TEKNILLINEN KORKEAKOULU
Sähkö- ja tietoliikennetekniikan osasto

Hannu Napari

Magic Lens user interface in virtual reality

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 19. 4. 1999.

Työn valvoja	Professori	Tapio Takala
Työn ohjaaja	Professori	Tapio Takala

26 -05- 1999
TKK Sähkö- ja tietoliikennetekniikan kirjasto
Otakaari 5 A
02150 ESPOO
20591

TEKNILLINEN KORKEAKOULU Diplomityön tiivistelmä

Tekijä:	Hannu Napari	
Työn nimi:	Magic Lens user interface in virtual reality	
Päivämäärä:	19. huhtikuuta 1999	Sivumäärä: 91
Osasto:	Sähkö- ja tietoliikennetekniikan osasto	
Professuuri:	Ohjelmistojärjestelmät	Koodi: Tik-76
Työn valvoja:	Professori	Tapio Takala
Työn ohjaaja:	Professori	Tapio Takala

Keinotodellisuusympäristöjen tai ylipäätään kolmiulotteista grafiikkaa hyödyntävien sovellusten yleistyessä käyttöliittymien kehittäminen niihin muodostuu entistäkin tärkeämmäksi. Nykysin yleisesti käytössä olevien käyttöliittymien metaforat eivät välttämättä sovellu suoraan keinotodellisuusympäristöön. Syinä tähän ovat erilaiset fyysiset interaktiovälineet sekä immersiivinen ympäristö.

Tässä työssä tarkastellaan erilaisia kolmiuloitteiseen ympäristöön soveltuvia käyttöliittymäkomponentteja. Näistä keskitytään Xerox:in kehittämään Magic Lens - käyttöliittymäkonseptiin. Magic Lens - konsepti kehitettiin alunperin kaksiulotteisen käyttöliittymän interaktiovälineeksi, mutta sama metafora on myös siirrettävissä kolmiulotteiseen virtuaaliympäristöön.

Magic Lens - implementaatio on kaksiosainen. Ensimmäisessä osassa on yhdistetty kaksiulotteinen Magic Lens - työkalu kolmiulotteiseen visualisointiin. Tätä yhdistelmää voidaan käyttää virtuaalimaailmassa tapahtuvan navigoinnin apuvälineenä sekä siirtymissä että etsittävien kohteiden spatiaalisen sijainnin paikallistamisessa.

Toisessa osassa Magic Lens - käyttöliittymäkomponentti on siirretty kolmiuloitteiseen virtuaaliympäristöön. Tässä tapauksessa Magic Lens on teksturoitu geometrinen objekti osana virtuaalitilaa. Toteutus hyödyntää SGI:n O2 - työaseman muistiarkkitehtuuria siten, että implementaatio on mahdollisimman tehokas. Magic Lens - konseptia on myös laajennettu siten, että virtuaalitilaan ei enää varsinaisesti upoteta käyttöliittymäkomponentteja vaan ne korvataan tilaan projisoidulla tekstuurilla, Magic Light:illa.

Avainsanat: käyttöliittymät, keinotodellisuus, tietokonegrafiikka, 3D, Magic Lenses, VRML, Open Inventor, Java

HELSINKI UNIVERSITY OF TECHNOLOGY — Abstract of the Master's Thesis

Author:	Hannu Napari	
Name of the Thesis:	Magic Lens user interface in virtual reality	
Date:	19. April 1999	Number of pages: 91
Department:	Electrical and Communications Engineering	
Professorship:	Software engineering	Code: Tik-76
Supervisor:	Professor	Tapio Takala
Instructor:	Professor	Tapio Takala

User interface research and development for virtual environments gains more importance as 3D graphics and virtual reality applications become more common. The standard 2D user interface metaphors are not necessarily the most efficient for virtual reality applications. The different user interface devices and immersive virtual environments can cause problems with the traditional user interface components.

This thesis reviews first different user interface components suitable for 3D environments. The Magic Lens - metaphor, first introduced by Xerox is then discussed in detail. The Magic Lens concept was originally meant for 2D interaction, but it can also be used for 3D environments.

The Magic Lens implementation is presented in two parts. The first part is a combination of a 2D Magic Lens - interface and a 3D visualization. This combination can be used for navigation and wayfinding in a virtual environment. It also helps the user to better visualize the spatial location of the objects of interest.

In the second part, the Magic Lens user interface component is implemented as a 3D widget, embedded in the virtual environment. The Magic Lens is a textured, geometric object in the virtual space. The implementation utilizes the SGI O2 workstation unified memory architecture to provide an efficient solution for 3D Magic Lenses. The lens concept is also expanded by using a projected texture, Magic Light, instead of a geometric lens object.

Keywords: user interfaces, virtual reality, computer graphics, 3D, Magic Lenses, VRML, Open Inventor, Java

ALKULAUSE

Tämä diplomityö on tehty osana NetVE-tavoitetutkimushanketta (Networked Virtual Envoronments). Erityisesti haluan kiittää professori Tapio Takalaa, joka tarjosi minulle mielenkiintoisen työympäristön sekä mahdollisuuden saattaa opintoni päätökseen.

Kiitokset johdattamisestani tietotekniikan pariin kuuluvat opettajilleni Aulis Puttoselle sekä Pekka Bergmanille, joiden innostamana päädyin harrastamaan tietotekniikkaa noin 18 vuotta sitten. Vuosien saatossa harrastuksesta kehkeytyi huomaamatta työ.

Haluan myös muistaa Silicon Graphics OY:tä, joka suuntasi kiinnostukseni tietokonegrafiikkaan ja keinotodellisuuteen. Tämän lisäksi Silicon Graphics tarjosi mielenkiintoisen ja opettavan työpaikan huippuluokan tietotekniikkaratkaisujen parissa viiden vuoden ajan.

Kiitokset myös DIVA- ja CAVE- projektien työntekijöille sekä avusta että riittävästä työmäärästä huolehtimisesta: Andy, Ebu, Janne, Rami, Tapsa, Lauri, Tommi, Jarmo.

Haluan kiittää vanhempiani tuesta ja huolenpidosta elämäni sekä opiskeluni alkutaipaleella. Rakkaimmat kiitokset avovaimolleni Niinalle epäsäännöllisten työaikojeni ja elämäntapojeni sietämisestä sekä siitä, että elämässäni on muutakin kuin tietotekniikkaa.

Espoossa 19. huhtikuuta 1999

Hannu Napari

TABLE OF CONTENTS

1 Introduction 1

2 3D User Interfaces 4

2.1 Mapping of input devices 5

2.1.1 2D input devices 5

2.1.2 3D input devices 5

2.2 Interaction with 3D scene 7

2.2.1 Direct manipulation widgets 8

2.2.2 Selections and picking 11

2.2.3 Visualization widgets 13

3 Magic Lenses and Toolglass 15

3.1 Composition of Lenses 17

3.2 Implementation alternatives 19

3.2.1 Benefits and disadvantages of the different filtering implementations 20

3.3 Examples of the 2D Toolglass metaphor 21

3.3.1 CorelDraw 22

3.3.2 Kai's Power Tools 22

3.4 Examples of the 3D Toolglass metaphor 23

3.4.1 Virtual Tricorder 23

3.4.2 Volumetric Magic Lenses 25

4 2D Magic Lenses and 3D visualization 28

4.1 2D Magic Lenses as navigational aids 28

4.2 Mapping of the 2D lenses for 3D space 30

4.3 Implementation of 2D lens and 3D visualization 31

4.3.1 Java 31
4.3.2 VRML 35
4.3.3 Liquid Reality 37
4.3.4 Communication between the Magic Lens and VRML applets . 40
5 Flat 3D Magic Lenses 41
5.1 The relationship between the lens image and scene 42
5.2 Magic Lens semantics in multiuser environments 44
5.3 Projected Magic Lenses - Magic Lights 45
5.4 Functionality of 3D lenses 48
5.4.1 Changes in rendering 48
5.4.2 Changes in lens frustum 48
5.4.3 Visual queries 48
5.4.4 Modification of the geometry 49
5.4.5 Temporal modifications 49
5.4.6 Object manipulation or Toolglass 49
5.4.7 Using Magic Lenses As Portals 50
5.5 Composition of lenses 51
5.6 Implementation 52
5.6.1 Unified Memory Architecture - UMA 54
5.6.2 OpenGL Digital Media Buffers 56
5.6.3 Virtual Graphics Context - VGC 59
5.6.4 Forming the lens image 61
5.6.5 Picking and other interaction 72
5.6.6 Open InventorTM 73
5.6.7 Extensions to OpenInventorTM 74

6 Conclusions 81
6.1 2D Magic Lenses and 3D visualization 81
6.2 3D Magic Lenses 82
6.2.1 Magic Light 83
6.2.2 Augmented Reality 83
7 References 84
Appendix A: Bugs in the O2 OpenGL Implementation 89

SYMBOLS AND ABBREVIATIONS

AGP	Accelerated Graphics Port
API	Application Programming Interface
AR	Augmented Reality; Simultaneous combination of the real, perceived world and a synthetic world.
AWT	Abstract Window Toolkit
CAVE	Cave Automated Virtual Environment
CFD	Computational Fluid Dynamics
CSG	Constructive Solid Geometry
DOF	Degrees Of Freedom
DSO	Dynamic Shared Object
EAI	External Authoring Interface
FOV	Field Of View
GUI	Graphical User Interface
HUD	Heads Up Display
IO	Input / Output
KPT	Kai's Power Tools
LCD	Liquid Crystal Display
LOD	Level Of Detail
LR	Short for Liquid Reality
MIMO	Model-In Model-Out
MRE	Memory and Rendering Engine
OI	Short for Open Inventor^{TM}
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant

RGB	Red, Green and Blue
RGBA	Red, Green, Blue and Alpha
SGI	Silicon Graphics Incorporated
UMA	Unified Memory Architecture
URL	Universal Resource Locator
VGC	Virtual Graphics Context
VRML	Virtual Reality Modeling Language. A file format standard for 3D multi-media and shared virtual worlds on the Internet.
WIM	Worlds In Miniature
WIMP	Windows, Icons, Menus and Pointers
WWW	World Wide Web.

Separator - group node that saves and restores traversal state

Light

Camera

Switch

Shape

1 INTRODUCTION

This thesis is a study about 3D user interfaces for virtual reality. The main emphasis has been on implementing and developing the Magic Lens interface paradigm (Bier 1993, p. 73-80) for immersive and non-immersive virtual environments.

In most virtual reality interaction research the emphasis has been on methods that employ very sophisticated hardware devices, i.e. gesture tracking via data gloves. Most of these techniques have also been developed solely for immersive environments. Today, the advent of desktop virtual reality in the form of games and different 3D visualization software packages makes new demands for user interfaces for VR (Virtual Reality). This development has been accelerated by relatively inexpensive 3D hardware designed for personal computers and by new memory and bus architectures.

Because the normal, everyday home computers and equipment do not usually provide means for immersive environments and complex input devices because the price or cumbersomeness of such devices, the user interface software technologies are very critical to the success of VR at home, as well as for commercial or industrial VR usage. A good VR user interface (any interface, in fact) utilizes a simple, easily understandable metaphor from everyday experience.

Currently, there are no generally accepted standards or guidelines for 3D interaction. The research on the subject has shown that the traditional 2D user interface techniques do not bring any added value to virtual environments. Methods that make the interface between human and the virtual world efficient are not clear. The traditional 2D interface model, WIMP (Windows, Icons, Menus and Pointers) may not be the best solution for 3D interaction, especially with input devices that have more than two DOF (degrees-of-freedom). The WIMP-style interface also appears as a distinct layer between the application and the user interface devices. In 3D environment the interface should be embedded in the virtual environment, because the WIMP layer may distract the user and cause a loss off immersion, especially when used in combination with stereoscopic displays.

Green (1990, p. 229-235) gives the following requirements for non-WIMP interfaces:

1. High Bandwidth Input and Output
2. Many Degrees of Freedom
3. Real-Time Response
4. Continuous Response and Feedback
5. Probabilistic Input
6. Multiple Simultaneous Input and Output Streams From Multiple User

In immersive VR environments this list should be augmented by adding two-handed or other multimodal interfaces. The non-WIMP interfaces discard the traditional interface styles, but they should also allow for multiple input devices with many degrees of freedom to be used at the same time.

The tasks normally performed in 3D or VR applications according to Green are:

- Object creation, model definition
- Object selection
- Object placement and editing:

  affine transformations
  modification of surface characteristics
- Viewpoint control
- Perception:

  extracting cognitive information from an environment
- Programming:
  defining behavior of objects and relationships between objects

The first tasks that consist mainly of object manipulation can also benefit from perceptual changes of the environment, as well as the viewpoint control or navigation. The tools for direct object manipulation are out of the scope of this work, but I will

show that direct manipulation can be augmented by using perceptual filtering via the Magic Lenses interface.

Magic Lenses are transparent user interface components, which change the representation of the data visible through the Magic Lens. The lenses can be combined to form more complex filters.

The Magic Lenses interface also reduces the screen area of the user interface components by embedding the interface directly to the virtual environment. If a GUI (Graphical User Interface) component reserves a lot of space from the viewport, it can cause loss of immersion. Because Magic Lenses are transparent widgets, they do not essentially reduce the available viewport region. Three dimensional widgets can also be freely positioned in the virtual environment, so there is no need for a HUD (Heads Up Display) type of usage for the Magic Lenses widgets, thus freeing even more viewport space for the main visualization. If the Magic Lenses interface is used to filter the environment to extract cognitive information or to change the rendering parameters, it limits the changes to a spatial region. This limit helps the user to maintain the immersion, because drastic, sudden changes in the environment do not happen.

In this work I will show how the Toolglass / Magic Lenses interface, which is a kind of mix between the WIMP and non-WIMP interfaces, can be implemented to fill these requirements.

This thesis will first present an overview of 3D user interfaces, followed by a more accurate overview the Magic Lenses / Toolglass interface and the existing implementations of the metaphor. After the reader is familiar with the concept of Magic Lenses, I will to present the 2D Magic Lenses work made by the author, followed by the 3D Magic Lenses implementation.

2 3D USER INTERFACES

Most of the research on 3D user interfaces has been concentrated on 3D widgets, i.e. GUI components for manipulation of 3D objects. This is perhaps caused by the problems in 3D input and display devices. For example, tracking, interpreting and mapping the user's movements to enable interaction with the virtual world with many degrees of freedom creates many problems with both the hardware and software implementations. There is no such thing as a generic 3D GUI toolkit. The current user interfaces for 3D interaction typically present one or more views to the virtual environment, augmented by 2D user interfaces, like menus, sliders and dialogs, taking no advantage of the third dimension. The direct interaction with the 3D scene is limited primarily to interactive viewing, selection, translation and rotation.

Most of the paradigms and metaphors for 3D interfaces are less developed than the 2D counterparts. Some of the metaphors for 3D interaction are borrowed directly form the standard 2D interface concepts. These metaphors can find their place in 3D interaction, but they are inappropriate for immersive environments or applications that require direct manipulation. On the other hand, bringing in the real world tools and devices to help in the interaction misses an important point. For example, if the user wants to modify an object, bringing in a toolbox where the user would pick up a virtual spanner and then continue to use the spanner on a virtual object would make the interaction tedious. Who wants to operate in a virtual environment using the clumsy tools from the real world? In certain simulations this type of interaction might be the correct way of doing things, but for example, in scientific visualization the user interface should be as direct as possible.

The metaphors for 3D interaction must draw from the user's everyday experience, and the interaction techniques must take advantage of the possibilities of 3D. In VR systems, the interfaces should also account for multiple simultaneous inputs/outputs and multiple participants. The input devices may have multiple degrees of freedom, or they can be based on speech recognition. On the output side, the feedback might be provided in the form of visual, auditory and haptic displays.

2.1 Mapping of input devices

2.1.1 2D input devices

In desktop virtual reality the IO devices have generally a limited number of DOFs. The standard devices are the mouse and the keyboard. The interaction with a virtual environment requires that this 2D input is somehow mapped to the 3D environment. To augment the interface, professional animation and modeling tools usually have a possibility to attach other 2D IO devices, such as buttons and dials to reduce the number of functions and operations attached to a single device. To interact with the scene, the user typically selects an operation and then manipulates the desired object.

For example, rotational movement can be mapped to the mouse X- and Y-axes directly. Because a mouse-like device can have only two degrees of freedom, the mapping of the mouse movements is changed via a keyboard interface, a menu or using the buttons on the mouse. This causes extra levels of interaction. In a dial box user interface device, the different dials have separate functions: some of the dials can control the rotation, while others are used for translation, scaling etc.

As the movements are mapped to less than three dimensions, these interaction devices can maintain a high level of accuracy. The user is able to modify only one or two parameters at the same time. This applies to true 3D widgets as well, if appropriate constraints are utilized.

2.1.2 3D input devices

3D input devices, or devices that have more than two degrees of freedom are more versatile interaction devices than the previously mentioned 2D input devices. Examples of these type of devices are the SpaceBall (SpaceTec), different motion trackers and data gloves. They make it easier to combine basic operations to a single interaction event, i.e. translating and rotating an object at the same time.

In general, the device must still be mapped to the 3D environment. The mapping can include scaling of the motion of the device to fit the sensitivity to the expectations of the user. Also, the accuracy of the device can suffer if the movement is not constricted to a lesser number of degrees. For example, if the user wants to scale an object only in the X-Y plane, the operation must be limited to that plane to provide any accuracy, especially if a motion tracking type of system is used.

Using devices with more than two degrees of freedom can also be a difficult task. Experience has shown that even a relatively simple device, SpaceBall, can be too difficult for most users. Most people can not do several operations at the same time and instead do the tasks sequentially. This is probably because of the lack of appropriate feedback from the multiple operations.

An interesting variant of 3D interface technologies is the Lego™ Interface Toolkit (Ayers 1996, p. 97-98) displayed in Figure 1 on page 7. The problem with the current physical interface devices is that they are not reconfigurable. The physical configuration is the same, regardless of the software interface. Ayers presents a solution for rapid prototyping of different physical interface devices. The user can freely construct a suitable interface device by using three basic components: rotational sensors, linear

sensors and push buttons. The physical components of the interface are ordinary Lego™ blocks on which the sensor devices are attached.

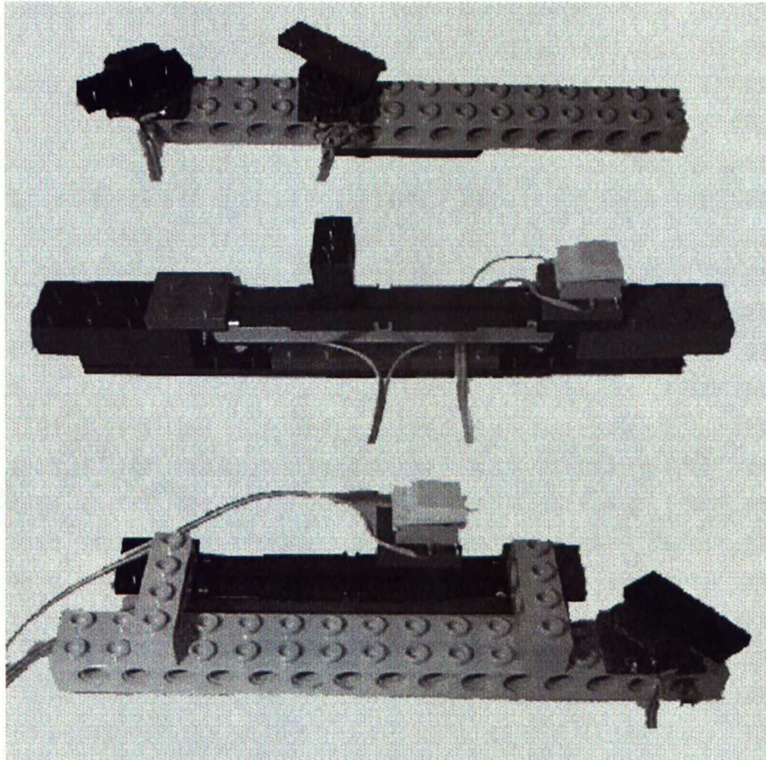


Figure 1 **The Lego interface toolkit samples (Ayers 1996, p. 97-98). The interfaces in the figure are constructed for manipulation of the Virtual Wind Tunnel widgets (see Figure 5 on page 14)**

2.2 Interaction with 3D scene

To augment the interaction with 3D scenes, we also need a set of manipulation devices, or widgets. The widgets make it easier to select different types of operations for the selected objects. 3D widgets are functional, geometric objects embedded into the scene and linked with the selected objects. The user interacts with the widgets through manipulation involving motion, selection and simple gestures.

As with 2D widgets, the 3D widgets should be responsive, using easily understandable paradigms or metaphors. An additional requirement is that the size of the widgets should be non-intrusive because the widgets are embedded in the view of the user.

The metaphors created by 3D widgets can also have properties that are not generally implemented in 2D widgets: the widget can either continue to operate after the interaction has stopped or it can stop at the same time. For example, a trackball metaphor can be thought to be given an impulse and it will continue to rotate the object indefinitely. In the three dimensional space the widget metaphor can be in addition commutative or non-commutative. This means that the result of the interaction can be dependent only on the end result of the interaction or it can depend on the path of the movement.

2.2.1 Direct manipulation widgets

Direct manipulation widgets are components which directly affect the properties of the object to which they are linked to. These widgets help the user to visualize the mapping of the IO devices to the virtual environment. In practise, there is no generally accepted set of metaphors for 3D widgets, in contrast to 2D GUI guidelines and toolkits.

