The implementation of a SBM can never be unique, since it is intimately dependent upon the machinations of the developer, which are themselves determined by culture, background and ability. Never-the-less, in order to demonstrate how the SBM might be constructed, the author has chosen a system that, hopefully, will be generally accepted by most readers. Thus, consider a record structure called **TSurface** that embodies within it the essential characteristics of a spherical surface, and which can be described by the following type declaration:

```

TSurface = record
    ap   : double; (* aperture           *)
    cv   : double; (* curvature          *)
    ndx  : double; (* prior refractive index *)
    sep  : double; (* post surface separation *)
end;

```

In addition, let us declare new types that describe a ray of light and a lens system:

```

TRay = record
    x,y,z : double; (* position coordinates *)
    l,m,n : double; (* direction cosines   *)
    wvl   : double; (* wavelength          *)
end;

```

```

TLensSystem = array[1..MaxSurfaces] of TSurface;

```

...where the lens system is seen to be comprised of an array of surfaces.

In support of these basic entities it is necessary to have various functions and procedures that will undertake to perform the various operations and transformations that characterise an active optical system. As an example, consider the case of ray-transfer, which is the process whereby the intersection of a ray of light with a surface is determined.

```

procedure RayTransfer(const surf:TSurface; var ray:TRay);

```

```

var
  xt,yt,F,G,cosI,delta:extended;
begin
  (* transfer to the tangent plane *)
  xt := ray.x + (surf.sep - ray.z) * (ray.l/ray.n);
  yt := ray.y + (surf.sep - ray.z) * (ray.m/ray.n);
  (* transfer to the surface *)
  F := surf.cv * (sqr(xt) + sqr(yt));
  G := ray.n - surf.cv * (ray.l * xt + ray.m * yt);
  cosI := sqrt(sqr(G) - surf.cv * F);
  delta := F / (G + cosI);
  (* set the ray intersection coordinates *)
  ray.x := xt + ray.l * delta;
  ray.y := yt + ray.m * delta;
  ray.z :=      ray.n * delta;
end;

```

In its simplest form, we can also consider the process of ray propagation through an optical system to be adequately described by the following code fragment:

```

for k := 1 to MaxRays do begin
  for j := 1 to MaxSurfaces-1 do begin
    RayTransfer( surf[j], ray[k] );
    RayRefract(  surf[j], ray[k] );
  end;
  (* transfer to the image plane *)
  RayTransfer( surf[MaxSurfaces], ray[k] );
end;

```

...where RayRefract() is a procedure that accepts both surface and ray arguments, just as RayTransfer, and transforms the ray argument to conform to the new refracted ray.

A simple inspection of the above code shows that all the activity connected with ray propagation occurs within the two procedures RayTransfer and RayRefract, while surface and ray references are reduced to arguments of the aforementioned procedures. It is clear

Surface	Curvature	Sepn	Glass	Diameter
1	0.086318	0.70	Ge	10.50
2	0.070305	10.33	Air	10.20
3	0.146526	0.50	Ge	6.30
4	0.115933	3.32	Air	6.20
5	0.000000	0.00	Air	1.00

Table 2.2: Petzval Lens Data

that though the process of refraction may be associated with a ray interacting with a refractive index boundary represented by a surface, the above code does not make any such distinction. Similarly, the process of ray transfer is a function of the space between two surfaces, but here again the subtlety is lost. In other words, the surface-based model cannot represent in code what we intuitively perceive to be happening.

It is not just at the level of code that the above disparity is apparent, but also at the level of the user interface. Consider the usual form in which lens data is presented, either for inspection or for editing. Table 2.2 represents a simple Petzval lens of the type commonly used in infra-red imaging, comprising two separated single lenses of Germanium. For the purpose of demonstration, let us attempt to reverse the order of the two singlet Germanium lenses by swapping surfaces 1 and 3, and surfaces 2 and 4, resulting in Table 2.3. Inspection of this new table reveals that while the order of the lenses has indeed been

Surface	Curvature	Sepn	Glass	Diameter
1	0.146526	0.50	Ge	6.30
2	0.115933	3.32	Air	6.20
3	0.086318	0.70	Ge	10.50
4	0.070305	10.33	Air	10.20
5	0.000000	0.00	Air	1.00

Table 2.3: Petzval Lens Data

reversed, the spacing between them has unintentionally changed and leading to an unforeseen configuration change. The reason for this arises from the unfortunate association in any row of both surface curvature and separation, and so long as this lens description

format is adhered to then such surface exchanges will lead to spurious results. We say that the table representation of a lens system is not commutative with row exchanges, which implies that the surface-based model is also non-commutative.

We may draw three conclusions based upon the above analysis of the surface-based model:

1. In terms of coding effort, the SBM is an efficient model;
2. The visual interface (spreadsheet) is a familiar one;
3. The SBM does not conform to our intuitive understanding of a lens system.

Both Items 1 and 2 are, without a doubt, strong reasons for adopting this approach. That is not to say that we should be complacent and restrict our attempts to improve upon this model. Indeed, the spreadsheet has, over the years, been endowed with ever-more useful features such as embedded drop-down list boxes (useful for the selection of glass types) and graphical cells that can show an image of a surface profile. These features all add to the functionality of the spreadsheet which in turn aid the user in inputting data and comprehending the system that is entered. Item 3, on the other hand, tells us that this interface is only an abstract and not ideal representation of a lens system. In addition, neither is it possible to visualise the system by casual inspection, nor can it be manipulated in a similar manner as a real lens system might.

2.3 The Component-Based Model

The essential idea behind all object modelling is that of abstraction, that is, a model that provides the required functionality and data encapsulation while hiding the unnecessary details that can be overlooked by the user. Ideally, a model will make visible those aspects of a system which it is designed to express. In this regard the component-based model (or CBM) can be viewed as expressing a higher level of complexity when compared to the SBM, and this becomes evident to the user by the provision of a truly graphical interface to the layout of the lens system and a greater facility for system synthesis.

The basic component of the CBM is the *lens*. While a lens may be understood, in its simplest sense, to be a glass element bounded by two regular surfaces and providing

some form of imaging quality, in the model to be described a lens represents an abstract element that may be a mirror, a diffraction grating or even a light source. In fact, the abstract lens may represent any physical entity that one might associate with the major elements of any imaging system.

In order that this lens be manifested to the user, it becomes necessary to imbue it with some form of graphical quality that distinguishes it from any other lens. In Windows programs it is quite common for minimised executables or programs to be represented by an icon having some manner of graphic depicted within its bounds. In doing so, the underlying program is effectively announcing its presence to the user, and distinguishes itself from any other program by the uniqueness or characterisation afforded by its graphical icon. Similarly, a lens may have a picture-icon that suggests that the lens is a mirror, a singlet lens or a source, as the following Figure 2.1 shows.

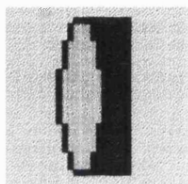


Figure 2.1: The Doublet-Lens Icon

There is obviously more to the lens icon than its simple arrangement of coloured pixels. In much the same way as the visible portion of an iceberg is a portent of something much larger (and occasionally sinister) below the waves, the lens icon is simply the visible manifestation of an object that encapsulates far more data and information than it is possible to represent graphically. But the importance of the icon is that it isolates the lens in space (*viz.* - the computer screen) and promises other behaviour (which will be discussed later in this section) more in tune with a physical entity, as opposed to the rather complex and not-so-independent surface representation of a row in a spreadsheet.

In addition to the graphical nature that has been attributed to a lens, we can also consider how a lens system, comprising several individual lens components, may be represented. Again, we borrow from the many graphical structures that the Windows environment provides, and note that a class called **TForm** (generally known as a *window*) exists which acts as a container or containment vessel for other Windows elements such as buttons and labels. The fact that **TForm** has properties that enable it to identify and reference all objects within its confines makes **TForm** an ideal candidate for this purpose.

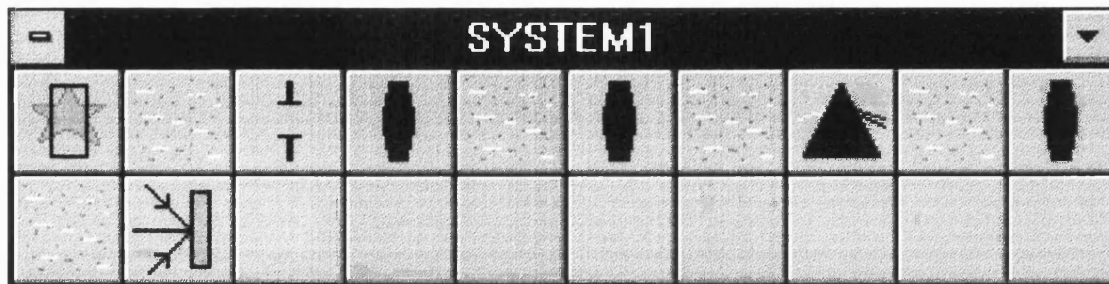


Figure 2.2: The Lens-System Container

Figure 2.2 shows such a form acting as host for several differing lens types, including a source (top-left) and a prism (the eighth from the left on the top row). The order of components within the system, or the direction in which light travels, commences from the top-left and proceeds to the right, and continues at the left-most component of the following row, and so on.

A simple description of a lens component is given below:

```
TLensIcon = object
    picture : TIcon; (* picture icon *)
    lens    : TLens; (* lens object *)
end;
```

TLensIcon is a class that encapsulates two other objects, **TIcon** and **TLens**, each of which is recognisably an essential element of a lens-component. In the language of OOP, they are also referred to as *properties* of **TLensIcon**. The **picture** property is the icon that is visible on the computer screen, while **lens** is a reference to a generic lens description. For example a singlet lens, as described below, is derived from a largely abstract class called **TLens**, and so may be assigned as a **lens** property of **TLensIcon**.

```
TSingleLens = object(TLens)
    surface1 : TSurface; (* first surface *)
    surface2 : TSurface; (* second surface *)
    sep      : TSpace;   (* inter-surface space *)
```

```

    glass      : TGlass;  (* glass type      *)
    procedure   ProcessRays(rays : TRayArray); override;
end;

```

Notice that the properties of the singlet lens are all familiar quantities, but in keeping with the object paradigm they have been converted into new class descriptions that themselves encapsulate the required data and functional properties of the abstracted entities. One other noteworthy feature in the above class definition is the presence of a procedure called **ProcessRays**. This procedure accepts as a parameter an array of rays, which are then refracted and transferred from surface to surface until the rays are in a suitable form to be passed on to the next Lens component for processing. **ProcessRays** is identified as a procedure associated with all classes derived from **TLens**, but since each newly derived lens component will interpret **ProcessRays** in a different manner, then this procedure will need to be re-coded to reflect this, hence the need to **override** the procedure for each descendent of **TLens**. Naturally, for simple refractive elements, the underlying code to **ProcessRays** is similar to that we have seen earlier in the previous section, i. e. **RayTransfer** and **RayRefract**.

In keeping with the previous section, we shall develop the necessary structures and code that describe a lens system and the manner in which rays are propagated from the source to the image plane. Firstly, a lens system may be described as an array of lenses, i. e.

```

TLensSystem = array[1..MaxLenses] of TLensIcon;

```

In practice, the **TLensSystem** would be described as a class having the array of **TLensIcon** specified as a property of the class, again in accordance with the aim of describing all entities as class objects:

```

TLensSystem = object
    MaxLenses : integer;
    property Lens[n:integer] of TLens read GetLens write SetLens;
end;

```

The property description specifies that the identifier called **Lens** should be interpreted as an array, where the two procedures **GetLens** and **SetLens** define how reference and

assignment of `Lens[]` is handled. Thus a reference to the n^{th} lens in a system called `ALensSystem` is given by `ALensSystem.Lens[n]`. With this information we can now write the code that implements raytracing in an object-based lens-system.

```
with ALensSystem do begin
  for j := 1 to MaxLenses do
    Lens[j].ProcessRays(rays);
  end;
```

The important difference to note between this piece of code and that associated with the SBM is that the subject within the loop is of type `TLens` while in the SBM code it is the ray-transfer and refraction procedures. The OOP rendition of raytracing results in both subject and action being tightly bound to one another, as allowed by the class definition, but the purely procedural form shows no such tight binding. The reason why binding in these circumstances is so beneficial stems from high degree of security that exists when employing methods that originate from and have been developed within the same class structure.

There are other ways in which the component-based model is superior to the surface-based model. Firstly, due to the similarity of structures that have been developed in code and that exist in the real world, it is possible for individual lenses to undergo physical transformation in a manner that would be difficult to consider in the lower order SBM. As an example, take the case of lens reversal. In its implementation we must develop reversal procedures for the surfaces first, and then introduce a reversal method for the lens itself, as the following code fragments indicate.

```
TSurface = object
  x,y,z : double (* position *)
  l,m,n : double (* orientation *)
  glass1 : TGlass (* prior glass *)
  glass2 : TGlass (* post glass *)
  procedure ProcessRays(rays : TRayArray); virtual; abstract;
  procedure Reverse; virtual; abstract;
```

`TSurface` is the ancestor of all surface types, where the data-sets (x, y, z) and (l, m, n)

record the position and orientation of the surface, and **glass1** and **glass2** are the glass types on either side of the surface boundary. Since we haven't given **TSurface** any specific geometrical form, then the methods **ProcessRays** and **Reverse** cannot be properly defined and so are specified initially to be **virtual** and *abstract* methods. That is, they cannot be employed as is, but must be overridden and defined by a descendent class first. A descendent class might be **TSphere**, **TAsphere** or possibly **TTorroid**. Each descendent will have its own method of implementing these abstract methods of **TSurface**.

```

TSingleLens = object(TLens)
    surface1 : TSurface; (* first surface      *)
    surface2 : TSurface; (* second surface     *)
    sep       : TSpace;  (* inter-surface space *)
    glass     : TGlass;  (* glass type       *)
    procedure ProcessRays(rays : TRayArray); override;
    procedure Reverse; override;
end;

```

This slightly modified version of the earlier **TSingleLens** now includes a **Reverse** method which is implemented below:

```

    procedure TSingleLens.Reverse;
    begin
        Surface1.Reverse;
        Surface2.Reverse;
    end;

```

Thus, assuming that the **Reverse** method for any particular surface type has been defined, then the reversal of any single-lens component is given by: **ASingleLens.Reverse**. The equivalent method for a set of surfaces within a SBM environment will not be as simple to implement. Firstly, there is no clear way of knowing or distinguishing which surfaces belong to which lens, since the concept of a lens is not defined in the SBM. This argument is simplistic since the originator of a lens system will have had some clear goal in mind during the construction process, but it does not contradict the original premise that the single-lens is an artificial construct within the domain of a SBM. The implementation of

Reverse in a SBM will require a process of inspection and the application of logical rules in order to isolate various lens combinations prior to implementing any lens transformation. This procedure will be laborious and contrary to the methods that a lower order model should support or provide.

In contrast, the CBM does indeed support commutative properties with respect to lens exchanges and movements. If we consider the graphical representation of a lens system, as in Figure 2.2, then this does not appear at all surprising since the lenses are clearly to be seen as independent entities, and lacking any degree of cross-coupling as in the case of the row-data in the SBM. How this, in terms of code and modelling, is possible will be the subject of Chapter 5.

Chapter 3

Object Oriented Programming

The last forty or so years has seen considerable development in the field of computers and computer architectures. In order to harness the ever increasing power that these hardware platforms provide requires programming languages to be regularly updated and improved. Since the late 1970s when the 8-bit CP/M operating system prevailed, the advent of the IBM-PC has resulted in operating systems progressing from the 16-bit code of Windows 3.X to the recent releases of the 32-bit operating systems, Windows NT and Windows 95. At each stage of the operating system development cycle, new compilers and language enhancements have been required to match the new capabilities offered. The principal driving force behind this frenzy of activity is the goal of a fully object oriented system.

At a simplistic or naïve level one may state that “God is the greatest programmer of all time”, where the proof is around us for all to see. How can we, as imperfect mortals, hope to emulate such creativity in our programming efforts? Not, it seems, by improving our ability to code using conventional languages such as FORTRAN and C, but by developing a new methodology that harnesses the power and simplicity of creation. The object-oriented paradigm recognises that systems exist as distinct entities and that they may also be evolutionary. For support of such a premise we look to biological systems (God’s *objects*) for inspiration. In reality, those biological systems that have arisen from some obscure evolutionary process have not done so according to some well defined plan, but have in fact arrived at their stage of existence having meandered, seemingly un-directed, through the multiple-spaces of infinite possibilities. This is most definitely not the approach that transitory creatures such as ourselves can afford to take. Indeed, in

essence, what we have chosen to do is take what is most simple and efficient from the biological evolutionary process and applied it to a new set of computer languages. These we have chosen to call 'object-oriented' languages.

Object oriented programming, or OOP, represents the latest and most potentially rewarding paradigm shift since assembler language. Since the dawn of the computer, man has sought to gain control over the massive computing power that lies dormant within the few square centimetres of silicon that is the playing field of the central processor. The central processing unit, or CPU, is inactive unless instructed otherwise. A computer language is the means by which instructions are given to the CPU - ignoring for the moment the necessary intervention of the compiler. There have so far been four stages in the evolutionary process of the computer language, where the emphasis of each has been placed upon a basic unit of data and/or code. They are:

machine-code – this uses the native language of the CPU. Each instruction normally comprises two parts: op-code and operand. The op-code indicates which operation is to be performed, while the operand supplies the necessary data;

assembly language – by replacing the machine language instruction with a mnemonic code and symbolic address, the task of writing a program is made easier. Prior to execution the assembler code requires translating to machine code;

procedural language – FORTRAN, Pascal, Basic and C are typical procedural (*high-level*) languages. Their overall structure is considerably closer to our everyday language and are much simpler to use. Structured programming methods are applicable to this class of computer language as they all support procedural and modular code blocks.

object oriented language – there are two forms of object oriented languages; the pure OOP languages as typified by Smalltalk, and the hybrid types such as Object Pascal, Ada and C++. Both provide new constructs that support the necessary features of a OOP language (see later in this chapter).

In terms of object-orientation, it should be pointed out that not all languages or tools that describe themselves as object-oriented are in fact so. An OOP tool must pass four basic criteria [8] in order to be truly object-oriented:

- **Encapsulation** — Data and program code must be locatable within single entities. That is, an object must be able to store both data elements (as a record structure does) and procedure elements (called *methods*). Procedural elements within an object must have automatic access to data elements within the object.
- **Inheritance** — New object types must be able to be synthesised from existing ones by inheriting their attributes and method procedures.
- **Polymorphism** — Object methods must be callable without respect for the actual object type in which the method resides. For example, the **Show** method performs radically different tasks when drawing a button control as opposed to drawing a grid control, though the call is identical. Also, provided they both descend from a common ancestor, calling the ancestor's **Show** method using an instance of either the grid or button control should properly display the correct control.
- **Primary methodology** — The object-orientation of a tool must be the primary method of constructing program code, not an afterthought or add-on.

The following sections will shed more light on the meaning and importance of the first three of the above items. Borland's implementation of Object Pascal will be used to illustrate these features, though the code should also be familiar to C++ and Ada programmers alike. *Note: It is not possible in this chapter to convey to the reader all of the finer details regarding object oriented programming, but it is hoped that the general points will come over and that the reader will seek further information in the bibliography.*

3.1 Encapsulation

Languages such as Basic and Pascal are typified by their clear discrimination between data and procedural code-blocks. Virtually any program may be considered to possess a data-body that is subsequently processed by numerous code-blocks and resulting in a transformation of the original data. The two entities remain separate at all times.

The paradigm shift associated with OOP stems from the argument that rather than data and procedures being separate entities, they should indeed be considered to be separate aspects of a single new entity. Consider the case of a vector of type TVector, where the coordinates of the end-point are represented by the two real numbers (x, y) . In

the simplest case these data are bound to the type **TVector** by the record, or structure, definition given by:

```
TVector = record
    x , y : real;
end;
```

Let us declare a variable **vec** of type **TVector**, then a reference to the vector coordinates occurs using the *dot* notation, i. e. **vec.x** and **vec.y**. If now a procedure called **length** is introduced which accepts a **TVector** type as a parameter and returns the length of the vector, then it might be defined as follows:

```
function length( vec : TVector ) : real;
begin
    with vec do
        result := sqrt( x * x + y * y );
    end;
```

...where a reference to the length of a vector is **length(vec)**. Now, one could argue that both the vector type and the function are intimately connected to one another since the function may only operate on vector parameters. This being so, one could also consider other functions and procedures that act only upon entities of type **TVector**. Going one step further, could not a type be considered that bound both the vector and its supporting procedures into a new entity? Let us call this new entity an *object* and we shall refer to its definition as a *class*. **TVector** is now a class, not a record, and is defined below:

```
TVector = class
    x , y : real;
    ndims : integer;
    function length:real;
end;
```

...where **ndims** contains an integer value of the number of dimensions occupied by the vector. The **length** function is referred to as a *method* of **TVector** and is implemented as follows:

```

function TVector.length : real;
begin
    result := sqrt( x * x + y * y );
end;

```

This class is similar in some respects to the earlier definition of the **TVector** record, and is accessed in a similar manner, i. e. the length of the vector **vec**, is given by **vec.length**. Note how the **length** function no-longer requires a vector parameter to be passed as an argument. The reason for this is that since **length** is now embedded within the new class **TVector** it automatically has access to data and other procedures declared within any instance of this class.

The process of melding both data and data-related procedures into a new structure is called *encapsulation*, and is one of the core concepts of object oriented programming. It arises through the recognition that programs almost always possess certain data-sets that appear to be bound to specific procedures, in the same way as we have seen with **TVector**. A recognised practice during software development for identifying classes is to inspect the system model with a view to isolating nouns, such as vector, lens or ray, and then to assign methods that correspond to natural operations or transformations that these classes would be employed in. For illustrative purposes the class **TVector** is shown below in a more complete form.

```

TVector = class
    x , y : real;
    ndims : integer;
    constructor create;
    procedure init(const a : array of real );
    function length:real;
    function cross( vec : TVector ) : TVector;
    function dot( vec : TVector ) : real;
end;

```

Thus, if u, v and w are vectors, then the scalar triple product $u.(v \times w)$ is given by **u.dot(v.cross(w))**. Note the appearance of a new procedure with the title of **constructor**. The creation of a class instance requires a single statement of intent to that

effect ¹ which is the purpose of the constructor `create`. Given a variable `vec` declared to be of type `TVector` then the creation of an instance of that type is through the statement `vec := TVector.create`, and initialised by `vec.init([1,2])`. The constructor and initialising procedure is given below:

```

constructor TVector.create;
begin
    inherited create;
    { the values of x and y are by default equal to zero }
    ndims := 2;
end;

procedure TVector.init(const a : array of real );
begin
    x := a[0];
    y := a[1];
end;

```

It is clear from the above code how the array argument `a[n]` copies over to the internal variables `x` and `y`. The presence of the line `inherited create` in the constructor is necessary to set aside the memory space for the declared variables and to maintain links to the declared functions for this class, as well as performing the necessary book-keeping chores that support all declared classes.

3.2 Inheritance

The previous section has demonstrated the useful facility for creating new types called classes that enable the form and function of objects to be encapsulated into a single code-block. If this reflected all there was to the much touted OOP paradigm then indeed there would be little benefit. To illustrate this point, consider developing a new class that is specific to three-dimensional vectors, and call this class `TVector3`. It might be defined as follows:

¹C++ offers a limited form of automatic construction of objects

```

TVector3 = class
  x , y , z : real;
  constructor create;
  procedure init(const a : array of real );
  function length:real;
  function cross( vec : TVector3 ) : TVector3;
  function dot( vec : TVector3 ) : real;
end;

```

To complete this specification the above functions need to be rewritten to account for the new type to which they belong, i. e.

```

function TVector3.length : real;
begin
  result := sqrt( x * x + y * y + z * z );
end;

```

The OOP vision is an organic one, where a class may be the progenitor of other, more complex, classes. In this regard, the above example fails to satisfy since there is no apparent inheritance mechanism in evidence during the definition of **TVector3**. Since the three-dimensional vector has much in common with the two-dimensional vector, we would expect some degree of repetition in the code implementation. What is required is a mechanism that will allow **TVector3** to be a descendant of **TVector**. In doing so, the various new methods might be updated in a similar manner, through *inheritance*. A clearer understanding will be obtained by re-writing the above example to include the new features for inheritance. Firstly we shall redefine the **TVector** class to make it suitable as an *ancestor* class.

```

TVector = class
  x , y : real;
  ndims : integer;
  constructor create; virtual;
  procedure init(const a : array of real );
  function length:real; virtual;

```

```

    function cross( vec : TObject ) : TVector; virtual;
    function dot( vec : TObject ) : real; virtual;
end;

```

There is one important addition to this new definition which is worth pointing out, and that is the presence of the directive **virtual** following each of the methods. This declaration indicates that the function may be overridden or redefined in a descendant class. Continuing on, a descendant of **TVector** is now defined that caters for three-dimensional vectors:

```

TVector3 = class( TVector )
    z : real;
    constructor create; virtual;
    procedure init(const a : array of real );
    function length:real; override;
    function cross( vec : TObject ) : TVector; override;
    function dot( vec : TObject ) : real; override;
end;

```

It is with this new definition of a descendant class that inheritance features are evident. Firstly, the appearance of (**TVector**) following **class** states that **TVector3** is a descendant of **TVector**. Secondly, the variables *x* and *y* are not present as they have been inherited from **TVector**, and so only *z* is required to be declared; and thirdly, the methods that follow are to be redefined due to the inclusion of the directive **override**. Fortunately, inheritance can make this process simpler as the following two function overrides make clear:

```

    constructor TVector3.create;
begin
    inherited create;
    ndims := 3;
end;

```

```

procedure TVector3.init(const a : array of real );
begin
    inherited init([ a[0],a[1] ]);
    z := a[2];
end;

```

and...

```

function TVector3.length : real;
begin
    result := sqrt( sqr( inherited length) + z * z );
end;

```

The procedures above illustrate the use of the `inherited` keyword, showing that the ancestor `TVector` is still visible to `TVector3`, although the contrived use of `inherited` in the `length` function is inefficient and so not appropriate nor absolutely required. One other interesting feature is the presence of `TObject` as the argument type in the functions `cross` and `dot` where one would expect `TVector3` in a class of the same type. Obviously a cross product cannot be envisaged between a vector and an object of unknown type, but this incongruity arises from the need to maintain an identical interface to both ancestor and descendant, when functions are to be over-ridden. This dilemma will be resolved in the next section.

3.3 Polymorphism

The word *polymorphism* originates from the Greek word meaning *many shapes*. It is through the virtual method facility that polymorphic objects are realised — literally object instances that can assume different forms when the program runs. A polymorphic object instance might take on the identity of itself or any of its descendants. The following code segment will help to illustrate the multi-personality traits that polymorphism allows:

```

1.  var
2.      vec1          : TVector;
3.      vec2          : TVector;
4.      vec3          : TVector3;
5.      len1, len2, len3 : real;
6.      dot           : real;
7.  begin
8.      vec1 := TVector.create;
9.      vec1.init( [1,2] );
10.     vec2 := TVector3.create;
11.     vec2.init( [1,2,3] );
12.     vec3 := TVector3.create;
13.     vec3.init( [1,2,3] );
14.     len1 := vec1.length;
15.     len2 := vec2.length;          (* Wrong answer *)
16.     len2 := TVector3(vec2).length; (* Correct answer *)
17.     len3 := vec3.length;          (* same value as len2 *)
18.     dot  := vec1.dot( vec2 );

```

Lines 2—3 declare `vec1` and `vec2` to be of type `TVector` while line 3 declares `vec3` to be of type `TVector3`. These three variables are then created and initialised in lines 8—13. The point to notice here is that `vec2` is created using the `TVector3` constructor that is assigned to 3-dimensional vectors, while the vector is declared as a 2-dimensional vector. That this was possible indicates that the constructor was successful in creating the necessary memory resources and assigning the appropriate values to the vector variables.

A clearer picture emerges when the `length` method is accessed for all three vectors in lines 14—17. The vector length `len1` produces a correct result ($\sqrt{5}$), but the value of `len2` is identical when it should be $\sqrt{14}$. The reason why this is so stems from the fact that `vec2` is declared as a 2-dimensional vector and may only legally access the internal variables `x` and `y`, and the 2-dimensional `length`-method. On its own, `vec2` cannot gain access to the 3-dimensional `length`-method. Line 16 shows how this may be achieved using the technique known as *typecasting*. This technique, as demonstrated in line 16, forces the compiler to view `vec2` as a variable of type `TVector3`, which then makes it possible for

vec2 to employ the 3-dimensional **length**-method. Finally, as one might expect, line 17 also results in a correct value being passed to **len3**.

The previous section concluded by noting that in order that equivalent methods should be available to all descendent classes, then the form of the method definition should remain unaltered. Thus, in the case of the **dot**-product method, the argument type is **TObject** in order to allow both 2-dimensional **TVector** and 3-dimensional **TVector3** parameters to be passed. *Note: Object Pascal declares a type **TObject** that is the primitive ancestor of all classes; in other words, **TObject** is to **TVector** as Adam is to Mankind.* In achieving this compatibility across descendant classes, one must now ask how it is possible for a method to determine what kind of object has been passed as an argument, since we cannot speak meaningfully of a dot-product between a vector and a generalised object of unknown quantity.

RTTI, or run-time type information, is the means by which a class variable may be interrogated in order to ascertain its type. This is implemented in Object Pascal by the new operators, *is* and *as*. As an illustration of this powerful enhancement to OOP languages (also recently added to C++), the **length** method for the **TVector3** class is given below:

```
function TVector3.dot(vec : TObject):real;
begin
  with vec as TVector do begin
    if ndims >= 3 then
      result := sqrt(x * vec.x + y * vec.y + z * vec.z)
    else result := inherited dot(vec)
  end;
end;
```

This new implementation of the 3-dimensional **dot** method initially checks if the **vec** argument is of type **TVector**, and if so, performs the 3-dimensional dot-product if the number of dimensions is 3 or greater, else it defers to the inherited method of the ancestor to implement the 2-dimensional dot-product. On the other hand, if **vec** is not of type **TVector** then an exception will automatically be raised. In a similar manner, the inherited method will check for a 2-dimensional vector as an argument (which in this case

will always be true), and then return the 2-dimensional dot-product, i. e.

```
function TVector.dot(vec : TObject):real;
begin
  with vec as TVector do begin
    if vec.ndims >= 2 then
      result := sqrt(x * vec.x + y * vec.y);
    end;
  end;
end;
```

Such a cascading sequence of events allows for any size vector to be treated accordingly and correctly, whilst the exception is raised only in the event of a non-vector parameter being passed.

It has become clear that polymorphism enables complex and evolving structures to come into existence, which, along with the supporting operators that implement RTTI, also allows for graceful method transfers along the evolutionary chain. In the case of n-dimensional vectors that have been developed as part of a structured development plan, polymorphism allows convenient and efficient handling of non-matched vectors in the case of the dot-product method. Similar benefits are to be derived from any class hierarchy that has been constructed with polymorphism in mind.

Chapter 4

Implementation Details

When commencing a project such as this, particularly in an academic environment, there are inevitably two questions that arise before any progress may be made. These are: “What is the computer platform on which the application will run”, and “What computer language(s) will it be coded in”. In some situations the answers will be obvious where no other choices are available. In other situations the answers will depend upon factors that may be outside of one’s control.