Usually the direct manipulation widgets provide an interface for the basic geometric transformations: translation, rotation and scaling. The widgets can provide additional geometry to help visualizing the possible operations or they can use the geometry in the scene. Open Inventor (Strauss 1992, p. 341 - 349) defines these as:

- Draggers, which use the geometrical representation of the selected object as the manipulator
- Manipulators, which present their own geometry around the selected object

Open Inventor manipulators implement widgets that allow more than one operation with a single widget. Examples of Open Inventor widgets can be seen in Figure 2 on page 10. The widgets in the figure are:

1. Handlebox - Translation and scale factor editing.
2. Jack - Rotation, scaling and translation.
3. Trackball - Rotation and scaling
4. Transformer - Interface for changing scaling, rotation and translation.

The user can also change the default geometrical representation of the different manipulators.

The operations are grouped logically, but the downside is that the widgets are big. On the other hand, different operations can be done without changing the widget type or device mapping explicitly.

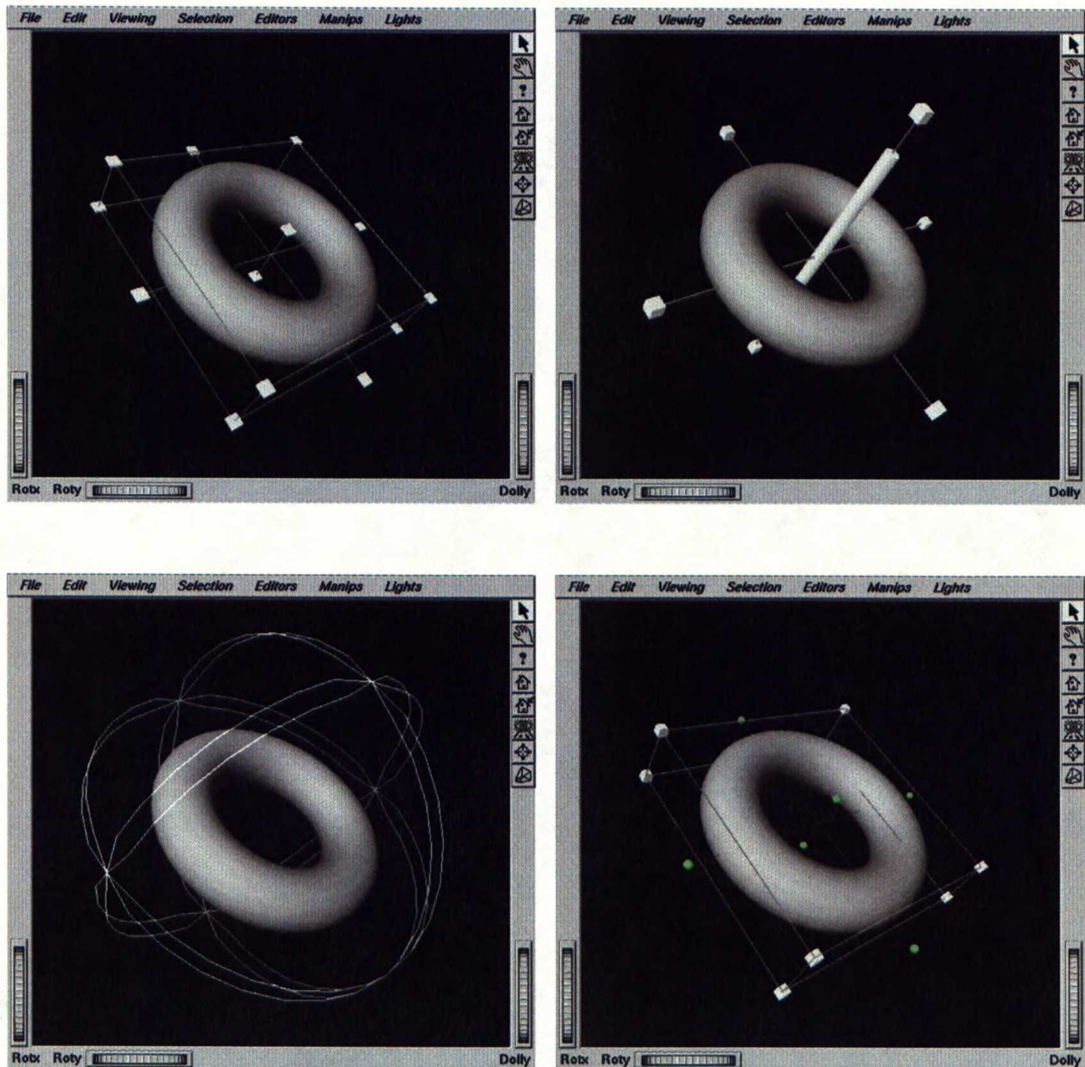


Figure 2 Examples of direct manipulation widgets (Open Inventor manipulators): A handlebox, a jack, a trackball and a transformer.

A relatively widely used 3D toolkit, VRML (VRML Architecture Group, 1997), has draggers with the following IO device mappings:

- Translation - The translation maps the motion of the pointer device to motion in a plane.

- Spherical mapping - This dragger type translates the pointer device to a virtual trackball, mapping it to a rotation.
- Cylindrical mapping - Cylindrical mapping maps the motion to a rotation around a specified axis.

2.2.2 Selections and picking

Picking, or dynamic target acquisition refers to target selection tasks, such as picking a 3D point or object. To make 3D interaction work, the selection process must be very natural and unobtrusive.

Ray-pick. Ray picking means that a ray is cast to the scene to pick objects or points on the objects. The picking ray can be formed by casting the ray from the eyepoint through the location specified by the pointer device to the scene. The ray can also be formed by using a spatial input device to define the ray and then modifying the ray interactively. For object-level picking the ray must be checked for intersections between the ray and the bounding volumes of the objects.

Cone-pick. Cone picking is a variation of the ray pick method, but instead of casting a ray to the scene a cone or a cylinder is used for the intersection testing. This can ease the picking task because the method is more tolerant to errors or noise in the input device. The picking cone can also be visualized more clearly than the picking ray. Forsberg (1996, p. 95-96) presents a method to interactively manipulate the aperture of the picking cone to enable the users to change the accuracy of the pick.

Volume-pick. The "Silk Cursor" method by Zhai (1994, p. 459-464) implements picking by utilizing a semitransparent volume for picking objects. Zhai tested the volume picking technique for picking a single object, and the results showed that the method provides enough depth cues to make the picking operation easier. However,

the method was not tested for picking a group of objects. An example of silk cursor usage is in Figure 3 on page 12

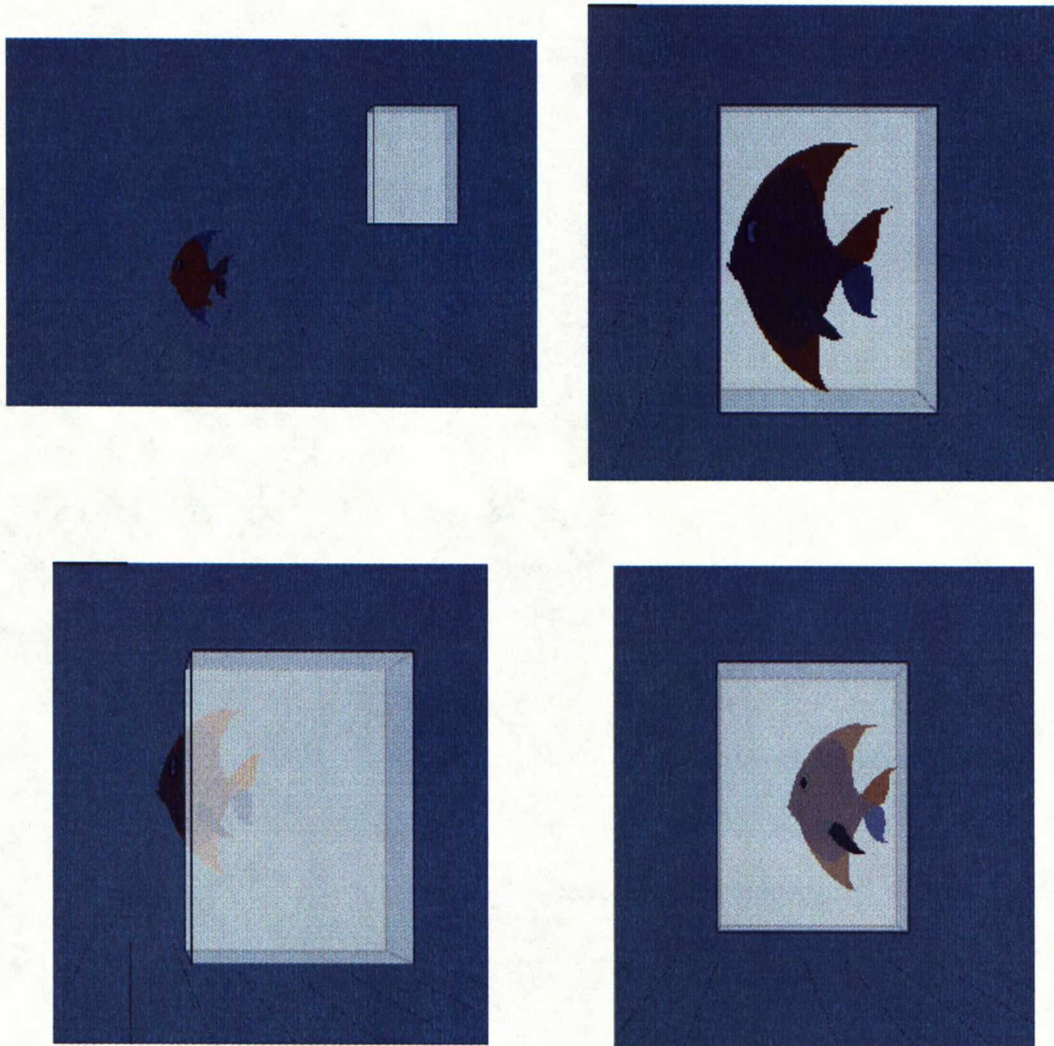


Figure 3 **Examples of Silk Cursor (clockwise from top left):**
A fish outside the silk cursor, a fish in front to the selection volume, partly inside the selection volume, behind the selection volume

Picking by orientation. Forsberg (1996, p. 95-96) also describes an orientation based picking method. This method uses the orientation of the interface device to single out

operations. See Figure 4 on page 13 for an example. The user can select only the vertical or horizontal parts of the widget at a time. This lessens the possibility of errors.

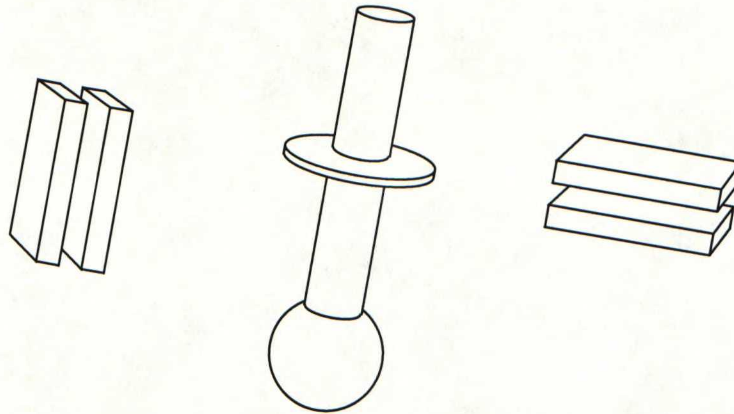


Figure 4 **Orientation based selection. The leftmost widget would select the rake frame, the rightmost would select the cylindrical slider.**

2.2.3 Visualization widgets

Aside the direct manipulation of the 3D scene, the visualization is also a problem in virtual reality. Visualization widgets implemented by Herndon (1994, p. 69-70) are targeted to interactive visualization of CFD (Computational Fluid Dynamics) data sets. The widgets are true 3D widgets, or 3D objects that each represent some properties of the visualized scene (i.e. glyphs), embedded to the scene. For different visualization needs, Herndon caters for 1D, 2D and 3D sampling widgets. The same metaphors have been used by the Brown University Computer Graphics group (See Figure 5 on page 14). The 1D widget samples a single point in the dataset, 2D creates a rake alike those used in real wind tunnels to display currents by inserting smoke in to the airstream. The 3D widget controls an array of vector samples. All these widgets

contain manipulators to change the parameters of the visualization widget. The widgets in the images are (clockwise from top left):.

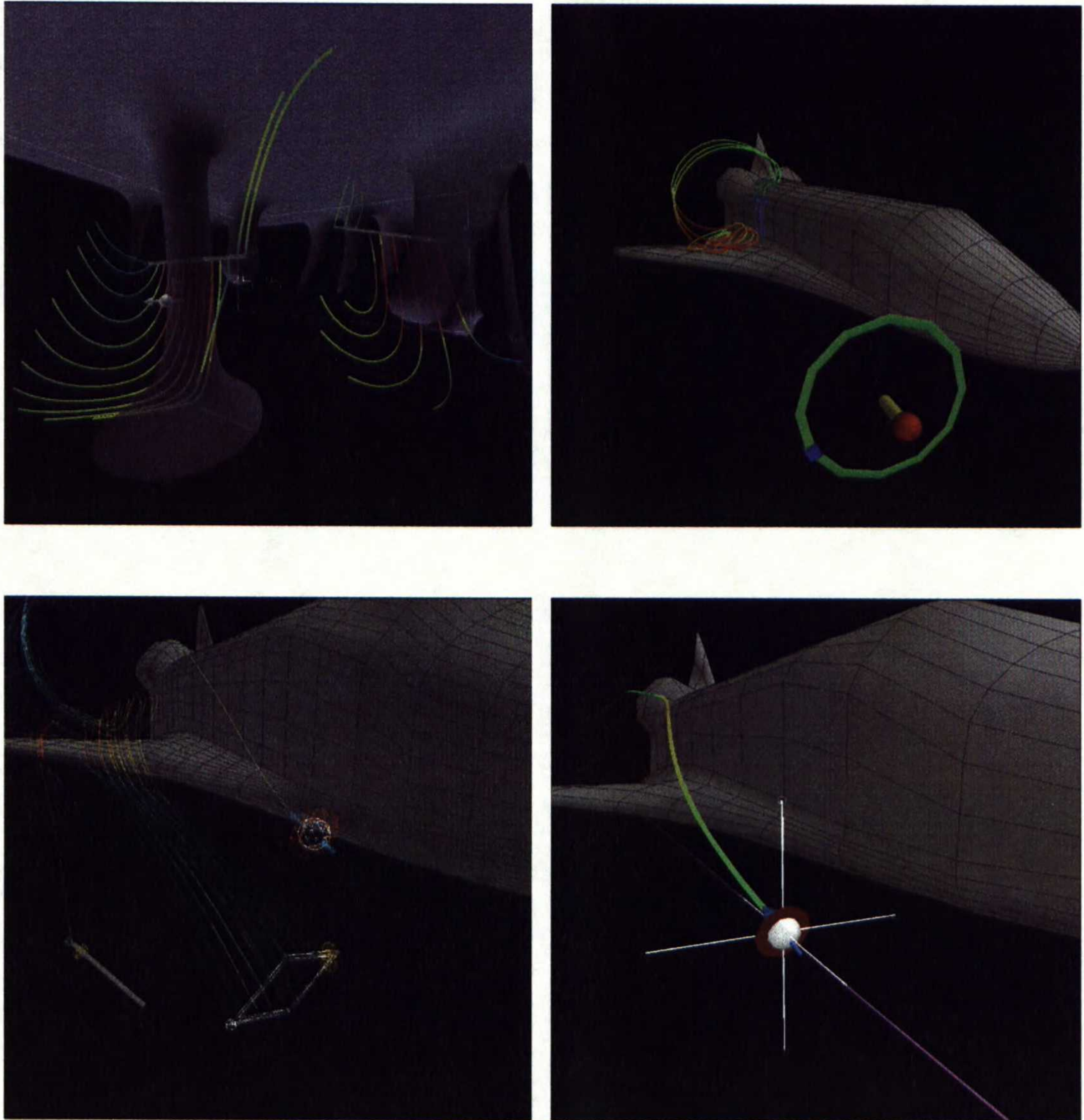


Figure 5 Visualization widgets (Brown, 1996)

- 2D rakes, the density and mount of the streamlines is controllable.
- 2D ring, the size and density controllable, as well.
- 2D point sampler, the radius of the streamer can be changed.
- Rake, point sampler and a 3D streamer grid.

3 MAGIC LENSES AND TOOLGLASS

Toolglass and Magic Lenses are relatively novel user interface metaphors presented by Xerox (Bier 1993, p. 73-80; Bier 1994, p. 358-364). These widgets are semitransparent interactive tools, which have the appearance of a virtual sheet - a Toolglass sheet - of semitransparent material like glass. Each sheet is a freely movable arbitrarily-shaped screen region together with an operator (filter or selection of tools) which affects the objects viewed through the region. This layer appears between the application and the traditional cursor. Figure 6 shows the location of the Toolglass layer. These widget types may provide a set of tools to manipulate the data, or they may customize the view of the application using different filtering methods. Bier divides the widgets into two classes depending on their functionality and properties:

1. Toolglass

   These widgets provide a set of tools with which the user can manipulate the objects underneath the toolglass region.
2. Magic Lenses

   Magic Lenses are Toolglass widgets combined with a filter to enhance the visualization, not necessarily containing manipulation tools.

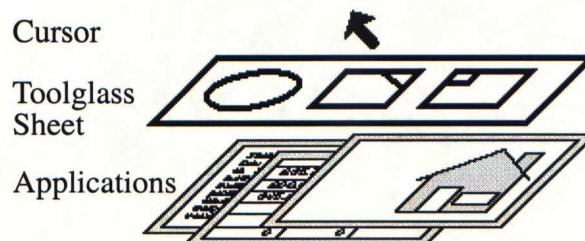


Figure 6 Toolglass interface (Bier 1994, p. 358-364)

Figure 7 shows the analogy between the traditional 3D interaction model based on glyphs and widgets versus the Toolglass/Magic Lens When a visual representation of

data is presented, it forms a glyph. When the glyph in turn is combined with direct manipulation, we can call the combination a widget..

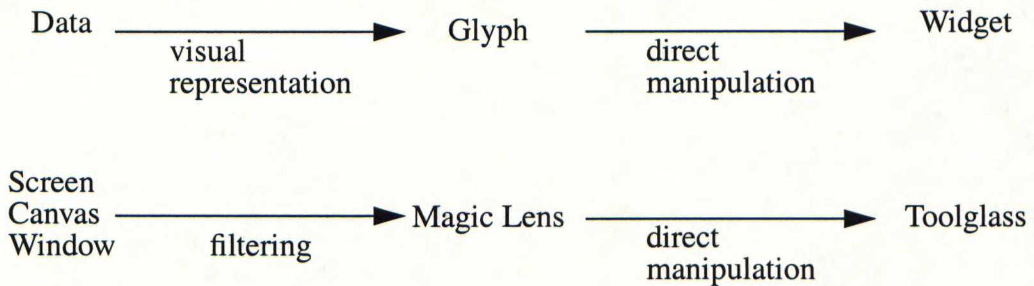


Figure 7 Analogies with traditional 3D interaction model and Magic Lenses

These tools use a spatial mode of interaction instead of the typical temporal mode, enabling the user to simultaneously have multiple, differently filtered views to the same data set. Temporal mode means that the user first selects either the object or the operator and then the other. The spatial mode allows the user to perform these tasks simultaneously through the toolglass region with a single operation. When the user positions the toolglass region over an object, both the tool and the object will be selected simultaneously. Spatial modality is especially suitable for large datasets, or when interfaces using the temporal mode might cause distraction or disorientation. For example, sudden changes in the visualization when using an immersive display will surely cause disorientation.

The lens/sheet metaphor also opens up a way for natural two-handed operation. The non-dominant hand can be used to position the lens, and the dominant hand to operate the tools on the lens or to interact with objects visible through the lens. This would be impossible using the temporal modes of the traditional user interfaces, so the lens metaphor also can enhance the responsiveness and speed of the interface in environment where two-handed operation is available. Kabbash (1993, p. 474-481) has proven the effectiveness of two handed user interfaces. In practice, the 3D first-person games have also proven the effectiveness of two-handed operation. In most of these games, users can freely alter the configuration of different controllers available for the gameplay, and the majority of the users have opted for two-handedness. The usual configuration uses the dominant hand for accurate tasks, like changing the

orientation of the viewpoint in the virtual environment and the non-dominant hand for translational movement (front, back, left, right, up, down).

Because these tools are movable and the operation applies only to a part of the screen, the visual cluttering or overloading of the interface is limited only to a small region. Also, the computational burden of changing the visualization is greatly reduced when the operation only applies to the Magic Lens area instead of the full dataset.

3.1 Composition of Lenses

One of the main benefits of toolglass metaphor is the possibility to combine the effects of several lenses by stacking them. If we consider a lens as a "visual query" to a data set, the stack is a combined query or macro. The visual query can be formed incrementally by adding more lenses to filter the data. This enables the user to more easily construct the query even if the user does not have a clear understanding of what combination of the filters might produce the wanted result. The resulting visual macro can then be saved as a compound lens for future use. An example of composition of the lenses can bee seen in Figure 8.