4.1 Computer Platform

What has an academic environment to do with the decision making process? Well, firstly, the nature of an academic environment is to minimise the constraints on creativity, and in this case we recognise the diversity and power afforded by the considerable computing resources that all universities make available to both staff and students alike. Almost invariably this takes the form of networked facilities that are underpinned by powerful workstations. In general the operating system is almost always some variant or flavour of Unix, which has been the mainstay of universities throughout the world since the late 1970s. It is characterised by its ability to provide multi-tasking services and by the relative stability of the open standards to which it conforms. The picture is not so clear-cut today due to the marked market impact of MSDOS/Windows during the last decade, and to a phenomenon known as the ‘Unix wars’. The nett result is that local Windows networks are becoming increasingly common in environments such as offices and laboratories.

The ever increasing thrust of the personal computer, and particularly of Windows, into all avenues of computer usage has also had a dramatic effect upon the number and types of software that have been developed over the past few years. It is almost true to say that there is no human activity to which some computer related software or data is not pertinent or available. This is especially so in the field of programming tools, such as computer languages, 'rapid application development' tools, and new 'object technology' - based controls. A cursory glance through any bookshop will show that the vast majority of computer-related titles on sale belong to the class dedicated to the PC user, and it is here that the move to operating systems such as OS/2, Mac System/7, and Windows95/NT has occurred. Today's users of computer technology require it to be a personal experience, where the operating system is not overbearing and counter-intuitive, but friendly and accessible. No-one can say that Unix is a friendly system, or one that is easily mastered, although when seen in the guise of a shell such as X-Windows then usability is markedly improved.

The move of the computer out of the cloistered halls of academia into virtually every office and living-room of the western world is reflected by the ease with which computer power is made manageable via the modern operating systems and newly developed programming tools that make the task of creating usefull programs far simpler than ever before. As a result of the ever-increasing size of the user base, the costs of both hardware and software are much more affordable than their equivalents in the Unix world. In addition, due to the much higher cash flows involved, (successful) commercial software developers are able to fund new development programmes on an almost never-ending basis. This factor, coupled with the ever present competitive edge that drives all companies, leads to products that are state-of-the-art for maybe a year or so, only to be superseded by more sophisticated products. As an example of this market driven development cycle, during the last four years the author has witnessed, and even indulged himself in, several seminal products that will be remembered as milestones of the personal computer-age, such as Windows 3.XX, Visual BASIC, Delphi, and Windows 95. So long as raw processing power continues to increase year after year, then this trend is unlikely to falter.

Though the 'C' language was originally developed for Unix systems programming, it really came to the fore when it was upgraded to encompass greater functionality and object-oriented extensions in the guise of 'C++'. It only really gained world visibility and acceptance through the commercial language implementations that were released onto the

mass of eager personal computer programmers that constituted the vast majority of programmers in general. The most sophisticated and powerful of all C++ implementations today are to be found serving the needs of Windows programmers in all walks of business, and they are produced by some of the largest and most successful organisations in the world, the largest being Microsoft Corporation, naturally. Other languages have also been successful on both MSDOS and Windows operating systems, particularly Pascal, BASIC and Prolog. As in the case of C/C++, these languages have also seen numerous new developments and enhancements that have considerably increased their productivity and scope. Thus it is clear that while the two principal computer platforms are Unix and Windows, they are differentiated from one another in the extent and sophistication of developer tools provided, with Windows easily being the target for a sizeable majority of system and product developers. So, if the most modern programming tools are to be sought after, at reasonable cost and with better than adequate support in both journal and book form, then the choice must be Windows.

Another factor to be considered in the choice of computer platform is the intended audience or user. Where an application is to be produced within an academic environment and then scrutinised by one's peers in the same environment, then whichever platform is prevalent will be a suitable one. If on the other hand one's peers are of differing locales and available computer resources then it makes more sense to aim at a platform that is common to both. Invariably this common platform will be Windows. It is a fact that the Windows operating system is ubiquitous and the paradigm that it expresses, similar to that of the Macintosh System/7, OS/2, and X-Windows, is by now a familiar one, not only to programmers and users but also to the children in our schools.

4.2 Programming Languages

Life would have been much simpler if we (human beings) had chosen not to design any more programming languages other than the original machine code language, which came about out of necessity rather than a conscious decision. However the price of such a decision would have been enormous, ruling out all the benefits that we would have derived from procedural and object-oriented languages, such as the very large and sophisticated applications we 'enjoy' today, with the likes of Microsoft Office and even the Rapid Ap-

plication Development tools themselves. Languages have evolved out of necessity, arising from a need to fulfill some specialised task. For example, Lisp was developed as a tool to facilitate list processing, Pascal was originally used as an educational tool, 'C' was targeted at systems level programming, and the business environment was served by Cobol. Many other languages also came into being during the last two decades, although few still serve the needs of today's programming community. Those that have survived do so because they have changed in some way to meet a more general requirement. As the programming languages and their operating environments have become more complex to deal with, the programmer has preferred to employ a language that is both general purpose and powerful rather than adopt a multi-'lingual' approach. This strategy is also the preferred course for the language developers, since it makes more business sense to support one or two mainstream languages than several of the specialised languages that in time may eventually fold.

At any one time during the past decade or two the computer industry may be seen to have been in a state of flux. That is, one can view the past and review all the innovations that have contributed to the then current state of technology, and one can also be aware of new developments that are still in the pipe-line and are soon to be established. Nothing stays still, not even the operating systems that we have discussed above, although it is to be hoped that their longevity will far surpass that of those software components that rely upon a stable operating system. In the context of this thesis such software components are represented by the computer language tools. During the first year of this research programme the author became extremely aware of the changing tides of software technology, and the difficulty involved in choosing the 'right' tool for the task; not only was it a matter of selecting the best tool, but one also had to discover whether the best tool was the right tool. For example, the author has spent many years using procedural languages such as HPBasic and Pascal on a casual basis. It may be that such an apprenticeship might create difficulties when given the task of determining the suitability of other computer languages for this project. In hindsight this was probably true.

Another factor that influences choice is the available timescale that the project extends over. During a short development programme it is important that the necessary resources are acquired almost immediately, since any undue delay will significantly cut into the remaining time. On the other hand, a longer programme will not suffer from the

same urgency, but a situation may arise when more suitable resources are not immediately available but are ‘reported’ to be in the pipe-line. In such a case, as the author knows from his own experience, it can be tempting to gamble with the remaining time in the hope that the rumoured tools will be on sale soon enough, so as not to seriously jeopardise what time remains. This approach is not for the faint-of-heart, and is not recommended.

The following sections provide summaries of the three principal computer languages that were considered. Other development tools were also investigated, including Borland’s ObjectVision and Computer Associates’ CA-Realizer. It was fairly soon established that ObjectVision could not provide all the facilities that one would require of a general-purpose programming language since it was designed as a specialised database tool having a scripted language at its core. In contrast, CA-Realizer is based upon a superset of BASIC, a language that is familiar to almost all programmers. Though BASIC has not been highly acclaimed as a language for purists or academics, it has never-the-less carved out its own niche in the computer world. In its favour CA-Realizer has managed to extricate itself from such views by improving the capabilities and language syntax sufficiently so as to create a far better BASIC than its predecessors. Unfortunately it was released shortly after Microsoft’s Visual BASIC, and as such it has had to suffer unfair comparison against what is now the world leader in Rapid Application Development tools. It seemed to the author that Realizer could not hope to keep up with all of the latest developments in Windows and object technology, an opinion that has not changed with time.

4.2.1 C++

The C programming language was designed and implemented by D. M. Ritchie, and it was published in the book *The C Programming Language*[6] by B. W. Kernighan and D. M. Ritchie in 1978. In its first years C was popular only in connection with the Unix operating system. Subsequently, it turned out to be a very good programming language for microcomputers as well, and at the time of writing C can be used with almost any computer type or operating system. Due to its widespread popular acceptance it soon became necessary to introduce some form of standardisation into the language, and so in 1983 the American National Standards Institute (ANSI) formed a committee (X3J11) to provide such a standard, known as ANSI C. This version of C did much to rationalise previous inconsistencies in the language, but it was only three years later, in 1986, that B.

Strousrup was spurred to write his book *The C++ Programming Language* [7]. This was almost two decades after the first object-oriented language, ‘Simula 67’, was demonstrated.

Despite the fact that ‘C’ was preceded by ‘B’ which in turn was developed from BSPL, they were all substantially *small* languages that could be mastered in a few days. In particular, C was developed as a language that would bridge the gap between the efficient and fast assembler language, and the higher level languages such as Cobol and Fortran. One of the drawbacks (C programmers would consider it a strength) of the low-level bias exhibited by C is the ability to mix types, i. e. characters and bytes may be interchangeable, resulting in a language considered not to be *type-safe*. For instance, a function might be passed two integer arguments m and n and will return an integer that corresponds to the evaluation of m^n . This represents an integer being raised to the power of another integer, but C will also allow characters to be passed to the function and in this case the result has no bearing on the data-type of the supplied parameters. The compiler will allow this whether the programmer intended it or if a genuine error was made. A type-safe language, such as Pascal or Ada, will allow only functions and operators to act upon data if the required data-types are compatible. In so doing the compiler forces the programmer to be explicit about the operations that are to be performed and the data-types upon which they operate. Later versions of ANSI C rectified, or improved matters by closing this loop-hole and so strengthening C’s type-safe features.

On the other hand, C++ was designed to provide both object-oriented extensions and stronger type-safety to the C language, and so we find such constructs as *classes*, *polymorphism* and *inheritance*. Not content with single-inheritance, version 2 of C++ implemented multiple inheritance that allowed a new class to be derived from more than one base class. Simple and complex data structures, including arrays and user-defined data structures are supported, where object classes represent the most complex structures of all, incorporating both data and procedural references. A rich set of operators is also provided that enable the processing and transformation of these same data structures. One of the features that distinguishes C++ from most of its current rivals is the ability to overload operators and functions. What this means in practice is that the same operator symbols or function names may be redefined to operate on differing argument types. Thus, if x and y are integers then $x + y$ will return an integer result, but the ‘+’ symbol may also be selected as a summation operator for vector quantities and so, if a and b are vector quantities then $a + b$ will now return a vector quantity. In both cases the summation

symbol ‘+’ has been used. Though this facility could be useful in some situations, the disadvantages are that the program code may become more difficult to interpret by those programmers other than the originator, and the compiler (linker) is required to work even harder in an attempt to resolve code features that are context sensitive. Hence the often referred to ‘programmer’s coffee-break’, signifying a very long program compile.

In an effort to create a language that is both flexible and terse, C++ has a myriad of symbolic operators that to the untrained eye may appear intimidating:

+	-	*	/	%	^	&
!	=	<	>	+=	-=	*=
==	!=	->*	->	.	::	#
.*	?:	##	&=	<<	>>	<<=
>>=	&&	—	++	--	,	[]

The author is not ashamed to admit to the possible encroachment of a new form of dyslexia whenever these operators are viewed in the midst of a segment of code. Unfortunately nearly all of these operators may be overloaded so that their original meanings are modified, and they may also function in a way that is not expected!

In the process of melding all that was essential in C with the new object-oriented features that typify C++ as an object-oriented language, it was necessary to make a few compromises, leading to a language that is both complex to learn and unwieldy for compilers to process. During compilation and the process that converts the textually coded program instructions to a set of machine-code instructions, several passes of the program will be required, commencing with the preprocessor expansion that will initially expand macro definitions and then insert all the referenced `include` (dependent) files. One of the compromises alluded to earlier concerns the need to maintain compatibility with its progenitor. A consequence of this approach is that all C++ constructs are converted by the preprocessor into standard C code; thus, C++ is actually a refined form of C.

Both C and C++ typically employ a *multi-pass* compiler that can result in greatly extended compilation times compared to *single-pass* compilers as used by such languages as BASIC and Pascal. The combination of features such as virtual functions and operator overloading means that in any large application it simply isn’t possible to look at a line of code and be able to deduce for certain which functions are to be called. The multi-

pass compiler thus becomes an absolute necessity if such references are to be adequately resolved.

Other features such as class templates, function templates and exception handling (the ability to safely intercept and proceed from run-time errors) result in a language that is both flexible and multi-purpose. It should therefore come as no surprise to learn that the majority of applications today are coded in C++, and even the very operating systems that these applications run under (Windows and Unix) are also coded in C++. But, through its immense popularity are sown the seeds of a growing discontent and uneasiness that perhaps C++ might just be too powerful, that maybe each new version produces a behemoth of ever increasing complexity requiring ever more effort on the part of the developer to master. In particular, both C and C++ offer a flexibility of expression that allows for more than one method to implement many of the standard operations, such as looping and iterating, and which inevitably leads programmers to develop a *style* that might prove difficult for other programmers to penetrate. This *flexibility* is supposedly in the name of code efficiency, but in these modern times where multi-team programming projects are increasingly becoming the norm it is imperative that individual programmers should not have to overcome varied syntax-dialects and other cumbersome idiosyncrasies that C/C++ deems to be legitimate. Despite these drawbacks C++ is probably the most powerful computer language available today and will continue to be so for some time to come.

4.2.2 Visual Basic

If C++ is the vanguard of programming languages, then BASIC[4] is the humble foot-soldier. Originally developed at Dartmouth College in 1962 as a simple language that would allow students to write quickly and test programs of their own, it has a reputation for being not overly-complex to learn and reasonably safe in usage. BASIC is an interpreted language, that is the textual code comprising a program is not compiled directly to machine-code as in a compiled language such as C, but instead the code is converted to a tokenised form¹ which is then executed by the interpreter. Unlike C/C++, the BASIC language has no facility for direct access to computer memory through the use of pointers, and so both variables and procedures remain in the abstract space of the program. In this

¹a computer code intermediate between machine code and a high-level language

manner the environment of a BASIC program can be reasonably well isolated from the operating system, and in so doing the operating system and other program components that exist in memory remain protected from the otherwise possibly unlawful or roguish behaviour of true compiled programs that allow access to all areas of memory, if permitted by the operating system.

During the intervening years BASIC has undergone one transformation after another, either as a reaction to a change of platform or, alternatively, in an effort to bolster an otherwise mundane language with features having increased functionality and scope. Since it was originally developed of unambitious goals, BASIC has had many reasons why it should need to evolve. Due to its simplicity the BASIC interpreter required little memory to operate, and for this reason it was chosen to be implemented on many of the pioneering and popular 8-bit computer platforms of the late 1970s (Altair, Commodore Pet, Tandy TRS-80) and the very first 16-bit IBM PC in 1985. In fact, it was the rapid expansion of the personal computer market that enabled the BASIC language to position itself as the *de facto* high-level development language of the masses. And so it has remained to this day.

More recently, when BASIC was thought to be showing its age in the light of new operating systems, Microsoft decided in 1992 to resurrect it as a much-improved language (Visual Basic) for the development of Windows programs. Considering that Windows had been written entirely in C and assembler language, it was indeed a brave and calculated move by Microsoft to market a new version of BASIC that would greatly facilitate the creation of Windows programs that were previously the sole province of more serious-minded programmers. Microsoft correctly reasoned that by leveraging the power and ease-of-use that Visual Basic brought to program development, the world would become more receptive to the adoption of Windows-type operating systems, and since Microsoft owned both products then its interests would be doubly served by such a strategy.

Unlike its predecessors which were procedural languages (where the application determines the flow and execution of the program in a logical manner), Visual Basic manages to encompass the event-driven architecture (program instructions execute only when a particular event calls that section of code into action) that underpins those dynamic aspects we associate with the Windows environment. It is this harnessing and proper integration of the event-driven paradigm into an easy-to-use programming environment that has made

Visual Basic such a popular Windows development tool. Visual Basic has not attempted to implement some of the user-definable language features found in its more powerful stablemate C++, but instead has retained the spirit of the original Basic while attempting to modernise the language through the inclusion of object-oriented facilities.

The term *rapid application development* tool, or RAD, came into being shortly after Visual Basic was presented to the world. It is through the success of Visual Basic that other RAD tools have had the opportunity to come into existence and show their own particular strengths. Naturally, this situation can only persist so long as RAD tools provide opportunities for improvement, as in evolution. In the case of Visual Basic there are some areas which may and have been improved on, and so at the present time there are several heirs apparent, including those offerings from Borland (Delphi), and PowerSoft (PowerBuilder).

The cornerstone of program development is the component or Visual Basic control (VBX). The VBX is a *dynamic link library* (DLL) that exports the methods and properties of an object instance, be it a particular type of Windows form or a non-visual entity such as a database link. In order to access any one of the VBXs that ship with Visual Basic (or those that have been obtained from third-party vendors) a specialised toolbox is provided that contains, in iconic form, all the components. In order to utilise a visual component it is firstly dragged from the toolbox by using the mouse and then dropped onto the form, whereupon it may be repositioned or resized using the various *handles* that all visual controls possess.

Though initially hyped as an object-oriented development tool, in fact Visual Basic falls short of these claims. In truth the only objects evident in the Visual Basic environment are the forms and their associated visual components. The BASIC language has been extended to accommodate interaction with objects, such as object referencing and property referencing, but it does not allow the creation of new object or class structures; the only objects that are realised within the Visual Basic development environment are those that have been obtained from the toolbox. In other words, Visual Basic will only allow the object-oriented paradigm to operate over those visual and non-visual elements that derive from the toolbox, and the programmer/analyst is prevented from applying the paradigm to any other self-inspired structures or models ². That is not to say that Visual Basic does

²Visual Basic version 4, the current release, partially rectifies this shortcoming

not support the creation of complex data structures or that it cannot readily be employed in the modelling of complex systems, but the point to using OOP is that the entities one creates (classes) can bear striking similarities to those entities of our own experience when considered in an objective manner. If we attempt to use Visual Basic to emulate the syntactical cohesiveness that derives from adopting the OOP approach then we may begin to lose sight of the problem and become overwhelmed instead by the necessity of performing the requisite book-keeping chores that an OOP language automatically manages.

At the present time Visual Basic has claimed for itself a significant proportion of the RAD development tool market. Not only is it being used by both the casual and serious-minded developer who might, for example, be interested in creating multi-media programs for education, but it is also to be found in the 'city' where it is normally employed as a 'front-end' to much more sophisticated programs that specialise in trading derivatives and the like. It is the latter that provides a glimpse of the true power and usefulness of Visual Basic. The language of Visual Basic is an interpreted one, and as such the speed of operation is limited by the ability to process the higher language elements at run-time, resulting in an operational speed of $\sim 20\%$ of compiled code, but while Visual Basic is executing code that resides within a control (VBX) or dll (dynamic link library) then speed is optimal. This leads us to two very important conclusions as to when and how VB should or should not be employed:

The generation of user interfaces. Virtually all the elements of a user interface are constructed from calls to the Windows API (application program interface), either directly or indirectly via a control. In each instance it is the code within the various Windows API routines that are executed and which result in the particular look and behaviour of a Windows element, such as a form being minimised or an edit component handling textual input. The component that one associates with a control is, in essence, a wrapper for a collection of API calls, but it also manages to *surface* various properties of the control in order to facilitate interactive editing by the developer. Thus, in general a Windows interface constructed with Visual Basic will not appear to be unduly slow, as long as there is not a great preponderance of BASIC code associated with the creation or destruction of the Windows elements. It is in this rôle that Visual Basic is now being used to revamp the legacy code (code that

was originally developed in the pre-Windows era and employing such languages as Cobol or standard BASIC) that companies rely upon for current operations but are unwilling to invest in a complete rewrite in a modern Windows language.

Multi-media or database applications. Interpreted development tools similar to Visual Basic rely totally on the availability of Windows components for their apparent power and flexibility of design. We have already seen in the previous paragraph how the construction of a visual interface of a program is a simple affair, but Visual Basic is capable of much more and is only limited by the power invested in the various controls that it can call upon. The advent of such operating systems as Windows 3.X and Windows 95 has led to a profusion of applications that utilise sound and video effects, but of more importance to the ‘business’ user is the ability to access information stored in a database which Visual Basic comfortably manages to do. All of these technologies are provided by the Visual Basic custom controls (VBXs) that are either supplied with the Visual Basic development tool itself, or obtained from third-party vendors.

4.2.3 Delphi/Object Pascal

Pascal is a computer language which was designed by Professor Niklaus Wirth at *Eidgenössische Technische Hochschule* in Zurich. The first draft was completed in 1968 although the first definitive text on the language was published in 1975 [9]. Since that time Pascal has become more and more popular for teaching principles of programming but also as a language in which to develop sophisticated software. The previous two sections described languages that are positioned at opposing ends of the language spectrum, where one is viewed as a powerful but, in the wrong hands, an unsafe language, whereas the other is seen as a language for the masses, but not terribly sophisticated or extendible. Pascal, on the other hand, attempts to tread the middle road by providing the flexibility and power of C while providing a syntax, much more similar to the English language than C, and a development environment that is both simple in concept and yet capable of providing sufficient power for almost all the tasks that a programmer is likely to encounter.

We have already seen how the C language treats both character and integer types as virtually identical (other anomalies also exist), only differentiating between the two when required to. The Pascal compiler will not allow such ambiguity and so the programmer

must explicitly declare variables of the appropriate type and ensure that they are employed correctly. It is for this reason that Pascal is said to be a 'type-safe' language. However, it is possible to circumvent type-safe restrictions, but in so doing the programmer must make clear the intention to do so rather than allow the compiler to rely upon a default mode. In addition Pascal requires the programmer to declare all variables (unlike C) and public functions prior to use, and it is this methodical and structured approach that appeals to the educationalist.

Pascal owes much of its current popularity to the implementation introduced by Borland in 1984 and known as Turbo Pascal. As part of Borland's continuing improvements to the language, in 1989 a new version was introduced (v5.5) that implemented the much acclaimed object oriented extensions, much of it borrowed from the C++ language. This chain of events finally (at the current time of writing) culminated in a version of Pascal called Delphi that encompassed not only a very sophisticated implementation of OOP but also a development environment that put it at the forefront of *rapid application development* tools. Unlike other RAD tools, the power and sophistication of Delphi is easily demonstrated by the fact that the whole of the visual development environment was created using the version of Object Pascal that underpins Delphi.

Program Development in Delphi has all the benefits of development in Visual Basic, plus a few more:

- **Enhanced Compiler and Run-Time Performance** – as discussed in the previous section, the Visual Basic compiler actually converts the source-code of a program into a set of intermediate instructions, sometimes called p-code, which are then interpreted and executed at run-time. An interpreter will generally perform the compile phase much quicker than a true compiler, but will suffer in terms of execution speed when the program is eventually run by a user. In the case of Delphi and Object Pascal this no longer applies. Object Pascal is a true compiled language, meaning that compilation will reduce the source-code to machine instructions directly rather than intermediate p-code. The fact is that compilation under Object Pascal is as fast as compilation by an interpreter such as Visual Basic, and is attributable not only to the compiler being an optimising compiler, but to Borland substituting a more powerful and efficient object file for the Microsoft/industry standard object (.OBJ) file that is employed by C/C++ and other similar language compilers. Another

contributing factor is that the Pascal language was designed to undergo compilation in a single-pass of the source code, unlike the C/C++ language, and it is for these two reasons that the Object Pascal compiler is demonstrably the fastest compiler among its peers – the programmer’s coffee-break is now redundant! Naturally, a fast optimising compiler results in a run-time performance that is considerably enhanced, and is the equal of applications created using C++ and equally sophisticated compiler technology.

- **Reusable Libraries** – one of the essential requisites of a high-level language is the ability to create and use code libraries. A library is a file that contains any or all of the following: constant declarations, e. g., `pi = 3.14159265359` , and global variables; type definitions, e. g.

```
type
    vector = class
        x : real;
        y : real;
        z : real
    end;
```

...and functional/procedural interface definitions, such as:

```
procedure InverseMatrix( mat : TMatrix );
function Determinant( mat : TMatrix ) : Real;
```

...that are exported to client applications as library resources. In the case of C/C++ the definition file³ is a separate text-file while the machine code associated with the implementation of the library is kept in an object file which is *linked* into the executable during the final linkage process. Alternatively, Visual Basic provides library re-use to an application in the form of a global .BAS file that contains both interface and implementation details within a single text-file. In contrast, Delphi provides the same facility but instead uses a precompiled Pascal unit (.DCU) that combines both the interface section and the object code relating to its implementation. This form of library offers the advantage of very fast linkage. Additionally, unlike Visual Basic, Delphi libraries (units) are completely divorced from any necessity of belonging to

³The definition file is synonymous with *header file*.

a form or application, and in this respect they are truly reusable libraries, without restriction.

- **Custom Components** – custom control components, or VBXs, can be written for Visual Basic that provide functionality across projects. A disadvantage to custom VBXs is the high degree of low-level Windows expertise required to create them. Controls in Delphi are called VCLs, and are much easier to create. Unlike Visual Basic, where controls must be created with an external C/C++ compiler, VCLs (Visual Component Library) are created from within the Delphi development environment itself, in the same Object Pascal language that is used for normal Delphi development. One other benefit afforded to the Delphi programmer is that a VCL may be subclassed in order to create a new and customised version of a VCL component that is modified or enhanced according to the programmer's inclination. Such a facility in Visual Basic is not available. Custom controls in the C/C++ language are not explicitly supported, although recent versions of C++ produced by Microsoft, Borland and other vendors have the capability to import VBX controls as well as the more up-to-date and sophisticated OCX (OLE⁴ derived) components.
- **A True Object-Oriented Language** – The basis of all that is Delphi is the Object Pascal language and its associated compiler. All of the core object-programming techniques are fully implemented in Object Pascal. For instance, it supports virtual methods, real encapsulation (including the use of the **Public**, **Private** and **Protected** keywords), real polymorphism, and real inheritance. It is not restricted to applying these techniques to forms alone, as in the case of Visual Basic. It also adds features such as runtime type checking, the **Published** keyword, properties, and functions that return complex data-types. In order to demonstrate the power that a fully object-oriented language such as Object Pascal wields, it was employed in the creation of Delphi itself, a development environment that is both modular and extendible in a fully OOP sense. Delphi is the product of hundreds of separate classes being developed and then melded together to form one operating whole, and to its credit all of these classes are made available to the programmer, whether to modify existing or create completely new VCL components that will integrate with the Delphi environment, or to develop totally new applications.

⁴Object Linking and Embedding – a standard for embedding one application within another.

4.2.4 Other Development Tools

Applications and languages such as FoxPro, PowerBuilder, Modula 2, and CA-Realizer were also possible candidates as the development tool for this research, but each suffers from some limitation, attributable to either the target technology or degree of maturity displayed by the product. In particular, both FoxPro and PowerBuilder are products that are aimed at serving the database development market, and although both are endowed with reasonably sophisticated macro languages, they cannot be considered as general purpose languages, nor do they support a complete application framework that is capable of harnessing the complexity or totality of the Windows operating system. On the other hand, Modula 2, though regarded as a powerful language in its own right, has not at the present time made any impact at all as a language for commercial use. Instead, during the last 8 years it has served the educational community in much the same way that early versions of Turbo Pascal did, that is to teach good programming practices. Other than serving this niche market, Modula 2 has lacked the necessary tools, support and enhancements that are prerequisites for commercial take-up. Similarly, CA-Realizer, though supporting a much enhanced version of BASIC, appears to have gone the same way.

4.2.5 Conclusion

The choice of computer language or development tool is not a simple matter, and just as the choice of our next car is dependent on such factors as cost, design, colour and utility, so the choice of language may be dependent on other similar criteria. Unfortunately, in spite of all the facts the final decision is usually a matter of personal choice; and so it is with programming languages. The differences between various language implementations are usually quite significant, and may amount to purely syntactical, technological, or even visual differences. While such criteria as fuel efficiency and power output may be sufficient to separate one vehicle from another, other less tangible criteria such as image conveyed (with all its the cultural overtones) and manufacturer perception (Lada vs Range Rover?) will probably have equal importance in the decision making process.

Bench-testing is the process whereby a product is tested against various performance criteria in order to assess its ability to meet or exceed said criteria. In most cases this amounts to a review of several products of similar class, such as modems, and the results

are then tabulated showing the performance level reached for each test criterion. In the case of a modem it is quite possible to select that one which provides the highest performance, based purely upon the objective measurements. In contrast, software applications have few features that are objectively quantifiable, and in today's world of very fast computers it might be irrelevant if one word-processor requires 0.13 seconds to complete a word search of a document while another requires 0.26 seconds for the same operation. Though one has twice the performance of the other, the actual perception is not so apparent. Thus, given that simple bench-testing cannot be regarded as the sole criterion for the selection of a development language, and the realisation that the decision making process is as much subjective as it is objective, then the rationale behind any choice is, to some extent, irrelevant. This is the conclusion that the author has arrived at, and the final decision to adopt Delphi and the Object Pascal language as the system of choice is one based on those reasons given above and in the previous sections.

During the early stages of the research programme most of the initial coding was undertaken in C++, and although this proceeded satisfactorily it was not until the Windows interface needed attention that the difficulties of contending with the Windows framework, as provided by Borland ⁵, became apparent. Since it was anticipated that a major part of this programme would require close interaction with Windows components, a tool that facilitated this process in a user-friendly manner would be of great advantage. It was for this purpose that Visual Basic was adopted, not for the reason that it provided a powerful language, but for the innovatory way it allowed the developer to easily create and manipulate Windows components. Unfortunately it later became clear that the language itself would be the main limitation due to its inability to cater for true object programming techniques, although much work had been undertaken in designing the visual interface. It was at this time that Delphi appeared on the market and the decision was taken to transfer all work over to this new development system. On the whole this cycle of change must have impacted on the work schedule, but it was not totally without benefit since much research had already been accomplished and ideas had been tested. Ultimately the move to Delphi proved to be a rewarding experience as the author began to realise that technology no longer proved to be a hindrance, and instead all possibilities became tangible ones that could be explored in their entirety.

⁵Borland's implementation of the Windows API is called OWL, or Object Windows Library.

Chapter 5

Description of Model

The Component-Based Model (CBM) is not the be-all and end-all of the optical analysis program as discussed in this thesis, but it is simply one of the leading *actors* in a play that depends very much on the props and supporting cast. In isolation, each cannot function without the cooperation of the other, and it is through the interplay of characters and plot that the whole process becomes meaningful and tangible.

In contrast, the Surface-Based Model (SBM) exists within the boundaries defined by the spreadsheet editor. The user is confined to viewing the surface relationships as a list of surface data that conforms to a linear sequence, and all manipulation and interaction between user and system takes place in the same domain. This scenario obviously limits the types of behaviour that may be exhibited to those that can be implemented within this overall scheme.

The framework of components and their interactions that are required to support the CBM are, without doubt, more complex than would be required of the SBM, and unlike the latter they are capable of a greater degree of independence. Whereas virtually all operations on a SBM are undertaken within the confines and limitations imposed by a spreadsheet editor, the CBM recognises that the sequential property of a lens-system is independent of the more localised properties of a lens, such as its shape or power. Thus, the CBM will distinguish between the global and local properties by offering facilities that decouple the operations that enable such property editing to take place. Similarly, the general lens system is derived from a number of lens components that are recognisably distinct. In the case of the surface-based model where lenses are constructed from surfaces alone, the

component-based model provides building blocks that already encompass such components as singlet lenses and prisms. The user is then only required to set the particular properties of each component.

In consequence, the component-based model, as briefly outlined above, is clearly a much higher level characterisation of a lens system than its predecessor, the surface-based model. There are other facets to the model that are necessary to maintaining the pretence of a real optical system, and these will also be described in more detail in the following sections.

5.1 The Toolbar

As already mentioned in the introduction to this chapter, the component-based model utilises components that embody complex arrangements of surfaces and glasses that combine together to produce essential lens-system subcomponents, such as lens singlets, prisms, or even at the lowest level, single surfaces. It is at this level that the distinction between a CBM and a SBM becomes most apparent, for the CBM actually defines the number of components that are available to the developer, while the SBM is restricted to a user-defined and application-specific collection of low-level surface descriptions. Component



Figure 5.1: The Toolbar

based applications have been familiar to users ever since Windows was launched, and are to be seen at their simplest level in such programs as File Manager¹ and Program Man-

¹File Manager has been superseded by Windows Explorer in Windows 95.

ager. In particular, such tools as CAD² and graphics programs have easily lent themselves to component-centred interaction. The principal vehicle for the display and storage of components is the **Toolbox** or **Toolbar**, where an example of the latter is shown in Figure 5.1. The distinction between the two is that the former is implemented as a moveable window whereas the latter is fixed and usually located beneath the menu bar. In either case a collection of buttons is presented to the user with associated icons that provide visual clues to the functionality of each one. Naturally, each button provides a different service, much in the same way as a spanner or a screwdriver would, when selected from a hardware toolbox.

The construction of the lens toolbox is simple in essence, although there are a few steps to be covered for a fuller understanding. Firstly, the icon buttons are derived from simple speedbuttons, as the following code shows:

```
TLensIcon = class(TSpeedButton)
public
    LensRef    :TLensRef;
    hBmp       :HBitmap;
    Dragable   :boolean;
    CompType   :TLensType;
    CompName   :string;
    Constructor create(AOwner:TComponent);virtual;
end;
```

where...

TLensIcon is the name of the new class, derived from **TSpeedButton**, that describes the behaviour of the buttons on the toolbar;

TSpeedButton is the standard button, from which **TLensIcon** is derived;

LensRef is a reference to the type of lens that the **LensIcon** represents;

hbmp is the graphic that appears on the **LensIcon**;

²VISIO is an excellent example of a component-based CAD application.

CompType is a variable of type `TLensType` which characterises lens components as belonging to one of several types, defined as:

```
TLensType = (alens, aspacer, atransform, asource, adisperser,  
areflector, amodifier, aregulator, anaperture, asurface, none);
```

CompName is the name of the lens component, in string form, that the actual instance of `TLensIcon` represents.

Any lens type may be selected and then placed on a System Form by employing the common procedure of ‘drag-and-drop’.

The contents of the Toolbar are dynamically determined at compile-time, and is made possible by the initialisation³ feature of Pascal’s unit modules. Each lens type has been developed as a separate module, and included with each module is an initialisation section that loads details of the exported `LensIcon` into an array that is owned by the main form. When the main form is about to *paint* itself onto the display (the method is known as `FormCreate`) for the first time, it checks the contents of this array and displays those buttons and bitmaps that were found. This procedure was adopted in order to isolate the main form from the numerous lens modules, the result being that both could be developed in virtual isolation from one another, and new lens modules could be developed and included into the main application in a transparent and standard manner.

5.2 The System Form

While the visual representation of the `LensIcon` component carries with it few implications, the way in which these components are to be grouped together in the manner of a real optical system proves more problematic. Previous sections have discussed the differences between the surface-based and component-based models, and it is clear that most arise from the visual models upon which they are individually based although, so far, little detail has been given of the component model. Historically, the spreadsheet has been the sole means for viewing a lens system in its non-visual and numeric form, neglecting those earlier programs that utilised the simple command-line. One exception is a program

³Initialisation code associated with each unit module is executed prior to the execution of the main program.

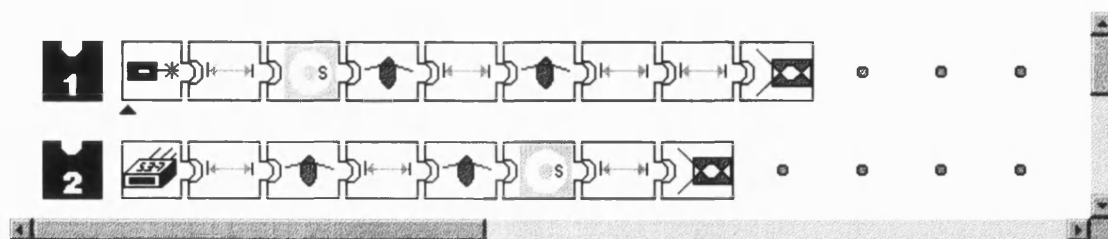


Figure 5.2: The OptikWerks 'Worksheet'

called OptikWerks⁴ and initially released in 1993, one year after the commencement of this research programme. This program was probably the first to describe a lens system in terms of components, and to provide a configuration editor that maintained this emphasis.

Figure 5.2 shows the configuration editor supplied by OptikWerks, otherwise referred to as a 'worksheet'. Notice that two system worksheets are provided, possibly to enable the copying and moving of components from one system to the other. We may also observe that the component sequence in each system proceeds from left to right, commencing with the source, and proceeding uninterrupted. While this complies with accepted practice⁵, it may pose a problem when dealing with systems whose components are so numerous that they are no longer visible on the form. It was for this reason that the traditional approach to designing a configuration editor was abandoned in favour of one that is more closely allied to more modern practices.

Figure 5.3 shows the configuration editor as implemented in the author's program⁶. Note that the components are now distributed within a form, and that the sequence proceeds from left to right, commencing with a source.

When created, the form is initially empty and comprising of blank icon buttons only, as shown in the bottom-most row. The process of populating the form is achieved by dragging components from the toolbar and then dropping onto the chosen empty icon

⁴A lens design and analysis program, produced by OptikWerk, Inc.

⁵It is 'traditional' in optical diagrams to have light rays travelling from left to right, prior to entering a system.

⁶The program is called IRIS, named after the Greek word for *Rainbow*.

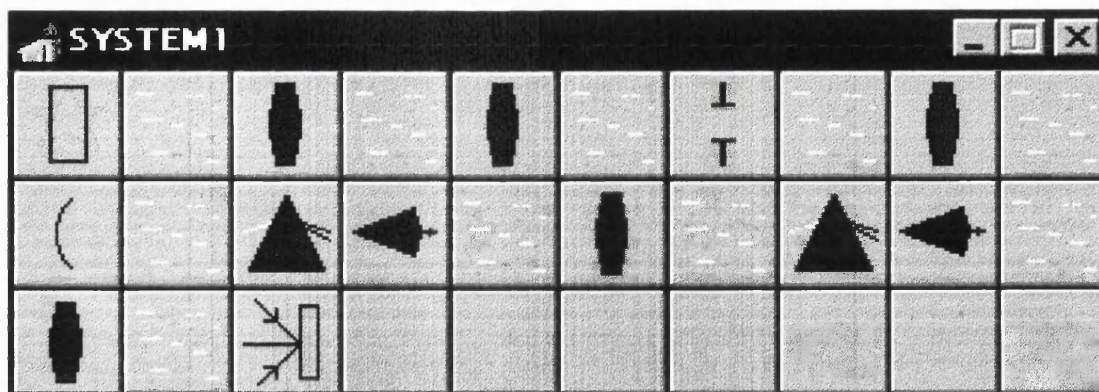


Figure 5.3: IRIS Configuration Editor

button. When a row has been completed then component placement proceeds from the left-most button on the next row. Thus the complete description of a lens system is arrived at by visually scanning the complete array of lens icons, in the same manner as one would read a piece of text, i. e. scanning a row of text from left to right, and then proceeding onto subsequent rows in a similar fashion. In this way very complex optical systems may be viewed in their entirety, unlike the approach taken by OptikWerks, where only discrete segments of the sequence are visible at any one time. The same comment will also apply to spreadsheet editors, although the problem is much more marked since no visual clues are available (other than numeric data in textual form) that might describe the configuration of the system or the nature of the individual components.

One other point worth noting is that the configuration editor is a form and not a Windows element, as in the case of the Optikwerks Worksheet, and so retains all the properties of a normal window. As such, more than one form may be present, where each may contain a different lens system and components may be moved and copied from one form to another. Additionally, forms may also be minimised in order to increase free space on the desktop. The choice of a form as the repository for the configuration editor thus reflects the same object characteristics to be found in a lens system, that is: it is unique, bounded, derived from components and may enter into any legal transaction with other similar forms.

5.3 The Lens Container

When a lens icon is dragged from the toolbox and dropped onto a system form it might appear that a direct copy has been made. This is not so, and the truth is that while the lens icon is only a reference to the lens-type that it represents, the lens-button is actually a container for the lens itself. The following extract of code is from the declaration, or definition, of the `TLensButton` class:

```
type
  TLensButton = class(TSpeedButton)
  protected
    {Protected declarations}
    procedure MouseDown(Sender: TObject; Button: TMouseButton;
                        Shift: TShiftState; X, Y: Integer);virtual;
    procedure DragOver(Sender, Source: TObject; X, Y: Integer;
                        State: TDragState; var Accept: Boolean);virtual;
    procedure Click(Sender: TObject);virtual;
    procedure DragDrop(Sender, Source: TObject; X, Y: Integer);
    procedure MouseUp(Sender: TObject; Button: TMouseButton;
                        Shift: TShiftState; X, Y: Integer);
    procedure DeleteLens;
    procedure CopyLens(src: TLensButton);
    procedure NewLens(ic: TLensIcon);
  public
    {Public declarations}
    copy: boolean;
    ndx: integer;
    TheLens: TLens;
    occupied: boolean;
    constructor create(AOwner: TForm; n: integer);virtual;
    destructor destroy;override;
  end;
```

TLensButton is derived from **TSpeedButton**, a standard Windows component in toolbars, and so its visual representation is naturally that of a speedbutton. In addition, several procedures have been added that facilitate such editing functions as move, copy and delete. Probably the most important item that has been added to this newly derived class is a variable of type **TLens** called **TheLens**. This variable is essentially a pointer to an instance of a **Lens** object. When an instance of **TLensButton** is first created it will also generate a new instance of **TLens**, the default, which has very basic properties that support all the normal properties of a regular lens of whatever type, but without effect. A consequence of this is that a lens system appearing within a lens form remains unchanged when one or more default lensbuttons are inserted at various points within the same sequence. Though such a feature is not to be found in any other program, the author considered it a necessary feature for inclusion in any program that purportedly supports graphical in-place editing, since the alternative would have been counter-intuitive⁷. Other given properties of **TLensButton** include **occupied**, which is **false** when the **LensButton** is of the default variety and **true** when it is not, and **copy** which is **true** when a **LensButton** is being copied from one position to another and **false** when the lens is being moved, instead. Both of these properties enable the instance of **TLensButton** to query its own state, which will be useful in some situations.

One other property worth mentioning is the integer variable **ndx**. During the creation of the system form all lensbuttons are given an index, commencing with the value of 1 for the first lensbutton and incrementing this value for subsequent lensbuttons. These values remain with the lensbutton throughout the lifetime of the form and are unaffected by a lens being copied or moved. The purpose of the index is to give each lensbutton a sense of its own position relative to other neighbouring lenses. It manages to do this by using two methods, **Next** and **Previous**, that are declared within the base class **TLens**. A definition of the **Next** method is given here:

⁷When inserted into a system, a non-effective lens should have no affect – any other result would be undefined and so unsupportable.

```

function TLens.Next:TLens;
begin
  try
    with owner as TLensform do
      begin
        {get the next lens reference from array Lens[]}
        result := Lens[ndx + 1];
        {if a default lens...}
        if result.comptype = none
        {...then ask the next lens for the next lens}
        then result := result.next;
      end;
      {if the search falls outside array Lens[]}
      {then return a null lens}
      except on Exception do result := nil;
    end;
  end;
end;

```

Since every lens has a reference to the form in which it was created (**owner**), then it is able to access the lens array **Lens[1..MaxLenses]**, in which is stored a reference to every lens *owned* by the form. If a lens needs to know what is the next neighbour in the sequence, then the following call is made: **self.next**, which will return a reference to the next lens in the sequence. The type of lens that is returned in the **result** variable is tested and if valid then all is well, but if the type of lens found is a default type then the **Next** method for that lens is executed. This process cascades down the sequence until a valid lens is returned or the search falls outside the **Lens** array.

The previous two sections have described the System Form (**TLensForm**) and the Lens-Buttons (**TLensButton**). Both are containers: the LensForm is the host/owner of the LensButtons, which are in turn hosts for their own **TLens** reference. The following two sections will concentrate upon **TLens** – after all, the lens is the principal ‘actor’ in this production, while the System Form may be considered as the stage and the LensButton as the corporeal presence of the actor.

5.4 Methods and Properties of TLens

Methods are the procedures and functions that are tightly bound to a class description. When an instance of a class (an object) is created, it is guaranteed that the object will have a data-set that is exclusive to itself and no other object. A method that is declared within the same class will, by default, have access to this data and any other methods that are declared within the same class. When viewed anthropomorphically, the data may be considered to represent the characteristics of the object, while the methods reflect its behaviour. The properties of a class are slightly more complicated in that they may refer directly to class variables, or alternatively they may return a data type via a specified function call. The following segment of code is a complete description of the **TLens** class and its associated methods and properties.

```
TLens = class(TObject)
protected
    compname      :string;
    fRef          :integer;
    function GetNdx : integer;
    function GetLensName:string;virtual;
    function GetMedium:TGlass;virtual;
    function GetSurface:TSurface;virtual;
    procedure SetARef(newval:integer);
public
    comptype      :TLensType;
    lensref       :TLensRef;
    EditorActivated :boolean;
    parent        :TObject; {reference to LensButton}
    owner         :TForm;   {reference to LensForm}
    editor        :TForm;   {reference to LensEd}
    constructor create(par:TObject);virtual;
    destructor destroy;override;
```

```

procedure    assign(lens:TLens);virtual;
procedure    reverse;virtual;abstract;
procedure    ProcessRays(rays:array of TObject);virtual;
procedure    WriteToFile(const fname:string);virtual;
procedure    ReadFromFile(const fname:string);virtual;abstract;
procedure    SetRef(j:integer;newval:string);
procedure    LoadRef(s:TComboBox);
procedure    GetPropertyList(lst:TStringList);virtual;
function     initialise(var msg:string):boolean;virtual;
function     Next:TLens;
function     Previous:TLens;
function     ActiveLensDirn:TDirn;
function     IsName(id:string):boolean;
function     GetRef(j:integer):string;
property     Surface:TSurface read GetSurface;
property     ndx:integer read GetNdx;
property     name:string read GetLensName;
property     medium:TGlass read GetMedium;
property     Ref:integer read fRef write SetARef;
end;

```

The above listing shows the complex interface that **TLens** presents. Some of the above methods and properties will be discussed in more detail in later sections and so we shall concentrate on those that are particularly relevant to **TLens** alone and which reveal important insights into the inner workings of this class.

When deriving a base class it is important to identify those methods that may be easily overridden by subsequent descendants. For instance, a single lens will significantly differ from a diffraction grating in how it will affect incident rays, since the surface types are so different – the former being comprised of refractive surfaces and the latter having only a single diffractive surface. The solution adopted by the author is to define a method **ProcessRays** that accepts a parameter **rays** of type **array of TObject**. **ProcessRays** is a method that implements whatever the action of a general lens-type (descended from **TLens**) is to an incident collection of rays, **rays**. When a lens is of the default type then

the rays will be unaffected, the lens behaving as if it had no refractive power or spatial extent. Alternatively, a descendant of **TLens** will reimplement the **ProcessRays** method to handle its own specific raytrace algorithm.

The argument of the procedural method is of type **array of TObject** and not **array of Rays** because the array contains not only finite rays but paraxial rays. Due to the lack of similarity between these two types of rays it was not considered appropriate to make one the descendant of the other, and instead two ray classes were devised: **TParaxialRay** and **TFiniteRay**. Since both are automatically descendants of **TObject** then the type **array of TObject** will always allow either or both types of rays to be included in the method call.

The **assign** method is used to copy lens data from the lens specified as the argument of the call to the lens whose method made the call, thus: **Lens1.assign(Lens2)**, which will copy data from **Lens2** to **Lens1**. In the case where **Lens1** is of the same type as **Lens2** then there will be an exact correspondence between data fields of the two lenses, and the assignment of data from one to the other proceeds in a straightforward manner. Where the two lenses are different, i. e. **Lens1** is a singlet lens having spherical surfaces while **Lens2** is also a singlet lens but having an aspheric surface, then by virtue of polymorphism and RTTI (Run-Time Type Information) it is possible to ensure that only the fields that are common to both lenses are employed in the assignment transaction. In the case given above, this results in only the spherical form of the aspheric surface being passed over to **Lens1**, where anything more will probably result in an error that will be either caught at compile-time or will result in a run-time error.

One of the truly innovative features that has been included with **TLens** and other associated modules is the support for multi-pass components. Multi-pass optics, though not common occurrences, are to be found in such systems as the Super-Schmidt camera (see [10]). This is a catadioptric imaging system comprising a double concentric meniscus surrounding a central crown-flint doublet Schmidt plate. The image surface is positioned on the object side of the final shell and so image rays will have to traverse this shell twice. Conventional optical design programs do not cater for such a circumstance but, instead, it is left to the designer to include the shell twice, where the second time it will be entered in reverse order and reversed curvatures. A more general system having component tilts will demand very much more care of the designer when entering the data. **TLens**, on

the other hand, has been designed to operate with incident rays travelling from left-to-right (the conventional direction **and** right-to-left. In order to achieve this feat it must firstly recognise in which direction the rays are travelling when a call to the **ProcessRays** method is made. It achieves this through a method called **ActiveLensDirn** which returns a value of type: **TDirn = (left,right,here)**, where **left** and **right** indicate that rays are coming from the left or the right, and **here** indicates neither.

The key to determining the ray direction is a variable called **ActiveLens**, of type **TLens**, that is stored in the system form and which is readily accessed by any lens component through the **owner** field defined in **TLens**. Whenever a lens becomes engaged in processing rays it will inform the system form⁸ of the fact by assigning its object address to the value of **ActiveLens**, thus allowing any other object that has read privilege to access the same lens. This is, in fact, what happens whenever the **ProcessRays** method is called. Prior to processing rays, a lens will refer to the system form for the index of the lens that was last active in processing the rays. If the index is less than its own index then the rays are travelling in a normal direction (left-to-right), but if the index is greater then the rays have reversed their normal direction. When reversed rays are detected the lens will instigate a method called **Reverse**, which is a virtual method that is to be overridden by all descendants of **TLens**. In the case of a singlet spherical lens described by the following simple parameter set $[r_1, t, r_2]$, and the order is important, then a reversal of this lens will result in the set: $[-r_2, t, -r_1]$.

5.5 Editor Linkage

Modern program development environments that support visual class and object modules, such as Visual Basic and Delphi, recognise that these custom controls require to be configured for the particular task at hand. For instance, both Delphi and Visual Basic provide form controls, the same rectangular form or window from which dialogue and application main forms are constructed. Customisation of these particular controls may necessitate the setting of around twenty individual properties, such as the height, width and position of the window when it first appears on the display. Rather than set these properties in code, the development environment provides an *Object Inspector* or *Editor* that appears

⁸The system form is also known as the Configuration Editor, and in code is referred to as **TLensFrm**.

in Figure 5.4.

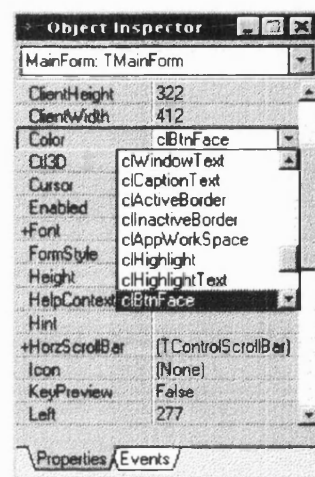


Figure 5.4: The Delphi Object Inspector

The presentation of the object editor appears similar to a spreadsheet, with the left-hand column providing a list of property names and the right-hand column the current property values. Editing of a property is achieved by simply scrolling to the desired property and entering a value via the keyboard. In cases where the property has a value that is a member of a set, such as the colour of a form, a drop-down list replaces the edit field and the user simply selects a member from the displayed set of values.

The object inspector has become a common-place item in today's applications that employ an object-based development environment, not just for the reason that Microsoft invented it when Visual Basic was first released and so everyone else should copy it, but because it seamlessly fits in with the visual object-based scheme that is the backbone of the main application. The art of interface design is knowing how to convey the essence and supported interactions of a program in a manner that appears to be both efficient and intuitive, and the fact that this solution has been widely adopted leads us to conclude that it should not be ignored. Both the optical analysis program and those applications mentioned above share the same component centred philosophy, and so it seems reasonable that the IRIS program should be adapted to provide a similar editor, which we shall call the *Lens Editor*.

Figure 5.5 shows the equivalent editor for the IRIS program. Note that it maintains the outward appearance of its ancestor, with the same two-column spreadsheet editor. The lens editor is activated when a lens is dragged from a system form and dropped onto the editor. The editor will inspect the object that is dropped by employing RTTI, and if

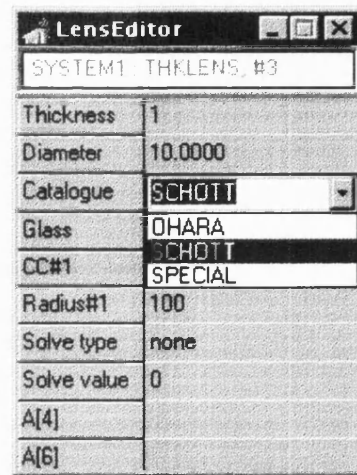


Figure 5.5: The IRIS Lens Editor Inspector

it is found to be a descendant of `TLens` then it is accepted as a valid object, otherwise it is rejected and no change to the editor is forthcoming.

How does the editor know what properties of the `TLens` descendant to show? The answer to this question lies in a method originating in the `TLens` class and is defined by:

```
procedure GetPropertyList(lst:TStringList);virtual;
```

When the lens editor has ascertained that the object dropped onto it is a `TLens` descendant, a call is made to the `GetPropertyList` of the object, supplying a `TStringList`⁹ as a parameter. The returned list supplies information on every property of the lens, including the property name, type of property, and if a set, then the values that comprise the set. As a property on the lens editor is selected, a routine determines these parameters for the particular property, and then implements a property editor in the right-hand column. This procedure is repeated as each row becomes active. In the case of the lens editor shown in Figure 5.5, we see by the title of the form that a thick lens has been selected from Button #3 of System 1. The active property is **Catalogue**, meaning the glass catalogue from which the glass comprising the thick lens is to be selected. Since this is a set of values, then the string that relates to each value is displayed in a drop-down list box, hence the values [Ohara,Schott,Special] that appear in the list box.

Unlike those editors supported by Delphi and Visual Basic, IRIS has been designed to work with more than one editor active on the screen at any one time. The reasons why this should be so are two-fold:

⁹`TStringList` is a container class for general objects and Pascal-style strings.

1. Lens designers often need to set some properties of a lens to be identical with corresponding properties of another lens. This is facilitated if the properties of both lenses are displayed simultaneously on separate editors;
2. It was anticipated during the design of IRIS that multiple editors would permit links to be made between similar properties of different lenses. For example, it is common practice among lens designers to link the shape of one surface to that of another surface of a different lens. By employing drag-and-drop techniques it would be possible create this link in a very simple manner.

The code that would support Item 2 has, due to insufficient time, not been included in the current version of IRIS.

5.6 Support Modules

The mathematical equations normally associated with raytracing are usually derived in vector form and then reduced to a corresponding set of equations in scalar form, usually as a means of increasing the numerical precision of the calculation. Occasionally when attempting to arrive at a solution for some other related problem it may not be possible to arrive at a satisfactory scalar form and so we must proceed with the original vector form of the solution. Taking this latter course presumes that the appropriate software tools are available, in particular tools that enable mathematical operations to be undertaken on both vector and array quantities alike.

The C/C++ language has, over the years, been the beneficiary of many support modules that have been developed in order to enhance the capabilities of the language, so much so that every commercial implementation of C/C++ comes with modules (header files) that allow vector and array classes to be created. In addition, due to its ability to overload operators, C++ provides a simple and intuitive way of utilising these classes in a meaningful way. For instance, if we assume that **a** and **b** are instantiated¹⁰ vectors then the sum of the two is given by (**a+b**), i. e. the '+' operator has been overloaded to enable it to function not just with simple scalars, but also vector quantities.

The Pascal language does not provide a capability for operator overloading and

¹⁰Instantiate *verb* to create an instance (object) of a class

so, in general, does not offer any useful facilities for manipulating both vector and array quantities, other than the normal static types, i. e. types that are not classes. On the other hand, the Object Pascal implementation included with Delphi does offer a limited form of operator overloading that may be used to good effect in the creation of general non-scalar modules. Object Pascal classes are able to define array-like index properties that simulate the *access* and *assignment* of indexed classes. Access refers to obtaining the value of a indexed reference, while assignment involves the assigning of a value to an indexed reference, thus:

`2 * a[j]` is an example of an indexed access specifier, while...

`a[j] := 2` is an example of an indexed assignment specifier.

Unfortunately, Object Pascal does not allow any further form of operator overloading, and so we must rely upon simple function and procedure-type methods for the implementation of all other operations.

The IRIS program has made extensive use of the **Vectors** and **Arrays** modules, both designed by the author, in the various ray-vector manipulation procedures that were required to implement some of the special features that the application offered. The salient features of the interface and a small portion of the implementation to the **TVector3** class are shown below:

type

TVec3 = array[1..3] of double;

TVector3 = class

private

vec3:TVec3

protected

fnumber:word;

function GetVal(j:word):double;virtual;

procedure SetVal(j:word;newval:double);virtual;

function GetLength:extended;virtual;

```

    procedure SetLength(newval:extended);virtual;
public
    constructor create;
    constructor InitCreate(a,b,c:double);
    procedure Init(a,b,c:double);virtual;
    property value[j:word]:double read GetVal write SetVal;default;
    property length:extended read GetLength write SetLength;
    function plus_(v:TVector3):TVector3;virtual;
    procedure plus(v:TVector3);
    procedure cross(v:TVector3);
    function cross_(v:TVector3):TVector3;virtual;
    property x:double read vec[1] write vec[1];
    property y:double read vec[2] write vec[2];
    property z:double read vec[3] write vec[3];
    property l:double read vec[1] write vec[1];
    property m:double read vec[2] write vec[2];
    property n:double read vec[3] write vec[3];
    procedure axisrotate(axis:TAxis; angle:extended);
end;

function NextVector:TVector3;
var
    {define the unit vectors}
    x_vector, y_vector, z_vector : TVector3;
    (*****)
implementation
uses Matrices;
const
    MaxVec = 15;
type
    EVectorErr = class(Exception);
var
    vec      :array[1..MaxVec] of TVector3;
    count    :integer;

```

```

ExitSave :pointer;

function NextVector:TVector3;
{returns vec[count] as a temporary null-vector}
begin
    result := vec[count];
    result.init(0,0,0);
    {cycle 'count' to next available vector}
    count := (count mod MaxVec);
    inc(count);
end;

```

The first point to note is that the Vector class defined above is not an implementation of a generalised n -dimensional vector, but is in fact simply a 3-dimensional vector. While the former is not too difficult to design, the specialised 3-dimensional case was chosen for reasons of efficiency and speed.

The TVector3 class has at its heart an array `vec3`, of type TVec3 which is defined as `array[1..3] of double`, i.e. a simple indexed variable having three values of type `double`, where each element may be accessed or assigned to. In actual fact this is not strictly true, since `vec3` is to be solely a container and all procedures and functions appertaining to this class will mostly utilise the container indirectly through the use of properties.

The creation of a vector may be undertaken through one of the above constructors. For instance, `create`, as in `a := TVector3.create` will instantiate a vector called `a` and set its elements to a value of 0.00. Alternatively, `InitCreate` may be used if the vector is to be initialised to a particular set of values, as in `a := InitCreate(1,1,1)`. The property definitions that follow enable the vector elements to be referenced either as direction cosines that use the notation (l, m, n) , such as `a.m` and `a.n`, or as Cartesian coordinates, `a.x`, `a.y` and `a.z`, both using the familiar dot notation. The length of a vector is also accessed as a property value with supporting `GetLength` and `SetLength` methods, not listed here. The `length` property allows us to obtain the length of a vector by `a.length`, which will return a scalar value, or to set the length of a vector, `a.length := 2`.

The Vectors module also implements a sophisticated method for implementing intermediate return values. What does this mean? Well, consider the expression $(x * (y^2 + z^2))$. The evaluation of this expression involves the calculation of y^2 and then z^2 , storing these as temporary values, summing the two and storing again, and finally multiplying this value by x . In total, the evaluation of this expression requires three temporary values to be stored. By the time that the expression has been completed all the temporary, or intermediate, values will have been destroyed and the memory reclaimed. This procedure is followed when the variables are of a simple type, such as a **real** or **ordinal** type, but if instead they are dynamic objects, such as vectors, then how can intermediate dynamic objects be created and then automatically destroyed when no longer required?

The C++ language implements the auto-destruction of objects when they are no longer in scope, but this is not the case for Object Pascal, where the responsibility for the creation and destruction of objects lies solely with the programmer. The solution to this problem, as devised by the author, relies upon the array:

```
vec : array[1..MaxVec] of TVector3;
```

which is declared in the implementation section of the Vectors module. As mentioned in a previous chapter, a Pascal module may utilise an **initialization** routine prior to loading, and a corresponding exit routine prior to the unloading in memory of the module. In this case the **initialization** routine creates the vector objects comprising the **vec** array, where the maximum number created, **MaxVec**, is any suitably large number (15 in this case). These vectors will represent our intermediate results and the number of intermediate vectors will determine the depth of the result stack. In other words, we can have upto fifteen pending results in any one or more expressions. This is usually sufficient for most needs. Let us consider the example of a vector triple product: $\vec{r} = \vec{a} \times (\vec{b} \times \vec{c})$. The Vectors module supplies two methods that will handle the cross product,

1. **procedure cross(v:TVector3);**
2. **function cross_(v:TVector3):TVector3;**

and the applicability of each is given by:

1. `b.cross(c);`
2. `vec[j] := b.cross_(c);`

where the result of the first cross product is stored in `b`, while the result of the second cross product is assigned to one of the temporary vectors. Which of the temporary vectors to employ is the purpose of the `NextVector` function, which provides as its output a pointer to the next available temporary vector in the `vec` array. The vector triple product may now be implemented by the following code:

```
r := a.cross_(b.cross_(c));
```

We note here that only two temporary vectors were created in this expression, and both were discarded once the assignment to the `r` vector was made. When, finally, the application is terminated, the Vectors module will execute the special exit procedure that will destroy the temporary vectors and allow the operating system to reclaim the allocated memory.

The one other supporting module worth mentioning in this section is the Arrays module, but in many respects it is very similar in principle to the design of the Vectors module, implementing as it does the same temporary 3-dimensional matrix array that allows for intermediate values to be created.

Chapter 6

Lens Internals

One of the first problems that the author encountered when attempting to describe the component based model was whether the lens components should have defined within themselves a description of the refractive media that would encapsulate the entity, much in the same way as a real lens is always surrounded by air. The surface based model, as an example, consistently binds a geometrical surface description with a glass identifier, either in terms of a glass-name or a refractive index. If this approach were to be adopted then a consistent mechanism would require to be developed that would allow lens components to be freely moved from one position to another. In such a circumstance it would be unreasonable to maintain the specific reference to a refractive medium contained within the component. The only logical course forward would be to assume that the lens component would exist without prior knowledge of its bounding media, and that at some point prior to raytracing all glass references would be resolved. This conclusion necessitates the existence of lens components that are purely refractive in nature, serving only to fill the implicit spaces that would in actuality comprise the immersing medium. A lens system is normally viewed as one or more lenses, be they simple or complex, that out of necessity must be immersed in a refractive medium. An alternative description, and one that is followed closely by the raytracing algorithm, is that a lens system is a series of refractive media that interface with one another across distinct boundaries. It is this description that is at the heart of the component model.