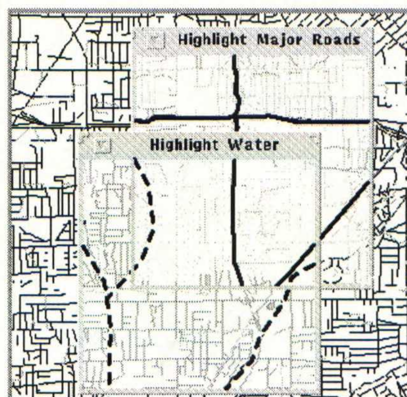


Figure 8 **A composition of two lenses, one showing the roads and the other waterways (Stone 1994, p. 306-312)**

Fishkin (1995, p. 415-420) has implemented Magic Lenses as an interface for database queries using scatterplot displays of US census data as the base dataset. In the article Fishkin defines formally a set of composition rules for Magic Lenses.

A lens is: **(EQ 1)**

$$L = (F,M)$$

where:

F is a filter that describes the output of lens for some datum.

M is a boolean operator (*AND, OR, NOT*) which describes how the result of the filtering function is combined with the output of lenses underneath.

For example, given L_1=(F_1, *AND*) and L_2=(F_2, *OR*), the result of positioning L_2 over L_1 is (F_1 *OR* F_2). Conversely, the effect of positioning L_1 over L_2 is (F_2 *AND* F_1).

The filtering operations can be further divided into two classes:

1. Filters that do selection, i.e. allow operations to be performed only on certain types of objects in the data set.
2. Filters that change the visual presentation of objects.

The *NOT*-operation can be understood as a lens that inverts the sense of data coming in to the lens. This operation applies only to the first filter class described above. The inverting lens is then L_i=(*NULL, NOT*). This means that the lens applies a *NOT* (or inversion) to the incoming data but it has no intrinsic filter function. It can be argued that the inversion/notation itself is a filter, but the definition given above helps in the construction of complex filtering operations. For example, the user might construct the query (F_1 *OR NOT* F_2) by using a lens L_2=(F_2, *) and positioning lens L_i=(*NULL, NOT*) on top of L_2 and then using lens L_1=(F_1, *OR*) on top of the resulting query. Composition of lenses reduces the amount of different Magic Lens filters needed in queries.

3.2 Implementation alternatives

Bier (1993, p. 73-80; Bier 1994, p. 358-364) describes three methods for the implementation of viewing filters:

1. *Recursive Ambush*, a procedural method
2. *Model-In Model-Out (MIMO)*, a declarative method
3. *Reparameterize and Clip*, a modification of MIMO method

The *recursive ambush* method is suitable for implementations where the original model is described procedurally as a set of graphics language calls, such as Postscript. The lens is actually a new interpreter for the graphics language, with new implementations for the graphics primitive operations. For example, a lens can modify a line drawing primitive to produce stippled line by first turning on a stipple mode and then calling the original line drawing primitive. Thus in a strict sense the recursive ambush does not need to actually change the primitive operations, if the same effect can be obtained by changing the attributes of the objects.

In composition of the recursive ambush lenses the primitive modified by the lens may not be the original graphics representation, but a modified version of the graphics primitive created by other lenses.

The *MIMO*-method is a modification of the previous method that involves copying of the original graphics model or representation. The model can be a data structure, like a high-level scene graph depicting the scene or a collection of graphics language calls or an array of pixels or some other image representation. Each lens first copies the structure and walks through it modifying it in accordance with the desired lens filter.

When MIMO lenses are composed, each lens takes the model produced by the lenses under it and modifies it. The modified version is then associated with the clipping region formed by intersecting the lens in question with that of the lens underneath. The resulting models are in turn passed to the lenses above.

The *Reparameterize-and-Clip* is a global modification of the renderer parameters. For example, a wireframe or zoom lens can be created by first drawing the original image and then changing the rendering parameters to produce the filter effect. After that, the renderer is asked to clip the drawing region to the lens boundary shape and then to redraw the modified image.

If the rendering parameter changes are compatible, lenses can be composited. With the Reparameterize-and-Clip method there is a possibility that the composition of the lenses becomes impossible or the result may well not be what the user expected. For example, the composition of different viewing frustums (different FOV etc.) can not be defined clearly.

3.2.1 Benefits and disadvantages of the different filtering implementations

Recursive ambush: This method is a very lucrative way of creating Magic Lens filters, because it does not need to allocate extra storage for the scene representation. It also works on the graphics primitive level, suggesting that it might be possible to implement this type of filtering very easily for many applications. However, the other methods can as well work on the primitive level. Usually the filtering effect of the lens is more complex than just a simple change in a certain graphics primitive, thus forcing the recursive ambush lenses to modify several graphics primitives to produce the filtering effect. Bier also suggests that debugging the composition of recursive ambush lenses is difficult because the effects of several co-operative interpreters are hard to understand. The effectiveness of this implementation can also be questioned, because the result of each lens has to be computed several times: the number of computations doubles with each overlap.

MIMO: The main drawback of the MIMO method is that the data depicting the scene needs to be copied and maintained for each and every lens. If the original scene graph is large, this can lead to excessive memory requirements, causing loss of performance. On the other hand, copying the data and maintaining separate copies pays off, because the filter needs to be computed only once per lens. The MIMO lenses can also perform complex filtering tasks more easily than the other methods.

Reparameterize-and-Clip: The problem of compositing Reparameterize-and-Clip lenses is the only major drawback in comparison to the MIMO-lenses. Implementing this filtering type is very straightforward, and does not require maintaining a separate copy of the scene graph for different lenses. Of course, the underlying software API may hinder the implementation of a Reparameterize-and-Clip filtering.

If direct manipulation is to be considered, the best alternative for filtering is MIMO, because the operation can be performed in the "local" scene graph of the lens. Also Reparameterize-and-Clip is suitable for direct manipulation.

3.3 Examples of the 2D Toolglass metaphor

Xerox has presented many sample implementations of the toolglass for different purposes, but commercial implementations have been very sparse. The implementations by the Xerox Magic Lens group have been mainly directed for the creation and manipulation of 2D graphics, but they have also implemented the metaphor as a tool for database visualization and queries (Fishkin 1995, p. 415-420) (See Figure 9.)

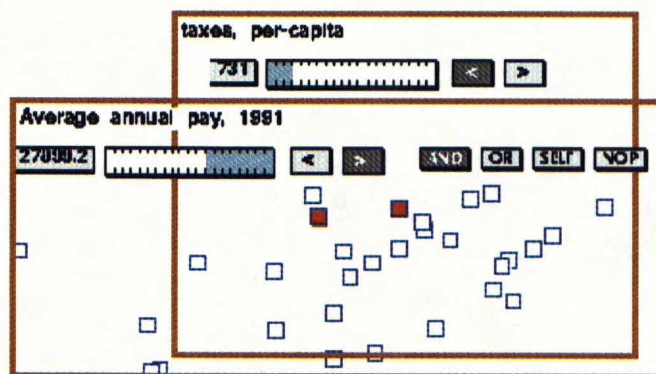


Figure 9 **Dynamic database query: high salaries AND low taxes (Fishkin 1995, p. 415-420)**

The database query operations included both boolean queries and real-valued queries, making it possible to implement arbitrary scoring functions for the data. In the imple-

mentation, scoring function is a filter which returns a numeric value between [0...1]. The data returned by the scoring function can the be subjected to arithmetic filtering.

The traditional 2D Magic Lens interface has been implemented also by a couple of commercial software manufacturers, mainly for commercial illustration systems, such as CorelDraw and Kai's Power Tools.

3.3.1 CorelDraw

The CorelDraw implementation contains a small number of Magic Lens filters, which can be composited by overlapping the filter regions. The filters include a magnifier and five color operators.

3.3.2 Kai's Power Tools

Kai's Power Tools (KPT) is a collection of plugin modules for the Adobe Photoshop (MetaCreations 1997). The interfaces provided by KPT have traditionally been very artistic; the tools have been decorated by unnecessary graphics just to make the tools stand out from the mass. The Magic Lens in KPT is a previewer toolglass. Without the composition of several lenses to form visual macros. As example, Figure 10 presents a view of the underlying graphics through a "smudge" lens.

Figure 10 Kai's Power Tools Magic Lens Implementation. An image of a car seen through a "smudge" lens.

3.4 Examples of the 3D Toolglass metaphor

For 3D environments the metaphor has not been researched thoroughly, though it provides a solid and easily understandable basis for interaction with virtual environments. The interface can also be scaled from palmtop displays to immersive wall displays or CAVE-environments. A CAVE-environment is an immersive VR environment, consisting of several video walls usually in a cube-like arrangement utilizing backprojected, computer generated stereo imagery. Also, the spatial mode of operation opens up a way for natural, two-handed interaction model, which suits well for immersive VR. Because the lens metaphor limits the changes in the visualization into a spatially bounded area, it reduces the possibility of disorientation or loss of immersion.

3.4.1 Virtual Tricorder

Wloka (1995, p. 39-40) has combined the Magic Lens metaphor with a universal toolbox, the Virtual Tricorder (see Figure 11). The virtual tricorder is a widget, reminiscent of a hand held remote or PDA (Personal Digital Assistant) with a flat screen.

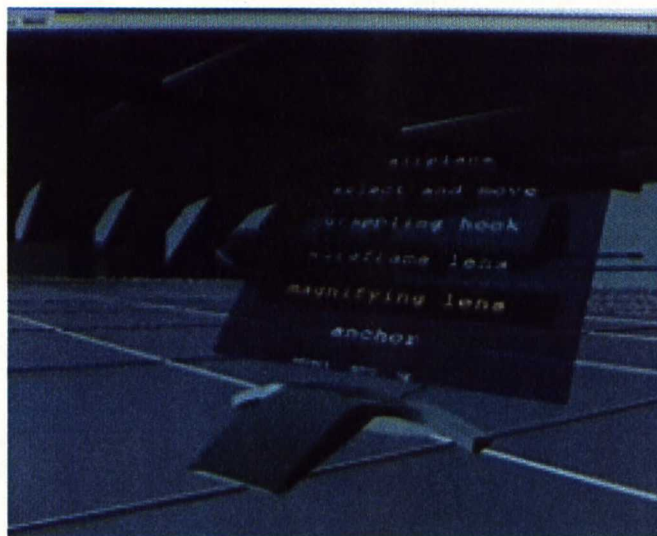


Figure 11 The Virtual Tricorder Interface (Wloka 1995, p. 39-40)

With this tool, the user can perform different tasks with the objects in the scene. In his implementation the Tricorder can be used both as a navigation and visualization tool in virtual environments. The main motivation for the Tricorder has been to create a large toolbox to visualize the hardware interface devices (wands, etc.) in the virtual environment, and to create a replacement for the standard 2D popup/pulldown menus for VR environments. When the Tricorder is used as a movement aid in the virtual environment, the user would first select a location by pointing the tricorder at the spot. After that, the tricorder would "reel in" a virtual line between the user and the location of interest, moving the viewpoint accordingly.

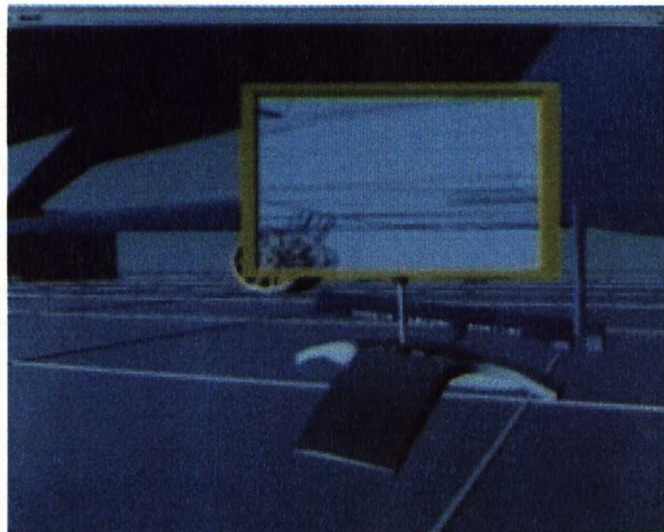


Figure 12 The Virtual Tricorder as Magic Lens (Wloka 1995, p. 39-40)

Wloka also alluded towards selective filtering and visualization of normally hidden objects through the Tricorder interface. The screen is both a menu selection device and a Magic Lens (see Figure 12). This combination of a navigational tool and a Magic Lens interface is a very attractive idea. However, the interface described in his paper missed some main points of Magic Lenses:

- Reduced visual clutter.

  The tricorder interface itself reserves quite an amount of screen space, which could perhaps be utilized for other purposes.

- The composition of lenses.

  With just a single Tricorder, the nature of Magic Lenses interface is not fully realized, because the Tricorder is able to display only a single lens filter at a time.

3.4.2 Volumetric Magic Lenses

Viega (1996, p. 51-58) has presented both flat and volumetric lenses. An example of a volumetric lens in action can be seen in Figure 13.

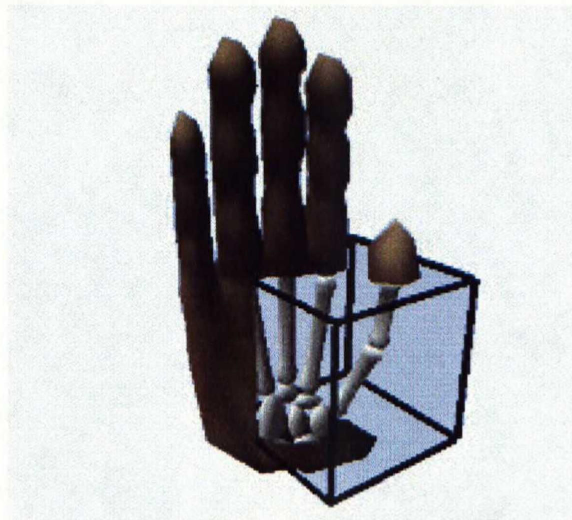


Figure 13 A volumetric X-Ray Magic Lens (Viega 1996, p. 51-58)

The implementation uses hardware clipping planes to draw the lens images. The downside of that particular method is that it requires the scene to be rendered from four to six times per lens image, and it limits the shape of the lenses to rectangular forms. This is because the current hardware implementations define the clipping planes as infinite half spaces. Thus it is impossible to arrange six clipping planes to form a rectangular region in space which would excludes what falls inside. Figure 14

on page 26 shows two facing clipping planes which combined clip all of space. The implementation also makes the composition of lenses overly complex task.

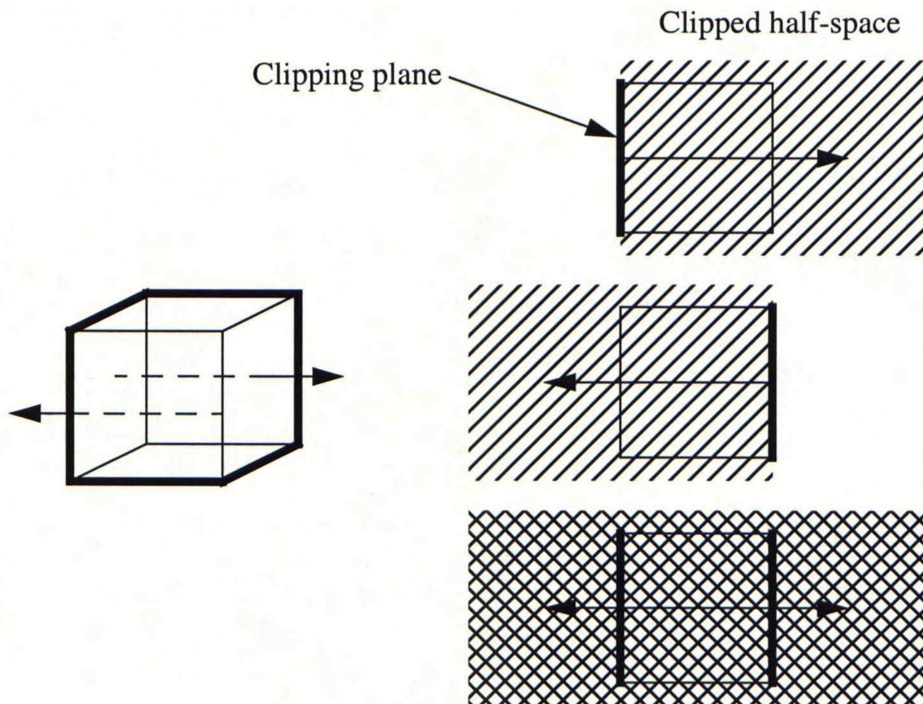


Figure 14 Two facing clipping planes clip all of space

In the paper Viega suggests that volumetric lenses could also be used to create Worlds In Miniature (WIM) (Stoakley 1995, p. 265-272) - style navigational aids for virtual environments by considering the WIM display of the environment as a reducing

volumetric lens encompassing the region in space around the viewer. In a strict sense this is not a volumetric Magic Lens. It is rather an additional display component. Figure 15 shows an example of a volumetric lens used as a navigation aid.

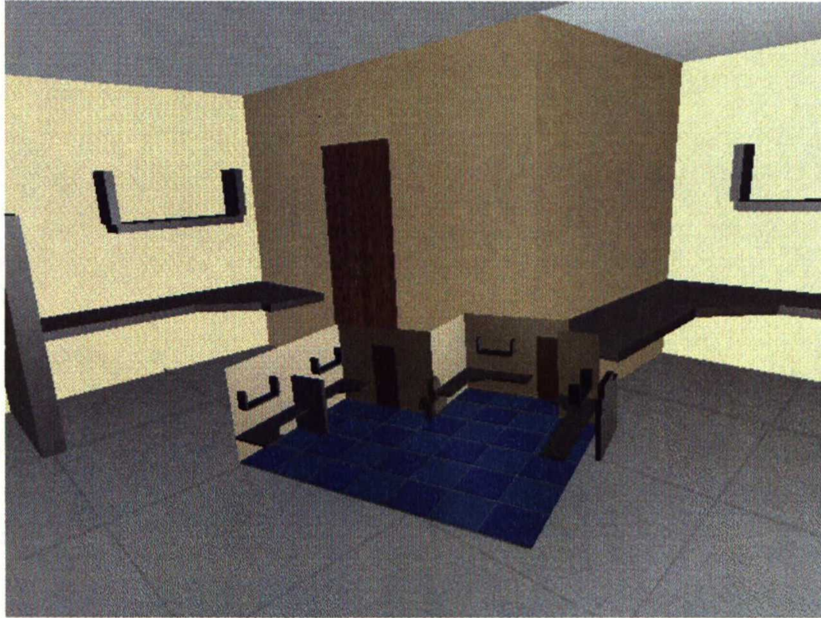


Figure 15 A room and the WIM navigation aid (Viega, 1996 p. 51-58)

Volumetric 3D Magic Lenses probably fit well to certain situations, like virtual surgery and other medical applications. However, the effectiveness of the metaphor is limited by the strict regionality of the volumetric lenses. In a sense, the lens metaphor reduces to a “virtual aquarium”-display or to a simple clipping / CSG (Constructive Solid Geometry) operation.

4 2D MAGIC LENSES AND 3D VISUALIZATION

The 2D magic lens interface has previously been used mainly to filter the information seen directly in an application. Combining the magic lens interface with a separate 3D scene visualization provides more possibilities for the lens metaphor. E.g. both navigation and locating structures or areas of interest in large scale virtual environments are often difficult, but by using a 2D map image with Magic Lenses to filter 3D visualization of the location, the navigation and wayfinding tasks can be made easier. Of course, all data representations which benefit from the 2D/3D combination, can be used in this way. For example, a network management program may use the 2D Lens Interface to display the logical structure of the network, and utilize the lenses to filter the routers and other network components based on fault types, equipment types etc. The 3D visualization will then display the physical structure of the network, helping to pinpoint the real location of the faulty equipment. Thus the two different topologies of the network can be visualized side by side, augmenting the management task.

4.1 2D Magic Lenses as navigational aids

In large scale virtual worlds, like cities or buildings, the sense of orientation can be lost easily, because the quality of the visual and aural presentation does not necessarily provide the same visual or auditory cues as the real world. The user can gradually learn to navigate in the world but this can take some time.

Darken and Sibert (1996, p. 142-149) divide wayfinding tasks into three categories:

1. *naïve search*
   The user has no a priori knowledge of the location of the target in question. Thus an exhaustive search must be performed.
2. *primed search*
   The user knows the location of the target. The problem that remains is wayfinding, i.e. the user must move to the location.
3. *exploration*
   A searching task where the user has no target; he is only exploring the environment.

These different searches are usually combined. For example, the user may know the approximate location of the target, so he first performs a primed search to find the general location, followed by a naïve search to pinpoint the target. This search can also be performed in the opposite way: naïve search to find the approximate location followed by a primed one.

VRML tries to overcome the navigation problems by making available a number of predetermined viewpoints. If the scene is simple, and only a few viewpoints are sufficient for the navigation, the viewpoint scheme works well. If the scene requires a large number of possible viewpoints, the navigation via a viewpoint list soon becomes very tedious. Also, this method may not provide enough clues to help the user to determine the relative location of the target in the virtual environment.

Combining the 2D magic lenses and a 3D virtual world provides a simple and easy interface for both navigation and searching tasks. The 2D lens interface is used both as a navigational aid and a tool to define the objects of interest in the 3D visualization. The lens interface can easily display an indicator showing the current location of the user in the environment, and additional data, like the field of view, or elevation.

For navigational purposes, 2D lenses can be used on top of a normal map-like representation of the virtual scene. The lenses can filter uninteresting parts of the scene

away or bring more information to the map. The filter, resulting from the combination of the lenses, can then be used to filter the 3D visualization of the same area, giving the user the possibility to see the spatial location of the interesting parts in 3D. Used side by side, the map interface provides a well known aid from daily experience for navigation and wayfinding. The 3D visualization also takes into account the actual position of the user in the virtual scene.