The IRIS program was designed in such a way as to hide the complexity that supported the concept of a lens system. It achieves this by breaking down a general lens

system into its natural component parts and then presenting these to the user in a much more user-friendly and idealised manner. Each optical or functional element of a system is represented by an icon, where each icon is in effect the surfacing of a particular property of the whole ensemble of properties that it supports. We have already seen in the previous chapter that the `TLensButton` class is actually derived from a `TSpeedButton`, which is why a lens element appears as a special type of Windows button, and it was pointed out that the specific lens properties of `TLensButton` were embodied in a property called `TheLens`. Similarly, as in the case of an onion, the `TLens` class is also the *home* for various surface and glass properties. This chapter will provide the reader with a detailed view of those properties that constitute the actual lens and which the lens designer or engineer has in mind when synthesising a lens system.

6.1 Glasses

Glasses, in general, encompass all media that are used to construct real lenses, and also the media that separate real lenses. From the theoretical point of view, one glass description is very similar to another glass, in that all perform the same function and all may be described by the same form of refractive power and dispersion characteristics. The real world, however, introduces non-essential complexities that originate from the various models that have been proposed which purport to describe in algorithmic terms the dispersion equation for a real glass. As yet, no uniform glass description has been adopted by the world's glass manufacturers, and so it behoves the application developer to treat each model separately.

The two principal models [11, Chapter 9] are the Sellmeier and Schott formulae:

$$\text{Sellmeier... } n^2(\lambda) - 1 = \frac{B_1\lambda^2}{\lambda^2 - C_1} + \frac{B_2\lambda^2}{\lambda^2 - C_2} + \frac{B_3\lambda^2}{\lambda^2 - C_3}$$

$$\text{Schott... } n^2 = A_0 + A_1\lambda^2 + \frac{A_2}{\lambda^2} + \frac{A_3}{\lambda^3} + \frac{A_4}{\lambda^4} + \frac{A_5}{\lambda^5}.$$

While the Sellmeier formula is based on a physical model of a dielectric medium (see [12], Chapter 2.3), in practice the Schott formula is found to give better results¹ for a wider

¹The current Schott catalogue has now adopted the Sellmeier formula

spectral interval (near ultraviolet to mid-infrared), achieving an interpolation accuracy of better than 0.00001 in refractive index.

Each of the major glass manufacturers usually chooses, for whatever reason, one of these formulae to represent the dispersion characteristics of any glass that is produced, and provides a catalogue that details the coefficient values for every glass that they supply. Fortunately, the same manufacturers also provide, on request, this same data in an electronic file format, usually a dBase or text file. Since there may be upto several hundred glasses in any one catalogue, the electronic form is undoubtedly a great time saver when it is required to include such data in third-party software.

The **TGlass** class is relatively simple when compared to previous classes discussed since its main function will be to return a refractive index value for any given wavelength. A listing of the class interface section is given below:

type

```
TCatalogs = (SCHOTT, OHARA, SPECIAL, NONE);
GlassCatalogs = set of TCatalogs;
TGlass = class
private
    coeffs:array[0..6] of double;
public
    name:string;
    catalog:TCatalogs;
    function    index(wvl:double):extended;virtual;
    procedure    assign(g:TGlass);virtual;
    constructor create(catalog:TCatalogs; aname:string);
    constructor createair;
    procedure    SetGlass(catalog:TCatalogs; aname:string);
    function    GetGlassname(j:integer):string;
    procedure    SetGlassname(j:integer;newval:string);
    function    GetCatalog(j:integer):string;
    procedure    SetCatalog(j:integer;newval:string);
    procedure    LoadCatalogNames(s:TComboBox);
    procedure    LoadGlassNames(s:TComboBox);
```

end;

The **create** method requires the glass catalogue and glass name to be passed as initialisation parameters: the glass catalogue is an enumerated parameter having values given by the type **TCatalogs** (American spelling for brevity), while the glass name is of type **string** and may include any name that exists within the specified catalogue, such as 'BK7'. The action of the **create** constructor is to search the relevant catalogue file and load the array **coeffs** with the correct dispersion constants.

The **index** method is used to obtain the refractive index of the glass when a wavelength is passed in the **wvl** parameter, i.e. **glass.index(0.589)** will return the refractive index of the glass at a wavelength of $0.589\mu\text{m}$. During the evaluation process the glass catalogue will be used as a pointer to the appropriate dispersion equation, so that the coefficients read from the database table will be correctly recognised as either Sellmeier or Schott coefficients.

6.2 Surfaces

Historically, the first optical surfaces were spherical in form since these were intrinsically easier to generate and test without specialised techniques. In this instance a surface is an abstract form devoid of any technical specification, but instead the abstract surface is developed as an ancestor for all real surfaces that are ultimately derived from this type.

6.2.1 The Abstract Surface

The general surface is unlike other components in that not only is it provided in component form and may be included within a system, but it is also the main class that is used to derive other more complex components, such as prisms and lenses. In its singular form, the surface does not **own** any refractive media that exist on either side of the boundary, but instead it **borrow**s the media that are provided by adjacent components. In contrast, a singlet lens component will **own** the internal medium but will still require to borrow adjacent glasses in order to fully implement itself. The following listing presents an edited view of those properties and methods that **TSurface** implements.

```

TSurface = class
protected
  norm :TVector3;
  fpoint:TVector3;
  glass1,glass2:TGlass;
  faperture:double; {aperture radius}
  q:extended; {ndx1/ndx2}
  function GetGlass(j:integer):TGlass;
  procedure SetGlass(j:integer;const newval:TGlass);
  procedure SetPoint(newval:TVector3);
public
  reflect:boolean;
  lensowner:TObject;
  formowner:TObject;f;
  constructor create(lensowner:TObject);virtual;
  destructor destroy;override;
  function Normal(point:TVector3;var failed:boolean):TVector3;
    virtual;abstract;
  procedure Transfer(rays:TObject);virtual;
  procedure Refract(rays:TObject);virtual;
  procedure Transfract(rays:TObject);virtual;
  procedure PxTransfer(rays:TPxRaySet);virtual;abstract;
  procedure PxTransfract(rays:TPxRaySet);virtual;abstract;
  function InsideAperture(ray:TRay):boolean;virtual;abstract;
  procedure PxRefract(rays:TPxRaySet);virtual;abstract;
  procedure reverse;virtual;
  procedure assign(s:TSurface);virtual;
  procedure initglasses(g1,g2:TGlass);virtual;
  function  property glass[j:integer]:TGlass
    read GetGlass write SetGlass;
  property aperture:double read faperture write faperture;

end;

```

The abstract surface defined above is bounded by two refractive media, **glass1** and **glass2**. When a surface is implemented as a component, part of the initialisation procedure that is performed by the surface is the method **initglasses** that takes two glass references, **g1** and **g2**. The code for this procedure is shown below:

```
procedure TSurface.initglasses(g1,g2:TGlass);
begin
    glass1 := g1;
    glass2 := g2;
end;
```

...where it becomes apparent that the glasses that are passed as parameters are only references, and that the apparent assignments, such as **glass1 := g1**, are simply reference assignments. What this means is that the surface never creates its own **TGlass** object, but instead the **glass1** and **glass2** properties are simply employed as *pointers* to neighbouring media that are not of its own creation, and until the surface is initialised the glass pointers are left *dangling* with a value of **nil**. The following snippet of code is from the component **initialise** method of the **TThkLens**:

```
surf1.glass[1] := previous.medium;
surf1.glass[2] := glass;
surf2.glass[1] := glass;
surf2.glass[2] := next.medium;
```

The component **initialise** method is called prior to any sequence of raytracing, and is necessary so that all surfaces may set their interfaces correctly. In the above case of a thick lens, each of the two surfaces sets its own interface condition. Note that the medium of the thick lens, **glass**, is directly referenced by both surfaces, whereas the outer media of the lens are obtained through the call to **previous.medium** or **next.medium**. The notation **previous** and **next**, introduced in the discussion of **TLens** methods and properties, comes into its own in this context.

The **TSurface** class also contains several methods that enable the processing of incident rays, the most important of these being **Refract** and **Transfer**. Both of these methods accept **rays** as a parameter, but note that **rays** is of type **TObject**. Recall from

earlier discussions of object oriented programming in Object Pascal that **TObject** is the ancestor of all defined classes, and it is used here because the underlying structure of **rays** comprises several different ray-sets, including paraxial and finite rays. The **Refract** and **Transfer** methods each interrogate the **rays** object, using run-time type information (RTTI), to determine the structure and entry points to each one of these ray-sets. When this has been accomplished the individual ray-sets are passed to other procedures that will correctly implement the method for that type of ray; this is the purpose of such methods as **PxRefract**, which only refracts paraxial rays using the standard paraxial raytrace formulae.

Another method that we shall refer to in the next chapter is called **Normal**, and is supplied with a vector **point** and a boolean **failed**. This is an abstract and virtual class which means that it may only be used in a descendant class that overrides the method. In operation, the descendant surface class will define the method in such a way that it will return a vector type that corresponds to the normal of the surface at the point given by **point**, where the latter is normally supplied by the positional data set of a finite ray (see Chapter 7.2). If **point** is invalid and does not correspond to an actual point on the surface then the **failed** parameter will be set to **true**, otherwise it will be set to **false**.

6.2.2 The Plane Surface

Descended directly from **TSurface**, the plane surface implements, or overrides, those abstract methods that were first introduced in the ancestor class, as the following extract of **TPlane** illustrates.

```
TPlane = class(TSurface)
private
    ftheta:double; {surface tilt}
    fphi:double;  {surface rotation - spherical polar coords}
    fwidth:double;
    {note : if fwidth = 0 then faperture corresponds to the aperture}
    {radius, otherwise the half-height, and fwidth is the half-width}
public
    constructor create(Insowner:TObject);override;
```

```

destructor destroy;override;
procedure initangles(const theta_,phi_:double);
function InsideAperture(ray:TRay):boolean;override;
procedure reverse;override;
procedure RayTransfer(ray:TRay);override;
procedure RayTransfract(ray:TRay);override;
procedure RayRefract(ray:TRay);override;
procedure PxTransfer(rays:TPxRaySet);override;
procedure PxTransfract(rays:TPxRaySet);override;
procedure PxRefract(rays:TPxRaySet);override;
end;

```

There are several points to notice in the above listing. Firstly, the perimeter of the plane surface may be defined as either circular or rectangular, depending upon the values of `fwidth` and `fheight`. Similarly, the plane surface may, in its default state, be considered as being orthogonal to the local z -axis, or alternatively may have a general tilt and rotation associated with the surface. The latter is implemented by a call to `InitAngles` with values of `theta_` and `phi_` passed as polar angles (θ, ϕ) of the surface normal. Since `TPlane` has been allowed to have a surface perimeter that is not circular, it will be important to know when a ray falls outside the defined surface, or not. This requirement is handled by the method `IsInsideAperture`, which accepts a `ray` parameter of type `TRay`. The ray in question will have been transferred to the surface prior to the call, and the result will be of type `boolean`, indicating that the result is either `true` or `false`.

Secondly, the methods that relate to the processing of rays have been overridden. The ancestor class defined these methods as `abstract` since they could not realistically be considered for a general surface description, but `TPlane` is now able to implement its own specific routines. The following two sections of code show how the algorithms that relate to the transfer and refraction of finite rays are implemented:

Transfer...

```

procedure TPlane.RayTransfer(ray:TRay);
var
    delta:extended;
    cosI:extended;

```



```

j:integer;
IgnoreAps:boolean;
begin
  IgnoreAps := TLensForm(formowner).IgnoreAps;
  with ray do
    begin
      if not RayCanProceed(ray) then exit;
      cosI := (norm.x*l + norm.y*m + norm.z*n);
      delta := -(norm.x * x + norm.y * y) / cosI;
      x := x + l * delta;
      y := y + m * delta;
      z := n * delta;
      fpoint.assign(ray.pasn);
      If not(IgnoreAps) and not(InsideAperture(ray))
      then failmode := failmode + [missed_aperture];
    end;
  end;
end;

```

and Refraction...

```

procedure TPlane.RayRefract(ray:TRay);
var
  cosI,cosId:extended;
  ndx1,ndx2,q,k,c,s:extended;
  j:integer;
  inray:TRay;
  dev:extended;
  rotvec:TVector3;
  rotmat:TMatrix3;
begin
  if not RayCanProceed(ray) then exit;
  inray := TRay.create;
  inray.assign(ray);
  try

```

```

ndx1 := glass1.index(ray.wvl);
if reflect then ndx2 := -ndx1
else ndx2 := glass2.index(ray.wvl);
with ray do
begin
  q := ndx1/ndx2;
  cosI := (norm.x*l + norm.y*m + norm.z*n);
  s := 1 - q*q*(1 - cosI*cosI);
  if s <= 0.0 then
  begin
    failmode := failmode + [internal_reflection];
    exit;
  end;
  cosId := sqrt(s);
  k := ndx2*cosId - ndx1*cosI;
  c := k / ndx2;
  l := (norm.x * c) + (l * q);
  m := (norm.y * c) + (m * q);
  n := (norm.z * c) + (n * q);
  {if the polarisation vector is not nil then}
  if polvec <> nil then
  begin
1    rotvec := TVector3.create;
1    rotmat := TMatrix3.create;
    {determine the ray deviation angle}
2    dev := vector.inclangle(inray.vector);
    {assign the cross-product of in/out rays to rotvec}
3    rotvec.assign(vector.cross_(inray.vector));
    {make rotmat a rotation matrix wrt rotvec and angle dev}
4    rotmat.rotvecmat(rotvec,dev);
    {multiply polvec by rotmat}
5    rotmat.vecmult(polvec);
    rotvec.free;
    rotmat.free;

```

```

end;
{if this is a reflecting surface then }
if reflect then
begin
  ForwardReflection;
  if polvec <> nil then
    begin
      polvec.reflection(x_y);
    end;
  end;
end;
finally
  inray.free;
end;
end;

```

The **RayRefract** method is complex in several ways, but most particularly because it demonstrates how polarised ray vectors² are handled. The first action of the procedure is to set the local variables **ndx1** and **ndx2** to the value of the refractive indices on both sides of the surface, making due account of whether the surface is a reflecting surface or not, where the usual convention in defining a reflecting surface is to set **ndx2** to the negative value of **ndx1**. Subsequent lines of code initiate the raytrace algorithm for refraction at a tilted surface, whereupon the ray-vector **ray** takes on the new value. The final part of the routine is involved in the processing of the polarisation vector, if one is present (see Chapter 7).

Note: one of the early objectives of this research programme, before it adopted its current form, was to provide a means for tracing polarised rays through a general optical system. System throughput, particularly in diffractive optical systems such as an astronomical spectrograph, is very much dependent upon the state of polarisation of optical rays in every space of the system. It was for this reason that polarisation vectors were originally

²According to the electromagnetic theory of radiation, a ray of light consists of an oscillating electric field and a magnetic field, mutually perpendicular to one other and propagating along the ray direction. A polarised ray is a ray in which the electric vector is confined to a single plane of vibration.

*included in the definition of **TRay** (see Chapter 9) and that some components, especially the plane surface, have been coded to correctly handle polarised rays.*

The problem of how to handle polarised ray vectors has been much simplified by the development of the **Vectors** and the **Arrays** classes, both of which allow complex operations involving both vectors and matrices to be undertaken in a straightforward manner. The sequence of operations is as follows:

1. If the ray contains a valid polarisation vector then create a vector **rotvec** and rotation matrix **rotmat**
2. Determine the angle of ray deviation following refraction, and assign this to **dev**
3. Determine the vector normal to the plane of incidence, and assign this to **rotvec**
4. Modify the value of **rotmat** such that it corresponds to a rotation matrix equivalent to a rotation from the input ray-vector to the output ray-vector
5. Finally, multiply the polarisation vector **polvec** by the rotation matrix **rotmat**
6. As part of the tidying-up process, release or **free** the original **rotvec** vector and **rotmat** rotation matrix.

6.2.3 The Conic Surface

The conic surface is extremely important in lens design and manufacture since it encompasses virtually all the non-planar surface types that the designer is likely to specify, the exception being the aspheric surface. Probably the most common conics in optics are the sphere, the parabola and the hyperbola. The general conic surface is defined by the following equation,

$$z = \frac{1}{2}c(x^2 + y^2 + \varepsilon z^2) \quad (6.1)$$

...where z is the sagitta of the surface for the coordinate pair (x, y) . It is easily seen that this equation represents a quadric of revolution about the z -axis, passing through the origin and having curvature c at the same point, and where the coefficient ε determines the conic form of the resulting surface, as follows:

$$\begin{aligned}
\varepsilon &> 1, && \text{prolate ellipsoid,} \\
\varepsilon &= 1, && \text{sphere,} \\
0 < \varepsilon &< 1, && \text{oblate ellipsoid,} \\
\varepsilon &= 0, && \text{paraboloid,} \\
\varepsilon &< 0, && \text{hyperboloid.}
\end{aligned}$$

The raytrace formulae that apply to the general conic surface are very similar to those we would use for a spherical surface, but with a minor correction for the aspheric sagittal term. This observation proves useful when coding the ray transfer equation, since the code and underlying logic is much simplified. As a result, the early development version of the IRIS program did not explicitly support non-spherical conics, although the surface editor did indeed provide the user with a selection of available conical forms, but all calculations were based upon a default base sphere.

In many respects the **TConic** surface is very similar to **TPlane**, where the only significant differences are to be found in the actual implementations of the raytrace algorithms. In one other respect, though, the **TConic** differs quite radically in that it makes available to the designer a surface curvature that is variable, that may be specified either by the designer at design-time, or alternatively, during active raytracing (run-time) where the curvature is resolved by what is normally referred to as a *solve*. Solves are a mechanism employed by designers that simplify the structure of an optical system. For instance, one of the more common solves will adjust the final surface curvature of a lens system so that the marginal paraxial ray emerges with a specified convergence angle. This solve is employed when the final image NA (numerical aperture) needs to be set to a predefined value. The normal procedure for achieving this result is to perform the paraxial raytrace and make the necessary adjustments to those surfaces that are involved in solves. Only when this process has been completed may real finite rays be propagated through the system.

In keeping with the philosophy underpinning much of this work, the implementation of the above scheme is both elegant and simple. We start with a shortened listing of the **TConic** class that highlights those properties pertinent to the above discussion.

```

Type
AsphericCoeffs = (A4,A6,A8,A10);
AsphericArray = array[AsphericCoeffs] of double;
ConicTypes = (SPHERE,PARABOLA,OBL_ELLIPSE,PRO_ELLIPSE,HYPERBOLA);

TConic = class(TSurface)
protected
    coeffs:AsphericArray; {aspheric coefficients}
    fcv :double; {curvature}
    fcc:double; {conic constant}
    solvetype:byte; {0-no solve; 1-PR solve; 2-Marg solve}
    solvevalue:double;
public
    constructor create(lnsowner:TObject);override;
    destructor destroy;override;
    constructor initcreate(lnsowner:TObject;curv,conic:extended);virtual;
    function Normal(p:TVector3; var failed:boolean):TVector3;override;
    function Sagitta(h:double;var failed:boolean):double;override;
    procedure RayTransfer(ray:TRay);override;
    procedure RayTransfract(ray:TRay);override;
    procedure RayRefract(ray:TRay);override;
    procedure PxTransfer(rays:TPxRaySet);override;
    procedure PxTransfract(rays:TPxRaySet);override;
    procedure PxRefract(rays:TPxRaySet);override;
    procedure Solve_U(ray:TPxRay);
end;

```

The above listing commences with a description of the various surface types that will determine the general form of the surface. **ConicTypes** is a set of constants that represent the fundamental forms of the general conic surface, while **AsphericCoeffs** and **AsphericArray** are used to describe the precise aspheric departure from the chosen conic, where the departure is given by the following:

$$\Delta z = A_4 h^4 + A_6 h^6 + A_8 h^8 + A_{10} h^{10} \quad (6.2)$$

... where Δz is the sagittal departure from the conic surface at height h , and the coefficients are given by the constants A_4 to A_{10} . Most optical aspheric surfaces are described by the above polynomial, although more precise descriptions will require higher order coefficients A_{12} , A_{14} , etc., although A_{12} is usually the highest coefficient for most optical design tasks. The **TConic** class is then defined, and unexpectedly is shown to be a descendant of **TSurface**. Most or all of the methods inherited from **TSurface** have been overridden, but note particularly the **protected** section that includes several new fields which together describe the surface asphericity and the solve conditions, if any, that currently apply. The **solvetype** variable may hold one of three values, which correspond to either *no solve*, a *PR solve*, or a *Marg solve*. The first is self explanatory, while the latter two refer to angle solves with either the principal ray or the marginal ray. If the **solvetype** is non-zero then the **solvevalue** field indicates the value required for the convergence angle of the particular ray. **TConic** implements the solve at the instant the appropriate paraxial ray is received by the surface for ray-processing, i. e. refraction. How the surface recognises a particular paraxial ray amongst all the other rays that it will receive is the subject of Chapter 7, "Optical Ray Implementation".

Chapter 7

Optical Ray Implementation

It is true to say that without rays we would be none the wiser on the subject of lens design, for it is the ray itself that carries the information that we process and interpret in order to ascertain the state of optical correction of a lens system. Though of an essentially complex nature¹, in most situations light may be suitably approximated by a simple vector. Whilst raytracing formulae are initially described in vector form, for example the refraction equation:

$$n'(r' \wedge n) = n(r \wedge n), \quad (7.1)$$

...it is only when the formulae need to be coded that these vector forms are dispensed with in favour of a scalar description. Thus the above vector equation is usually reduced to the more common form shown below.

$$\begin{aligned} n'L' - nL &= k\alpha \\ n'M' - nM &= k\beta \\ n'N' - nN &= k\gamma \end{aligned} \quad (7.2)$$

where k is called the *generalised power* and is a function of the angle of incidence at the surface,

$$k = n' \cos I' - n \cos I. \quad (7.3)$$

Other scalar formulae have been devised[13] that provide maximum information output whilst minimising the number of mathematical operations required to complete the calculation. Obviously such procedures were very specialised, as in the tracing of a marginal

¹(see [12], Chapter 8)

ray, but were considered necessary in that period prior to the development of the modern computer. One of the more important features of the early pioneering work into the design of raytrace algorithms was the need to maximise accuracy of calculation. This was considered an essential requirement in those times when most calculations were undertaken using 9-figure, or thereabouts, logarithmic and trigonometric tables. Such care in the design of modern raytrace algorithms is probably absent now that we have regular numbers that may be represented by upto 19 significant digits (double precision).

The major reasons for adopting the above approach were twofold:

1. vectors could not be adequately represented by the computing machines available at the time,
2. scalar forms of the raytrace equations could be designed to be more efficient in terms of CPU time.

Though the first is no longer applicable, with the advent of parallel computers and array processors, the second is as valid now as it has always been. Be that as it may, the author has implemented a ray class that exhibits both the scalar and the vectorial nature of the optical ray, including a ray class that has the capability to resemble a polarised ray, i. e. the ray has an associated polarisation vector that is maintained perpendicular to the ray direction.

7.1 Paraxial Rays

The paraxial ray owes its existence to the study of Gaussian optics, that domain of a rotationally symmetrical optical system where all ray angles, with respect to the optical axis, are vanishingly small and which are able to be numerically represented by first order approximations; hence the alternative name of *first order* optics. Valid rays within the Gaussian domain may be specified by a single coordinate pair (h, u) , where h is the height of the ray-surface intersection from the optical axis, and u is the convergence angle. The usefulness of paraxial rays and their own form of raytracing algorithm is diverse, and includes:

- the determination of the Gaussian properties of an optical system, such as the position of the nodes and principal planes, and quantities such as focal length;
- the calculation of third (Seidel) and higher order[14] aberration coefficients;

A complete Gaussian description of an optical system may be arrived at by the tracing of a single paraxial ray that connects a pair of axial conjugates. The technique depends upon a set of formulae developed by Ludwig Seidel in 1856, later referred to as the Seidel *difference formulae*. When paraxial raytracing is implemented in program code, the more usual approach to take is to use two paraxial rays, one referred to as the paraxial marginal ray, and the other as the paraxial principal ray. The former refers to a paraxial ray that starts at the axial position of the object and which also intersects the pupil/stop(iris) at the periphery, while the latter describes a paraxial ray that starts at the extreme edge of the object and then proceeds to intersect the stop at its very centre; see Figure 7.1. The class that encapsulates the paraxial ray is quite simple compared to

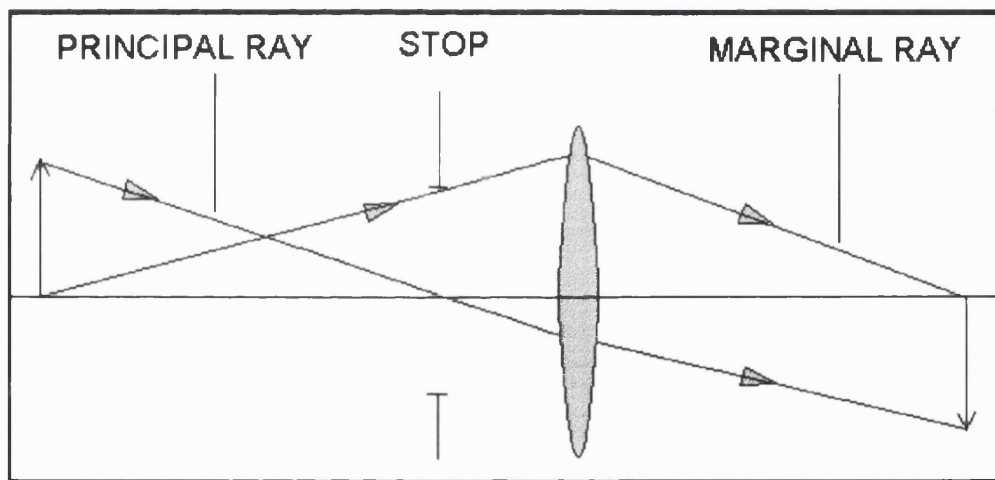


Figure 7.1: Paraxial Ray Types

the finite ray, and a listing of the class methods and properties is given below:

```
TPxRay = class
protected
    h1,u1,wvl1 : double;
public
    constructor create;virtual;
    constructor initcreate(const h1,u1,wvl1:double);virtual;
    procedure init(const h1,u1,wvl1:double);virtual;
```

```

procedure assign(ray:TPxRay);virtual;
function  datastr:string;virtual;
property h:double read h1 write h1;
property u:double read u1 write u1;
property wvl:double read wvl1 write wvl1;
end;

```

...where the first constructor will create a paraxial ray having default initial values, while the second constructor will perform the same function and also allow the initial values to be set programmatically. The three data fields `h1`, `u1`, `wvl1` contain the paraxial ray height, the convergence angle and wavelength (μm), and are accessible through their own specific property definitions, i. e. `a_ray.h`, `another_ray.u`.

7.2 Finite Rays

In most optical analyses the finite ray is probably the most used. Whereas the paraxial ray is an artificial mathematical construct that is limited in scope, the finite ray is as good a description of a real ray as we are ever going to get. Though it too is an artificial device, the results obtained from a finite raytrace may, using specific mathematical procedures, be transformed to provide information on the optical wavefront itself as it propagates from one lens space to another. Considering the wealth of information that the finite ray provides, it will not be too surprising to learn that the class that encapsulates the finite ray is considerably richer, as the listing below shows.

```

TRayFailure = (missed_aperture,missed_surface,parallel,
               TIR,illegal_gratingorder);
TRay = class
protected
    fposn      :TVector3;
    fdirn      :TVector3;
    fpolvec    :TVector3;
    procedure SetVector(newval:TVector3);
    function  GetDirn(index:integer):double;
    procedure SetDirn(index:integer; newval:double);

```

```

function GetCoord(index:integer):double;
procedure SetCoord(index:integer; newval:double);
procedure SetPosn(newval:TVector3);
procedure SetPolVec(newval:TVector3);
function GetHeight:double;
public
  FailMode : set of TRayFailure;
  wvl      : double;
  constructor create;virtual;
  destructor destroy;override;
  procedure initposn(a,b,c:double);
  procedure initdirn(a,b,c:double);
  procedure initpolvec(len,angle:double);
  function  tangential(norm:TVector3):double;
  function  sagittal(norm:TVector3):double;
  procedure RotateBy(mat:TMatrix3;pol:boolean);
  procedure MoveBy(x,y,z:double);
  procedure LinearMove(inc:double);
  procedure reverse;
  procedure ForwardReflection;
  procedure assign(ray:TRay);virtual;
  function datastr:string;virtual;
  property x:double index 1 read GetCoord write SetCoord;
  property y:double index 2 read GetCoord write SetCoord;
  property z:double index 3 read GetCoord write SetCoord;
  property l:double index 1 read GetDirn write SetDirn;
  property m:double index 2 read GetDirn write SetDirn;
  property n:double index 3 read GetDirn write SetDirn;
  property vector:TVector3 read fdirn write SetVector;
  property dirn:TVector3 read fdirn write SetVector;
  property posn:TVector3 read fposn write fPosn;
  property polvec:TVector3 read fpolvec write SetPolVec;
  property height:double read GetHeight;
end;

```

The first two lines of the listing supply an enumerated type definition, called **TRayFailure**, which defines several constants that correspond to the various failure modes that an active finite ray is likely to experience. This is introduced in order to provide a finite ray with a **FailMode** tag, a field that enables the failmode history of a particular ray to be discerned. Naturally, a ray that successfully completes a raytrace will have an empty **Failmode** field, i. e. a null set equivalent to `[]`, while a ray that intersects a surface beyond the physical aperture and which is subsequently internally reflected will have a **FailMode** of `[missed_aperture, TIR]`.

In stark contrast to the paraxial ray, the finite ray requires much more data to describe it. A finite ray is characterised, apart from the wavelength `wvl`, by two sets of data; the positional coordinates (x, y, z) and the directional cosines (l, m, n) . Both sets of quantities may be individually specified through the several property methods that are supplied with the class, i. e. `a_ray.x` and `another_ray.m`. The container class for both these sets of data is, unsurprisingly, a vector class: `fposn` for the positional coordinates, and `fdirn` for the directional coordinates. The distinct advantage that these containers provide is that both data sets may be manipulated by vector methods supplied by the `Vectors` and `Arrays` classes that the author has implemented for just this purpose. For example, the vector that is perpendicular to the plane containing the incident ray and the surface normal at the point of ray incidence on a surface, `perp`, is given by the following procedural fragment:

```
var
  failed : boolean;
  norm   : TVector3;
  perp   : TVector3;
begin
  norm := a_surface.normal(a_ray.posn, failed);
  if not failed then begin
    perp := a_ray.vector.cross_(norm);
    etc.
    etc.
  end;
end;
```

The *vector* property of a finite ray, `a_ray.vector`, enables the ray to be treated as a true vector quantity, so allowing it to be manipulated by those vector and matrix methods already developed by the author. This becomes particularly important when attempting to devise new raytrace algorithms for unusual surface types. It is often the case that the reduction of an algorithm proceeds from a vectorial form to a scalar form, and in so doing it becomes highly susceptible to errors being propagated through to the final set of results. The possibility of such a mishap may be avoided if the raytrace is implemented by both forms of the transfer algorithm and the final results compared for equality.

The vector property of a finite ray, as implemented in the above definition of `TRay`, also enables several other transformations to be undertaken with ease. For instance, the IRIS program allows for several localised axes to be declared by the user during the system design phase, although only one is ever allowed in any optical space. The purpose of such a feature is to simplify the user's task when complex non-axisymmetric systems are to be synthesised. For instance, a simple spectrometer having a prism that is preceded by a collimator and then followed by a decollimator is difficult to model both at and following the prism itself, for the reason that the optical axis is forever changing. Not only that, but the optical axis within the prism needs to be calculated by the designer so that the following surfaces are correctly positioned. This seems like an unnecessary burden for the designer, and so the author has implemented a scheme (see Chapter 8.1 for a more detailed description) whereby each new optical axis may be specified as a real ray direction at a particular wavelength², and the necessary transformations for each ordinary ray are undertaken in real-time during the raytrace. The principal procedure in this scheme is

```
procedure RotateBy(mat:TMatrix3;pol:boolean);
```

`mat` is a rotation matrix that has been derived from the relative directions of the previous optical axis and the new axis. When implemented for each ray in the ray-bundle, the `RotateBy` method will realign all rays so that their directions and positions will conform to the new axis as given by the direction and position of the reference ray.

The `TRay` class is also the repository for a polarisation vector, which may be initialised after the finite ray has been created by employing the following method:

```
procedure initpolvec(len,angle:double);
```

... where `len` is the length of the vector and is analogous to the intensity, and `angle` is the

²In the parlance of the IRIS program, this ray is referred to as a Reference Ray

angle of the vector in degrees with respect to the y-axis in the x-y plane. The initialisation procedure ensures that the polarisation vector is created orthogonal to the direction of the host ray, while subsequent ray transformations that may occur at surface boundaries during raytracing will always correctly maintain this relationship.

Before we move onto the next section, one other worthy feature of the `TRay` class is its ability to physically reverse direction. In Chapter 8.7 we shall have the opportunity to discuss at length the specialised component that permits us to have optical components that operate in double-pass mode, similar to the front surface of a mangin mirror. This mode makes extensive use of the `TRay` method called `ForwardReflection`. When a reflective surface is followed by a double-pass component it will cause all rays to physically reverse direction, which is not ordinarily the case for surfaces that are not participating in a double-pass. The procedure, or method, once reduced to vector and positional transformations, is relatively simple:

```
procedure TRay.ForwardReflection;  
begin  
    reverse;  
    posn.reflection(x_y);  
    dirn.reflection(x_y);  
end;
```

Again, note how the use of the vector properties of both position and direction of the ray are used to full effect. The `reflection` method applied to both of these vectors requires a constant parameter to be passed which specifies the plane about which the reflection is to be undertaken, in this case the x-y plane.

7.3 Ray Containers

In general, raytracing an optical system usually involves atleast two paraxial rays, and more often also includes several finite rays. Probably the most important reason for including paraxial rays is to provide the designer with the means to undertake various *solves*. A solve is an artificial construct that may be applied to a surface or space, and when applied will modify either the surface curvature or space separation in order to satisfy some

predetermined condition as required by the designer (see the following section for a more detailed explanation of what solves are and how they are implemented). Another reason for including a paraxial raytrace is that it also provides valuable data that is required to initialise the finite rays prior to launching. These data will be used to determine the paraxial positions of both the entrance pupil and image plane, which are absolutely essential for firstly, initialising the finite rays at the beginning of the raytrace, and secondly, for determining at what point (or plane) the raytrace will be concluded.

Though seemingly similar in nature, the actual data items that comprise both a paraxial ray and a finite rays are too dissimilar to warrant any form of inheritance between the two ray types, and so they must be treated separately. Accordingly, two ray-set types are defined, the first encapsulating both the principal and marginal paraxial rays, **TPxRaySet**, while the second encapsulates an array structure of finite rays, **TRaySet**, with the added bonus of also being able to refer to the finite rays in terms of spectral line sets, i. e. those rays that originate from a line source as opposed to a point source, and which are of the same wavelength. Thus, the two paraxial rays are referenced by:

```
a_pxray[PR]
```

and,

```
a_pxray[MARG]
```

...while the finite rays are referenced by:

```
a_rayset[j]
```

...for the j^{th} ray. If the source has been created to have a spectral line structure, then the middle ray of the the j^{th} line (assuming that a line is made up from three ray points) is referred to by:

```
a_rayset.line[j].middle.
```

7.4 The Component-Ray Interface

The component-ray interface, to be described further in this section, is a common interface for all components. The procedure that implements the interface is called **ProcessRays** and accepts a parameter that accommodates both paraxial and finite ray-sets. Each newly created component class that derives from **TLens** will override this method and

implement its own code that will be pertinent to that particular class. Components may not just be optical subassemblies, but may also undertake to be any valid form of ray processor. One such component, the **TRefAlign** component, concerns itself only with the task of redirecting a ray bundle in a direction specified by a reference ray. This facility is especially useful when a component such as a prism causes finite rays to be deviated far from the original optical axis. The **TRefAlign** component will create a new optical axis that conforms to the path of a particular ray (reference ray) defined in the source component by the user prior to the raytrace. More commonly, though, components and their associated **ProcessRays** method will actively transfer and refract rays from the source to the image plane, as is the case in other raytrace programs, although there will always be subtle differences that reflect the varying nature of each.

One of the most useful of all properties that some components provide is the ability to undertake *solves*. As mentioned in the previous section, solves are required to be undertaken prior to a finite raytrace if the results of the latter are to be valid. A solve is a technique that a designer may opt to use if it is required that a particular paraxial ray satisfy a certain condition. There are, in total, four types of solve that may be applied, but only one per surface or space:

1. Angle Solves

- (a) **Paraxial Principal Ray** - the curvature of the specified surface is adjusted so that the convergence angle of the ray is equal to a predetermined value;
- (b) **Paraxial Marginal Ray** - as above;

2. Separation Solves

- (a) **Paraxial Principal Ray** - the separation between the current surface and the following surface is adjusted so that the subsequent ray intersection height is equal to a predetermined value;
- (b) **Paraxial Marginal Ray** - as above;

The most usual circumstances in which these solves are employed are either to set the back focal length to that of the paraxial image plane, or to set the power/eff³ of the total system. In either case, any subsequent raytrace that utilises finite rays will correctly

³eff = equivalent focal length

reflect the state of the optical system as defined by the paraxial solves. If the order of the raytrace is reversed, and the finite rays are traced before the paraxial rays, then the final state of the optical system will not conform to the state as given by the finite ray analysis, since it is the data appertaining to the latter which provides an exact description of the aberrational correction of a lens system.

It is important that solves are applied before finite rays are passed through the system, otherwise the results of the finite raytrace could become invalid, which leads us to conclude that the paraxial raytrace must always precede the finite raytrace.

Since a paraxial raytrace appears to be a prerequisite for raytracing, and considering that it is much faster to undertake than a finite raytrace, there seems little reason not to include it on every occasion. Thus we must consider the mechanism of how the program is to handle a ray-set that comprises ray-types that are so dissimilar, since it was pointed out in the preceding section that `TFiniteRay` is not descended from `TParaxialRay` for just this very reason. There are several solutions to this problem, all apparently as valid as the other.

1. Commence the raytrace with a procedure that accepts paraxial rays, and then another procedure that accepts finite rays,
2. create a new class that contains both paraxial and finite rays and use this class as a parameter for a general raytrace procedure,
3. provide a procedure that accepts both paraxial and finite rays, which then passes each parameter to separate procedures as above.

The method opted for is Item 3, where the interface for a typical lens component, say of type `TThkLens`, would look similar to:

```
TThkLens.ProcessRays(rays:array of TObject);  
var  
    j,k:integer;  
    etc.;  
begin  
    inherited ProcessRays(rays);
```

```

for j := 0 to high(rays) do begin
  if rays[j] is TPxRaySet then begin
    {trace the paraxial rays}
  end
  else if rays[j] is TRaySet then begin
    {trace the finite rays}
  end;
end;
end;

```

...where a call to this method would look like

```

a_thklens.ProcessRays( [a_rayset, a_pxrayset]);

```

The structure [...] is the Object Pascal syntax for creating an *open array*, which allows for arrays of varying sizes to be passed to the same procedure or function. In this case the array is of type **TObject**, which will permit an array of any descendant of **TObject** to be passed. Naturally, this includes both the paraxial and finite ray-sets: **TPxRaySet** and **TRaySet**. Inside the procedure the elements of the open array are interrogated using Run-Time Type Information, or RTTI, in order to determine the correct action that needs to be taken for each of the array elements. This process is dependent upon the order of the ray-sets within the open array, i. e. for the paraxial raytrace to be accomplished before the finite raytrace, the paraxial ray-set must be the first element of the open array. It does not preclude either the paraxial ray-set nor the finite ray-set from being the sole member of the open array if this is exactly what is required by the programmer.

The implementation of solves is best illustrated by referring to the paraxial refraction method of the **TConic** surface type:

```

Procedure TConic.PxRefract(rays:TPxRaySet);
var
  j:integer;
  ndx1,ndx2,hcv:extended;
  px:TPxRay;
begin
  ndx1 := glass1.index(rays[PR].wv1);
  if reflect then ndx2 := -ndx1

```

```

else ndx2 := glass2.index(rays[PR].wvl);
q := ndx1/ndx2;
px := nil;
if not TLensform(formowner).doublepass then
case solvetype of
  ubar : px := rays[PR];
  uMarg: px := rays[Marg];
end;
if px <> nil then Solve_u(px);
for j := PR to Marg do begin
  px := rays[j];
  if (solvetype = j) then break;
  hcv := px.h * cv;
  px.u := q * (px.u + hcv) - hcv;
  if reflect then px.u := -px.u;
end;
end;

```

The interesting and relevant part of this code starts with the line **case solvetype of**; it is at this point that the **solvetype** property of the **TConic** surface is queried. If it has a value of either **ubar** or **uMarg** (both predefined constants) then this signifies that a solve is required and the appropriate paraxial ray is passed to the **Solve_u** procedure. This procedure is a method of **TConic**, and when initiated with a paraxial ray it will alter the value of the base surface curvature to a value that causes the convergence angle of the paraxial ray to be equal to the required value, as given by the **solvevalue** property of the surface. Only when this has been accomplished will the remaining paraxial ray be processed in the normal manner.

We have shown in this section that the component-ray interface can be relatively simple to implement in principle, and yet be capable of reflecting very sophisticated ideas. The application of solves is just one illustration of this, but in the next section we will have the opportunity to discuss other more specialised components that process rays in very different ways.

Chapter 8

Detailed Component Descriptions

So far in this thesis, we have had the opportunity to review the minor players and supporting cast in what could be described as a very complex stage production. We started with the basic ideas that underpin most of today's optical analysis models and showed that the data structures underlying the user interface are relatively primitive and unconnected with the essential model that users are familiar with. A more realistic model was then developed which used the laboratory optical workbench as its operational metaphor. There then followed a necessarily brief description of the modern object oriented programming technology that has impacted the current and future designs of major software application suites. It was shown that the model employed by these techniques could be adapted to simulate many physical processes, and in particular the simulation and analysis of complex optical systems, which is the main subject of this thesis.

The stage and props were then introduced. The toolbar was shown to be the repository for the main components of our system, providing everything from the light source to the lens elements and image plane; the so-called optical bag. In keeping with the need to keep everything on the user's desktop as simple and clear as possible, the lens form was introduced. In conjunction with the toolbar, optical elements of varying descriptions may be dropped onto the lens form to create virtually any combination of optical system desired, and employing current *drag and drop* techniques so prevalent in modern GUI applications these same components may be moved and copied in order to modify the same system.

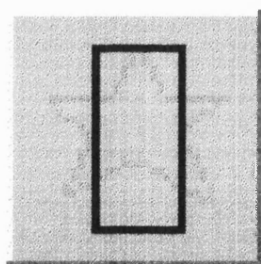
But these components are not just 'pretty' representations of the real thing, indeed

they are also containers for the data that makes each component so unique. The provision of a lens editor allows the user to edit every property of a component, from the selection of glass type for a thick lens to the number and wavelength of all rays provided by the source. Any number of active editors at one moment are supported, while the simple action of dragging a component from the lens form and dropping onto a lens editor is all that is required to initiate a view or commence editing of any visible component.

The last two chapters concerned themselves with the basic elements that all components are derived from. Glasses were straightforwardly dispatched while the general surface class was developed in such a way as to show how the various surface types (plane, spherical, ellipsoidal, etc) could be accommodated in any component supporting one or more surfaces. Simpler in essence and yet much more sophisticated in operation, the various ray types and their corresponding containers were then highlighted. The amalgamation of optical rays with the mathematical subtlety of vector and matrix properties was shown to provide a powerful tool in the armoury of general purpose raytracing algorithms.

In this chapter the reader will be introduced to some of the principal players of this cast. Though all are derived from the **TLens** class, which is a property of the **TLensButton** component, they each implement the inherited method of **ProcessRays** very differently.

8.1 The Source



Traditional raytrace programs have always considered the source to be a completely separate entity from the rest of the system, and so requiring an editing facility that is radically different to the surface editor. This approach is consistent with the surface based model adopted by virtually all commercial programs to-date. In contrast, the component based model does not have such a restriction and the editor employed by IRIS is equally at home supplying an editing facility to a source component as it is to a thick lens component. This is the legacy of modern development environments such as Visual Basic and Delphi, where

only a single editor is required to set the properties of windows components as diverse as a drop-down combo box (a visual component) and a system timer (a non-visual component).

The source, like all components within IRIS, is located in the toolbar and a copy may be placed in any system form by normal drag-and-drop techniques. A system form will only ever require a single source, and the possibility of more than one source being placed on a system form is precluded during run time by actively searching the current form for other sources and refusing the drag-and-drop operation if one is found.

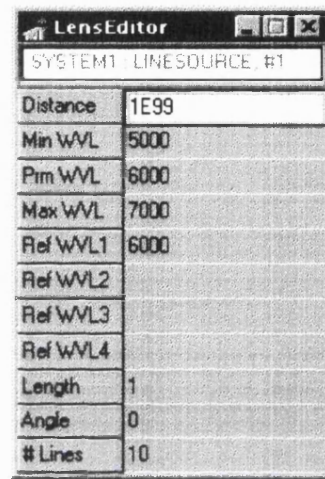


Figure 8.1: Source in Lens Editor

When a source component is dropped onto a lens editor the user is presented with several properties (see Figure 8.1) that may be configured to match the system requirements. These properties are described in more detail below.

Distance The distance in mm from the source to the next component in the system form;

Min WVL1 This is the shortest wavelength, in Ångstroms¹, of the spectral band in which the system is to operate;

Prm WVL This corresponds to the centre or primary wavelength of the systems operating band;

Max WVL As in *Min WVL* above, this is the maximum wavelength, in Ångstroms, of the operating waveband;

¹The Ångstrom (10Å = 1nm) is an accepted unit of wavelength in spectroscopic work, and is used alongside the nm throughout this thesis.

Ref WVLn where $n = 1 \dots 4$. Reference wavelengths are a new concept introduced into IRIS to facilitate the design of non-axisymmetric systems such as spectrographs. Specifying a non-null value² in either or all of the Ref WVL edit boxes will result in a finite axial ray of the specified wavelength being created, and which will be traced along with all other paraxial and finite rays at the commencement of a system raytrace. A special component called **TRefAlign** may be placed anywhere in the system form following the aperture stop, and will realign all finite rays such that the reference ray to which the component is ‘tuned’ now appears to be the new axis for the next component. A more complete description of the **TRefAlign** component is given later in this chapter;

Length IRIS was originally designed for the purpose of spectrograph analysis, and in keeping with this goal the type of source that has been implemented is a *line*-source. **Length** is the total length of the line-source or slit in **mm** for a source at a finite distance ($\text{Distance} < 10^{14}$), or the total angular length of the slit in arc-seconds for a source at an infinite distance ($\text{Distance} > 10^{14}$);

Angle This is the azimuthal angle of the slit with respect to the $+y$ -axis;

Lines Again, in keeping with the spectrographic theme, this property is used to set the number of specific wavelengths that are to be traced through the system. Each spectral line will comprise three finite rays emanating from the top, bottom and middle of the slit, where each ray is a finite principal ray, i. e. each ray passes through the centre of the aperture stop. A single spectral line will be interpreted as a line of wavelength equal to the primary wavelength, while two lines will have the wavelengths corresponding to either end of the spectral waveband; subsequently higher numbers of lines will have equally spaced wavelengths.

The activating method of the source is the **ProcessRays** procedure, and this is the case for all components derived from **TLensButton**. The listing for the source implementation of this method is given below.

```
procedure TSource.ProcessRays(rays:array of TObject);
var
  j,k:integer;
```

²The value of the reference wavelength must lie within the operating waveband.


```

PxRays:TPxRaySet;
RefRays, FiniteRays: TRaySet;
begin
  inherited ProcessRays(rays);
  try
    PxRays := GetPxRays(rays);
    if (PxRays <> nil) then aspace.transfer(PxRays);
    RefRays := GetRefRays(rays);
    if (RefRays <> nil) then begin
      for k := 1 to RefRays.raycount do begin
        aspace.transfer(RefRays[k]);
      end;
    end;
    FiniteRays := GetFiniteRays(rays);
    if (FiniteRays <> nil) then begin
      for k := 1 to FiniteRays.raycount do begin
        aspace.transfer(FiniteRays[k]);
      end;
    end;
  except on E:Exception do
    begin
      E.message := E.Message +
        #13#10'Source Unit: Raytrace failure at component #' +
        IntToStr(self.ndx);
      raise;
    end;
  end;
end;

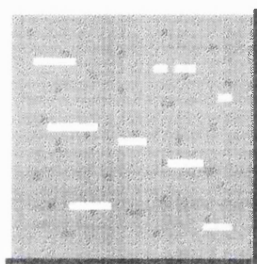
```

The three principal actions of this routine are to:

1. initiate the paraxial raytrace;
2. initiate the raytrace of reference rays, if any;
3. and finally, initiate the finite ray raytrace.

The end product of each step is the transfer of rays from the source to the vertex plane of the following component. This is achieved through the implied presence of a spacer component (see next section) that accommodates the **Distance** property of the source, and is referred to in the above listing as **aspace**. The initialisation of rays, both paraxial and finite, is undertaken by a few utility routines that will be described in the next chapter.

8.2 The Spacer



This component is probably one of the simplest components available. Its function is to provide the necessary separation between the more tangible components and is, in effect, much like the medium of air that surrounds all optical components. At a deeper level of understanding, the spacer also allows other components to have an existence independent of others within the lens form. One of the primary aims of the lens component is that it should, as much as possible, have an existence that does not depend in any way upon the existence, or not, of other components that may have been assembled within the lens form system. We have already seen in a previous chapter how the surface based model depends upon a linkage between a surface characterisation and knowledge of its position relative to its neighbours. Surfaces cannot be moved or copied from one location to another because of this linkage. In developing the component based model, of paramount importance was the capability to do just this, and if this could not be achieved then the *optical bench* metaphor that underpinned this work would have been sacrificed. The spacer component is central to realising this goal.

The space component is much more flexible than the simple analogy with the air medium that was alluded to in the previous paragraph; air is just one of the many media that may be selected to occupy the space between any two components. In common with the thick lens that we shall encounter in the next section, the spacer also provides a property editor that allows the user to select any type of glass from the glass lists

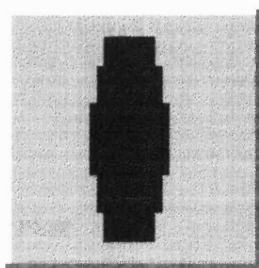
provided. Thus, with this facility it is possible to create more complex lens-types such as lens doublets and triplets. For example, the triplet lens is constructed by simply inserting a glass spacer between two thick lens singlets.

Additionally, whilst the user may specify a fixed thickness for the spacer medium, it is also possible to activate a *thickness-solve*. This operates in a similar manner to the angle-solve discussed in Chapter 6, but instead of causing the surface curvature to be modified during a raytrace, the action is upon the spacer thickness to be modified to accord with the type and value of the selected solve. There are two solves available:

1. **A paraxial principal ray-height solve** - the thickness of the spacer is adjusted to a value that results in the intersection height of the principal ray with the following surface being equal to the user specified value;
2. **A paraxial marginal ray-height solve** - similar to the above, but the thickness solve applies to a paraxial marginal ray.

The former may be used, for example, to correctly position the aperture stop when it is not possible to undertake design changes in the preceeding components (solve value = 0), and the latter solve is often used to position the final image plane at the paraxial, or gaussian, image plane (solve value = 0, also).

8.3 The Thick Lens



Probably the most prevalent and important of all powered optical elements, the thick lens may be considered to be the basis for other complex lens components, as shown in the previous section. It consists of two conical surfaces that bound the enclosed glass medium, all of which are properties introduced into the new class **TThkLens** that is derived from **TLens**. In component form, the thick lens provides property editors that allow the user

to select the type and thickness of glass, and the various parameters that both identify and quantify the conical surfaces (see Figure 8.2), and which are originally supplied by the respective surface and glass classes.

The screenshot shows a window titled "LensEditor" with a standard Windows-style title bar. Below the title bar is a text field containing "SYSTEM1: THKLENS, #4". The main area of the window is a table with two columns: a label column and a value column. The table contains the following data:

Thickness	1
Diameter	10.0000
Catalogue	SCHOTT
Glass	BK7
CC#1	SPHERE
Radius#1	100
Solve type	none
Solve value	0
A[4]	1.457000E-4
A[6]	2.360000E-7
A[8]	4.562140E-8
A[10]	
CC#2	SPHERE
Radius#2	100
Solve type	none
Solve value	0
A[4]	
A[6]	
A[8]	
A[10]	

Figure 8.2: Thick Lens in Lens Editor

The **ProcessRays** method implemented by the **TThkLens** class is relatively simple, since most of the processing actually occurs within the raytrace methods of the surface and glass types. A short section of the code is shown below:

```
for j := 0 to high(rays) do
begin
  if rays[j] is TPxRaySet then begin
    {now trace the rays}
```

```

if diagnostics then begin
    OutputAddStr('Lens ' + IntToStr(ndx));
    TPxRaySet(rays[j]).outputrays;
end;
surf1.transfer(rays[j]);
surf1.refract(rays[j]);
aspace.transfer(rays[j]);
surf2.transfract(rays[j]);
end
else if rays[j] is TRaySet then with TRaySet(rays[j]) do
begin
    for k := 1 to raycount do
    begin
        {now trace the rays}
        surf1.transfer(ray[k]);
        {call coating analysis here}
        surf1.refract(ray[k]);
        aspace.transfer(ray[k]);
        surf2.transfract(ray[k]);
    end;
end;
end;
end;

```

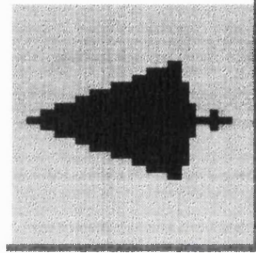
There are three features of this code block that are worth highlighting:

1. The first of the two `for...loop`s undertakes the raytrace of the paraxial rays³. This loop must be undertaken before any other in order to complete any necessary *solve* computations;
2. The second `for...loop` completes the raytrace by tracing all the finite rays;
3. the simplicity and efficiency afforded by the adopted lens model may be discerned

³In order for a surface to support *solves* it is essential that paraxial rays precede finite rays during a raytrace. Thus, while the above procedure appears to treat both types of ray simultaneously, in actuality the procedure is entered for the first time with paraxial rays alone, and later only finite rays will be passed to the procedure.

by the comparatively few lines required by the `ProcessRays` method. The core of each of the above loops consists of only four lines that include: the transfer of rays to `surface1`, followed by the refraction at the same surface, transfer of rays across the `aspace` property, and finally the transfer and refraction of rays at `surface2`.

8.4 The RefAlign



The source component was introduced in the first section of this chapter, and reference was made to the rather specialised finite rays called *Reference Rays*. The RefAlign component, that we introduce here, is the only component to properly utilise these rays. `TRefAlign` is derived from `TLensButton`, but unlike previous components its purpose is to transform all finite rays in such a way as to realign the optical axis. This is best understood by referring to Fig. 8.3 where an arbitrary set of rays is shown on the left along with a designated

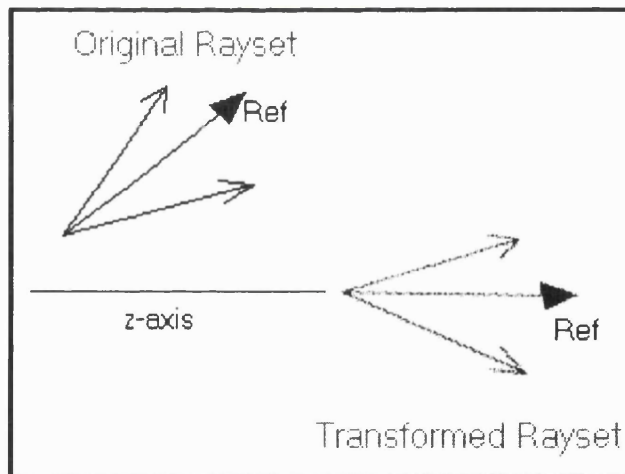


Figure 8.3: The action of `TRefAlign` upon Reference Rays and Ray-Sets

reference ray, *R*. When these rays encounter a RefAlign component that is *tuned* to a designated reference ray, the following actions will occur:

1. RefAlign will determine the displacement vector of the reference ray, which is given by

```
dispvec.assign(refray.posn.negative_);
```

- i. e. the displacement vector `dispvec` is assigned the negative value of the ray position vector;

2. a matrix `rotmat` is assigned the value of a rotation matrix that corresponds to realignment of the `refray` with the z -axis, and then inverted, or

```
rotmat.VecAlignMatrix(refray.dirn, z_vector); rotmat.transpose
```

where the transpose is equivalent to a matrix inversion when the original matrix is orthogonal;

3. finally, all the rays in the ray-set are transformed, as in the code snippet below,

```
with rayset[k] do
begin
    posn.plus(dispvec);
    vector.matmult(rotmat);
    if polvec <> nil then polvec.matmult(rotmat);
end;
```

where each ray is displaced and then rotated. The resulting action leads to the reference ray having a directional vector of $(0, 0, 1)$ and a positional vector of $(0, 0, 0)$, equivalent to a ray travelling along the z -axis. All other rays in the ray-set will maintain their exact positions and directions relative to the reference ray.

One other feature that is not shown is a **Locked** property. The **RefAlign** component, in common with other components, is capable of being used within a *double-pass* sequence (see final section of this chapter) where light may not only traverse the component in the normal direction, but it may also pass through the same component in reverse direction, as in when reflected by another component further along the sequence. In this circumstance the **RefAlign** component will save the transform matrix and vector until the rays return in double-pass mode. In this way the spatial relationship between this component and the next is preserved, as it should be. Naturally, all components, including **RefAlign** are able to determine if rays are in double-pass, and so take suitable measures to compensate or modify their behaviour when in the **ProcessRays** method.

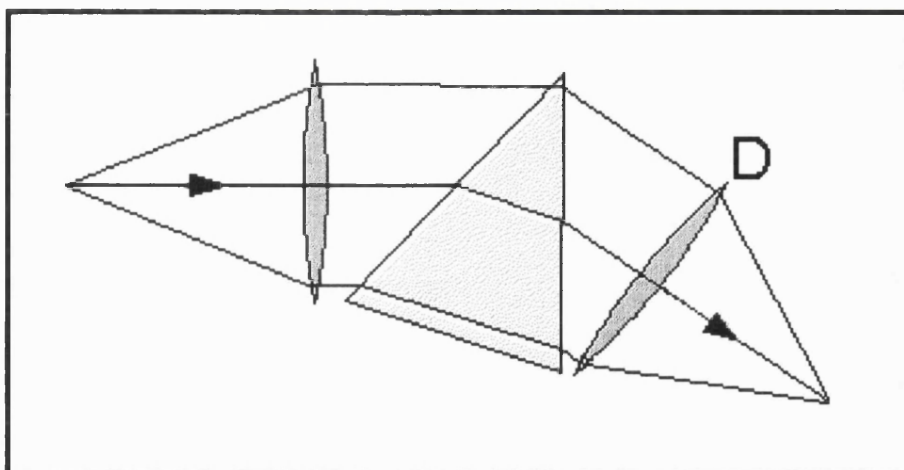


Figure 8.4: A Simple Spectrograph

Consider the case of a spectrograph based upon a simple prism (see Figure 8.4), where the optical axis is strictly no longer defined after the first prism surface. In such a situation we may decide that the de-collimating lens (D) should sensibly be centred upon the original axial ray, and at a wavelength corresponding to the middle of the waveband and in the space following the prism, as shown in the figure. This scenario may be easily created by the adoption of the following component scheme, as shown in Figure 8.5, below. The source has been modified to have a single reference ray with a wavelength

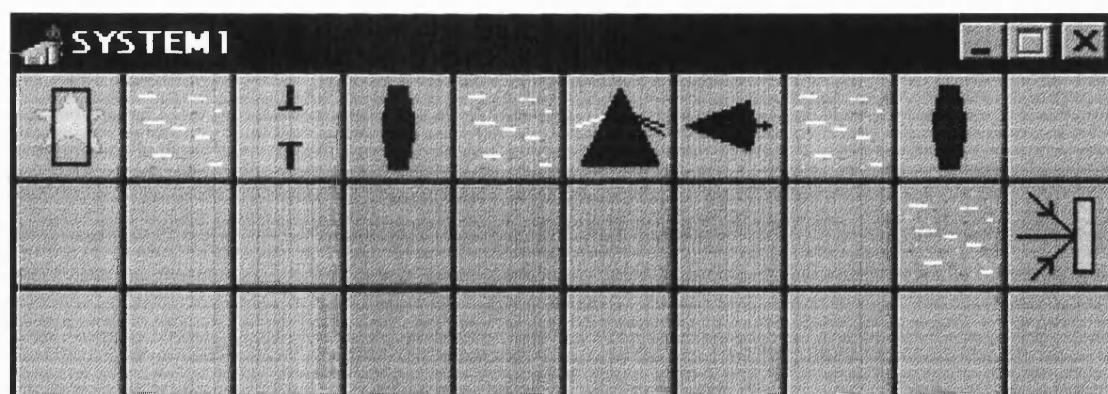
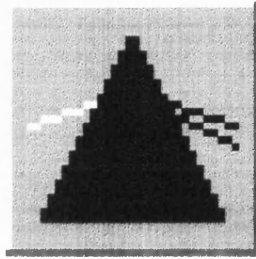


Figure 8.5: The Spectrograph Form

in the middle of the specified waveband, while the RefAlign component has been tuned to the same reference ray. Axial rays that commence at the source may be shown to pass through the system and leave the RefAlign component still travelling along the z -axis, exactly as required (see Chapter 10 for a more detailed analysis of a spectrograph).