4.2 Mapping of the 2D lenses for 3D space

If the traditional 2D lens interface is combined with a 3D visualization, the effect of the lenses for the virtual environment can act in three ways:

1. The lens interface acts as a general filter selection tool
2. The lenses affect only the region under the lens in the 2D display
3. The lenses react to user entering and exiting the lens region

These options can be used in combination. The first of these options is quite straightforward, but it is as usable as the rest of the mappings. For example, the Magic Lenses interface can be used to select the type of network wiring from the floorplan of an office building. The selected wiring can then be seen in a separate 3D visualization with the model of the building. This type of functionality is needed for example when the user knows how to construct filters to display entities in a smaller context, but needs also to see the same filters in the whole context. The selection of filtering is accomplished by clicking through the wanted lens combination in the 2D lens interface.

The next choice is a direct mapping from the bounded 2D lens region to the 3D display. For this type of filtering, the 2D lenses must specify a region in space, i.e. restrict the filter effects in the vertical direction. To define a region in space, the 2D lenses can be used to specify the third dimension by adding two scalar values to the properties of the lens. These values specify the bounding region in the vertical direction. Scalar valued lens filters can also be used for this purpose.

For general purposes, the avatar[1], or the projection of the user in the virtual space must also be taken into account. Avatar means the embodiment of the user in the virtual space. Avatars do not necessarily need any geometrical representation, but they must have a defined size to help to establish the sizes of openings the user fits through, the maximum step size and other values regarding the movement in the virtual world.

Because the 2D->3D mapping creates in fact a volumetric lens, the filtering can provide the same functionality as real 3D volumetric lenses. In this implementation the lens regions are generally larger than the user's avatar, so the second method can be varied to produce yet more lens-to-visualization mappings. The lens effect on the visualization can be dependent of the user's location in the virtual environment. For example, the filter can be visualized constantly, providing a real volumetric lens, or the filter can be turned on when the user enters the lens region.

4.3 Implementation of 2D lens and 3D visualization

Lately, different virtual cities implemented in VRML have appeared on the net. This work presents one solution for more versatile navigation and wayfinding in the virtual city. This part of the work is based on Java and VRML for platform independentness.

4.3.1 Java

Java is a platform independent, interpreted object-oriented programming language developed by Sun Microsystems Inc. (1991). Java language contains the language core, plus additional, rudimentary functionality for media processing and a GUI - toolkit. Abstract Window Toolkit (AWT) is a simple toolkit for creating interfaces for Java applets or applications. AWT is based on the least common denominator of the native window systems on different platforms. Other reasons for the minimalism of the interface components might be that Java was originally designed as a

1. **Merriam-Webster's Collegiate® Dictionary:**
av.a.tar *n* [Skt *avatara* descent, fr. *avatarati* he descends, fr. *ava-* away + *tarati* he crosses over--more at ukase, through] (1784) **1:** the incarnation of a Hindu deity (as Vishnu) **2 a:** an incarnation in human form **b:** an embodiment (as of a concept or philosophy) often in a person **3:** a variant phase or version of a continuing basic entity

programming environment for embedded systems, i.e. intelligent remote controllers with LCD (Liquid Crystal Display) displays and the like. AWT is missing some basic functionality, like overlapping windows.

Overlapping windows with AWT. Because the AWT does not cater for overlapping GUI components, this functionality had to be implemented to provide a platform for 2D Magic Lenses, where the controlled overlap of the windows is a requirement. The abstract window toolkit allows overlapping GUI components, but there is no way to control the drawing order of the components. To create this functionality, a drawing component, `java.awt.Canvas` is subclassed to implement a simple windowing interface. This subclass maintains a stack of windows on a canvas, handles the drawing of the windows, resizing and other operations or event handling. When the user moves the mouse over a window, the canvas subclass calls the appropriate callbacks for the window. The windows themselves are implemented in class `CanvasWindow`, which draws the window frames and contains also the graphics context for the drawing. Both of these classes double buffer the drawing to minimize artifacts.

Magic Lenses interface for Java. The `MagicLens` class extends the `CanvasWindow` class mentioned above to implement the functionality for the lenses. The `MagicLens` class contains a `LensFilter` interface for implementing the filter methods. The lenses operate on image data and the filtering is based on color values using the image filters provided by the `java.awt.image.RGBImage-Filter` class. The best composition method for operations on images is MIMO. When the lenses are composited, the lens which is currently being drawn, requests the underlying lens to provide the source data. Because the filtering happens asynchronously, the Magic Lenses employ a MediaTracker object to block the execution until the filter has produced the final image. The applet is not threaded, because the drawing needs to be synchronized anyhow, so creating multiple threads for the

different lenses does not offer any advantage. An example of the lens applet is in Figure 16.

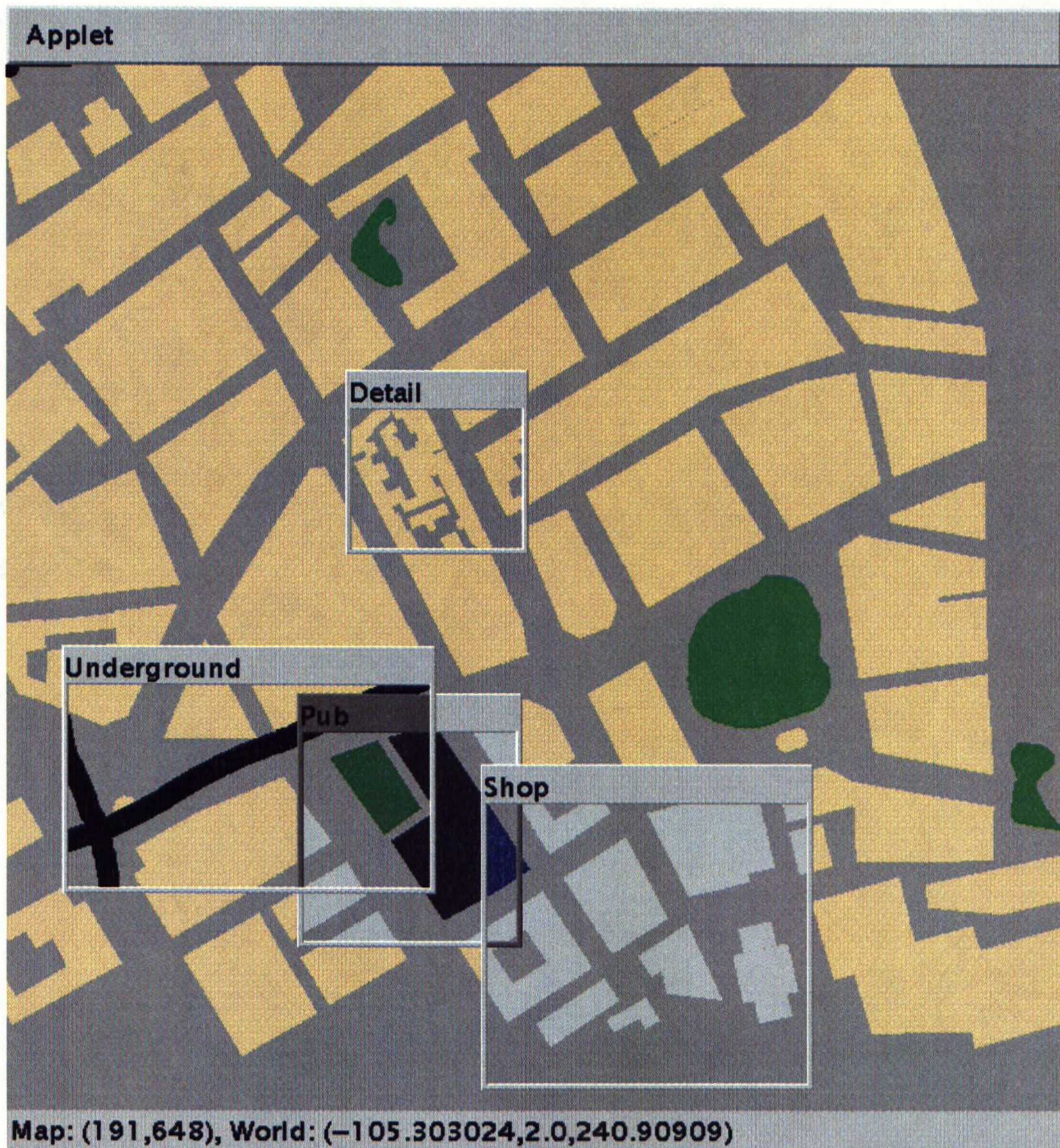


Figure 16 Magic Lens applet displaying different lenses

Interaction with the applet is straightforward. The user can move directly to a location in the scene by clicking the appropriate spot on the map. Depending on the mapping of the lenses, clicking trough the lenses can also select a combination of lens filters. The lenses are interactively movable, updating the displayed image constantly. When combined with a VRML viewer applet, the map also displays the location and the field of view of the user.

The applet is started from HTML code, and the available lenses for the Magic Lens viewer are defined as parameters for the lens applet:

```
<applet code="MagicLensTest.class" width=660
            height=660>
<param name="name" value="magiclenses">
<param name="receiverName" value="vrmlviewer">
<param name="baseImageURL" value="file:///home/napo
    /public_html/lens/images/london_crop.gif">
<param name="lens1" value="Detail-Lens,
    100,200,80,50,file:///home/napo/public_html/lens
    /images/london_detail_crop.gif,Detail,Detail">
<param name="lens2" value="ColorPassLens,
    20,20,50,50,0,128,0,Restaurant">
<param name="lens3" value="ColorPassLens,
    100,100,50,50,0,0,128,Shop">
<param name="lens4" value="OverlayLens,
    100,300,80,50,file:///home/napo/public_html/lens
    /images/underground.gif,Underground,Underground">
</applet>
```

The first parameter defines the lens type. A Java class with the same name is then instantiated, the window is positioned and sized according to the four coordinates: location (x,y), width and height. The rest of the arguments define the filter parameters or the image URL (Universal Resource Locator) for overlay-type of lenses, the lens name and the string to match against when filtering the VRML scene. The different lenses in this work are filters that operate on the RGB (Red, Green and Blue) data of the image and also overlay lenses, which import data from additional sources to create lenses that can for example enhance the detail on the image or show underground pipelines.

4.3.2 VRML

VRML (Virtual Reality Modeling Language, VRML Architecture Group 1997) is a file format for describing interactive 3D objects and virtual environments to be experienced and shared on the World Wide Web. The first release of the VRML specification was based on Open Inventor (Wernecke, 1994) file format, and it was created by Silicon Graphics in collaboration with numerous contributors. The first release was lacking on the interaction side, the interaction with the scene was limited to navigation and clickable, active objects called anchors. The anchors provided means to load new VRML scenes or other information from the web. This first version was actually a simplified version of the Open Inventor format, from which the interaction with the scene was removed. The reason behind the removal of the interaction was that the implementation of the VRML browsers would be an easier task. In principle, this first version was expandable, but in practice the expansion was difficult.

The second release, VRML 2.0 or VRML97, is based on the first version of VRML. This release incorporates more interaction to VRML, and it specifies a clear interface for expanding the functionality of VRML. VRML97 is capable of representing static and animated 3D content plus a wide variety of other media. There are also mappings between VRML objects and commonly used application programming interfaces (API).

The VRML scene description consists of a scene graph and functionality embedded to the scene graph. Scene graph is a hierarchical, usually a tree-like acyclic graph that contains nodes that describe a virtual world.

Metadata for VRML nodes. If the VRML scene is to be filtered using other properties than the ones specified for the VRML nodes, the scene objects need to contain additional metadata. For example, if a city scene requires that the different parts of buildings or blocks can be filtered based on their usage (shops, pubs etc.), the scene database must contain this information.

The VRML objects could have been designed to be more friendly for metadata, but as it is, adding metadata to the node descriptions would require rethinking and rewriting the VRML specification. There is one possible way to insert metadata to the scene without creating new nodes or node classes, though. The VRML event handling and instancing model requires that the nodes have distinct names, which can be specified freely. This `DEF/USE` construct can contain the metadata as the node name. Because the node names are used only for routing events from a VRML node to another, there are no shortcomings or loss of functionality if the names are used for other purposes too. VRML Database Working group has presented a proposal (VRML Database Working Group, 1998) for metadata nodes that provide access to arbitrary XML (eXtensible Markup Language) documents.

When the scene is filtered, the VRML browser can decide based on the node name the class of objects where it belongs. In this work, the node name is matched against a regular expression for the classification. The use of regular expressions simplifies the task of scene construction, allowing the creator of the scene to name the different objects using human understandable names to ease the task.

Unfortunately, the VRML specification does not provide means for complex operations on the scene graph. The External Authoring Interface (EAI), which is an external, usually Java-based API, was not ready at the time when this part of the work was done. EAI allows access to the VRML scene graph, but for example searching a named node from the scene graph is limited to exact names of the nodes. This causes problems if the metadata is to be inserted to the scene as node names. Of course, the metadata could be mapped to node names using an external database to provide the mapping.

Using Java in Script node inside the VRML scene, which also was not functional at the time this work was made, has the same problems as the EAI. Fortunately, Dimension X's Liquid Reality - Java toolkit for VRML was available for this task.

4.3.3 Liquid Reality

Liquid Reality (LR) is a VRML toolkit written in Java, using native methods where the execution speed is critical. It contains classes for basic geometric operations, the graphics primitives defined in VRML and the interface for embedding VRML content to Java applets or programs. LR also has complex searching operations for the scene graph and additional functionality, like bounding box and scene path operations usually found in high level graphics toolkits.

Dimension X was acquired by Microsoft in 1997, and the development of the toolkit as it was has been cancelled. The toolkit is no longer available.

Because of the limitations of VRML, I wrote a VRML browser applet using LR. The browser implements methods for searching the scene graph and manipulating the VRML content for filtering purposes. In this case, the VRML applet searches the scene graph for named nodes matching against a regular expression. When a list of nodes is found, the browser then modifies the `Appearance` node of the found object to affect the visualization of the scene. If the named node or group does not contain an `Appearance` node, it is inserted to the scene by the browser. The composed filtering effect of the lenses is handled in the Java-applet, so there is no need for more complex filtering capabilities on the VRML side.

The VRML applet also transmits information about the user's location in the scene and viewing parameters to the Magic Lens applet.

The Figure 17 displays an example VRML scene unfiltered side by side with the Magic Lens applet.

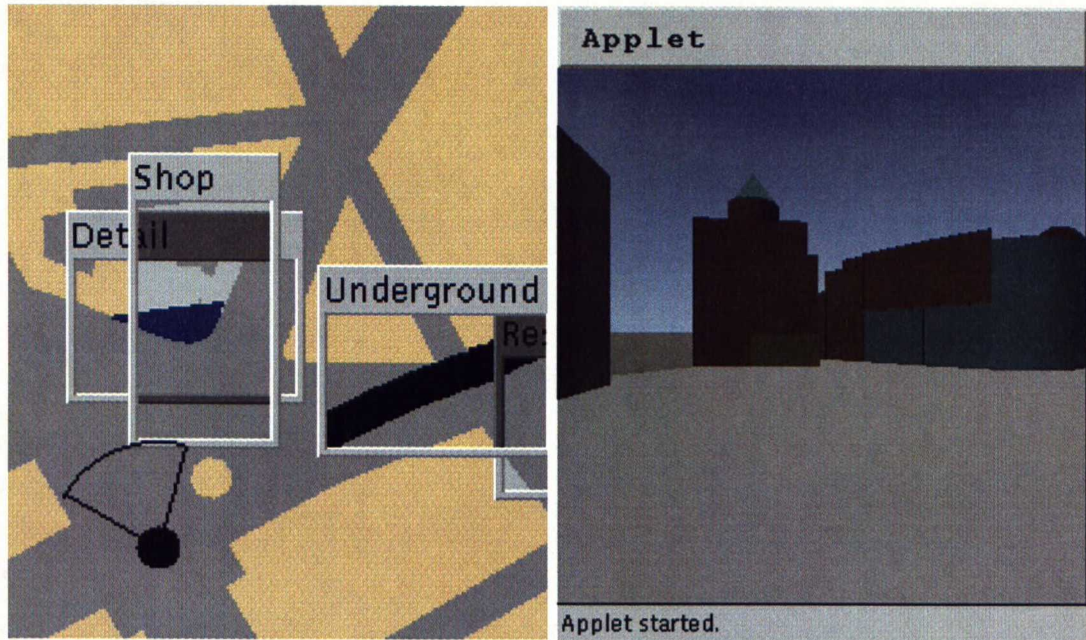


Figure 17 The lens applet displays the lenses and the location and FOV of the viewpoint

When the scene is filtered using the 2D lenses as general filter selection tool, the resulting VRML scene changes to portray the filtered parts of the scene semitrans-

parent, as can be seen in Figure 18. The transparency artifacts visible in the screen-

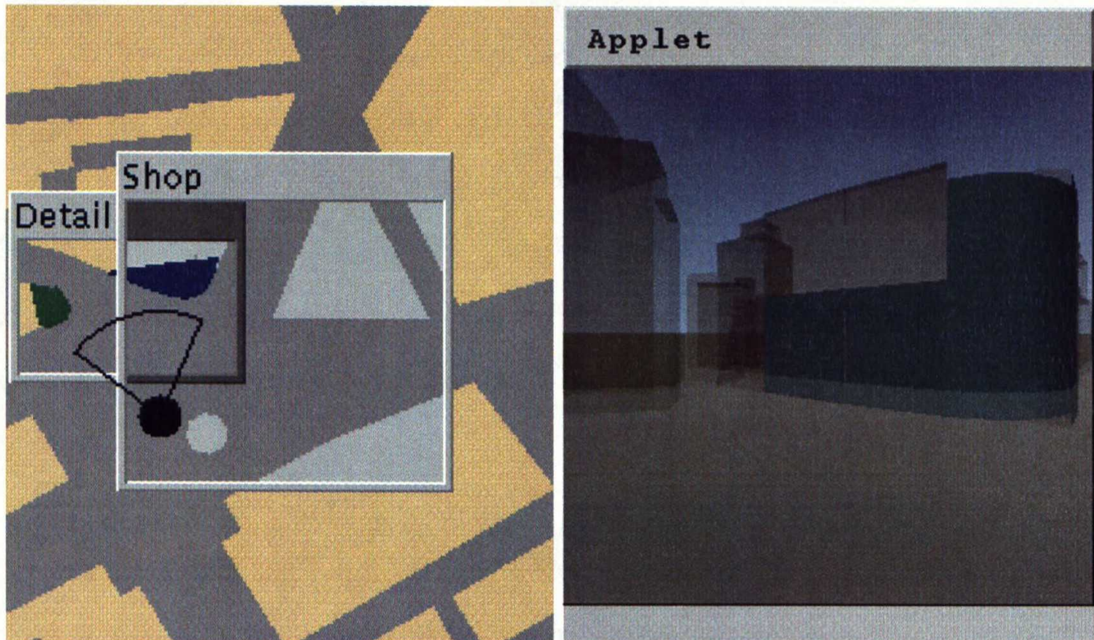


Figure 18 Scene filtered by a Magic Lens

shots are caused by bugs in the Liquid Reality implementation. The version used in this work was a beta and it was missing certain features.

As mentioned in the beginning of chapter "Mapping of the 2D lenses for 3D space" on page 30, the filtering of the 3D scene can be done in various ways. Figure 19 on page 40 shows the effect of lenses that operate like volumetric lenses. The user is

positioned outside the lenses, and the filtering affects only the parts of the scene inside the lens volume.

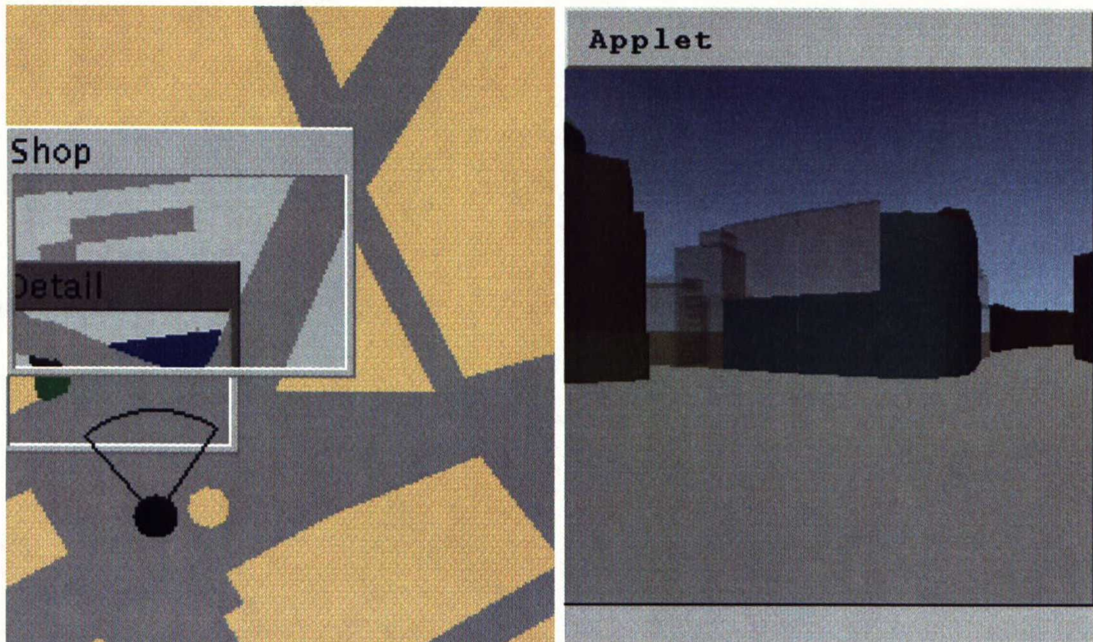


Figure 19 Scene filtered by aquarium type lenses

4.3.4 Communication between the Magic Lens and VRML applets

Both applets implement an interface for messaging, and because the applets are located on the same web-page, they can call each others methods. The messaging API is implemented as an interface (also known as multiple inheritance in object oriented programming), which consists of methods to initialize the messaging, sending and receiving single or multiple messages in a single method call.