8.5 The Prism



In common with **TThkLens**, the prism class (**TPrism**) has two surface types and a spacer, but unlike the former the surfaces are of type **TPlane**. The **TPlane** surface offers two important properties that makes it eminently suitable for inclusion within the **TPrism** class:

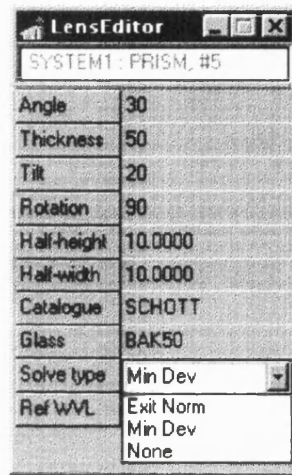
- the plane surface may be tilted, or rotated about the x -axis;
- it may also undergo an azimuthal rotation about the z -axis.

These properties are hidden from the user but are developed further internally in the surfacing of several other properties that have a more direct relevance to ones understanding of how a prism operates. These are:

1. Angle – the apex angle of the prism;
2. Tilt – the rotation angle of the first surface of the prism with respect to the x -axis;
3. Rotation – the azimuthal rotation angle of the whole prism with respect to the z -axis;
4. Solve type – usually, the designer has it in mind that the prism should operate in a particular manner that often makes it difficult to construct from single surfaces. In order to remove this obstacle, the two most common operational modes are offered to the designer as complete configurations:
 - (a) Exit Norm – the inclination angle (tilt) of the first surface and the prism apex angle are automatically set at those values which will result in the specified reference ray leaving the second surface perpendicularly;
 - (b) Min Dev – the inclination angle of the first surface is automatically set at that value which results in the reference ray entering and leaving the prism at the minimum deviation condition for the user specified apex angle;

- (c) None – the prism configuration is defined solely by the user specified parameters of **Angle** and **Tilt**.

The prism properties that the designer has to select from are shown in Fig. 8.6, below.

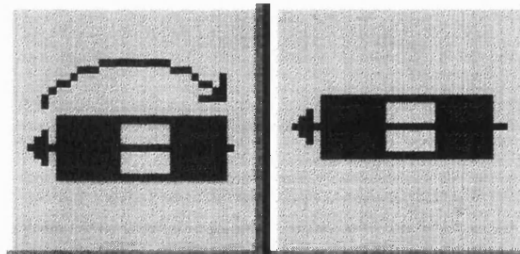


The screenshot shows a window titled 'LensEditor' with a tab labeled 'SYSTEM1: PRISM, #5'. Below the tab is a table of properties for the prism component.

Angle	30
Thickness	50
Tilt	20
Rotation	90
Half-height	10.0000
Half-width	10.0000
Catalogue	SCHOTT
Glass	BAK50
Solve type	Min Dev
Ref.WVL	Exit Norm Min Dev None

Figure 8.6: Prism in Lens Editor

8.6 The Double-Pass Components



One of the driving principles behind much of this thesis is that reality usually embodies within it a model that is both simple and efficient, if only sometimes we could see it, and in attempting to model reality we should embrace those ideas that harbour germs of the underlying truth. The double-pass component is a necessarily artificial construct that attempts to simplify what has, upto the present time, been a complex method for handling the problem of light re-traversing optical components.

The ‘unintelligent’ surface based model provides a singularly linear sequence of surfaces, where light appears to be travelling in only one direction alone. If we are to cause light to reverse direction and pass more than once through the same element, then we must

recreate that element further *downstream* in suitably reversed form. This is burden enough for the designer, but if the element is not axially symmetric then the task of reversing the element becomes even greater. The approach taken by the component based model is far more intelligent, where the model is designed from the ground up to accommodate such a scenario.

The **TLens** class has two very important features that make possible the notion of having light re-traversing components in reverse sequence. Firstly, all descendants of **TLens** possess a method called **reverse**, which allows each component to undergo a mirror reversal of itself. Secondly, prior to tracing rays (**ProcessRays**) each component will check with the lens form to determine whether light is travelling in reverse direction, and will undertake to **reverse** itself if true. The **RefAlign** component, as we have already seen in a previous section of this chapter, also undertakes special procedures to ensure correct and proper behaviour when positioned within a double-pass sequence.

There are two components that define a double-pass sequence, both of which have no published properties and so cannot usefully be viewed in the lens editor. More correctly, these components are used as *flags*, one to indicate the beginning of a double-pass sequence and another to indicate the end, although they both have an identity that is individually unique. The best way to illustrate how these components operate is by way of example. Consider a sequence of characters, as shown below,

. . A, B, X, (C, D, E), Y, F, G. .

where each character represents an optical component, but the letters X and Y represent the start and end points of the double-pass sequence, which in this case contains the sub-sequence (C,D,E). A ray of light that were to traverse this sequence would ‘see’ the components in the following order:

. . A, B, X, (C, D, E, Y, D, C), X, F, G. .

i. e. the sub-sequence is *reflected* about the component, E. Figure 8.7 shows a rather contrived lens system that illustrates more clearly the power of the double-pass components.

The components A, C, D, G are single lenses, B is an optical axis displacement component (for brevity, this component has not been included in this thesis), and F is a **RefAlign** component. It is quite clear that the two singlet lenses (C,D) are in double-pass mode as defined by the presence of the double-pass components X and Y, and initiated by the action of the mirror surface E. In fact, the component sequence given above is an accurate

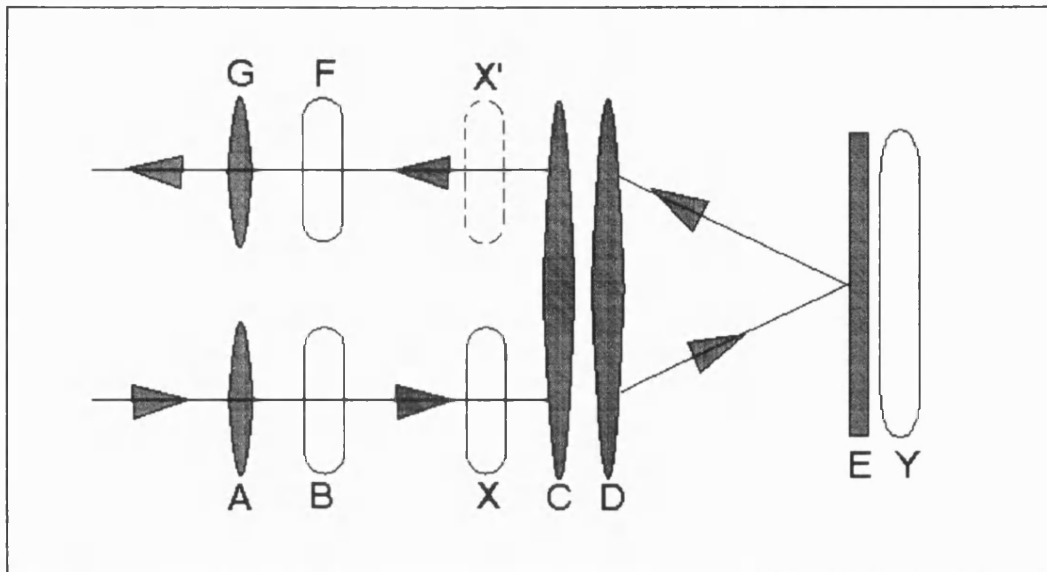


Figure 8.7: A Double-Pass Sequence of Lenses

representation of the component description within the figure, ignoring the absence of air-spaces. The sequence of events that the ray encounters as it passes through the system is as follows:

- an axial ray passes through lens A,
- the component B laterally translates the optical axis in preparation for the decentred lenses C and D,
- the **DoublePass** component X signifies the beginning of the double-pass sequence,
- the ray progresses through lenses C and D and is then reflected by mirror E,
- the presence of the **EndDouble** (Y) component causes the ray to reverse direction and reverse the sequence,
- lenses D and C are encountered next, each one detecting that the ray is in double-pass mode and accordingly reversing its shape,
- the **DoublePass** component (X') is encountered once again, indicating that the reverse sequence is now concluded. The sequence now continues at the component following **EndDouble**,
- a **RefAlign** component (F) is introduced here so that a new optical axis is created along the same ray that initiated the raytrace,

- finally, the ray traverses the lens G, and concludes our ray journey.

How does all this take place, and who or what is waving the baton? This is the subject of the next chapter.

Chapter 9

Optical System Processes

In this chapter we shall be reviewing those processes that exist at a level higher than those that might otherwise take place within the realm of the components described in the previous chapters. The *system-level* processes are what is required to breathe life and activity into otherwise static and directionless components. For instance, while we should now understand how all lens components are constructed and how they manage to uniquely define their own particular behaviour in respect of raytracing (**ProcessRays**), there still remains several questions that have direct bearing upon the final orchestration of all these processes, i. e.

- Is the system, or sequence of components as defined by the designer, a physically valid one?
- As described in the previous chapter, a lens sequence may not necessarily be a linear sequence, in which case how is the sequence coordinated, i. e. what is it that determines the sequence throughout the raytracing process?
- How are the rays to know their initial positions and directions, and what is it that launches them on their way?
- Finally, however the raytrace might be concluded, how is the image information to be obtained?

The answers to these questions are to be found in the following sections of this chapter, but before we embark upon this final leg of our journey it is probably opportune to review

the various structures and classes that have been the so-called stepping stones that have brought us to this point.

Figure 9.1 shows a hierarchical structure that depicts **TApplication** (the main application as seen on the computer monitor) at the head and several other recognisable classes that cascade down to ever lower, or more primitive structures. The **LensForm** is

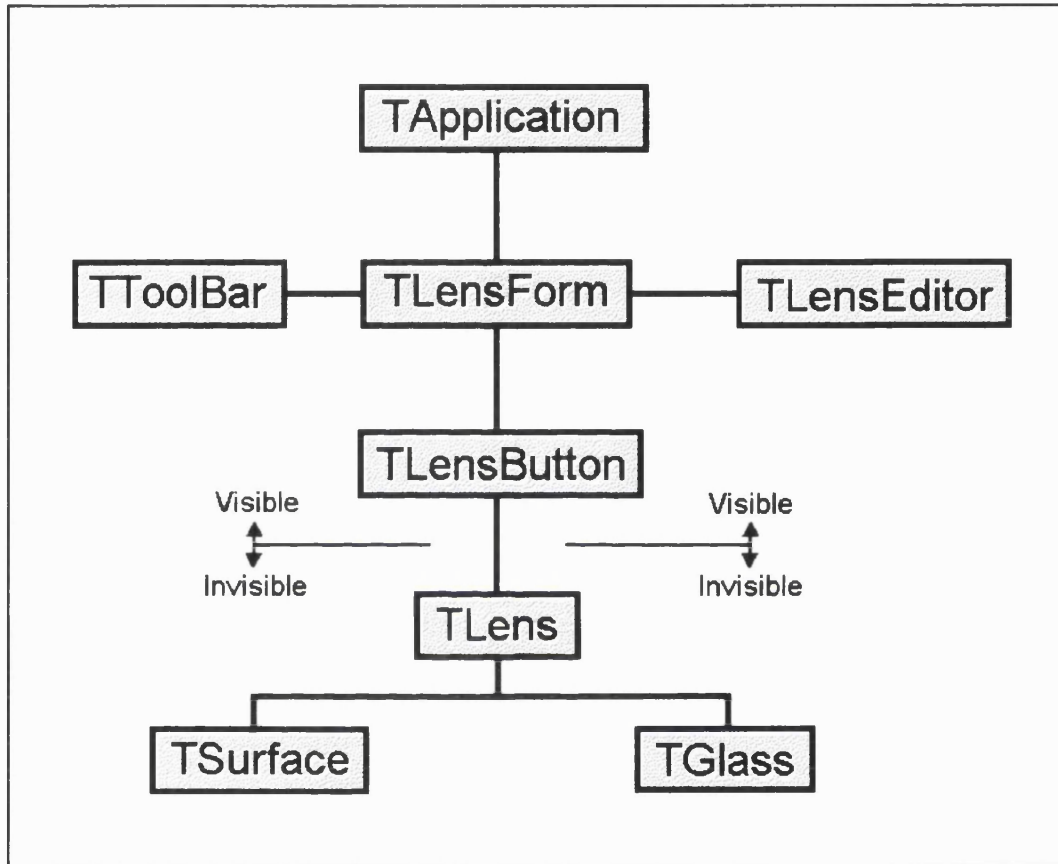


Figure 9.1: Hierarchical structure of the IRIS program

the visible manifestation of the lens system (see Section 5.2), while the various components of the lens system, (including all lenses, source, stop, image plane, etc.) are represented by the **TLensButton** speed-buttons (see Section 5.3), that populate the **LensForm** when selected from the **ToolBar**, according to user requirements. The heart of each lens component is the **TLens** class, which is also the repository of the fundamental primitive structures that all components are derived from. It is in the **TSurface** and **TGlass** classes that the real *number-crunching* takes place, where rays are transformed from one optical space to another, and where the imaging process becomes manifest.

But what is the ultimate driving force behind these processes? As in the age old question of what it is that drives a human being to his or her ultimate destiny, do we assign such a force to the whimsical (or not) notion of fate, or do we attribute it to our unique quality that we call ‘freedom of choice’? And if it is the former, do we presuppose, as the early Greeks did, that the Fates spun the destiny of mankind into an elaborate tapestry, or if this seems too complicated and contrived then do we *create* a God in Whom we place our destiny but Whose purpose is not available to us for scrutiny? These philosophical ideas also have their place when we try to consider what is the motivating force behind a simple raytrace. There are two possible solutions:

1. There is no external force beyond that of the lens component, as represented by **TLensButton** and its subordinate classes. Instead, each lens component is imbued with a self awareness (if that doesn’t appear too pretentious) that enables it to provide the correct response in any given circumstance;
2. The lens component is allowed to process rays, as that is its speciality, but the interpretation of the correct sequence, amongst other matters, is the prerogative of some external agent.

Both solutions have been considered in depth by the author, but not even the reality of the situation offers any clue as to which should be the final choice. Initially, the lens component was designed to be a complete and separate entity, complete in that all methods or modes of operation were self-contained, and separate in that it could not be considered to be an active part of a greater whole, except by association. In this way, according to the author, the lens component was as near to a true lens as it could possibly be. If this model was to be further extended to encompass the idea of a free-running system then an alternative approach was required.

In the same way that each lens class implements its own unique set of instructions within the **ProcessRays** method, it was decided that the **Source** component should also have some behaviour that would uniquely identify it as the instigator of such system-wide events as raytrace initiation. The source was chosen over other lens components because, quite naturally, this is where we expect rays to originate from. Rather than use a form-based menu structure to display the options, a selective pop-up menu is designed to appear when the source component is right-clicked, which is the currently accepted practice for

Windows 95 applications; see Figure 9.2. As the figure shows, the pop-up menu invites

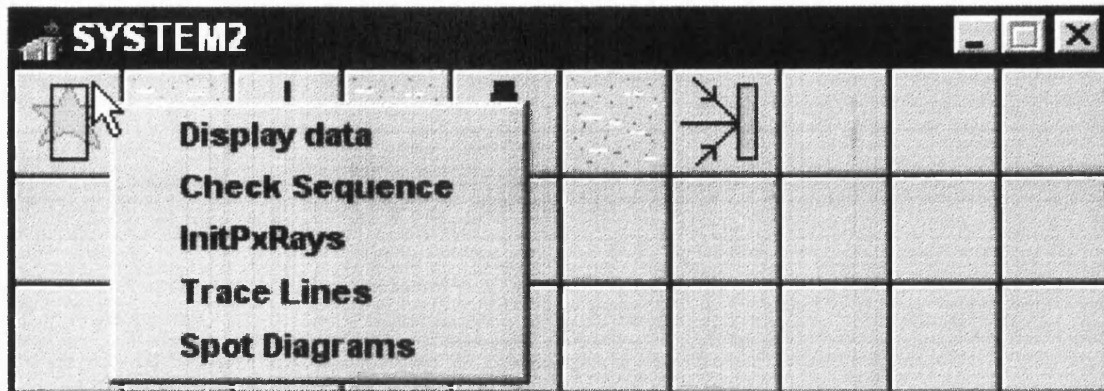


Figure 9.2: The Source Pop-Up menu

the user to select one of the five available options:

- print component data to the ‘Textual Output’ window. This menu item is also available to all other lens components;
- check if the sequence of components is a valid sequence;
- as above, but also initialise and trace the marginal and principal paraxial rays;
- as above, but also trace all the finite rays as defined by the user;
- displays spot diagrams in the ‘Spot Diagram’ window (see Figure 10.5).

Though it appears that all of these routines are actioned by the source component alone, in fact the source component will call routines that are really located **within** the LensForm, since the LensForm is the *parent* of the source and so will have access to all components that reside within the form. The following sections will describe in detail how some of these menu options are implemented.

9.1 Validating the System

Despite the designer having complete freedom to arrange the lens components in any particular manner, it is not necessarily the case that the sequence will be a valid one.

During the initial development period of the IRIS program it became necessary to decide whether the program should actively encourage the user to construct only valid sequences, or not. Active encouragement would take the form of the program vetting each component as it was dropped onto the form, or moved from one location to another, and allowing at each stage only a valid sequence to exist, i. e. refusing to accept an editing operation that would result in an invalid system. Ultimately this form of ‘censorship’ proved too restrictive and did not allow for an interim invalid sequence as a prelude to a final valid sequence, and accordingly this routine was dropped. The preferred solution proved to be much simpler and more forgiving, and is summarised by the following points:

1. a sequence, in the initial stages (prior to raytracing) could be formed from any number of lens components in any order;
2. a valid sequence must start with a source-component and end with a detector-component;
3. a valid sequence may exist within an invalid sequence

So, what is a valid system or sequence? Probably the most important feature of a valid sequence is that all the interface conditions for every component are unambiguously resolved, and this implies that each component must be aware of an external medium, and its corresponding refractive index, whether it ‘looks’ to the left or to the right of itself. Recall that in order for the component-based model to be successful it was necessary that each component should have an existence that would be independent of any other component. For instance, if we were to remove a single lens from a series of lenses, where each is separated from the other by an air-space, then we would expect that what remained would still be a valid system, even if its characteristics had changed. In other words, where the lens had once been we would not now expect there to be a volume of space that had no definable properties. In order to correctly resolve this possible incongruity, and yet retain the *quasi*-intelligent approach that we have come to expect from an object-oriented stance, it became necessary to instill within each component the ability to query its neighbours for a proper understanding of how the refractive interface is comprised. Consider the following scenario; a sub-sequence is defined by the following components, (see Figure 9.3): a spacer (air), an aperture stop, a lens, and another (air) spacer. While it is obvious that the lens ‘sees’ a glass/air boundary to the right of itself, how does it manage to

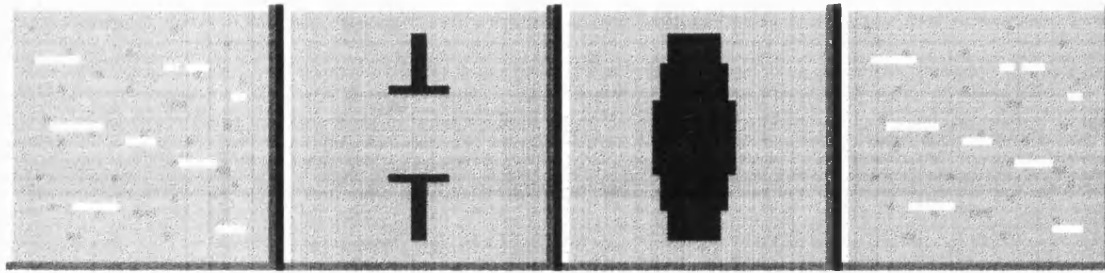


Figure 9.3: An Example of Component Interfacing

resolve the refractive boundary to the left, where its immediate neighbour is a stop that has no associated medium? In very simple terms, the lens undertakes a conversation along the following lines:

Lens: (Question) To the component to the left of me, what is your medium?

Stop: I don't have a medium, but I'll ask the component to the left of me the same question;

Spacer: My medium is air;

Stop: (Answer) My medium is air.

In such an apparently trivial fashion the lens manages to obtain a knowledge of the refractive boundary, thanks to the cooperative nature exhibited by its neighbouring components. But it doesn't stop there, since there is also another ambiguity that requires resolution, and that is in the surface interface between the stop and the lens. If the stop is considered to be a plane surface with a central aperture (the default case), then a ray proceeding from the edge of the aperture towards the lens must apparently traverse a space that we would assume to be air. If it is the case, then this example would be equivalent to having an air-space component of zero thickness interposed between the stop and the lens. Is this equivalence to be allowed, or should some other interpretation be adopted? The solution to this problem lies in an often quoted drawback associated with some of the simpler raytrace programs, that is, whether it is possible to have a stop that is coincident with a

lens surface, i. e. can the stop be ‘painted’ on the lens surface? Such a stop is allowed by the IRIS program according to the following circumstances:

- if the previous component has an associated surface then the stop-surface will be identical, else
- if the next component has an associated surface then the stop-surface will be identical, else
- the stop is a plane surface.

Applying this reasoning to the case cited above, it becomes clear that the stop is actually located at the first surface of the lens.

Thus, by employing the above two techniques for both surface and interfacial media allocation, it is possible for all components to initialise themselves prior to raytracing. In this light, system validation is simply a call to each component in the sequence to initialise itself, in the following manner:

```
if not lens[j].initialise(Msg) then begin
    Msg := Msg + ' : Initialisation error at
                component #' + IntToStr(j);
    MessageDlg(msg,mtError,[mbOK],0);
    exit;
end;
```

The `initialise` function accepts a message-string parameter and returns a boolean value (true/false). If it returns `false` then `Msg` contains an error string from the offending component, which is then concatenated with the `Initialisation error...` string and placed within a Windows style dialogue box.

It was stated earlier that the major factor in achieving a valid sequence was that all components should be capable of successful initialisation, but there is also one other more obvious factor that relates directly to the allowable configurations imposed by certain components. In locating a sequence, the program searches for the first available source and detector, and these two components will mark the beginning and ending of the sequence.

What then follows is a series of ‘interrogations’, the aim of which is to locate those more obvious errors, such as:

- Is there more than one source? (not allowed)
- Is there a stop? (necessary)
- Is there more than one stop? (not allowed)
- Is there a dispersing element before the stop? (not allowed)
- Are there any coordinate transform components before the stop? (not allowed)
- Are the double-pass components correctly configured?

...and finally, as already discussed above:

- Do all components initialise successfully?

The above interrogation is signalled by a call to the **IsSystemValid** function, a method provided by the **LensForm** which, naturally, returns a boolean value.

9.2 The Lens Sequencing Algorithm

The sequencing algorithm is at the heart of all raytrace operations, whether it is paraxial rays, finite rays, or both types of rays being traced. The purpose of the sequencing algorithm is to ensure that the **ProcessRays** method of each component within a valid sequence is triggered in the correct sequence. Recall that **ProcessRays** is implemented differently from one component to another, and that while it triggers a source component to generate the appropriate paraxial and finite rays, a lens component will interpret it to mean that it should transfer the rays from the first surface to the second surface using the specified transfer and refraction methods supplied with the lens component.

The final section of the previous chapter discussed the double-pass components and illustrated how a physical sequence in the lens form could be transformed into a very different logical sequence when double-pass components were involved. The presence of the double-pass components within a valid sequence results in a non-sequential flow of rays

along the component chain, and some means to control this flow is required. Two possibilities presented themselves during the initial stages of the IRIS development programme:

1. The data-flow could be controlled at the component level, where each component is capable of deciding whether optical rays are to be passed forwards or backwards within the sequence.
2. The `LensForm`, as the *parent* of all components within the form, could take external control of the sequencing, much in the same way as a musical composer determines what instrument is played, and when, within an orchestral framework.

Both solutions were considered to be equally complex in their application, but ultimately the second proved to be more adaptable and manageable, for the same reason that a conductor stands outside the orchestral assemblage: complexity is much easier to manage when it is viewed from an external vantage point (although it must be said that the idea of an autonomous lens component is much more appealing). The `LensForm` is just such a vantage point and so the principal routine that drives the sequencing operation, `ProcessSequence`, is located within the `LensForm` code module. The relevant portions of this procedure are shown below.

```
procedure TLensform.ProcessSequence(endndx:integer);
var
  srcndx,stpndx,detndx,ndoublendx:integer;
  lns,tmplns,tmplns2,tmplns3:TLens;
  doublepass:boolean;
  Msg:string;
begin
  if not IsValidSystem then exit;
  {Get positions of source, detector and stop}
  srcndx := GetALens(TSource,1,MaxLenses).ndx;
  detndx := GetALens(TDetector,srcndx+1,MaxLenses).ndx;
  stpndx := GetALens(TStop,srcndx+1,detndx-1).ndx;
  {if necessary, assign predefined components to endndx}
  if (endndx = 0) then endndx := detndx
  else begin
```

```

case endndx of
    THESTOP      : endndx := stpndx;
    THEDETECTOR : endndx := detndx;
end;
end;
{check if endndx lies within the sequence}
if (endndx <= srcndx) or (endndx > detndx) then
begin
    Msg := 'Terminating component (#' + IntToStr(endndx)
    + ') is outside sequence';
    MessageDlg(msg,mtError,[mbOK],0);
    exit;
end;
doublepass := false;
lns := lens[srcndx];
ActiveSource := lns;
try
{this is the RayTracing loop}
    repeat
        Application.ProcessMessages;
        tmplns := lns;
        lns.ProcessRays([mPxRays,mRefRays,mRays]);
        if (lns is TDetector) or (lns.ndx = endndx) then break
        else begin
            if (lns is TEndDouble) and doublepass then
                begin
                    .....
                end
            else if (lns is TDouble) and not(doublepass) then
                begin
                    .....
                end
            else
                begin

```

```

        if not doublepass then lns := lns.next
        else lns := lns.previous;
    end;
end;
until (lns = nil);
except on E:Exception do
begin
    E.Message := E.Message + #13 +
        'ProcessSequence failure at component #'
        + IntToStr(tmplns.ndx);
    raise;
end;
end;
end;

```

The `ProcessSequence` procedure accepts a single parameter: `(endndx:integer)`, where `endndx` denotes that component index at which raytracing is to be terminated. The first few lines of the procedure cover purely administrative tasks, such as checking whether the system contains a valid sequence, and assigning an appropriate value to `endndx` in the case where `endndx` refers to predefined constants: `THESTOP = -1` and `THEDETECTOR = -2`; `endndx` may also refer to any other component index within the valid sequence.

The main body of the routine lies within the `repeat...until` block of code. Prior to entering this block, the `lns` variable is set to the first component of the sequence, which is always a source component, and so the first the loop executes the line `lns.ProcessRays([...])`, resulting in the generation of optical rays that are then directed towards the next component in the sequence. If there were no double-pass components within the sequence, then the loop would continue to return to this line until the end of the sequence was reached, i. e.

```

repeat
    lns.ProcessRays([mPxRays,mRefRays,mRays]);
    if (lns is TDetector) or (lns.ndx = endndx) then break
    else lns := lns.next;
until (lns = nil);

```


Execution will exit the loop when either the detector component or the component with index of `endndx` is reached; while in other unforeseen circumstances, the loop will terminate when no other valid lens is available. The presence of double-pass components inside the sequence introduces considerable complexity within the main loop, where the prime purpose of the extra code is to update *book-keeping* chores and determine whether rays are to be transferred to the next or previous component.

9.3 Pupil Location

Virtually all conventional optical systems possess an aperture stop (iris) whose function it is to limit the size of the bundle of rays as it traverses the optical components that comprise the lens system. In some cases, such as binoculars or even holographic viewers, the entrance pupil is defined by the pupil of the eye that is viewing the scene. Alternatively, where normal Gaussian imaging becomes unnecessary, as in solar concentrators (see [15]), then the elements themselves will ultimately act as the final limiter for the incoming bundle of rays.

To the uninitiated, the positioning of an aperture stop within an optical system (or even within close proximity to an optical system) may seem arbitrary, or at best not well understood. In fact, the aperture stop performs two very important functions:

1. It provides an extra degree of freedom when designing a lens system. The position of a stop within an optical system determines the incident ray height and angle of the principal ray at every optical surface within the system, and since the principal ray is at the centre of the ray-bundle¹ then all other rays disposed about it must also be affected. The movement of the stop in one direction or another will usually result in minor changes to the Seidel aberrations which, in conjunction with other system variables, might be sufficient to improve the correction of an optical system².

2. In some instances it allows the optical designer to indirectly ‘control’ those aberrations

¹In *real* optical systems that operate at large field-angles the principal ray is not usually coincident with the centre of the ray-bundle, and in such cases the term *centre ray* is often employed to denote the actual ray at the centre of the ray-bundle.

²A set of formulae have been devised, generally known as the *stop-shift formulae*, that predict the variation of the Seidel aberrations with change in stop position; see Ref. [11], Ch. 7.5 for a detailed derivation of these formulae.

tions at the margin of the ray bundle. In general, the optical designer's goal is to ensure that all rays converge as near as possible to a single image point, ignoring for the moment those aberrations that are related to field curvature or distortion. This task is comparatively simpler for those rays at the centre of the ray bundle than it is for rays at the periphery. In addition, as the image (or object) moves away from the optical axis then these aberrations will also increase, and more so for the peripheral rays. The usual approach to controlling these 'rogue' rays is to remove them altogether, and this is accomplished by allowing one or more optical elements to vignette these rays. Since lens elements are of finite size, then as the field-angle increases the ray bundle will be seen to traverse the extent of the lens aperture until the periphery is reached and at which point rays will become increasingly vignetted as the field-angle increases further. Thus, by correctly placing a stop relative to some other lens element(s) it becomes possible to reduce the otherwise deleterious effects of the peripheral rays.

There is nothing mysterious about how a raytrace is initiated. If we were to mimic the real world then we would choose to send a great many real rays in the general direction of the lens and hope that a sufficiently large number would be transmitted through to the final image plane. Naturally, all these rays would have ideally originated from a single point on the object plane, and the only really useful rays would be those that successfully passed within the clear apertures of the various optical elements. This approach is somewhat hit-and-miss. A better approach is to determine the position and size of the entrance pupil³ by tracing two paraxial rays; a paraxial principal ray will determine the position of the pupil, while a paraxial marginal ray will determine the extent (in angular terms) of the pupil.

Once this information has been obtained, real rays may then be constructed that will originate from the chosen object point and be directed in some manner that will almost guarantee their successful passage through the optical system, whilst also ensuring that they pass within the periphery of the iris. A typical ray generation scheme attempts to send rays towards the pupil in such a direction as to intersect the pupil in a regular rectilinear grid, or alternatively two ray-fans may be constructed that intersect the pupil

³The *entrance pupil* is the image of the iris as seen from the object-side of the lens; conversely, the *exit pupil* is the image of the iris as seen from the image-side of the lens.

along the local x and y coordinate axes⁴. These ray-fans are referred to as the *sagittal* and *tangential* planes. The tangential plane remains constant from surface to surface, but the sagittal plane almost invariably is not.

The implementation of the pupil location algorithm is comparatively trivial, as it employs a simple one dimensional search based upon the Newton-Raphson method (see Ref. [16], Ch. 9.4). For the special case of an object at infinity, the initial starting conditions of the paraxial ray are a convergence angle equal to the field angle, and an arbitrary intersection height at the first surface of zero. A programmatic loop is then entered that traces the ray through to the stop or iris plane, and this is then repeated within the loop for another starting ray height that is incremented by a very small amount. The difference in ray height at the stop plane is then used to obtain a derivative from which an improved starting height is calculated. The loop is continuously repeated until the paraxial principal ray intersects the stop plane at a value approaching zero. The routine as shown below is fairly *robust*, i.e. it will converge in virtually all cases, since the paraxial raytrace equations represent simple monotonic functions. When the position of the pupil is finally ascertained it is a simple matter to construct real rays that are directed towards this plane, employing one of the ray generation schemes described above.

```

procedure pxInfinityInit(ray1,ray2:TPxRay; src,stp:TLens);
const
    dh = 1e-9;
var
    count,stpndx:integer;
    n,u,h,dy,y,MidWvl:extended;
    lns:TLens;
    lf:TLensform;
begin
    {obtain the form that owns the lens components}
    lf := TLensform(src.owner);
    stpndx := TStop(stp).ndx;
    {get the mid-wavelength}
    MidWvl := TSource(src).MidWvl;

```

⁴The y -axis of a surface is orthogonal to the axis of symmetry (z -axis) and lies in the plane that contains the z -axis and the object point.

```

{get the field angle in radians}
u := -ArcSecToRad(TSource(src).ObjHt);
{initialise the two paraxial rays}
ray1.init(1,u,MidWvl);
ray2.init(TStop(stp).epr,0,MidWvl);
h := ray1.h;
lf.ProcessSequence(stpndx);
y := ray1.h;
count := 0;
{repeat until the ray height at the stop is < dh}
while not(Equal(ray1.h,0,dh)) do begin
    h := h + dh;
    ray1.init(h,u,MidWvl);
    lf.ProcessSequence(stpndx);
    dy := ray1.h - y;
    n := y/dy;
    h := (h - dh) - n * dh;
    ray1.init(h,u,MidWvl);
    lf.ProcessSequence(stpndx);
    y := ray1.h;
    inc(count);
    if count > 100 then
        raise Exception.create('InitPxRays - convergence error!');
end;
{set Stop aperture}
TStop(stp).surface.aperture := ray2.h;
{set paraxial rays to their new initial values}
ray1.init(h,u,MidWvl);
ray2.init(TStop(stp).epr,0,MidWvl);
end;

```

9.4 Other System Processes

The principal system processes have been described above, and though they are fundamental to all forms of optical system analysis, there are also several other less important routines that have significant rôles to play. These are not described in detail here, but the singular case of ray aiming will be briefly discussed below.

There are two distinct cases that need to be considered when designing a ray aiming routine: i) Infinite Source, and ii) Finite Source. The first case is relatively simple to handle since the directional vectors of each ray are equal prior to the first surface, while the spatial coordinates of the rays are set to some coordinates at the pupil plane: either a rectilinear grid if a spot diagram is the final outcome of the raytrace, or to various points along the local x and y axes if a ray fan analysis is required. The case of the finite source differs in that the directional coordinates of each individual ray will need to be determined, since each ray vector will have a unique value.

Chapter 10

E v a l u a t i o n o f M o d e l

The previous chapters may be considered as a narrative, setting the scene, so to speak, for the real expression of those ideas and architectures that have underpinned the basic premise of this thesis: that current software models in the field of optics are comparatively simple, and that they may be much improved by the harnessing of modern concepts of modelling (OOP) that are becoming increasingly prevalent in current state-of-the-art software development tools.

The purpose of this chapter is to provide *proof-of-concept* for the model described within these pages. This will take the form of several tests, where each will provide results that either may be compared against theoretically obtained data for the system under test, or against similar results obtained from a commercial lens design program, which in this case will be Zemax v6.0. The subject of these tests will naturally be the IRIS program, developed by the author over a period of two years, initially in Visual Basic but latterly converted to the Object Pascal language. All of the modules and components that have been described are included in this program, although several other components were developed but not considered complete or stable enough to warrant inclusion.

The simple spectrograph, as depicted in Figure 8.4, presents several major hurdles that IRIS must successfully negotiate:

1. Raytracing through simple lenses: the first lens of Figure 8.4 is required to collimate the light from the source for the subsequent prism to operate correctly. This will test the solve capabilities built into the lens component;

2. Configuring the prism: the normal configuration of a spectrograph requires that the prism operate at minimum deviation, or that the incidence angles of the input and output beams (at a particular wavelength) at their respective prism faces are equal to one another. Such a disposition results in the image of the source slit being free from astigmatism. Unfortunately, at the time of writing this facility had not been successfully implemented on the prism component, and so the prism will need to be configured manually;
3. Retrieving the new optical axis following passage through the prism: the task of positioning the decollimating lens is made far simpler if the specifying coordinates are referenced to the deviated optical axis. This will be a test of the **RefAlign** component;
4. Decollimating and focussing the output beam: the final imaging lens is required to bring the dispersed and collimated beam to a focus. As in 1), this facility will be provided by the solve capability of the lens component. The positioning of this lens is much simplified with the introduction of the **RefAlign** component. Finally, the air space between the decollimating lens and the detector will need to be adjusted such that the detector is accurately placed at the focus. The **Spacer** component provides a solve that will ensure that this is the case.

The final test for IRIS is probably the most rigorous and concerns the modelling of a simple spectrograph conforming to the Littrow design. This system particularly will test the capabilities of the Double-Pass components in an environment where the optical axis is deviated by the presence of a refractive prism (see Figure 10.5).

The verification of the model, and the program code that supports it, will be undertaken in stages, each one corresponding to those items listed above. Data from the raytrace is made available on-screen within a text window that has been designed solely for this purpose. The screen-shot in Figure 10.1 shows this window with the caption 'Textual Output'. The data in this window, created directly from the raytrace routines, will be compared to the results obtained from direct calculation. The final test will provide a set of results in spot-diagram form (also shown in the above figure), and these will be compared to the spot-diagram analysis performed by Zemax on an identical system.

Since the primary impetus for this work had its origins in spectrograph analysis,

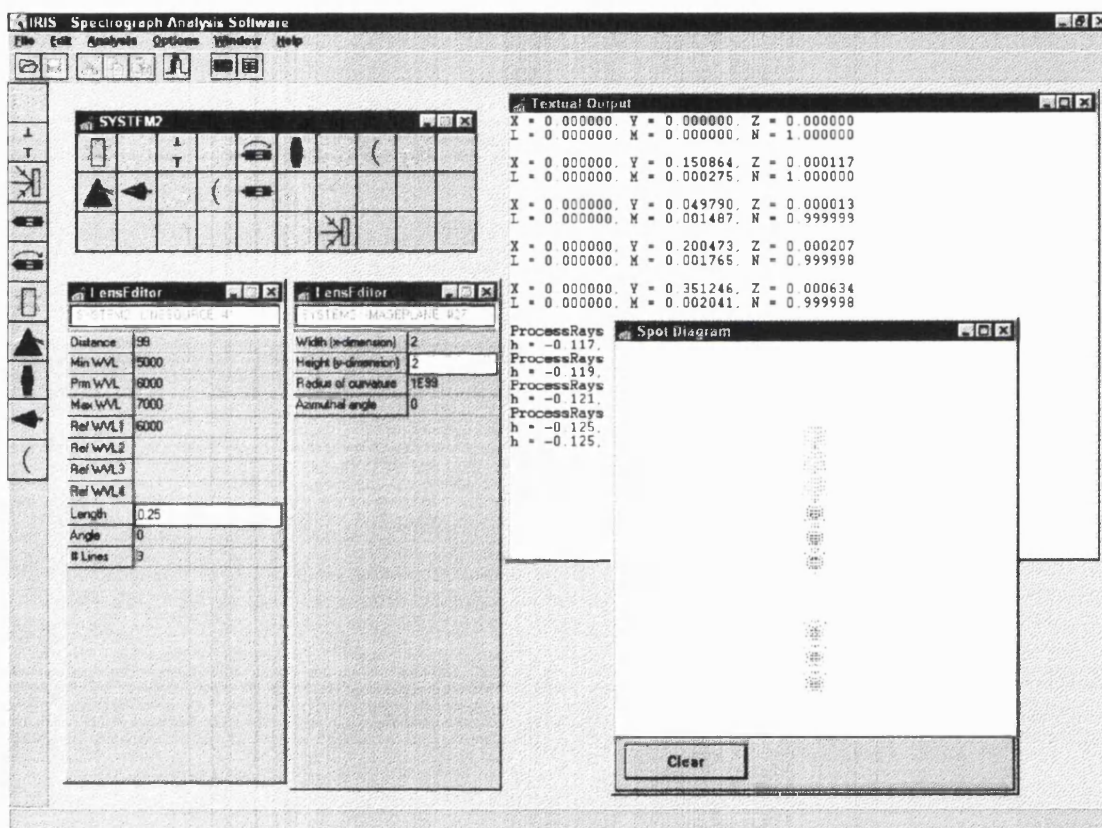


Figure 10.1: The IRIS Program, showing analysis of Littrow Mount

both finite rays and principal rays have been implemented. It is the latter rays that will provide spectral information when they are created at differing wavelengths. Thus, when the source component is edited to provide several spectral lines, in actuality several sets of principal finite rays (each set comprising of three individual rays) will be created at the required wavelength interval, and where each principal ray in a chromatic set will correspond to the two ends and the centre of each spectral line. Alternatively, the main purpose of the finite rays is to provide a set of rays (approximately 100) that are constructed to *fill* the entrance pupil. These *pupil-rays* are created for each of the principal rays mentioned above, i.e. a set of pupils-rays for each wavelength and point on the object slit. Pupils-rays are employed in the creation of the traditional spot-diagram analysis, a feature that will be exploited in the final test.

10.1 Raytracing Through Simple Lenses

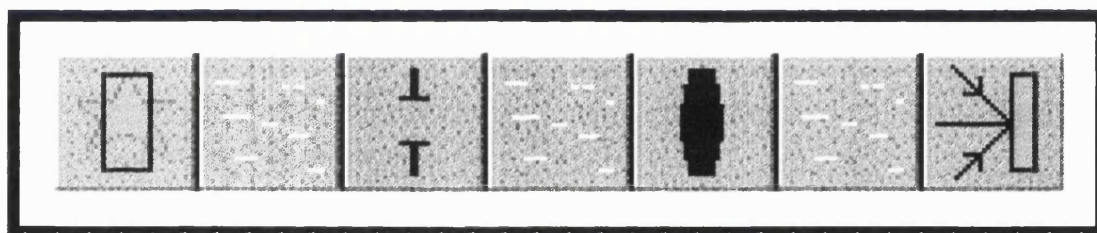


Figure 10.2: Arrangement of Components for Test #1

The first test involves those components shown in Fig. 10.2, comprising a line source, an air space, an iris (stop), a spacer, a lens, another spacer and a detector (image plane) component. The source and lens components are configured as follows:

SOURCE (Component #1)

Distance	= 99.0mm
Minimum Wavelength	= 486.132 nm (F line)

Primary Wavelength	= 587.561 nm (d line)
Maximum Wavelength	= 656.272 nm (C line)
Reference Wavelength1	= 587.561 nm (d line)
Slit Length	= 2.00mm
Slit Angle	= 90.00deg
Number of Lines	= 1

LENS (Component #5)

Thickness	= 0.0mm
Diameter	= 10.00mm
Glass	= BK7
Radius #1	= 1E99 (plano)
Radius #2	= 1E99 (plano)
Solve Type	= u_marg
Solve Value	= 0.00

...while the remaining components are set-up as below:

Component #2, Air Space, Thickness	= 1.00mm
Component #3, Stop, Aperture Radius	= 5.00mm
Component #4, Air Space, Thickness	= 0.00mm
Component #6, Air Space, Thickness	= 20.00mm
Component #8, Detector, Radius	= 9E99mm (plano)

The salient points of the above system may be summarised in the following way:

1. the source is positioned 100mm from the lens stop;
2. the slit is 2mm in length and is orientated to be coincident with the local x -axis;
3. the entrance pupil radius is 5mm;
4. the stop is coincident with the first surface of the lens, which is a plane surface;
5. the second surface of the lens has a solve placed on it; this solve will alter the radius of curvature of this surface to provide a marginal paraxial ray with a convergence

angle of $u = 0$, i. e. the diverging beam that originates from the source is collimated (made parallel);

6. the lens has zero thickness, which for the purpose of a paraxial raytrace is referred to as a *thin-lens*.

The raytrace is initiated by right-clicking the source component and selecting ‘Trace Lines’ from the pop-up menu. Automatic generation of output results by each component as the rays progress from one component to another is available to the user by ensuring that the menu item ‘Options — Diagnostics’ is checked; when this is so, each component will report the state of both paraxial and finite rays to the Textual Output form that is supplied by IRIS as default.

The first result of interest concerns the solve placed on the second surface of the lens. Raytracing of any component with a solve will automatically cause the appropriate parameter to be re-calculated so as to satisfy the solve criterion, and in this case the surface is adjusted to provide a collimated beam as the rays leave the lens. The new radius of this surface is obtained by dropping the lens onto the Lens Editor, which shows a value of -51.68mm, where prior to the raytrace it was 9E99mm. That this is the correct value may be shown by using the following thin-lens formula (see Ref. [11], p35, Eq. 3.56):

$$\frac{1}{f} = (n - 1) \left(\frac{1}{r_1} - \frac{1}{r_2} \right) \quad (10.1)$$

...where $1/f$ is the inverse of the focal length (equivalent to the lens power, K), $1/r$ is the inverse of the surface radius (or curvature, c) and n is the refractive index of the lens at the wavelength at which the rays are traced. In the test configuration used above, $f = 100\text{mm}$, $1/r_1 = 0$ and $n = 1.51680$. Substituting these values into the above equation results in $r_2 = -51.680$, in exact agreement with IRIS.

The paraxial raytrace results at the detector (component #7) are also given below:

Ray Type	Height(mm)	Angle(rad)
Principal	-0.20	-0.01
Marginal	5.00	0.0

We may glean several important facts from this data: firstly, the principal ray angle is 0.01 radians, which corresponds to the subtense of half the slit length at the lens, i. e. $1/100$;

secondly, the principal ray height at the detector is -0.20mm, equivalent to the product of the final air-space thickness and the principal ray angle, $20 \times -0.01 = -0.2$; and thirdly, the marginal ray height is 5.0mm, the same value as the entrance pupil radius, as it should be if the beam is collimated. Collimation of the beam is also indicated by the value of the marginal ray angle ($= 0$).

In addition to paraxial raytrace results, IRIS also provides finite raytrace data at the detector for those finite rays generated by the source. Since the source was configured to provide just one spectral line, then only three finite principal rays are traced. The finite raytrace data from the previous run is shown below.

Ray	X	Y	Z	L	M	N
Top	0.2	0.0	0.0	0.01	0.0	0.999950
Middle	0.0	0.0	0.0	0.0	0.0	1.0
Bottom	-0.2	0.0	0.0	-0.01	0.0	0.999950

The above data corresponds very closely with the paraxial data presented in the previous paragraph. Again, the principal ray heights for the top and bottom of the slit are identical to the paraxial case ($X = \pm 0.2\text{mm}$), while the L -direction cosine is also identical ($L = \pm 0.01\text{rads}$). In general, ray data from equivalent paraxial and finite rays do not agree with one another, but in this case the finite principal ray angle with the optical axis is so small (0.01 rads) that it may indeed be considered as a paraxial ray.

10.2 Configuring the Prism

Having established that the beam exiting the lens is collimated and that the principal rays emanating from the slit are in accord with our expectations, the next step will be to introduce a dispersing prism. This time, instead of tracing a monochromatic set of slit rays, a chromatic set will be employed. This set will comprise of three rays at the minimum wavelength, as specified by the source component, and three rays at the maximum and mid- wavelengths. Since we are concerned only with the mean deviation and the chromatic dispersion of the beam, only ray data for the (slit) centre rays will be presented.

The layout of the new scheme is shown in Fig. 10.3 while the prism specification is

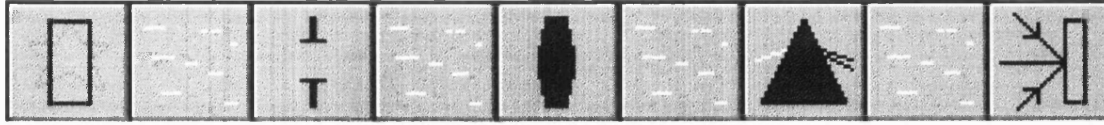


Figure 10.3: Arrangement of Components for Test #2

described by the following:

PRISM (Component #7)

Angle	= 45deg
Thickness	= 20.0mm
Tilt	= 22.5deg
Rotation	= 0deg
Glass	= BK7
Reference Wavelength	= 1

The initiation of a raytrace produces the following results:

Ray	X	Y	Z	L	M	N
MinWvl	0	-12.931	0	0	-0.469048	0.883173
PrmWvl	0	-12.761	0	0	-0.463406	0.886146
MaxWvl	0	-12.686	0	0	-0.460907	0.887449

The angular deviation of the axial ray at the mid-wavelength is given by $\arccos(0.886146) = 27.60712^\circ$, while the angular dispersion is given by $\arccos(0.883173) - \arccos(0.887449) = 0.52689^\circ$. These same values may be derived theoretically from the following equation ([17]) that predicts the angular deviation of a ray (D), given the incidence angle on the first face (θ), the prism apex angle (α) and the refractive index of the prism (n).

$$D = \theta + \arcsin \left[\sin \alpha \sqrt{n^2 - \sin^2 \theta} - \sin \theta \cos \alpha \right] - \alpha \quad (10.2)$$

Substituting the following values: $\alpha = 45^\circ$, $\theta = 22.5^\circ$ and $n = 1.51680$, the mean angular deviation is calculated as $D = 27.60711^\circ$, virtually identical to the result obtained from the raytrace data. Using the same equation again, but this time calculating the value of $D_{\lambda_{min}} - D_{\lambda_{max}}$ leads us to a value for the chromatic dispersion, δD_λ :

$$\delta D_\lambda = 27.97235 - 27.44548 = 0.52687^\circ$$

...essentially identical to the raytrace obtained value of 0.52689° , allowing for six-figure round-off errors in the above calculations.

10.3 Decollimating and Focussing of the Output Beam

The collimated beam that is leaving the prism is now required to be focussed onto the detector. This poses two problems:

1. the beam has been deviated by the prism and makes an angle of approximately 27° with respect to the original optical axis. The placement of a lens in this beam is complicated by the fact that the positional component becomes a function of both y and z , and we also have to consider the generalised directional vector of the lens surfaces. IRIS was never conceived to handle such a problem directly, but rather it was designed to circumvent these difficulties by the introduction of the **RefAlign** component (see Ch.8, Section 4). Introducing this component directly after the prism will create a *new* optical axis that is coincident with the original axial ray at the ‘tuned’ reference wavelength;
2. the focussing of the beam necessitates that the beam convergence angle is set to the required value and that the detector is accurately positioned at the lens focus. The convergence angle will be controlled by a solve on the second surface of the lens, while the lens-detector spacing will be adjusted by another solve on the intervening spacer component.

The final layout of the system is shown in Figure 10.4 where the additional components commence after the prism. Notice the RefAlign component following the prism and the decollimating lens. The lens has been configured almost identically to the collimating lens,

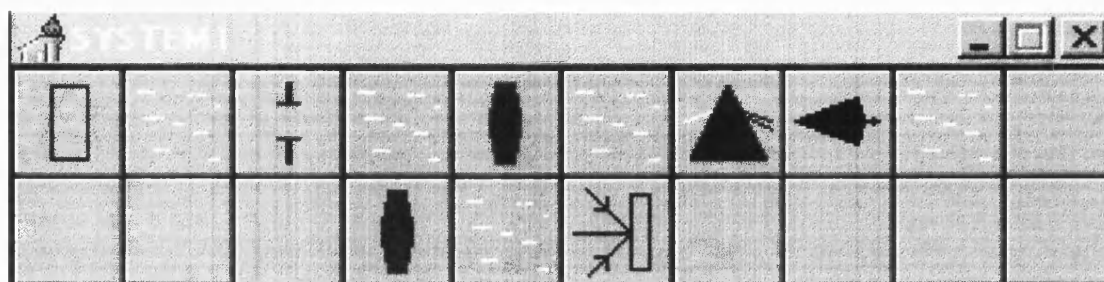


Figure 10.4: Arrangement of Components for Test #3

but instead a solve has been placed on the final surface to give a convergence angle of -0.05, while the following air-space has a thickness solve to ensure that the paraxial marginal ray will intersect the next component with a ray height of 0.0. These components are specified thus:

REFALIGN (component #8)

RefWVL = 1

LENS (Component #13)

Thickness = 0.0mm
Diameter = 10.00mm
Glass = BK7
Radius #1 = 55.68
Radius #2 = 1E99 (plano)
Solve Type = u_marg
Solve Value = -0.05

AIR SPACE (Component #14)

Solve Type = h_marg
Solve Value = 0.0

A raytrace of this system produces the following results:

PARAXIAL RAYTRACE

hbar = -1, ubar = -0.00468 h = 0, u = -0.05

FINITE RAYTRACE

	X	Y	Z	L	M	N
RefWVL1 (=PrmWVL)						
1	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000
MinWVL						
2	0.994026	-0.640987	0.000000	0.004533	-0.004904	0.999978
3	0.000000	-0.636090	0.000000	0.000000	-0.004867	0.999988
4	-0.994026	-0.640987	0.000000	-0.004533	-0.004904	0.999978
MaxWVL						
5	1.002466	0.277282	0.000000	0.004610	0.002122	0.999987
6	0.000000	0.282161	0.000000	0.000000	0.002160	0.999998
7	-1.002466	0.277282	0.000000	-0.004610	0.002122	0.999987

At the paraxial level the system may be viewed as a simple unit magnification system (magnification = -1), and so the paraxial raytrace results at the detector are not completely unexpected. The original slit height (referred to the optical axis) has a value of 1.0, and so at the image plane (detector) of a unit magnification system (magn. = -1) the corresponding image height will be -1 (hbar = -1). Similarly, the image height of the marginal ray will be 0.0 (h = 0), since it is at the focus, and the convergence of the same ray will be equal and opposite to the value at the start of the raytrace (u = -0.05). These results confirm the paraxial expectations of this system.

Of greater interest are the data for the finite raytrace. The first ray (#1) is an axial ray at a wavelength equal to the Primary or Mid-Wavelength, while the remaining two sets correspond to the ray-sets at the minimum (2,3,4) and maximum (5,6,7) wavelengths specified by the source component. The first and last ray of each set originates from either end of the slit, and the middle ray originates from the centre of the slit and initially travels along the optical axis. The points to note are:

1. ray #1 shows an intersection at the origin of coordinates of the detector, and a direction vector equal to the $+z$ -axis. In other words, though ray #1 was deviated by the prism through an angle of 27° the RefAlign component has successfully managed to establish a new optical axis that is coincident with the reference ray, as required;
2. the rays that originate from the ends of the slit (2,4 and 5,7) have an x -coordinate value of approximately ± 1 , indicating that the finite raytrace results are very similar to the paraxial result for **hbar**;
3. it is noticeable that the image of the slit is slightly smaller at the short wavelength end of the waveband (0.994) than at the longer wavelength (1.002). This is attributable to an effect known as *transverse chromatic aberration* and originates from the absence of achromatism¹ provided by the two lenses;
4. a phenomenon that is common to spectrographs is known as *slit curvature*. This effect is evident in the above results if we compare the y -values at the end and centre of each slit image. For a 2mm slit (± 1) the difference between the centre and end of the slit (*sagitta*) is 0.004897mm, but if we perform the same raytrace with a 20mm slit then the difference becomes 0.484mm. The parabolic relationship between slit-length and sagitta ($\text{sagitta} \propto \text{slit-length}^2$) is clear and consistent.

10.4 Raytracing a Littrow Spectrograph

This final test includes all the above features but with the added complexity of simulating a double-pass over two components. The general form of the system under test is known as the Littrow Mount, and comprises a doublet lens and prism in close proximity to one

¹Achromatism – the ability to bring rays of more than one wavelength to the same focus

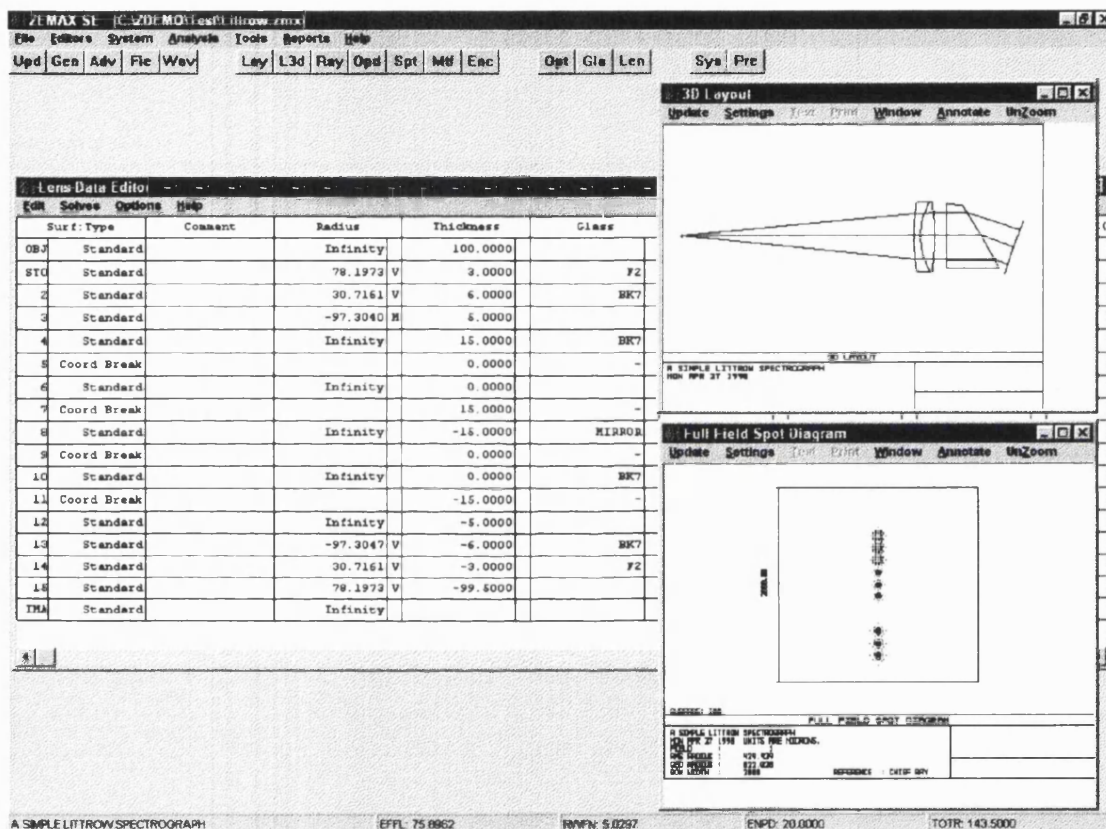


Figure 10.5: The Zemax Program, showing analysis of Littrow Mount

another followed by a retro-reflecting mirror, as shown in the '3D Layout' window of Figure 10.5. A line-source is placed at the focus of the doublet, where subsequently the rays leave the doublet as a collimated beam. The rays then pass through a 30 degree prism which, for ease of laying out, is positioned such that the first surface is normal to the beam. A reflecting surface is then placed normal to the deviated beam which causes the rays to re-traverse the same paths back to the source. The normal form of Littrow spectrograph deviates from that shown in that it is the second prism surface, and not the first, that is normal to the beam, and it is usually this surface which supports the reflective coating that provides the means for auto-collimation.

The implementation of such a system within the Zemax framework is shown in the 'Lens Data Editor' window of Figure 10.5. In keeping with a surface based model and in order to incorporate the various surface tilts and axial decentrations Zemax, in common with other similar programs, introduces a new surface type which Zemax refers to as a '**Coord Break**'. Unlike a regular optical surface, the **Coord Break** only utilises the following six parameters: three decentrations and three tilt angles. The first **Coord Break** is shown as Surface #5 and corresponds to the tilt of the second prism surface, while the **Coord Break** of Surface #7 implements a new optical axis, which in this case corresponds to the deviated axial ray. Following the reflection from Surface #8, Zemax requires that those prior surfaces that are in double-pass mode be entered in reverse order, where all surface radii, tilts and thicknesses have the negative value of their former value. This procedure for handling surfaces in double-pass mode is common to all lens design programs that are based upon the surface model. Naturally, the need for negative thicknesses in the Littrow design is determined by the necessity or not of requiring that the drawing of the system correspond to the actual physical layout, which in this case is true.

In contrast, the IRIS program is able to represent the construction of the Littrow system in purely pictorial form, as a sequence of icons (Lens Buttons) laid out upon the Lens Form, as shown in Figure 10.6. Three points worth noting in this scheme are: 1) due to the absence of a distinct doublet component, the doublet lens is represented by the combination of a singlet lens, a spacer having the appropriate glass as the medium, and a surface; 2) the double-pass sequence commences with the singlet lens (Lens #6) and culminates with the mirror surface (Lens #15), i.e. that lens sequence within the limits defined by the double-pass components at Lens #5 and Lens #15; 3) a RefAlign

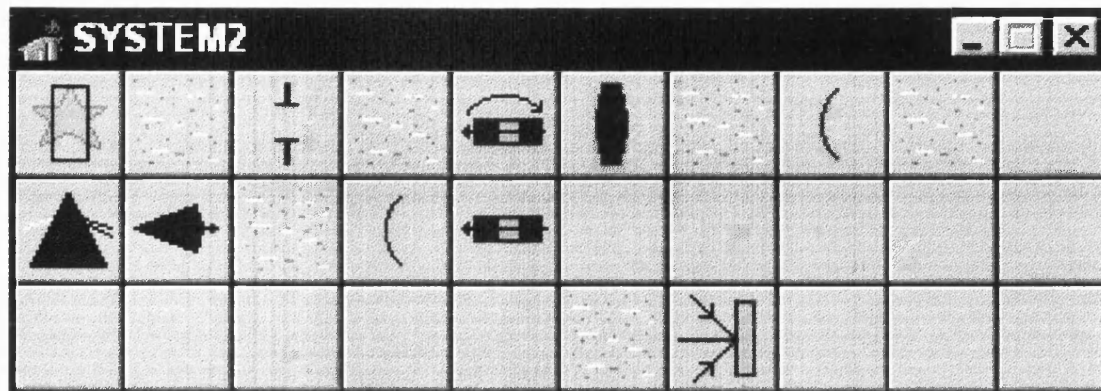


Figure 10.6: The Littrow system as represented by IRIS

component is shown at Lens #12, performing the task of re-aligning the optical axis with the deviated axial ray following deviation by the prism (Lens #11). Unlike the Zemax approach, a double pass sequence such as required by the Littrow mount is implemented almost automatically by the simple expedient of introducing the double-pass components at the appropriate places within the system. It is completely unnecessary to repeat the double-pass sequence in reverse order nor, as in this case, is it necessary to manually configure a coordinate change arising from the deviated axial ray following refraction by the prism.

The raytrace results for the Littrow mount as modelled by Zemax and IRIS are presented in Figure 10.7. Source specification is: line height (y-dirn) = ± 0.125 mm, three equally spaced points per line, and wavelengths of 500nm, 600nm and 700nm. Note that the prism dispersion is parallel to the line source, at variance with the normal approach where the line (slit) is perpendicular to the dispersion. The square frame surrounding each set of spot diagrams has side-lengths in the image plane of 2.0mm. In order to present the spot diagrams with some internal structure it was necessary to defocus the image plane by 0.50mm towards the collimating lens. The similarity between the two sets of spot diagrams is very marked, although very slight discrepancies are more than likely due to the differing number of rays traced in each case and the differing ray distributions at the pupil plane. Despite this it is quite clear that IRIS does indeed produce results that are in agreement with those obtained by Zemax.