The Java virtual machine is relatively slow, so it can cause delay for the messages. Because most of the data is transmitted from the VRML viewer to the lens applet in the form of viewpoint location, the messages are not handled in a timestamped manner. Instead, only the latest message is used.

5 FLAT 3D MAGIC LENSES

When the Toolglass metaphor is moved to a 3D environment, most of the properties of toolglasses remain unchanged. However, three dimensions create more possibilities and challenges for interaction. For example, the geometry of the lens can be of any shape, even a curved surface resembling a conventional lens. For immersive worlds, the display of the lens can be stereographic or flat depending on the needs of the user. An important element is that the lens geometry and image does not need to be constrained to be two dimensional region aligned to face the viewer, but it can be located anywhere in the virtual world. Even the source data for the lens does not need to be directly visible from the current location of the viewer. This advances the possible ways of using lenses to filter the environment but it also makes the composition of the lenses a more challenging task. It should also be noted, that even though the lenses are described as "flat", they operate on a volume in space.

The previous work on 3D lenses has mainly concentrated on implementing the Magic Lens interface for 3D graphics. The implementations have not taken into account the possibility of multi-user environments and the methods presented are not very effective. For example, the volumetric lens implementation by Viega (1996, p. 51-58) had to draw the same scene at least six times to properly construct the lens image.

Virtual environments are usually computationally expensive, but the spatially bounded region of the lens interface can ease this burden. Because the geometrical representation of a virtual environment is complex, it is usually stored in a hierarchical structure: the scene graph. VRML and Open Inventor scene descriptions are examples of scene graphs. Typically the scene graph contains high-level descriptions of the objects in the scene, not just the basic graphical primitives. The structure can also be used to optimize the rendering of the scene by culling the non-visible parts of the scene before it gets rendered. Because Magic Lenses operate only on a part of the scene, the operations done by/via the lens can be directed to a smaller portion of the scene graph.

5.1 The relationship between the lens image and scene

The 2D Toolglass constrained the effect of the lens to the region bounded by the lens. The 3D version of Toolglass or Magic Lens uses instead of a two dimensional region a volume in space, *lens frustum* (see Figure 20), which specifies the area on which the lens operates. The Magic Lens images are generated as an additional view to the scene. The lens frustum is analogous to the normal viewing frustum in 3D graphics, but it does not need to be constrained to the actual viewpoint of the user. The lens frustum is formed by the standard matrices used for 3D graphics that define the transformations for the camera: orientation, location and projection.

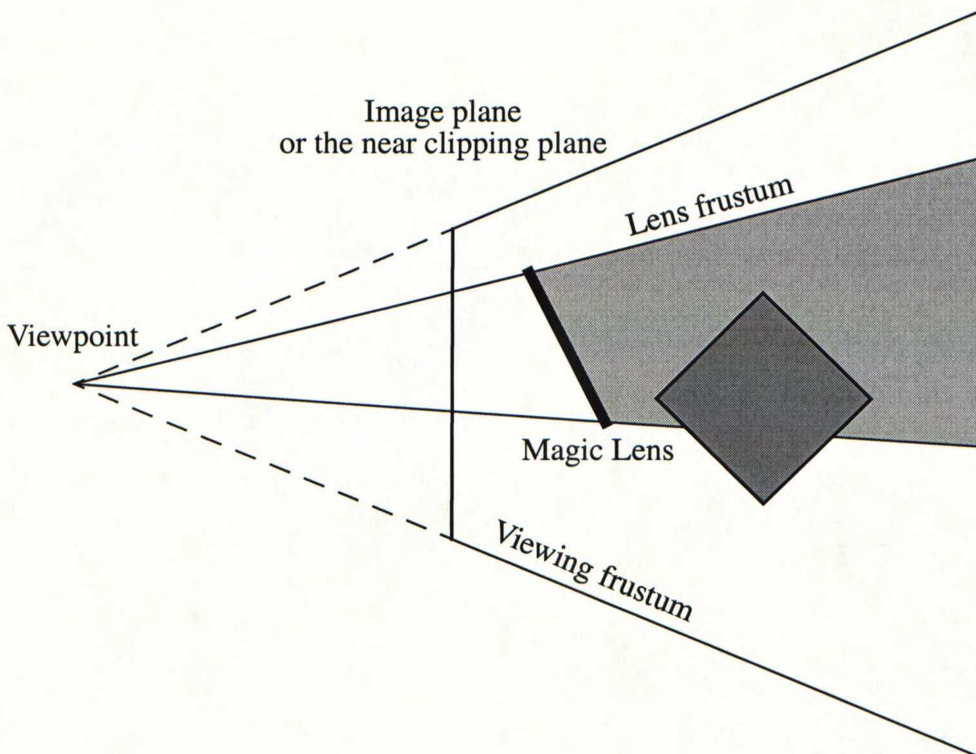


Figure 20 The lens frustum formed by the Magic Lens

Even though the lenses are three dimensional objects, they can be used to construct a traditional 2D lens interface by constraining the lenses to be visible in the viewport regardless of the position of the user in a Head-Up-Display style. A real volumetric

lens effect can be obtained by combining several lenses together in a cubical formation with a suitable filtering operation, like modifying the clipping planes or changing material properties seen through the lens to be semitransparent.

A more viable usage of flat lenses is as “real lenses” in the virtual world. This means that the geometry of the lens acts as a filter through which the scene is viewed. When the user changes position in the virtual world, the image formed by the lens changes accordingly, displaying the scene as it would appear from the viewpoint of the user as shown in Figure 21.

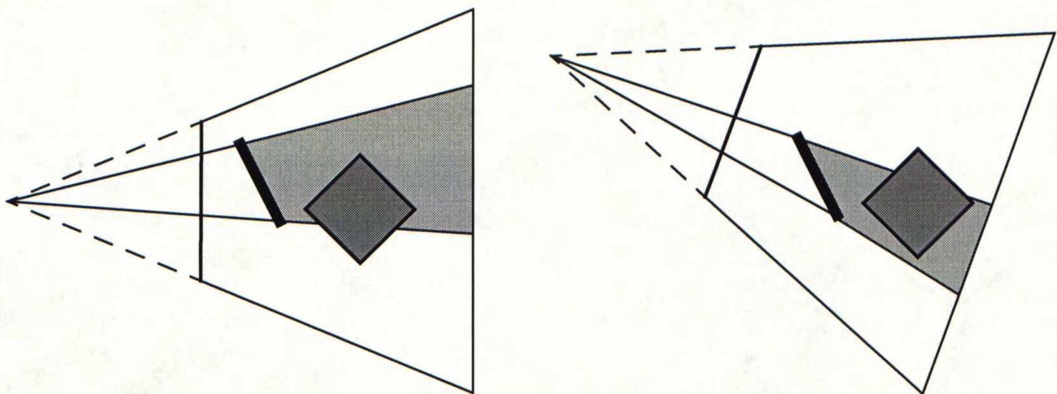


Figure 21 Scene viewed through a lens from different directions

Because the position and other aspects of the lens frustum can be easily defined, it allows us to expand the possibilities of a Magic Lens - user interface. The image displayed by the lens can show a location anywhere from the scene, even from a view which is not currently visible in the viewport, e.g. the source of the image can be a location behind the viewpoint. The lens frustum can also be attached to the lens

geometry itself (see Figure 22), so that the lens image does not change unless the orientation or position of the lens changes.

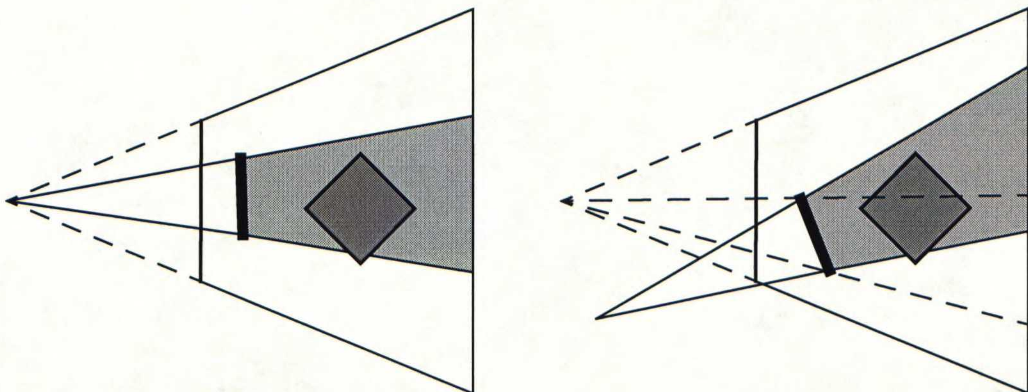


Figure 22 Scene viewed through a lens frustum attached to the lens geometry

The different ways to connect the frustum to the actual lens image enhance the possibilities of 3D Magic lenses. On the other hand, the composition of the lenses with different lens frustum - geometry relationships is not straightforward or can even be impossible to implement.

Also, the lens geometry itself can be transformed regardless of user's actions. For example, billboards, animation and HUD - like lenses can be used to automate parts of the interaction. This reduces the amount of "secondary" interaction, like moving and positioning the lens thus freeing the user to perform more complex tasks with the lens.

5.2 Magic Lens semantics in multiuser environments

When the Magic Lens metaphor is used in VR environments, the semantics of the interface can be expanded, even for single user virtual worlds. Previous work has mainly shown that Magic Lenses interface can be used with visible information, as a see-through tool.

If the environment is populated by multiple users, the question "Who can see and what through the lens?" must be answered. The lens can be thought to be a part of the

world that anyone can see and operate, or it can be strictly a local, private entity for each user. For example, collaborative scientific visualization applications could require the lenses to be a part of the common virtual world.

For shared magic lenses, the image of the lens can be formed by two different ways:

- One of the participants "owns" the lens, and is able to see through the lens. Other participants just see the image of the lens much like a movie or slide.
- All of the participants in the VR environment experience different results when looking through the lens, depending on their viewpoints.

Of course, the ability to manipulate the geometrical representation of the lens can be limited too. An additional problem with multi-user environments and physical lens objects in the world arises if the avatars in the world have physical boundaries or limitations. The avatars might not be able to gather to the same location to peer through the lens if the avatars can not share the same volume in the virtual space. If we can remove the physical geometry of the lens representation from the environment, this problem gets easier to solve.

5.3 Projected Magic Lenses - Magic Lights

If we discard the idea of separate piece of geometry as a "canvas" on which the image formed by the lens is displayed, but use surfaces existing in the virtual environment, we have a new concept: Magic Lights. Magic Light can be thought to be a slide

projector or flashlight which makes the lighted surfaces to behave specially, e.g. the light marks the surfaces which display the Magic Lens filter attached to the light. An example of a Magic Light can be seen in Figure 23.

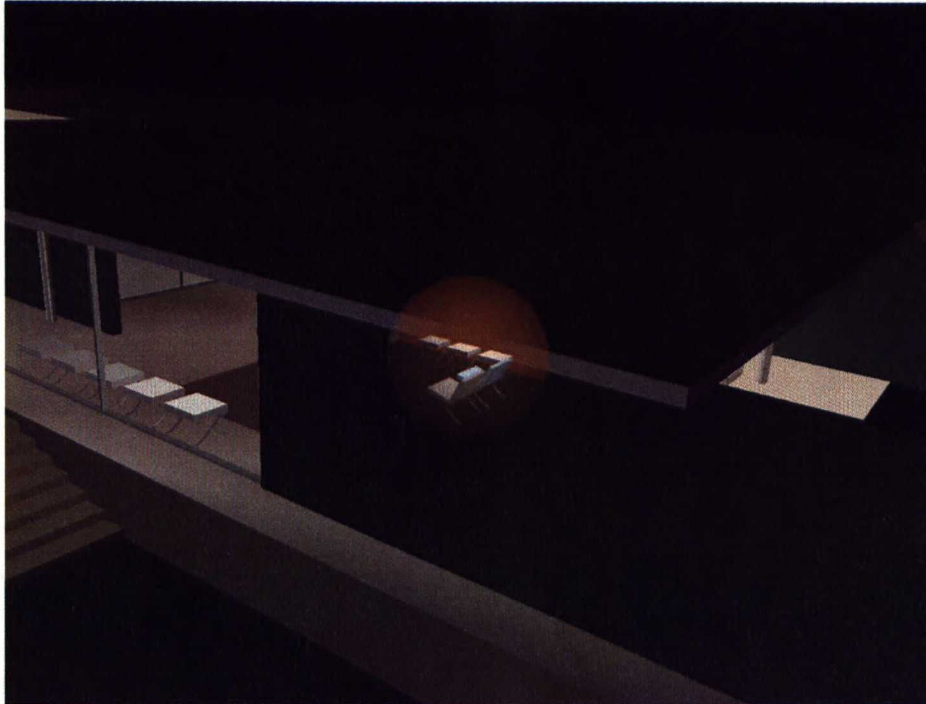


Figure 23 A Magic Light displaying structures below the roof.

This new user interface metaphor reduces the amount of geometry needed to create the Magic Lens - interface, and it suits well to shared multi-user environments. For example, if the lens-interface is used in shared virtual spaces, all of the participants must share the same lenses, either locally or by sharing the same viewpoint to get similar filtered views to the scene. But with Magic Lights the change in the visualization or filtering effect would be seen directly on the surfaces of the objects in the scene.

The manipulation device for Magic Lights can be a flashlight-like device, which can be pointed at the scene. For immersive environments, like CAVE, a wand interaction device can be used as a controlling device for Magic Lights. The Magic Light can also be implemented as a headlight-like interface, where the light always points to the direction of the view.

The Magic Light paradigm consists of three concepts:

1. The viewing frustum
2. The lens frustum
3. The Magic Light texture projection

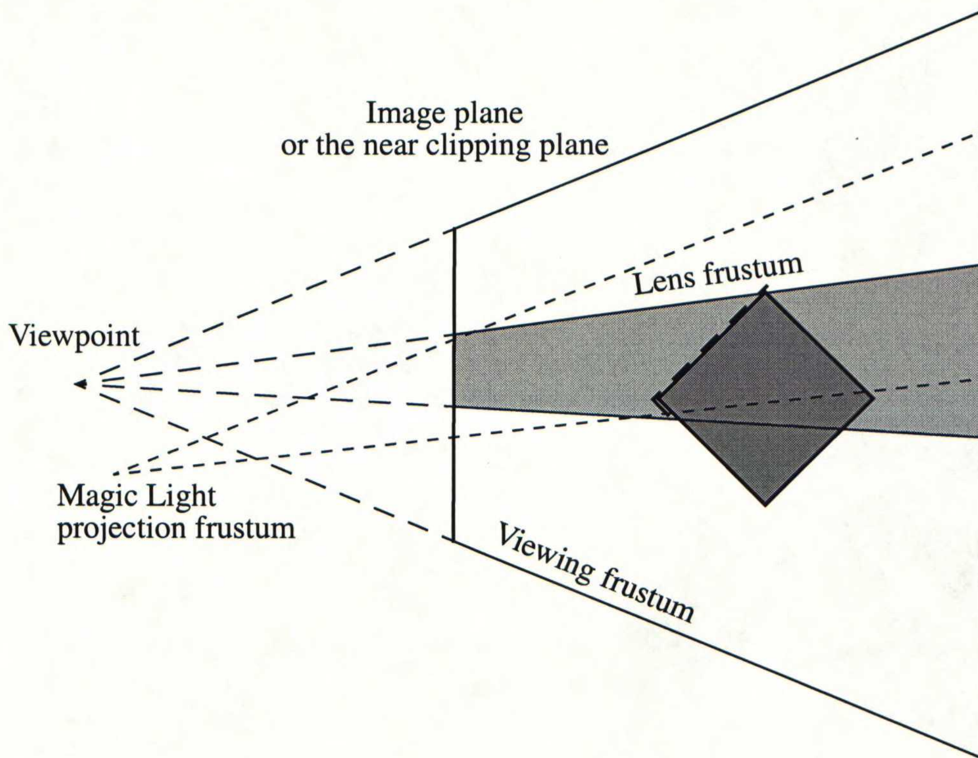


Figure 24 Lens frustum formed by Magic Light, dotted line shows the object surface hit by the Magic Light.

The viewing frustum defines the view to the world. The Magic Light projection frustum encloses the objects like a flashlight beam. The enclosed objects or object surface areas define the region which is filtered. The Magic Light image is generated by using separate lens frustum for projection. See Figure 24 for a diagram.

As with the flat lenses described above, the lens image can be formed in many ways. Some of the more useful ways are:

- The image is from the viewpoint of the Magic Light control device and it is projected onto the geometry using the same projection as for the image.
- The lens image is projected to the scene from the viewpoint of the user, and the light mainly controls the location of the Magic Light image on the viewport.

5.4 Functionality of 3D lenses

5.4.1 Changes in rendering

The simplest way to use Magic Lens filters with a 3D scene is to change the rendering of the selected scene. The rendering modifications can be changes in the shading model of the scene or in the general material properties, like making all materials semitransparent regardless of the original material specification. Also the drawing mode can be changed to transform for example a previously solid scene into a wireframe or bounding box representation.

5.4.2 Changes in lens frustum

The Magic Lenses can also perform a variety of viewing transformations, like field-of-view and clipping. This enables us to create enlarging or shrinking lenses, for example. By modifying the lens frustum suitably, a see-through or X-Ray lens effect can be achieved without the need to use wireframe rendering to achieve the same result.

5.4.3 Visual queries

Another important aspect of Magic Lenses is the ability to do "visual queries" or search for objects that have certain properties. The source scene for this query can be dynamically obtained from a database as well as from a static scene description. The query can be made to select groups of objects with certain properties. These properties can be both visible, graphical properties (material, color) or nonvisible properties, metadata attached to the objects.

These queries can enhance the visible representation of the data: if the visualized dataset has several dimensions, it might be reasonable to limit the number of visualized dimensions. Using Magic Lenses to filter the dataset to a reasonable amount of dimensions can clarify and help explorative searches or visualization.

The visual queries can also import additional data to the visualization, like pipelines or telecommunications cabling which are normally not visible or even not a part of the actual visualized dataset (Stone 1994, p. 306-312).

5.4.4 Modification of the geometry

3D Magic Lenses can be used to change the geometrical representation of objects viewed through the lens. For example, in large scale virtual environments the objects usually consist of several different level-of-detail (LOD) representations for performance reasons. The lens interface could be used to show only a certain LOD level to allow Toolglass functionality regardless of the LOD chosen by the visualization outside the region bounded by the lens.

5.4.5 Temporal modifications

The Magic Lens interface can also be used to modify the time frame of the scene viewed through the lens, if the scene is simulated or recorded as a series of events. For example a time modifying lens can be used to view the scene in the past or peek to the future, if the scene contains objects whose properties change with time. Also the sampling or rendering rate of the lenses can differ from the update rate of the main viewport.

5.4.6 Object manipulation or Toolglass

If the lens metaphor is combined with a set of tools for manipulating objects, the resulting interface is generally faster to use than the normally used sequential mode of interaction. In the sequential mode the user selects an operation from a palette, and then chooses the object to operate on. With Toolglass interface, the palette can be

located over the chosen object so that the selection of both operation and the object happens simultaneously. Thus the set of tools for interaction is more readily available, and the filtering capabilities can ease the task of selecting suitable objects or graphical components as the target of the operation.

For example, the material properties of a rendered object can be changed through a toolglass lens, and at the same time have also the original material definition visualized outside the toolglass region. The two different views available to the scene help in experimentation with the properties, unlike the normal temporal mode, where the user would select a material from a library of materials and then apply the change to an object. Also, with toolglass interface the material palette is available on top of the part of the scene where the interaction takes place, thus minimizing the need for secondary interaction. An excellent example of a Toolglass interface as a drawing and composition aid is presented by Bier (1997, p. 62-70).

Also, a form of X-Ray lens can be used to help with the object manipulation in a virtual environment. For example, the user wants to manipulate an object which is not directly visible because another object or wall blocks the line-of-sight. This can be accomplished by using a lens that changes the graphical representation to be wireframe rendering. A better way to achieve this is to use a lens that clips away the object blocking the view. Thus the user can see the solid rendered object and more easily change its material properties, for example. By modifying the field-of-view of the lens frustum, a telescopic image can be formed to augment in manipulation of objects far away from the user.

5.4.7 Using Magic Lenses As Portals

One interesting application to lens-like interfaces is to use them as surveillance/ teleportation devices. The user could change the lens frustum to visualize objects totally elsewhere in the scene, and thus see the other location while working somewhere else. If something interesting is seen through the lens, the user could select the lens or just walk through it to arrive at the location seen through the lens. A similar approach has been presented by Elvins (1997, p. 21-30). While that imple-

mentation stored static, captured parts of 3D scenes, the VGC (Virtual Graphics Context, to be explained in section "Virtual Graphics Context - VGC" on page 59) / Magic Lens implementation can present the user real-time scenery from selected spots.

5.5 Composition of lenses

Fishkin (1995, p. 415-420) has implemented Magic Lenses as an interface to database queries using scatterplot displays of US census data as the base dataset. In the article Fishkin defines formally a set of composition rules for Magic Lenses. These rules apply to the composition of 3D lenses as well.

Bier (1993, p. 73-80) describes the three methods for producing the composed image of two or more lenses. The most suitable methods for 3D lenses are *Model-In Model-Out* (MIMO), or *Reparameterize-and-Clip*. MIMO-method takes a model representation of the scene, modifies it and passes the modified version to other lenses. Reparameterize and clip modifies parameters, the clips to a region and draws the modified version of the data. From these the latter one is the easiest and fastest to implement in cases where the filter does not change anything else except the drawing parameters of the renderer.

Also, one of the shortcomings of the MIMO-method is that it requires copying of the original model, changing it and possibly passing the modified copy to other lenses. This is surely computationally too strenuous task if the scene graph is large. Using this approach to composite lenses would require a copy of scene graph to be maintained for each lens indenpendently. Thus the possible changes in the main scene graph should also be copied to the scene graphs of the lenses. With small changes to the MIMO-model the effectiveness of this method can be greatly increased.

This can be accomplished by letting the different lenses to modify the copied scene graph by inserting switches (scene graph nodes which can be used to select a single children node, see Figure 25) to suitable points instead of copying the scene graph

structure for each lens separately. Using this method the lenses simply implement the filtering effects by manipulating the switch nodes inserted by the lens that is being rendered at the time. If the scene hierarchy is relatively small memory-wise, the real MIMO-method can also be used.

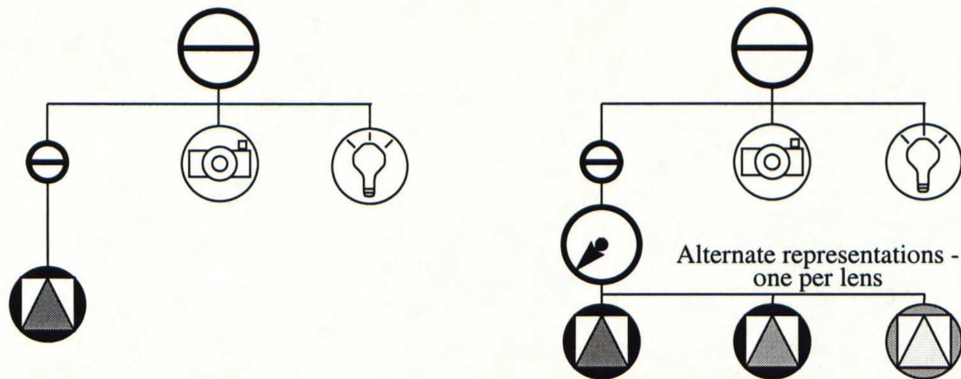


Figure 25 Original scene graph and scene graph after inserting a switch node

Depending on structure of the scene graph and the desired filtering, the *Reparameterize-and-Clip* method can also be very effective, if the effects of the lenses are just changes in the drawing modes or in the lens frustum. For example, wireframe or hidden line rendering and magnification filters are easier to implement with this method. The downside of this method is that it does not cater for all possible compositions, because the composed lenses must have compatible filters.

5.6 Implementation

Flat 3D Magic Lenses can be implemented using several different methods. The earlier implementations of flat lenses have used clipping planes (Viega 1996, p 51-58). This is very ineffective, because the scene needs to be rendered several times per lens. Another approach to this problem is the use of *stencil bit planes*. Stencil bit planes restrict drawing to certain regions of the viewport for multipass rendering. The regions may be of arbitrary shape. With stencilling it should be relatively simple to create flat lenses, and there would be no need for perspective correction of the lens image when the lens is viewed from an angle. This is because the stencilling method just clips the rendered area to be the shape of the lens.

The algorithm to create flat Magic Lenses with stenciling is:

1. Fill stencil mask with value 0
2. Draw the lens shape to stencil buffer with value 1
3. Draw the scene with stencil testing enabled
4. Reverse the stencil testing to enable drawing only to the lens shape
5. Draw the lens image

The stencilling method works well, but it has certain shortcomings:

- Stencilling does not provide the possibility to create true 3D flat lenses for stereographic displays. The true flat lens in this case means that even when the scene is viewed stereoscopically, the image of the lens is always non-stereoscopic.
- It would be difficult to create other image-geometry mappings except see-through lenses.
- The depth of the stencil bit planes is usually limited, which can cause additional problems. A straightforward see-through Magic Lens can be implemented with just a single bit stencil bit plane, though.

Using texture as the lens image, most of these problems can be circumvented, and the flexibility of texturing provides a lot of other possibilities for Magic Lenses. Figure 26 describes the rendering pipeline needed for texture based Magic Lens implementation. The different filtering operations are optional, i.e. the lens image can be passed to a selection of object, scene and image level filtering as needed. The texture

generated by Magic Lens filters can be subjected to even more complicated imaging operations in the graphics pipeline, as opposed to the stenciling or other methods.

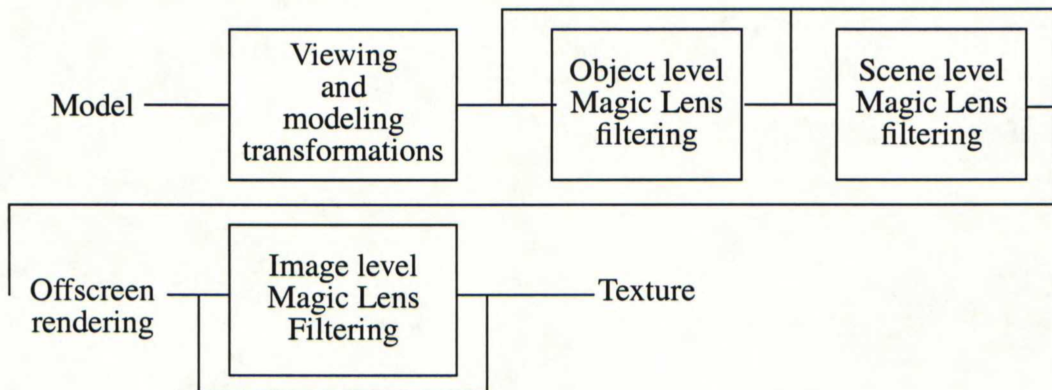


Figure 26 Texture based Magic Lens rendering pipeline

For an efficient texture based approach, a relatively fast path from offscreen rendered images to the texture memory is needed.

The optimum transfer rate in bits/s is: **(EQ 2)**

$$\mathrm{lensRefreshFrequency} \times \mathrm{lensImageResolution} \times \mathrm{lensImageBitDepth}$$

Current high end hardware is generally fast enough for this purpose. In the desktop VR, a more efficient solution is needed because of the required bandwidth. One suitable hardware solution is Unified Memory Architecture - UMA.

5.6.1 Unified Memory Architecture - UMA

The UMA (Silicon Graphics 1997; Kilgard 1997) memory architecture of SGI O2 workstation has no separate frame buffer or texture memory. This architecture was originally meant for enhanced digital media capabilities, like enabling movie clips or video as a texture without a penalty due to constantly copying of data from one memory location to another. One of the useful side effects of this implementation is

that an offscreen graphics context can be used as a texture. When the context is referenced as a texture, the contents of the buffer are read via copy-by-reference instead of copying the contents to a separate texture memory.

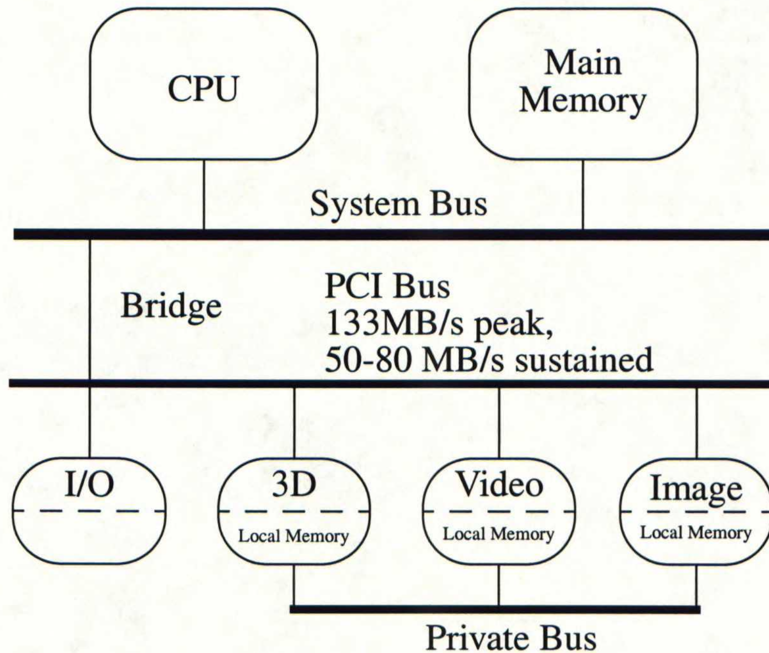


Figure 27 Traditional PC memory architecture

The traditional memory architecture (see Figure 27) limits the transformation rate between additional hardware (Video, 3D graphics accelerators) and the main memory. This limitation is imposed by the relatively slow transfer rate of the PCI bus or the possible private bus and also by the fact that the different hardware components have their own local memory, which cannot be easily accessed from outside of the hardware board. The local memory is usually also specifically constructed to be suitable for a single purpose.

The O2 UMA implementation consists of a single memory unit, which can be accessed from all of the hardware components via the Memory and Rendering Engine (MRE). Figure 28 shows the connectivity between the memory, CPU and co-processors. Because of this configuration, almost all data types stored in the memory

can be used as a source data for more operations performed by the graphics and imaging hardware.

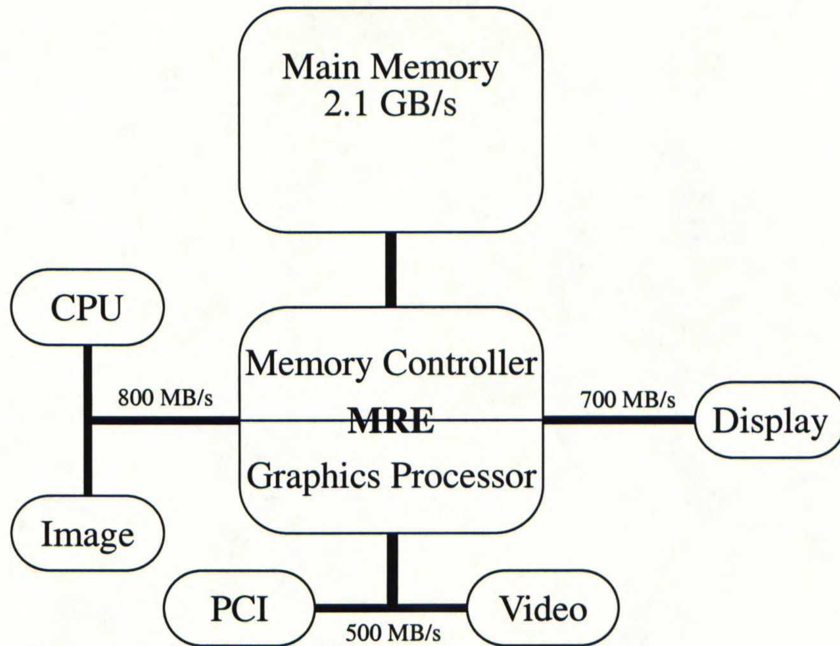


Figure 28 O2 UMA memory architecture

The current trend in PC bus architecture is Accelerated Graphics Port (AGP), in which the bandwidth is notably larger than in the previous PCI (Peripheral Component Interconnect) bus. This architecture can also be used to implement relatively fast Magic Lenses as textures, though there is a penalty because of the copying of the data between the different memory locations.

5.6.2 OpenGL Digital Media Buffers

Digital Media Buffers (DMbuffer) buffers are used to store and exchange time sensitive data, like images in various formats, which can be passed from video to graphics, graphics to video, video to programs, programs to video and as a side effect, graphics to texture. This buffering method used in conjunction with UMA architecture helps to create fast rendered texture implementation.

The DMbuffer extension to OpenGL allows to create an OpenGL pbuffer (pixel buffer) without a color buffer, and then attach the DMbuffer to it as the color buffer. The creation of a pbuffer in pseudocode:

```
GLXFBConfigSGIX config = ...;
int width = 640;
int height = 480;
GLXFBconfig int attribs [] = {
    GLX_DIGITAL_MEDIA_PBUFFER_SGIX, True,
    GLX_PRESERVED_CONTENTS_SGIX, True,
(int) None
};
GLXPbufferSGIX pbuffer;
pbuffer = glXCreateGLXPbufferSGIX(
    display,
    config,
    width,
    height,
    attribs
);
if (pbuffer == None) error();
```

After this, a DMbuffer is allocated from a buffer pool:

```
DMbufferpool pool = ...;
DMstatus s;
DMbuffer buffer;
s = dmBufferAllocate(pool, &buffer);
if (s != DM_SUCCESS) error();
```

The next step is to associate the DMbuffer with OpenGL pixel buffer:

```
DMparams* imageFormat = ...;
DMbuffer  buffer = ...;
Bool ok;
ok = glXAssociateDMPbufferSGIX(
    display,
    pbuffer,
    imageFormat,
    buffer
);
if (!ok) error();
```

When the DMbuffer is used as the target for rendering, the buffer must be made the current drawing buffer by calling `glXMakeCurrent()`:

```
GLXContext context = ...;
Bool ok;
ok = glXMakeCurrent(display, pbuffer, context);
```

To enable the texture read operations to operate using the DMbuffer as source, it must be first attached as readable context with `glXMakeCurrentReadSGI()`. When the texture is copied from the DMbuffer to the texture memory area using `glCopyTexSubImage2DEXT()`, the conditions that enable copy-by-reference texture use DMbuffer are:

- The parameters to `glCopyTexSubImage2DEXT()` must have these values:

```
target = GL_TEXTURE_2D
level = 0
xoffset = 0
yoffset = 0
x = 0
y = 0
width = width of pbuffer
height = height of pbuffer
```

- The format of the already-existing texture must match the format of the pbuffer. These are the pairs of DMbuffer formats and texture formats that match:

  `DM_IMAGE_PACKING_RGBA` and `GL_RGBA8_EXT`

  `DM_IMAGE_PACKING_XRGB1555` and `GL_RGB5_A1_EXT`

- If the image layout is `DM_IMAGE_LAYOUT_GRAPHICS`, then the mipmap extension must be disabled.
- If the image layout is `DM_IMAGE_LAYOUT_MIPMAP`, then the mipmap extension must be enabled.
- No pixel transfer operations can be enabled.

5.6.3 Virtual Graphics Context - VGC

A graphics context consists of a drawable area (memory location, frame buffer) and the current graphics pipeline state. In the past, display of a graphics context has generally been limited to a rectangular viewport, i.e. multiple contexts have been displayed in a rectangular screen region. Because of the UMA architecture the concept of graphics context is somewhat changed. The image of the context can be generated from a viewpoint by a synthetic camera. For this, I introduce a new concept called a *Virtual Graphics Context* or VGC. VGC is a graphics context that can be freely positioned inside the virtual environment, and it is actually a texture image that is dynamically updated when the contents of the VGC change. Due to the UMA architecture, the VGC can be displayed without any need to copy the rendered image from a location to another. The resolution of the virtual context can also be changed dynamically to reduce the sampling artifacts by the texture implementation to produce visually better results. Depending on the implementation, the texture resolution can be obtained by using for example OpenInventor class methods or by more generic method presented in equations below. The *screenWidth*, *screenHeight*, *textureVerticalResolution* and *textureHorizontalResolution* are in pixels, lens image coordinates and distance are in world coordinates.

The best vertical resolution for lens image: **(EQ 3)**

$$\text{textureVerticalResolution} = \frac{\text{screenHeight} \times (\text{lensImageTop} - \text{lensImageBottom})}{2 \times \text{lensDistance} \times \tan(\text{verticalFieldOfView}/2)}$$

The best horizontal resolution for lens image: **(EQ 4)**

$$\text{textureHorizontallResolution} = \frac{\text{screenWidth} \times (\text{lensImageRight} - \text{lensImageLeft})}{2 \times \text{lensDistance} \times \tan(\text{horizontalFieldOfView}/2)}$$

To prevent scaling artifacts from appearing, the resolution must be selected so that the texture is either used as it is or downsampled to a smaller resolution.

From the viewpoint of a programmer the VGC does not differ from a normal graphics context. It can be addressed the same way as an ordinary context.

The flexibility of VGC makes it possible to have multiple freely locatable graphics contexts inside other graphics contexts, thus enabling graphics contexts inside of a virtual space. Because the VCG is a texture, the texturing pipeline and different texturing operations available can be used to enhance the concept of a rendering context even more.

Benefits and disadvantages for 3D Magic Lenses. The benefit of the UMA architecture for flat 3D lenses is the greatly reduced overhead. On the other hand, the solution described will suffer performance loss when implemented on another memory architecture, unless the memory paths between frame buffer and texture memory are sufficiently fast.

The VGC also makes it possible to use the imaging operations of the OpenGL graphics pipeline to generate more complex filtering effects. Because the data is already in a texture format, this reduces the overhead needed for the imaging operations.

Of course, stencilling or other methods for 3D lenses could provide better quality for the image because the image would then be formed with the actual resolution and size. In this implementation OpenGL limits the size of the texture to be a power of two, which will cause small artifacts because the image must be resampled to fit the actual resolution of the lens on the viewport.

5.6.4 Forming the lens image

The basic component forming the lens image is the lens frustum, which contains the position, orientation and the viewing frustum of the camera looking at the scene. All these properties of the camera can also be modified freely. The viewing frustum defines the field-of-view and clipping planes for the camera. The near and far clipping planes define depth limits of the region in space the camera can see. The resulting projection of the scene viewed through the lens camera is then rendered to an offscreen buffer, which in turn can be subjected to imaging operations as texture data. The resulting texture is mapped on the surface of the lens geometry. The texture data can also be projected on surfaces, using an additional frustum or camera to provide the projection matrix. This pipeline in presented in Figure 26, "Texture based Magic Lens rendering pipeline," on page 54.

Mapping the lens image on the geometry. For the last two lens image mappings - the lens frustum follows the position and orientation of the geometry or the frustum is static (see Figure 21 on page 43 and Figure 22 on page 44) - the mapping is straight-forward. If the lens frustum viewpoint follows the viewpoint of the user, the mapping of the image on the geometry can be done by utilizing bounding boxes.

Because the rendered image is always a rectangle, we can use the projection of the bounding box of the lens geometry to the view plane as the basis for the lens frustum. After the bounding box is calculated, the lens frustum is formed by using the corner points of the bounding box to define the corners of the frustum and setting the near

clipping plane (i.e. the image plane, see Figure 29) of the lens frustum to coincide with the nearest point of the lens geometry.

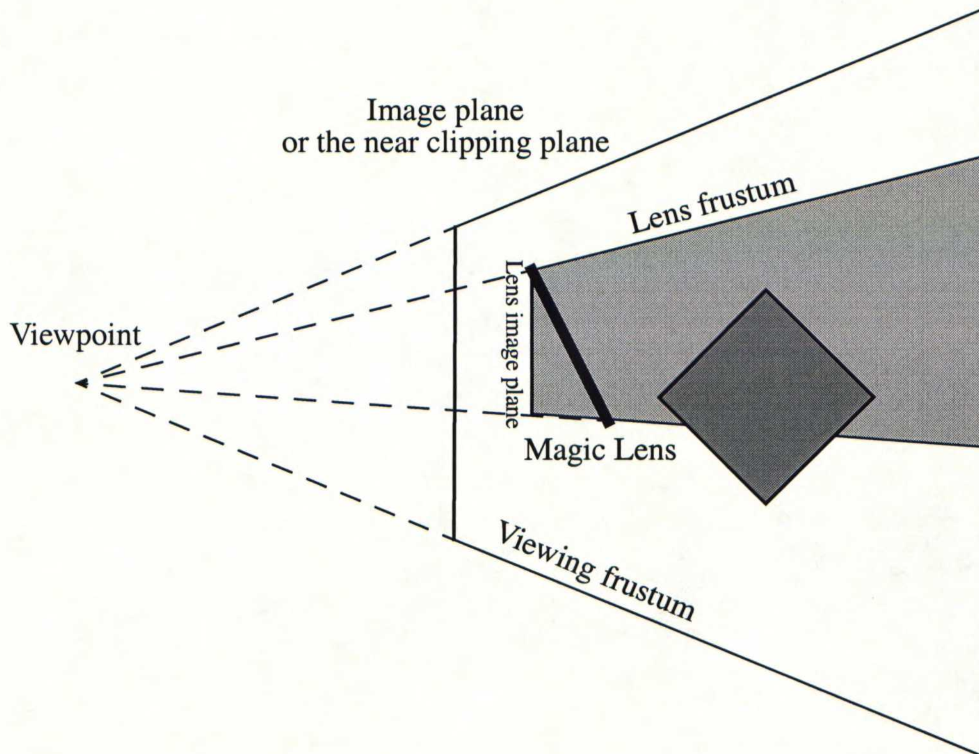


Figure 29 Lens image plane

Perspective correction using projective texturing. Usually texture projection is used for generation of cast shadows or computationally light spotlight effects in real-time graphics rendering APIs. This involves projection matrices, which are normally used in graphics pipelines to calculate the final projection of the scene to the viewport. These projection matrices can also be used to generate texture coordinates in order to make slide-projector like textures in the scene.