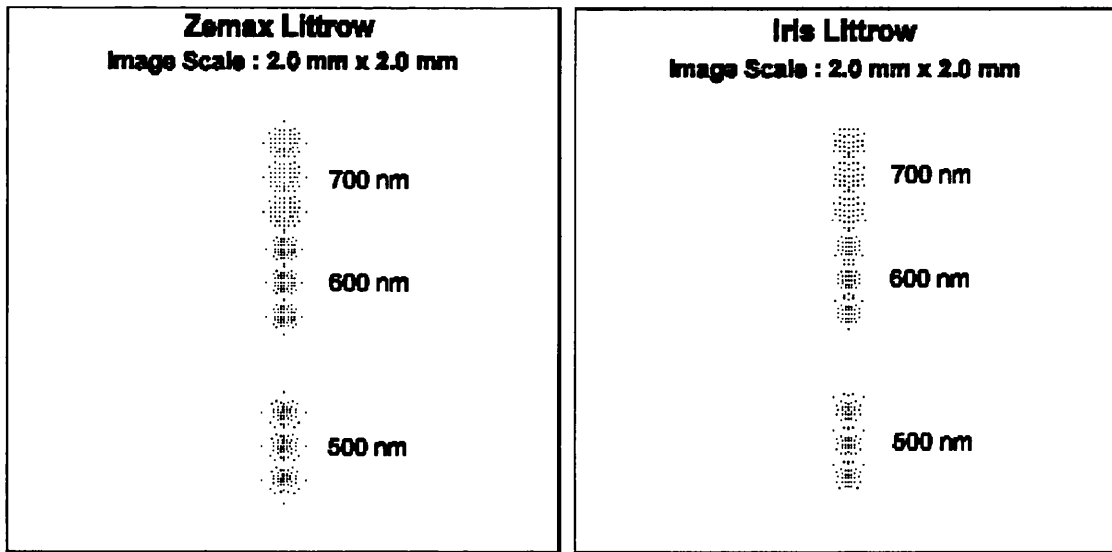


Figure 10.7: Full-field spot diagrams of the Littrow Mount as obtained by Zemax (LHS) and IRIS (RHS).

10.5 Conclusion

The purpose of this chapter was two-fold: firstly, to establish that the component based model could be made to function in a cooperative manner, and secondly, to determine if the numerical procedures embedded within these components did actually provide the correct results. Though both have been successfully demonstrated, it is quite clear that, independent of the verity of the numerical results, the outstanding feature is that, as far as optical design is concerned, a component based model is a reality. The very simple spectrograph that was explored at the beginning of this chapter provided ample verification that each component successfully played its rôle as part of a team, although some executed their parts more purposefully than others. The RefAlign component was particularly important in removing one of the major obstacles that dog most conventional design programs, namely the construction of non-axisymmetric optical systems. What unfortunately the reader cannot properly appreciate is the ease with which this system was synthesised, and only made possible through the advanced user interface provided by the IRIS program. What applies to a simple spectrograph can also be made to apply to much more complex optical systems, as was ably demonstrated in the previous section. Ordinarily, whereas the modelling of a non-axisymmetric system with double-pass sections would pose considerable difficulty, IRIS has been able to show that the component based

model offers unique opportunities in the way that strategies may be devised to rationalise otherwise complex configurational issues.

Chapter 11

Future Trends

Lens analysis, in its modern interpretation, has been with us for almost 50 years, and although classical optics has remained relatively unchanged during this period, the means of analysis has undergone one revolution after another. The modern day computer shares little with its ancestors other than the same common purpose. In other respects the modern desk-top computer utilises resources and power that formerly would have been unthinkable. Current trends in systems integration include multimedia (sound and vision, static and dynamic) and virtual reality (VR). It is probably the latter that promises the most since it offers a model of abstraction that closely approaches that which we, at a conscious level anyway, are aware of. Unfortunately, bridging the gap between our sense of sight and our other senses remains an obstacle, and is only practicable in certain specialised activities such as molecular modelling, architectural design and games playing, for example. Similarly, of equal interest during the past decade is the work being undertaken in the area of artificial intelligence, or AI. Such curiosities as neural networks and genetic algorithms are now making deep inroads into those problem areas once considered to be mathematically intractable. Is it possible, or even plausible, that our lens components could be imbued with a degree of intelligence? Another area that may offer further possibilities is that of optimisation. Not only does the new system model offer an enhanced and improved way of constructing complex optical assemblies, but it might also provide us with new features that current optimisation programs could profitably employ.

11.1 The Object Model

One of the aspects of object modelling that has not been overly discussed up until this point is that of *events* or *messages*. The event is a necessary feature in software systems such as the Windows operating system, where many components exist in virtual isolation from one another except for the ability to send, or post messages to each other. The operating system handles these messages and ensures that they are sent to the correct window or application. When received by a component, the message is interpreted and acted upon in a manner dictated by the software instructions already embedded within the procedure handler. Events are generally numerous and prolific in any application that provides a graphical user interface in which elements are in various states of flux. Some typical events include MouseDown, Click and Paint, where the first two events obviously relate to mouse events, and Paint is a message directed at a component that will result in the component redrawing itself upon the graphical display unit.

While most events that occur in a GUI-based operating system relate to visual events, there is no real reason why they cannot also be generated in a non-visual situation. Consider the case of a ray of light traversing several lens elements and then encountering an obstruction such as an aperture stop. This loss of a ray through total absorption would ordinarily be signalled to the user as a reduction of throughput which is only realised when all rays have been traced and counted. A different approach would be to generate an event, such as 'wm_RayObstructed', which would be handled by some procedure that could, for instance, display the number of obstructed rays as and when they occurred.

The advantage of this approach is that the program code of the raytrace procedure does not have to be interrupted by a call to another procedure, with all the ensuing changes at the machine code level, but instead, the raytrace code proceeds normally and the message will only be handled (in the case of a *post*) when the program allows. This will occur when either the program has stopped executing and is awaiting some input from the user, or when the program allows Windows to dispatch its messages through a specific call to an application-specific method called *ProcessMessages* as in Delphi. If the lens system object is provided with a default message handler for this particular message, then the program developer has the option of either ignoring the message and letting the default handler handle it, or alternatively, the default handler may be overridden with a handler that will undertake some other more useful action when activated, as in updating a

display component, as discussed above. In actuality, messages and events are far easier to implement than the above description would seem to indicate, since most of the hard work is undertaken by Windows and the methods that have been inherited from the base-class window.

We have also seen in earlier chapters how the lens object is designed to be a client of the container class, or form, and through this relationship the lens is able to query its host as to its own position relative to other lenses. This feature was exploited by the lens sequencing algorithm so that rays may be passed from one lens to another in the correct sequence. The same feature may also be employed in much more complex processes, for instance, in determining whether one lens will mechanically interfere with another, possibly as a result of a surface change. Consider the scenario where a lens is required to undergo a bend that, if allowed, may result in a mechanical collision with a neighbouring lens. Prior to this operation the lens will undertake a function call to both bounding spaces that a bend is to be undertaken, of the form:

```
function CanSurfaceChange(newsurface:TSurface):boolean;
```

The response to this query is obtained by interrogating each neighbouring lens component in turn, and the change is allowed only when both neighbours report no mechanical interference.

11.2 Improving the Picture

How can virtual reality carve in-roads into such fields as physics and mathematics when in most cases there is no phenomenological equivalent in a world that is so dependent upon our senses? This problem is one that faces all developers of models that are essentially non-experiential in nature, but success, or even partial success brings with it the opportunity to extend our frontiers of knowledge, notwithstanding the much hoped for financial rewards that occasionally accompany an evolutionary or innovative product. Inevitably we must initially look towards the object model that describes the system or application, with a view to encapsulating within it the behaviour and characteristics that are to be ascribed to the end-product. It is these latter two features which will largely shape the final experience of the end-user. It has long been the case that the user's interaction with a

software application, as executed on a modern computer, is largely a visual one. There are historical reasons why this should be the case, and also very good logistical reasons. The fact is that, as the saying goes, ‘a picture is worth a thousand words’, and in trying to describe the state of any particular system it is much more efficient in terms of computing power if a graphical representation is employed. This argument holds not only for the silicon processor at the heart of every computer, but also for the organic processor that we call the human brain. In terms of complexity and information content, a perceived image may convey large amounts of both qualitative and quantitative data, and which may be realised by the perceiver in the time it takes for one heartbeat to be executed.

While such techniques as 3-D rendering and virtual reality may be appealing to the less technically inclined, it is dubious whether they offer any real benefit for the purpose of interactive editing and design in an environment such as optics, where the precise configuration of a system is of more importance than a simpler generalised configuration. In such a situation it may be more expeditious to seek an alternative graphical model that will map onto the problem domain in a more efficient and constructive manner. We have already seen how an iconic representation of lens elements aids the user in constructing a lens system where the individual elements markedly differ from one another in size and shape, but is it possible to extend the model to include a more realistic graphical interface? One possibility is to accept that, for quantitative work anyway, VR cannot offer more than we already have, but for the semi-quantitative work undertaken by a mechanical engineer, for example, who is interested in such matters as the mechanical construction and fit of the optical assemblies, then the presentation of a VR image which may be manipulated (zoom, pan, magnify, etc.) would be of great benefit.¹

11.3 Intelligence and Optimisation

The lens model that has been discussed at great length so far will also allow for further and more complex forms of computation to be undertaken. In particular, one of the more often used forms of numerical calculation that is targeted at a lens system concerns optimisation, that is the automatic design of a system of lens elements according to some predefined search criterion, where the end goal is specified as a set of desired performance

¹Drawing options such as rendering and VR are becoming increasingly common in the more up-market optical analysis programs, e. g. OPTICAD by Focus Software, Inc.

parameters. In previous chapters we have discussed the various properties and behaviours that our lens object has been imparted with, and so at this point we may continue in the same vein by considering the manner in which a lens object might actively contribute towards the more general task of system optimisation. In addition, we may also ponder upon the possibility of there being a scheme whereby lens objects will actively cooperate with one another in order to achieve this same goal.

One of the more maturing technologies that offers great hope for the future is the field of artificial intelligence, of which the derived benefits have only become tangible through the ever increasing power and sophistication of computer hardware and the analysis techniques that have blossomed in the wake of the computer revolution. The term ‘Artificial intelligence’ or AI, is to some extent a misnomer, since in all its guises it is essentially an attempt at simulating certain processes that either occur in nature or that we, as intelligent organisms, undertake as a prelude to decision making. There are three fundamental technologies that fall under the umbrella of artificial intelligence:

1. **Knowledge Based Systems (KBS)** – sometimes referred to as *Expert Systems*.

This form of simulated AI attempts to develop solutions to problems by employing some kind of reasoning, rather than simple mathematical functions. Usually, this is done in the form of *IF condition THEN conclusion* rules, but other techniques are also employed. Typically, Knowledge Based Systems consist of two parts: the knowledge base (with the IF-THEN rules) and the inference engine (the algorithm to reason with these rules);

2. **Neural Networks** – Artificial neural networks are biologically inspired forms of computation that represent a departure from conventional forms of computing, in that many discrete artificial *neurons* are highly connected and operate in parallel. Based upon emerging knowledge on the operation of the brain belonging to the higher animals, the neural network takes its inspiration from the neurons, dendrites and axons that form the basis of the brain. In doing so, it is hoped that the neural network will exhibit those useful characteristics that we associate with the human brain: learning, generalisation and redundancy. The first two characteristics have been demonstrated, but the third is still awaiting further research prior to exploitation;

3. **Genetic Algorithms (GAs)** – While Charles Darwin[18] postulated upon the origin and survival of the (carbon based) species, similar theories are currently be-

ing employed to develop a system based upon purely algorithmic entities. Genetic algorithms are general heuristic global optimisation methods that operate upon a population of individuals that are the representations of solutions in the search space. Each solution is represented by a series of *genes* within the individual. Pairs of individual solutions bear *children* inheriting their features (sometimes with mutation). Weak individuals die, while strong individuals live for a longer period. The closer a solution represented in the genes to the desired optimum, the stronger that individual is considered to be. The GA has been particularly successful in solving those normally intractable problems such as the ‘Travelling Salesman Problem’.

All of the above AI technologies have demonstrated successful application in one or more fields of endeavour. In particular, KBS is usefully employed in those areas requiring high levels of logical reasoning and deduction, notably in current computer programs that aim to supplement and possibly enhance medical diagnoses. On the other hand, the application of neural networks favours those problem areas that are difficult to model in a mathematical sense, and yet would benefit from the ability of the network to be trained to a level of proficiency in that same area. Thus we see neural nets being developed that might, for instance, control industrial processes with a view to maximising efficiency, authenticate signatures or retinal maps during real-time financial transactions, or even provide economic and trade models that might be used by world governments to anticipate and react to change. GAs have similar applicability to neural nets, and in some areas such as financial modelling² the GA approach is supplanting the neural net, due to its ability to modify itself to changing conditions.

Lens optimisation has been a major area of research for almost half a century[19], where the principal techniques have been based upon either *least squares*[20] or *orthonormalisation*[21]. More recently, in a move to expand the search space to wider (global) areas, mathematical techniques that enable global optimisation (cf. *simulated annealing*, [16], p366) have been utilised. The problem of applying global optimisation methods to lens design is complicated by the very convoluted solution space of even the simplest of lens systems but, never-the-less, several commercial concerns (Optical Research Associates, Sinclair Optics Inc., etc.) have managed to integrate a global search algorithm into their current offerings.

²PAPAGENA – *Programming Environment for Applications of Parallel Genetic Algorithms*, European Community Esprit III – Project 6857

It is possible, though highly speculative, to consider a scenario where the TLens object is “blessed” with a certain amount of intelligence, possibly originating from an embedded neural net or genetic algorithm. The purpose of such a leap in functionality is to provide the lens with a self optimising capability. This faculty alone is not sufficient to enable the lens system as a whole to achieve an optimised state of correction, since each lens element will only be aware of the localised state of correction, and will be indifferent to the global picture. What is required is a lens that is able to cooperate with other lenses of the system during optimisation, in a manner that benefits the system and not the individual. How this might be accomplished could very well be the subject of another research paper!

Another area that might prove amenable to the AI approach is the initial selection and the later modification of systems during optimisation. It is well known among lens designers that some lens configurations are better suited to a particular task than others. For instance, we may consider a group of camera lenses, the standard F2/50mm for example, to share the same operating characteristics and performance, and though many companies are engaged in the manufacture of such lenses, the design adopted is predominantly of the double-Gauss variety. The same applies to many other similar examples, where various unique problems areas have been solved in similar ways. What a lens designer looks for when trying to identify a particular lens scheme (assuming that it is optimally corrected for the task at hand) is i) the general power configuration, ii) the bend of certain key lenses, and iii) the general glass configuration. In effect, this amounts to *pattern recognition*, a task that is most suited to the neural network. It is conceivable to imagine that a suitably trained neural network might be allowed to arbitrate over the varied and numerous configurational changes that an optical system will undertake during routine optimisation. For example, the network might realise that a given system is approaching that of a known and desirable configuration, and may provide extra weighting toward such a direction where the optimisation might indicate otherwise.

11.4 Classical Optimisation

One other interesting possibility that arises from the component model is the opportunity it allows us to modify the normal variable set that we ascribe to a lens. During the process

of optimisation, the most common design variable is curvature, but this variable will also alter the two principal parameters that we associate with aberration control: power (K) and bend (B). For example, the thin-lens equations that define these two quantities are given below:

$$K = (n - 1)(c_1 - c_2) \quad (11.1)$$

$$B = \frac{c_1 + c_2}{c_1 - c_2} \quad (11.2)$$

...and it is obvious that a change in either c_1 or c_2 will result in corresponding changes to both K and B . In some systems, e. g. the zoom lens, the power and possibly the glass configuration may already have been determined and all that is required is to optimise the system based upon lens bendings alone whilst retaining the original power distribution. This procedure will be difficult to achieve in the majority of design programs and usually requires that the designer implements a solve after each lens, setting the solve value to the original convergence angle of the paraxial marginal ray as it leaves the final surface. Though not a very satisfactory solution, it is perhaps the only method available for implementing bend changes. On the other hand, a lens component could be easily modified to include a **bend** property which would allow the following syntax:

```
ALens.Bend := ALens.Bend * 1.01;
```

...where the intention is to increase the current bend by one percent whilst maintaining the lens power. The converse is also possible:

```
ALens.Power := ALens.Power * 1.01;
```

...where the power of the lens is to increase by one percent while preserving the value of the lens bending. With these new properties in place it is now possible to undertake design modifications or even optimisation based upon a more intuitive and natural understanding of aberration control; instead of specifying simple curvatures as our design variables, we may directly employ lens bending and power to be part of the variable set. As an example of what benefits are to be derived from this approach, we shall take the case of the Cooke triplet photographic lens and consider the design approach we might take based upon these new variables. The Cooke triplet comprises three air-spaced lenses, two positive power singlets surrounding a negative singlet, where the stop is usually located close to

the latter lens. This combination provides a total of eight variables: two separations, three lens bendings and three lens powers; precisely enough to enable all primary aberrations to be controlled, provided the glass types are suitably chosen. The variable parameters may be divided into two groups, first the three powers and the two separations and secondly the three bendings. Four parameters of the first group control the overall power, the two chromatic aberrations and the Petzval sum; the remaining parameter of the first group (the choice is not critical) and the three bendings together control spherical aberration, coma, astigmatism and distortion. The two procedures defined above are applied iteratively until the lens system meets the specified requirements. These procedures are, in principle, all that is required to design the Cooke triplet (and with slight modification, many other systems) but in practice are largely ignored, for the reason that the power and bending parameters are not easily accessible or manipulated by the designer and are not supported by current analysis and optimisation programs. Until this situation is reversed, optical designers will continue to discuss aberration control with respect to bend and power, but will be unable to directly put such ideas into practice due to the limitations of the surface based model. On the other hand, a wider acceptance of the component model will, in principle, allow both the theory and the practice of optical design to merge into one another, and so offer the designer a closer and more intuitive relationship with the software tools that form part of the armoury of every optical specialist.

It is impossible to say what the future holds for lens design programs in general, or even if there is room for improvement, and if so, what form it will take. The ideas mentioned above will almost inevitably be incorporated into tomorrow's design programs, if not at the time of writing. One thing is certain though, the massive explosion in computing power and its ready accessibility will enable all people with open and enquiring minds to explore their own ideas in as much detail or sophistication as is deemed necessary. The Internet is one such vehicle whereby people may explore possibilities, and computer programming is another. Both are ultimately limited by our own imaginations and our ability to carry them through to fruition.

Chapter 12

Conclusion

This thesis is built upon three pillars that are crucial to a correct understanding of the work undertaken, and that are fundamental to the design of the IRIS program. The first of these concerns the nature and importance of models in general, and of the optical model in particular. Models determine the way we think about things, but occasionally they also prevent us from developing a better understanding. The belief that the world was flat probably originated thousands of years ago, and yet it has persisted up until the Renaissance period. Though it seems ridiculous today, this rudimentary world model was mostly successful because people then had a very localised and limited knowledge of the world that they lived in. It was only when the great explorers of the fourteenth and fifteenth centuries began to expand our horizons, both geographically, culturally and scientifically, that the world began to change for all time.

The optical model has, to some extent, shared a similar history to the flat world model. Ask any optical designer what he or she considers to be the internal model employed by lens design programs, and the answer will be, if forthcoming: “I don’t know. Is there one?”. The truth is that the model has always been subservient to the algorithm. It has always been the case that the mathematics and program code that enables raytracing to be undertaken is of central importance. In contrast, this thesis has largely ignored the mathematics of raytracing, although it is omni-present throughout the IRIS program, and instead it has highlighted a more fundamental and basic aspect of computer-based lens analysis: the identification and implementation of a suitable model.

The second pillar upon which this thesis has been built concerns the ideas and man-

ner of implementation associated with object oriented programming languages. While it is true to say that virtually any computer language may be used in the development of a sophisticated system model, it is only since the advent of true OOP languages that language constructs have been available to actively support this same process. Section 4.2 was devoted entirely to a review of three mainstream OOP languages. Visual Basic was shown to be an admirable development environment for prototyping and interface design, but was ultimately let down by a poor turn of speed and a dearth of true object oriented features. On the other hand, C++ (in the guise of Microsoft's Visual C++) provided both a high run-time speed and a wealth of OOP and other language features; it truly deserves to be the leading language for critical and complex development programmes. Notwithstanding the above, the author, in common with some leading computer magazines, has instead cast his vote for Borland's Delphi, generally recognised as the most productive of all rapid application development tools without sacrificing run-time speed or ease-of-use. The true worth of using such a tool as Delphi can only be fully appreciated when one moves up from a much simpler language.

The final pillar is the visual interface, which in scientific programs in general is probably not as well developed as it ought to be. In the author's opinion, such programs have suffered from excessive emphasis on the algorithm at the expense of the interface. This is particularly evident in the majority of lens design programs that still employ the spreadsheet data entry method, although one company at least (OptikWerks) has now adopted a component-based interface that is in keeping with current trends, and that hopefully is also a reflection of an underlying component based model. It is worth noting that all the major application programs (MS Word, Excel, Visual Basic, Delphi, etc.) and operating systems (Windows 95, NeXT, XWindows) are advanced from one version to the next by not only providing more features and enhancing existing ones, but by improving the functionality of the user interface. For instance, the evolution of the Windows OS has been most obvious in the changes made to the desktop. The early versions of Windows (v3.XX) were dominated by the Program Manager, whereas the current version of Windows 95 provides the user with the option of applications being selectable from the desktop (as applets), Windows Explorer and a drop-down menu (obviating the need for a Program Manager). The next version will be Windows 98 (codename 'Memphis') which will herald a new form of interface based upon Web browser technology. If lens design programs are to embrace those new ideas and concepts that are found in the man-machine

interface then they could do worse than emulate Microsoft's current offerings.

IRIS has attempted to redress the balance by showing the reader just what is possible. Through the use of rapid application development tools, interface design is no longer as onerous a task as it used to be. But in reality, companies that have been developing lens design programs over many years are unlikely to make the transition overnight, even if they wanted to. The truth is that in changing from a procedural/spreadsheet program to a full component based equivalent, only the numerical algorithms are likely to survive the conversion. In addition, scientists can be rather conservative when it comes to 'tampering' with their favourite program, and may not appreciate any 'unnecessary' changes. It is probably for these reasons that the only component based design program to come onto the market during the last five years is still OptikWerks, a product that has been developed from scratch.

In concluding this thesis, it is probably worthwhile to reconsider the major points and claims that the author has made:

1. Most design programs employ a simple surface based model that, in itself, limits the complexity and richness which we intuitively associate with an optical system.
2. If we wish to improve on this state of affairs then it becomes necessary to develop a new model which encapsulates within itself those features and properties that we would consider necessary.
3. Computer models are best described in terms of a language that actively supports the notion and manipulation of those entities that comprise the model. Object oriented languages, such as C++ and Delphi, are identified as such languages.
4. A real world model is developed that attempts to describe an optical system in terms of its major components and assemblies (lenses, mirrors, prisms, etc.); it uses the optical toolbox and lens workbench as its rôle model. This is called the *component based model*.
5. The structure of a generalised lens component is developed whilst also examining how more complex components are arrived at. Supporting structures such as the lens system are also described, and it is shown how these objects are simply integrated into standard GUI objects such as buttons and forms.

6. Rays are also treated thoroughly. Rather than being the poor relation in this tightly knit team, rays are extended to support vector and matrix operations that are shown to greatly enhance their performance and usefulness.
7. All models and systems must function according to a consistent set of rules, and the component based model is no exception. Rules are developed that determine the validity, or not, of an optical system configuration. These are extended to the notion of a lens sequence, or the order in which each component comes into play during a raytrace procedure. These ideas are then used to describe some of the fundamental system processes, such as pupil location.
8. All of the above elements are then combined together to produce a computer program called IRIS, which is then used to test the model in the simple case of a spectrographic camera. Raytrace results provided by IRIS show that, in all respects, the output data corresponds exactly to what is expected.

IRIS is not yet complete, and is unlikely to be further developed in its present form. On the other hand, IRIS has proved to be an excellent platform for obtaining a thorough understanding of the problems that need to be circumvented if it is to be successful at all in the future. As learning experiences go, the last three years have been truly worthwhile, for it is only through tackling real-world problems that any substantial learning takes place. Though object oriented programming is not, in itself, especially difficult to comprehend, the true art of any skill is in knowing how to use it in a concise and efficient manner. As a technology matures with time, the capabilities and potential of that technology become ever more apparent. So it with OOP, and Delphi. Since the time of the first release of Delphi, numerous articles, books and specialised magazines have appeared that have provided new insights into the various software features that were either poorly documented up until that time, or that were considered to be worthy of further elucidation. Much of the information acquired through these sources has direct relevance to any future redevelopment that IRIS may undergo, but while implementation details may change, for the better, the actual model will remain largely untouched.

Commercial lens design programs are usually the culmination of many years of devotion to the subject itself, and require the equivalent of many *man-years* of software development effort. As such, change is generally a very slow and incremental process. The ideas presented in this thesis are so fundamentally different to many existing program

codes that their adoption is unlikely without extensive ‘root and branch’ pruning of the original. Since this is unlikely, and probably not economic, then the more acceptable solution is to start anew. It is for this reason that I do not expect to see the adoption of the component based model in many commercial offerings, at least for a few years yet. This extended model is so new that we can expect a period of maturation to occur as developers begin to discover new possibilities and opportunities. At present, only one company is producing a lens design program that uses a similar model: OptikWerks. In my view, the visual interface is far from optimum, but with time and experience this situation may improve.

On a final note, I should add that the object based model is also applicable to many other types of problem. Science in particular is well endowed with models that attempt to describe a plethora of processes, in fields as diverse as physics, biology, chemistry and cosmology. If a physical model is well enough understood then it can also be successfully described in an object oriented software code. The only problem that remains is the design of the user interface, or how to transport the problem domain to a two-dimensional screen.

Appendix A

Glossary

Abstract Referring to a class: a class that is used only as a basis for deriving other classes; referring to a method: a method declared as abstract cannot be used in an instance of that class, but must be overridden in a descendant of the same class.

Achromatic The ability of a lens to bring rays of more than one colour to the same focus.

Astigmatism An imaging defect (aberration) of a lens system that results in tangential rays being focussed at a different plane to sagittal rays.

Ancestor An object from which another object is descended. Ancestor objects may have any number of descendents, but a descendent object may have only one immediate ancestor, a concept known as *single inheritance*. Some OOP languages, such as C++ but not Object Pascal, allow multiple inheritance, where descendent objects may have more than one immediate ancestor.

Class A synonym for an ‘object type’. A class defines the data fields and method headers of a single object, which might be descended from another object class.

Constructor A special object method that is called in order to create an instance of the class. A constructor allocates the necessary memory and initialises all properties. A class has atleast one constructor, and may have more.

Descendant Inherits all the data fields and methods from its immediate ancestor, which may have inherited properties from its own ancestor.

Destructor An object method that is called to destroy the object, releasing or deallocating the memory that the object occupied. A class may only have one destructor.

Encapsulation Refers to the manner in which object types are able to combine both data fields and methods.

Gaussian Optics That domain of an axially symmetric optical system in which all ray angles are very small and located close to the optical axis.

Inheritance Objects can inherit the properties (data fields and methods) of other objects. See also *Ancestor*.

Instance A variable of an object type, similar to the way an integer variable is an ‘instance’ of type **Integer**.

Instantiate *verb* – to create an instance of an object.

Method A function/procedure definition in a class. Methods describe the operations that objects *know* how to perform.

Method Implementation Contains the actual statements that flesh out an object type’s definition. The implementation of a method is shared by all object instances of the object type.

Object A variable belonging to a class or object type.

OOP Object Oriented Programming, a classification of computer languages that allows for complex class definitions that may encapsulate both data **and** procedural types; also supports *inheritance* and *polymorphism*.

Object type An object type defines an object’s data fields and methods. An object type is a definition for an object, not the object itself. See also *Class*.

Paradigm In programming, a paradigm is a conceptual model for building computer programs. An important paradigm in computer science history is the concept of structured programming.

Paraxial See *Gaussian*.

Polymorphism Refers to the ability of objects to have different forms/behaviour but still be type-compatible with existing code.

Sagittal Any plane that is not coincident with a tangential plane.

Static method Class methods declared as static (default in Object Pascal) are not virtual methods and so may not be overridden by descendant classes.

Tangential Any bilateral plane of symmetry associated with an axially symmetrical optical system.

Type A declaration; informs the compiler of new data types.

Virtual method Class methods that are declared as virtual may be overridden, or redefined, in descendant classes. It is through the virtual directive that polymorphism and polymorphic objects come about.

B i b l i o g r a p h y

- [1] Walker, D.D., Bingham, R.G., Diego, F.D., Fish, A.C. (July 1991) *Design Considerations for an Echelle Spectrograph* UK Large Telescope Technical Report, edited by Roger Davies.
- [2] Wirth, N. (1985) *Programming in Modula-2*, SpringerVerlag.
- [3] McCracken, Daniel D. (1965) *A Guide to Fortran 4 Programming* John Wiley & Sons Inc.
- [4] Albrecht, R.L., Finkel, L. and Brown, J. (1978) *BASIC 2nd Edition* NewYork: Wiley Interscience.
- [5] American National Standards Institute, 1983, *IEEE Standard Pascal Computer Programming Language*, New York: Institute of Electrical and Electronics Engineers, Inc.
- [6] Kernighan, B. W. and D. M. Ritchie (1988) *The C Programming Language, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall.
- [7] Strousrup, B. (1986) *The C++ Programming Language*, Reading, MA: Addison-Wesley.
- [8] Henderson, K. (1996) *Database Developer's Guide with Delphi 2*, Sams Publishing/Borland Press.
- [9] Jensen, K. and Wirth, N. (1975) *Pascal User Manual and Report*, SpringerVerlag.
- [10] Maxwell, J. (1972) *Catadioptric Imaging Systems*, p 49, Adam Hilger Ltd.
- [11] Welford, W. T. (1974) *Aberrations of the Symmetrical Optical System*, Academic Press.

- [12] Born, M. and Wolf, E. Third Edition (1965) *Principles of Optics*, Pergamon Press.
- [13] Conrady, A. E. *Applied Optics and Optical Design*, Part1: 1929, Oxford University Press; Part 2: 1960, Dover Publications.
- [14] Buchdahl, H. A. (1968) *Optical Aberration Coefficients*, Dover Publications.
- [15] Welford, W. T. and Winston, R. (1978) *The Optics of Nonimaging Concentrators*, Academic Press.
- [16] Press, W. H., *et al.* (1990) *Numerical Recipes in Pascal*, Cambridge University Press.
- [17] Hecht, E. and Zajac, A. (2nd Printing, 1977) *Optics*, Addison-Wesley.
- [18] Darwin, C. (1859) *Origin of the Species*, Odhams Press.
- [19] Rosen, S., and Eldert, C., 1954, *Least Squares Method for Optical Correction*, JOSA, **44**, 250.
- [20] Wynne, C. G., 1958 *Lens Designing by Electronic Digital Computer*, 1, Proc. Phys. Soc., **73**, 777. /
- [21] Grey, D.S., 1963, *Aberration Theories of Semiautomatic Lens Design by Electronic Computers*, JOSA, **53**, 672.
- [22] Born M. & Wolf E., *Principles of Optics*, Third Edition, pp401-407, Pergamon Press
Optical Astronomy, pp183-188, ed L.B.Robinson, SpringerVerlag, Berlin 1987