The method described in paragraph "Mapping the lens image on the geometry" on page 61 works well if the lens geometry is viewed from the direction of its normal. But if the lens is viewed from an angle, the texture will not display a correct image. If the angle between the normal of the lens geometry and the current viewpoint is small,

the distortion is only a slight perspective distortion. When the angle grows, the distortion becomes more noticeable. Examples of perspective distortion and corrected image can be seen in Figure 30 and Figure 31 on page 64. A simple, clarifying image using checker texture is portrayed in Figure 32 on page 64.

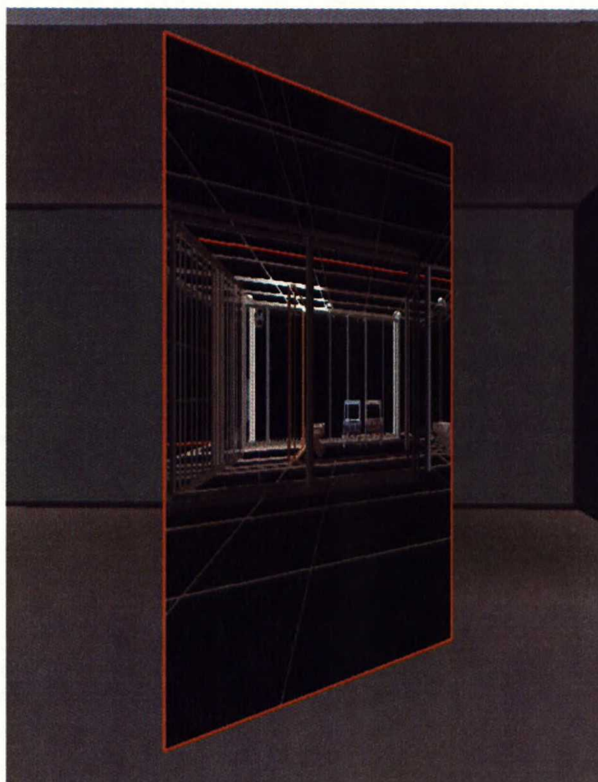


Figure 30 Distortion caused by incorrect perspective.

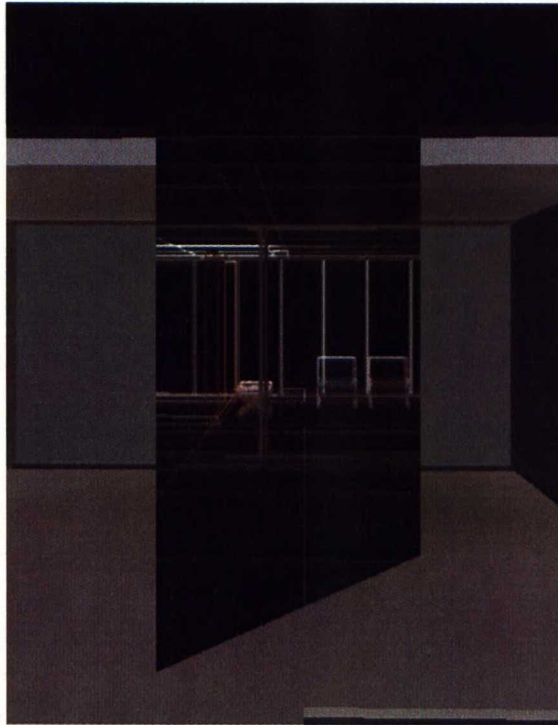
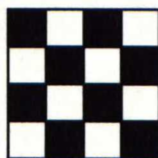
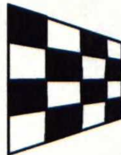
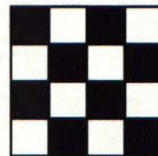


Figure 31 Perspective distortion corrected with projective texturing.







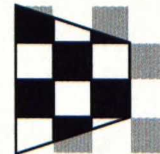


Figure 32 Lens distortion correction using projected textures.

The texture can in additionally suffer from distortion if the lens is located off the main viewpoint projection axis. The distortion is caused by the fact that the image plane (in OpenGL, the image plane and near clipping plane usually coincide) is always perpendicular to the view vector if the viewing frustum is symmetrical. This error presents itself normally as a slight registration error, though it is not caused by bad registration.

The image can be corrected by using an asymmetric frustum as the lens frustum, as depicted in Figure 33.

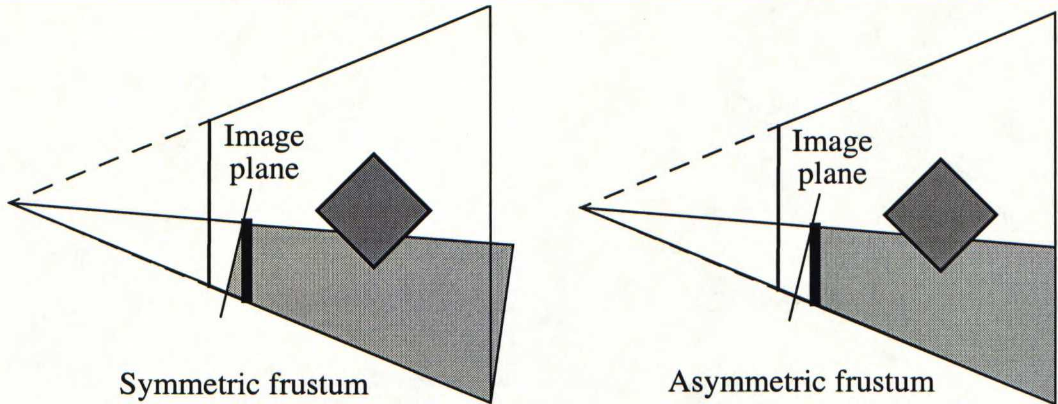


Figure 33 Lens distortion correction using asymmetric frustum

The asymmetric frustum is not necessity for the image correction. In this implementation, the image is used as texture. It can be corrected using the projection techniques described below. An example of the uncorrected situation is in Figure 34 on page 66 and the circled area enlarged in Figure 35 on page 67. The corrected image is in Figure 36 on page 68. Furthermore, asymmetric frustum can only be asymmetric in

relation to the view axis. Perspective distortion cannot be fixed, because in that case the shape of the frustum should be trapezoidal.

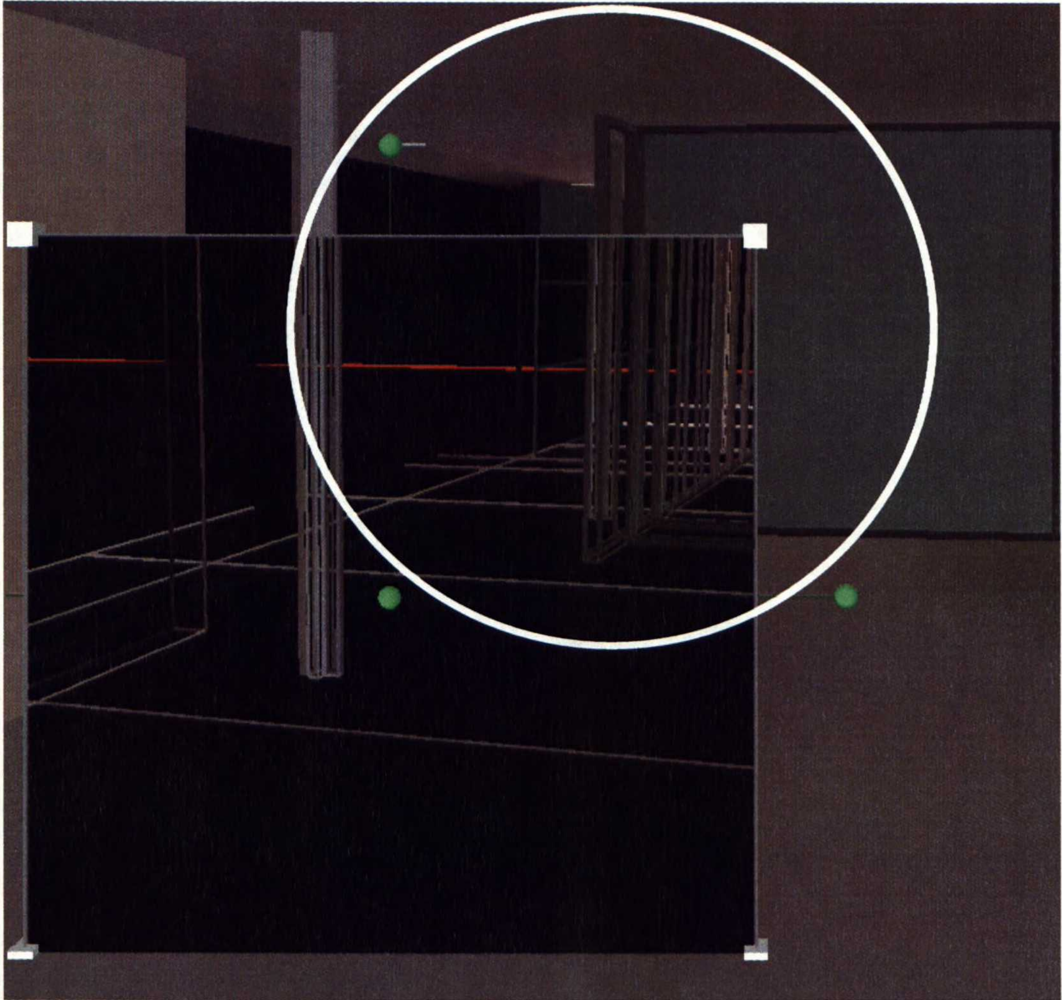


Figure 34 Off-axis distortion.

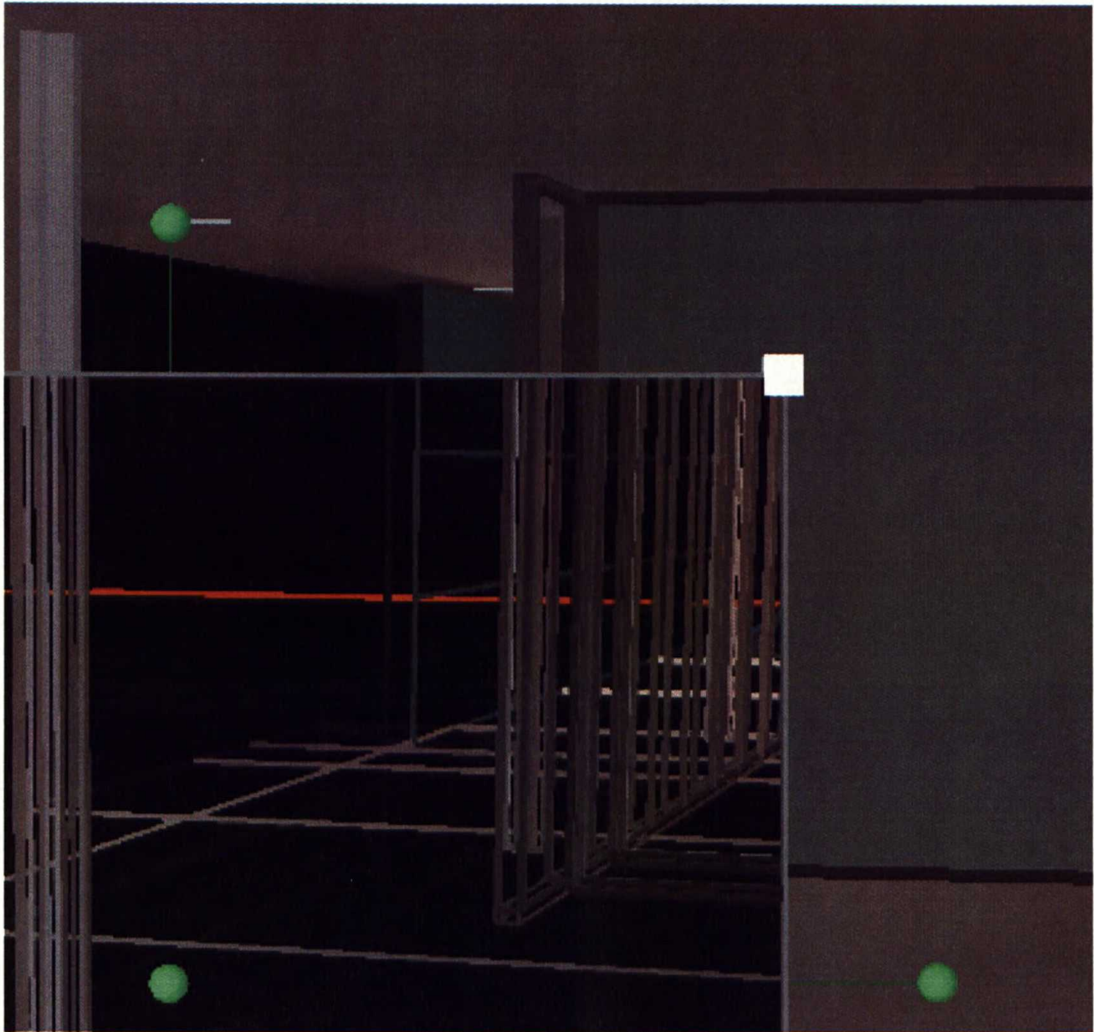


Figure 35 Off-axis distortion, enlarged view.

These errors can be corrected by using the same technique as for real-time satellite image orthorectification, because the cause of the problem is the same. In the image orthorectification, the photographed image is projected to the surface using the camera which took the picture as the projector, which automatically removes any distortion of the texture. By using projective texturing (Segal 1992, p. 249-252) to act as a projector through which the lens image is projected on to the lens geometry, the perspective correction is performed automatically. In this case, the texture coordinates are generated automatically for the lens geometry by the graphics pipeline, but they can also be calculated in the software. The software option does not provide pixel-

perfect result, because the texture coordinates can only be fed to the graphics pipeline per vertex.

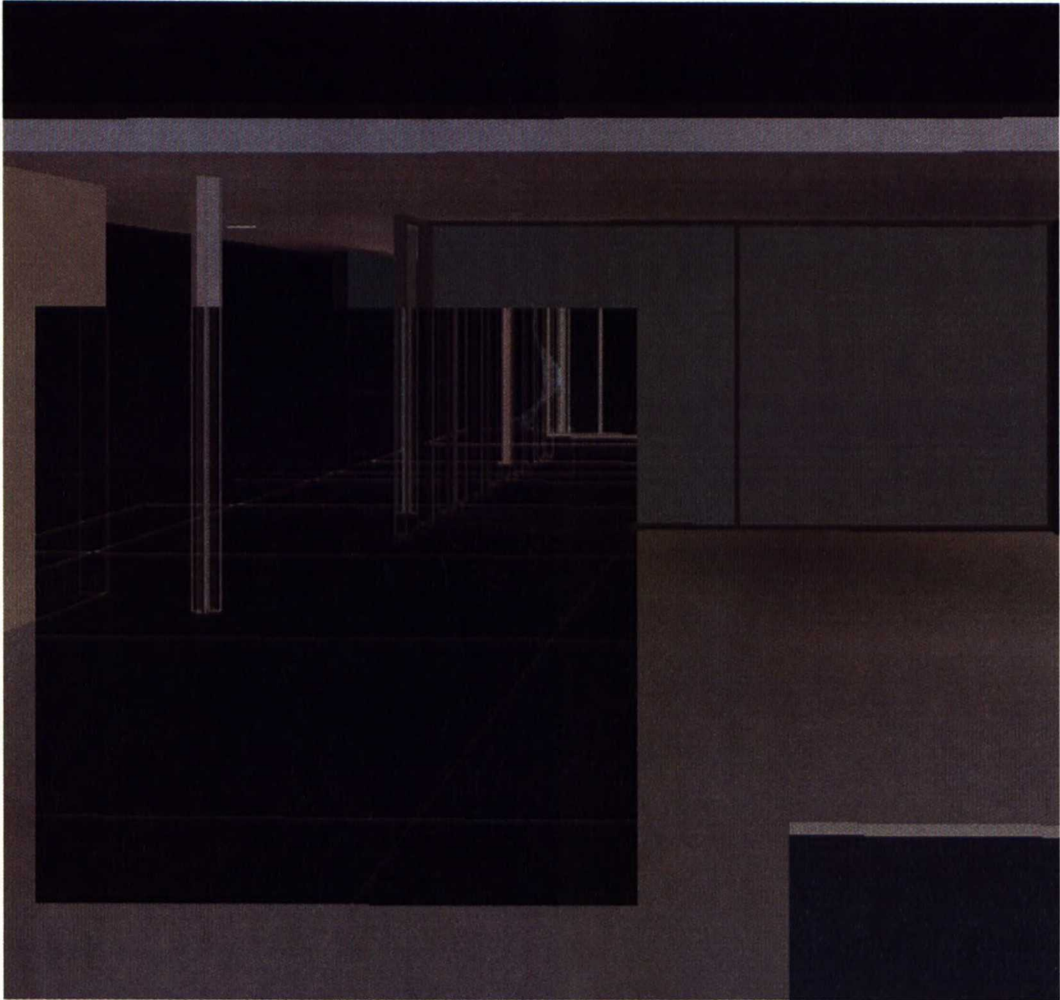


Figure 36 Off-axis distortion corrected by using projective texturing.

Because the approach of this work was to implement a fast texture based Magic Lens, doing projective texturing with software was out of question. Unfortunately, the O2 OpenGL implementation has a bug that appears when projective texturing is used in conjunction with digital media buffers (See appendix "Projective texturing bugs in O2 OpenGL implementation" on page 89). The bug has been acknowledged by SGI and the fix has been promised to be in IRIX 6.5.1 release, autumn 1998. The images that show the perspective corrected images are generated using a captured texture image as a normal texture source. The effect achieved is the same with digital media buffers, if they could have been used.

Texturing in OpenGL. OpenGL allows the user to define various parameters of the texture. These include the texture coordinates, texture modulation/blending and texture repeating/clamping. For the purposes of the implementation described in this work, modulation and clamping operations are important. The texture modulation provides the possibility to define textures as opaque or semitransparent with several variations considering the underlying fragment color and alpha values. OpenGL also allows the user to specify texture coordinates outside of the normal range [0,1]. In this case, the clamping modes describe how the texture coordinates should be handled.

The OpenGL specification defines two modes for texture coordinate handling:

1. Repeating textures
2. Clamped texture coordinates

For projective texturing, the repetition of the texture coordinates is not desirable. The clamping limits the texture coordinates to [0.0, 1.0] so that the rest of the surface outside of the texture area for borderless textures is painted using a linear combination of the two last pixels on the opposite edges of the texture.

To augment projective texture usage as spotlights, SGI has specified extensions for the texture clamping: `SGIS_texture_border_clamp` and `SGIS_texture_edge_clamp`. The border clamp extension will always use the specified border color or existing texture border for texels outside the texture coordinate range, thus making it simple to create nonrepeating textures with transparent border colors. Unfortunately he OpenGL implementation on O2 does not contain `SGIS_texture_border_clamp`-extension. This complicates the projective texturing implementation a bit. The texture clamping implemented in O2 allows the programmer only to clamp to the edge of the texture using `SGIS_texture_edge_clamp`, or to use the normal clamping. If the texture is projected using the above mentioned clamps, the last pixel or the combination of the two last pixels will repeat over the textured object. For Magic Lenses this is not a

problem, because the texture is formed using the bounding box of the object so that the geometry of the lens object will clip the texture correctly.

Using texture for Magic Lights. The Magic Light interface can be implemented by using the hardware accelerated texturing capabilities. The lens frustum and Magic Light texture projection can be implemented as different projection matrices.

The texture projection matrix forms an "invisible" Magic Lens geometry. If the lens frustum does not change the FOV (no zooming), the projection matrix can be the viewing frustum. The projection matrix can also be orthogonal instead of the normal perspective projection. See Figure 24 on page 47 for a diagram.

For Magic Lights, the repeating of the border pixels corrupts the rendered view. If the O2 OpenGL would allow the programmer to clamp to specified border color (RGBA), there would be no need for other techniques.

To prevent clamping artifacts, the texture image will have to be modified after the Magic Light image has been rendered by drawing at least one pixel wide frame with RGBA values 0,0,0,0 around the texture. Additionally, an alpha map can be used to

obtain different shapes for the Magic Light. Figure 37 displays an alphamap used for a spotlight effect.

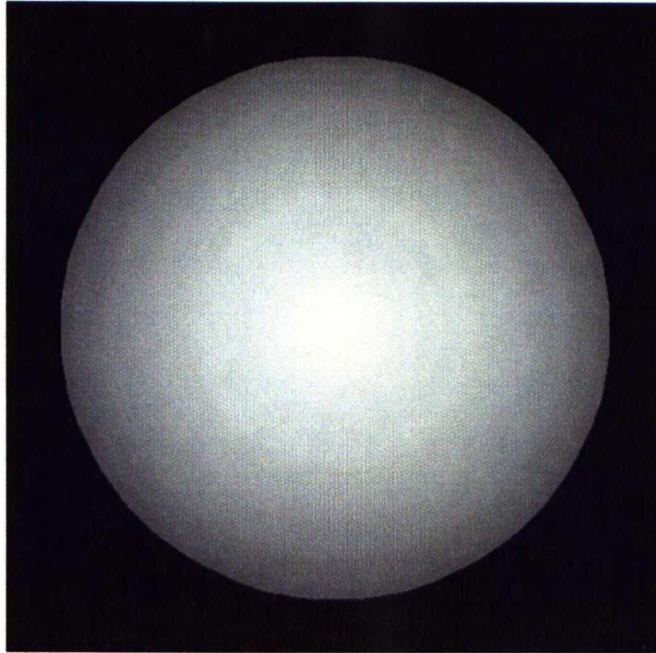


Figure 37 The associated alphamap for the Magic Light.

When projective texturing is used, the OpenGL-pipeline must be reversed to get the normalized texture coordinates to be in the usable [0, 1] range. The process for reversing the pipeline for texture matrix is:

1. Set the texture matrix to identity
2. Scale the matrix by 0.5, 0.5, 1.0
3. Translate the matrix by 1.0, 1.0, 0.0
4. Multiply this matrix with the projection matrix
5. Multiply the result with inverted model view matrix

Composition limitations with multiple lenses. For two dimensional Magic Lenses the composition of the lenses is straightforward, because the composition is always clearly defined by the filtering rule sets and the lens region. The same applies for 3D volumetric lenses, which will be composited only when the lens volumes overlap.

For flat 3D lenses the composition is more complicated, because the above mentioned different lens/geometry relationships and lens frustums might render the reasonable composition impossible in several cases. For example, if the lenses have different viewing frustums (different FOV etc.), the resulting intersection of the frustums can be asymmetrical. Rendering asymmetrical frustums is complicated with Open Inventor, and the result may well be something else than what the user was trying to accomplish.

5.6.5 Picking and other interaction

Because the lenses have their own graphics contexts, picking of the objects seen through the lens can be easily implemented by using a ray picking method. When the user selects or picks an object from the lens, the starting point of the pick is generated by first obtaining the texture coordinate of the pick event, and transforming that to the virtual graphics context viewport coordinates. The direction of the picking ray can then be calculated from the original screen coordinates and the picked point on the lens texture. By using the near and far clipping planes of the lens graphics context to limit the length of the ray to the visible portion of the scene graph, a ray pick action can then be constructed to select the objects both in the actual scene and the lens scene. For example, it is possible to easily pick objects that are inside other objects in the main scene. Because the selection was based on the spatial properties of the scene, accurate selection of objects both in the filtered scene graph and unfiltered is possible. Of course, if the lens displays the scene in the past or in the future, this selection method fails.

For scenes where the spatial or temporal location of the objects does not provide enough information to select objects non-ambiguously, other methods must be used. The most applicable is to provide additional metadata, or names, for the objects on the scene and use that information for selection in other scene graphs.

5.6.6 Open Inventor™

Open Inventor (Strauss, 1992; Wernecke, 1994) was chosen as the software platform because it has a reasonably flexible scene graph structure, which can contain *engine* nodes which enable dynamic content. Open Inventor (see Figure 38 on page 73) is a window system-independent object-oriented 3D toolkit, which provides a library of extensible objects. It can also be extended by using Open Inventor API in combination with OpenGL. The Inventor specification also defines an interchange file format.

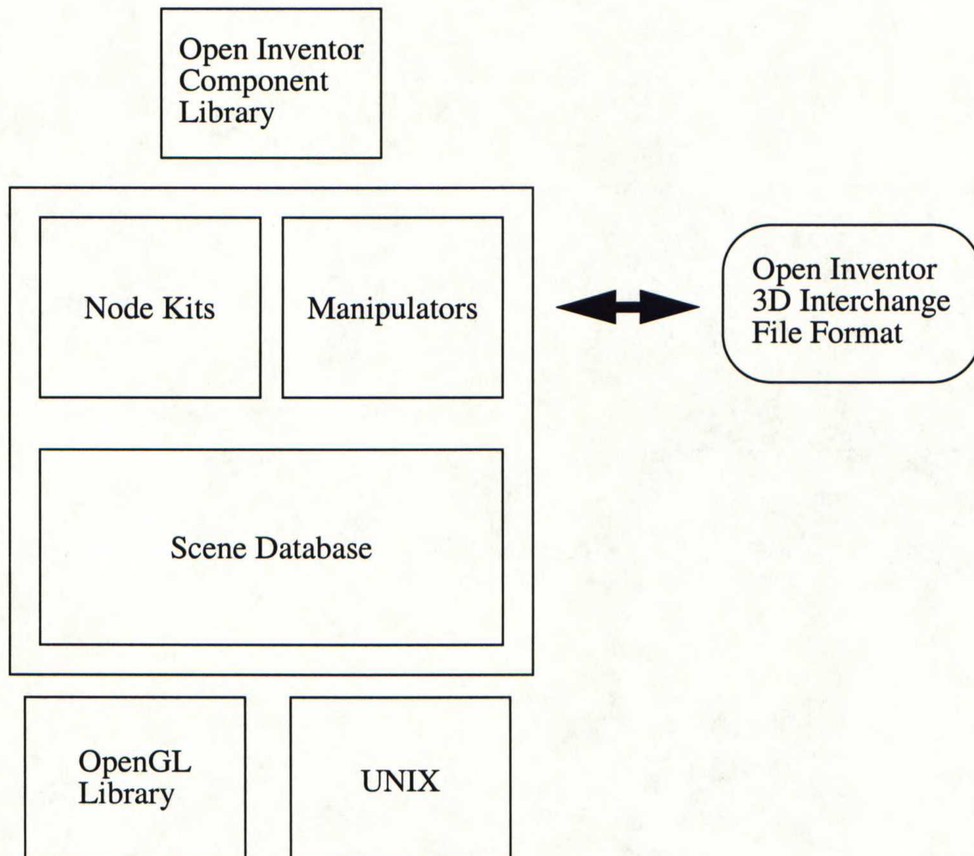


Figure 38 Inventor architecture

The Inventor objects include:

- Database primitives

  e.g. shape, property, group and engine
- Manipulators

  e.g. trackball, handlebox
- Components

  e.g. material editor, light editor and viewers

The engine nodes are objects that can be connected to other objects in the scene graph and used to animate parts of the scene or constrain certain parts of the scene in relation to other parts. The engines in the standard Open Inventor distribution can even be used to perform simple calculations without any explicit programming, if the creation of the scene graph does not count as programming.

5.6.7 Extensions to OpenInventor™

For the Magic Lens interface, several extensions based on the OpenInventor classes, both from the standard distribution and from the Digital Media Buffer extension library were made.

Lens viewer. To provide an interface for the Magic Lens operations, a generic Inventor viewer was implemented. The Lens Viewer handles the IO and provides a framework for interaction. It provides a way of opening Open Inventor scene graphs stored in the Inventor file format and adding lenses to the scene. The viewer handles the picking through the lenses by calling a callback defined for the lens picked lens instance. The operation returns the ray constructed for the pick through the lens, which is in turn used to pick objects in the main scene graph. The viewer also creates the menus and dialogs for changing the lens parameters.

Offscreen renderer. The Magic Lenses-interface in this work is built around an offscreen renderer which generates the texture needed for the lens. The offscreen renderer also implements the viewing semantics for the graphics context. The pixel

buffer for the offscreen renderer is a digital media buffer, which is associated with a graphics context. The graphics context is in turn utilized as texture data by using the `glCopyTexSubImage2DEXT()` library function, which references the texture instead of copying the data, if the current read buffer for OpenGL is a digital media buffer and the conditions mentioned in "OpenGL Digital Media Buffers" on page 56 are met.

Because the resolution of the offscreen buffer can be adjusted, there is no need for expensive mipmap filtering to reduce the sampling artifacts if the texture is viewed from the distance. This actually creates a dynamic LOD for the texture, thus reducing the computational load created by the lenses. Of course, the size of the buffer can also be fixed, if it is large enough for the needed texture sizes.

The offscreen renderer is an engine node, which enables the framerate of the lens differ from the actual update rate of the main viewscreen. Because of this we can ease the computational requirements if we can live with lower framerates. The independentness of the lens image update rate and time also makes it possible to peek to the future or to view the scene in the past. The output of the offscreen renderer or engine can be connected to a texture node as the image data anywhere in the scene; usually the target of the texture is the geometry of a lens. The shape of the lens can be arbitrary, because the texture coordinates can be explicitly defined in the lens geometry specification. Of course, the offscreen image is always a rectangle, so depending on the lens shape some of the texture data can be lost. The renderer, `SoDMBufferRenderEngine`, is subclassed from `SoDMBufferEngine` class, belonging to `libdmuiv` package provided by SGI to help DMbuffer development with Inventor. The `libdmuiv` package does the low level DMbuffer initialization and management. `SoDMBufferRenderEngine` replicates the functionality of `SoXtRenderingArea` without the Xt dependencies. It provides the basic functionality needed for rendering Open Inventor scene graphs to an offscreen buffer, which is in this case a DMbuffer. To add viewing semantics to rendering area, the class `SoDMBufferLensViewer` is derived from the `SoDMBufferRenderEngine`. This subclass provides the methods for camera manipulation, scene graph access and

draw style changes for *reparameterize-and-clip* type of lenses. Figure 39 describes the basic rendering hierarchy.

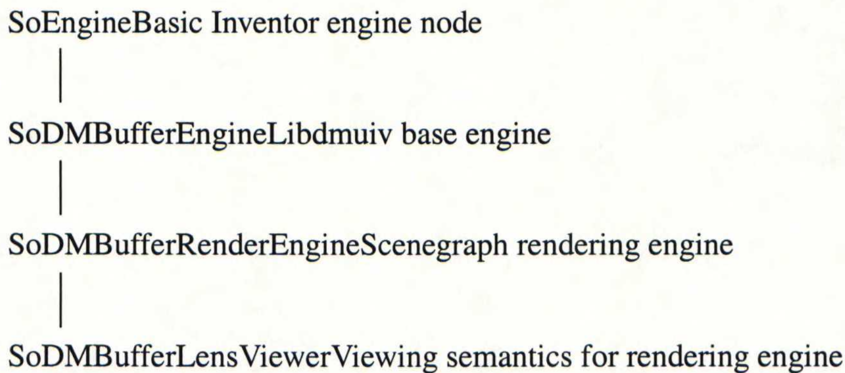


Figure 39 Digital media buffer renderer class hierarchy.

Supporting nodes. To help experimenting with different types of Magic Lens types and behaviors, some supporting node types were written. These are:

- Billboard, which maintains the orientation of its children nodes constant in reference to the viewpoint by manipulating the associated matrices according to the nodes parameters.
- HUD, which keeps the children nodes at a constant location in reference to the viewport, providing a Heads Up Display - like operation. This node type can be used in conjunction with 3D Magic Lenses to supply a traditional 2D Magic Lens operation.

Projective texturing for Inventor. Open Inventor allows the user to specify two different texture coordinate functions: plane and environment mapping. These classes are based on `SoTextureCoordinateFunction`-class. This class and its subclasses allow the texture coordinates to be generated either by the software itself or OpenGL. To create projective texturing for Open Inventor, `TextureCoordinateProjection` class is derived from the above mentioned base class. This class is contains only one additional field, a projector. The projector field can contain a

node derived from `SoCamera`. The camera node is used to provide a projection matrix for the texture, and by using a ready-made camera the functionality of the camera class can be used. When the rendering callback method, `GLRender()`, is called, it sets the current `TextureCoordinateElement` to use a callback function for the texture coordinate generation. This callback will enable the OpenGL texture coordinate generation in eye linear space. The nodes in the Open Inventor scene graph will continue to provide their own texture coordinates, but they are not accounted for when the automatic generation is turned on. Due to the structure of Open Inventor toolkit, this redundant workload cannot be turned off, because it would disable texturing altogether. The following sample from `GLRender()` - method shows how to create texture projection matrix (the process is actually a reversed OpenGL transformation path):

```
if (projector.getValue() != NULL) {
    // Model matrix from current model matrix
    // element
    modelMat = SoModelMatrixElement::get(state);
    invModelMat = modelMat.inverse();
    // Projection matrix from projector field
    projMat = ((SoPerspectiveCamera *)
        projector.getValue())->
        getViewVolume().getMatrix();

    SoTextureMatrixElement::makeIdentity(state,
                                         this);

    // Scale and translate to get the normalized
    // device coordinates to 0..1 range
    SoTextureMatrixElement::scaleBy(state, this,
                                    SbVec3f(0.5, 0.5, 1.0));
    SoTextureMatrixElement::translateBy(state, this,
                                        SbVec3f(1.0, 1.0, 0.0));
    // Multiply texmat element by projection matrix
```

```
        SoTextureMatrixElement::mult(state, this,
                                projMat);
        // And multiply it with inverted modelview matrix
        SoTextureMatrixElement::mult(state, this,
                                invModelMat);
        texMat =  SoTextureMatrixElement::get(state);
    }
```

MagicLensKit - actual lens operations. The Main Magic Lens component, `MagicLensKit` - class is derived from `SoWrapperKit` - class. The Open Inventor node kits are a way to create simple or complex collection of nodes, i.e. subgraphs. They can also contain other node kits to cater for more complex hierarchies. The `SoWrapperKit` allows wrapping of arbitrary, non-nodekit scene graphs within a SoSeparatorKit so that it can be used with other shape kits in a hierarchy. Similar kit is used for Magic Lights.

The MagicLensKit and MagicLightKit classes handle the instantiation of the lenses, construct the lens geometry and DMBuffer textures. The geometry construction methods are virtual to provide expandability for different lens geometries. They also provide methods for interaction with the lenses. The geometry contents can also be changed from outside of the kit. The kits handle the rendering of the lenses in a single callback, which in turn calls private methods to render different types of lenses:

- The lens image is from a static point in the scene
- The lens camera follows the viewpoint of the viewer, creating see-through lenses
- The camera is attached to the lens geometry so that the lens image changes only when the geometry itself is moved to another location.

Selecting objects through the lenses. The selection of scene objects through the lenses is implemented with a ray pick - method. This allows the user to select objects even when they are inside other objects, if the lens filter allows the objects to be visible. The following code portion constructs the picking ray:

```
// Get the lens camera and viewport
camera = viewer->getCamera();
camera->getViewportBounds(viewPort);
rpAction.setViewportRegion(viewPort);
// Get the texture coordinates and the viewport
// point of the pick
pickPoint4 = pick->getObjectTextureCoords();
pickPoint4.getValue(s, t, u, v);
viewPortPoint = SbVec2f(s, t);
// Construct a world space ray for picking
vVol = camera->getViewVolume();
near = camera->nearDistance.getValue();
far = camera->farDistance.getValue();
startPointVec = camera->position.getValue();
sightPointVec = vVol.getSightPoint(far);
planePointVec = vVol.getSightPoint(near);
directionVec = vVol.getPlanePoint(near,
    viewPortPoint) - startPointVec;
// Normalize dir vector due to Inventor bug
directionVec.normalize();
// Construct a ray pick action
rpAction.setPickAll(FALSE);
rpAction.setRay(startPointVec, directionVec,
    near,far);
rpAction.apply(viewer->getSceneGraph());
SoPickedPoint *pp = rpAction.getPickedPoint();
viewer->applyPick(pp);
```

Once the picking ray is constructed, the object selection is straightforward. The picking ray is constructed using the frustum of the Magic Light, so it shares the coordinate system of the main scene. Thus the same ray can be used in selecting the actual object from the main scene graph. The picking action can be simply applied again for the main scene graph. The picking action will return all of the objects which

intersect the path of the ray. For perspective projection, the picking ray is actually a cone shaped volume, extending outwards to help in the selection of the remote objects.

6 CONCLUSIONS

This work has shown that the Magic Lens and Toolglass metaphors are quite versatile and can be used also in virtual environments. The Magic Lens interface is a promising tool for manipulation and visualization in virtual environments. However, the Magic Lens research and development seems to be concentrating on the standard 2D interaction. The 3D interaction tools based on the lens concept is an interesting research area that should not be overlooked. This work provides an adequate basis for the continuing research on the topic.

6.1 2D Magic Lenses and 3D visualization

The 2D lenses combined with 3D visualization presents a solution for cases where the visualization benefits from the combination of these tools. In this work, the interface was used for navigation and wayfinding in virtual spaces. The 2D lens implementation showed that Magic Lenses interface can be easily and successfully implemented in a Java environment using the standard Java classes. The mapping between the 2D lenses and the 3D view can probably cause problems for the users, because it is not clearly visible in the user interface. There should be an indication of the lens types and their mapping in the user interface. Additionally, users should be able to modify the filtering properties of the lenses.

The 3D side of this part of the work would have benefited from a more versatile scene graph than the one used in VRML. Unfortunately, the Liquid Reality toolkit that was the basis for the 3D visualization implementation is no longer in existence. There is no suitable substitute for Liquid Reality because the EAI is not powerful enough to provide the functionality needed for the 3D visualization. Also, VRML seems to be losing popularity so there are no serious efforts to create a decent API for extending VRML. MPEG-4 could perhaps be used to implement this type of solution, but the standard is still incomplete. Also, the Java based implementation was not fast enough for large scale virtual environments. Both the lens interface and the visualization would have benefited from a faster run-time environment.

The solution should be tested in real world applications to find out the real benefits for users. Currently there are several projects concerning virtual cities and large virtual spaces using VRML for the modeling and interaction. These could be used as the test cases.

6.2 3D Magic Lenses

3D Magic Lenses - section of this work transported the 2D lens metaphor to a 3D virtual space. The metaphor works well in this environment, especially if the environment is an immersive one. Being able to manipulate visualization and/or objects through a single, easily understandable metaphor helps the interaction. This part of the work concentrated mainly on the implementation of 3D lenses and making the implementation as fast and versatile as possible. It should be noted, however, that the lenses need to be manipulated as well. The best choice for manipulation of the lenses is to use an external, physical control device. If the lenses themselves contain widgets for control and manipulation, one of the main benefits of the metaphor would be lost: reduced visual cluttering of the interface.

The implementation described in the work suffered from the bugs in the SGI O2 OpenGL. To test the effectness of the Unified Memory Architecture, the software will be ported to normal memory architecture. The implementation should have been more modular to allow the use of Magic Lens components in general Open Inventor applications by utilizing the Open Inventor built-in DSO (Dynamic Shared Object) loader. Currently the implementation uses a custom Inventor viewer. Additionally, different methods of embedding the lenses into the scene graph should be reviewed and tested.

The 3D lens metaphor should also be tested with a real world application to see the possible benefits for manipulation and visualization. The impact on immersive environments (such as a CAVE) should be tested, too. The suitability of different user interface devices, like wands, data gloves etc. for 3D Magic Lens manipulation and operation is also an uncharted territory. Probably the interface devices pose no difficulties and two handed operation is beneficial for the lens metaphor. Additionally,

new interface devices resembling a magnifying glass could be utilized for Magic Lenses in immersive environments.

6.2.1 Magic Light

The new Magic Light - concept presented in this work is more suitable for immersive virtual environments than to the normal desktop VR. It also will be tested in a suitable environment in the near future.

6.2.2 Augmented Reality

The Magic Lens tools could be beneficial for AR applications because of their unobstructive nature. Flat 3D Magic Lenses can be used as normal Magic Lenses but also as active post-it-notes in the augmented environment. These tools could be used to enable the user to see though walls to detect embedded wiring or pipelines. The Magic Light metaphor should work well in AR applications, because the user interface device can simply be a flashlight-like device, which the user can point at the real environment to control the location and other aspects of the Magic Lights.

7 REFERENCES

Ayers, Matthew; Zeleznik, Robert. 1996.
The Lego interface toolkit.
UIST '96. Proceedings of the ACM symposium on User interface software and technology, p. 97-98

Bier, Eric A.; Stone, Maureen C.; Pier, Ken; Buxton, William; DeRose, Tony D. 1993.
Toolglass and Magic Lenses: The See-Through Interface.
Proceedings of Siggraph '93 (Anaheim, August). Computer Graphics Annual Conference Series, ACM, p. 73-80.

Bier, E. A.; Stone, M. C.; Fishkin, K.; Buxton, W.; Baudel, T. 1994.
A Taxonomy of See-Through Tools.
Proceedings of CHI '94. p. 358-364.

Bier E.A., Stone M.C., Pier K. 1997
Enhanced Illustration Using Magic Lens Filters.
IEEE Computer Graphics and Applications
IEEE Computer Society, 1997, p. 62-70.

Brown University Computer Graphics Group. 1996.
3D User Interfaces for Scientific Visualization within the Trim and FLESH environment.
<URL:http//www.cs.brown.edu/research/graphics/research/vrinter/fflow.html>

Darken R.P., Sibert J.L. 1996
Wayfinding Strategies and Behaviours in Large Virtual Worlds.
In Proceedings of ACM CHI 96
ACM, NY, 1996, p. 142-149

Dimension X, 1996-1997

Liquid Reality,

<URL:http://www.dimensionx.com/products/lr/>

Elvins, T.T.; Nadeau, D.R.; Kirsh, D. 1997

Worldlets - 3D Thumbnails for Wayfinding in Virtual Environments.

Proceedings of UIST '97

ACM Press, NY, 1997, p. 21-30.

Fishkin, Ken; Stone, Maureen; 1995

Enhanced dynamic queries via movable filters.

Proceedings of SGICHI '95, p. 415-420.

Forsberg, Andrew; Herndon, Kenneth; Zeleznik, Robert. 1996

Aperture Based Selection For Immersive Virtual Environments.

Proceedings of UIST `96, p. 95-96.

Green, M.; Jacob, R. 1990

SIGGRAPH '90 Workshop Report: Software architectures and metaphors for non-WIMP user interfaces.

Computer Graphics, v. 25, p. 229-235.

Hartman, Jed; Wernecke, Josie. 1996

The VRML 2.0 Handbook - Building Moving Worlds on the Web

ISBN 0-201-47944-3

Addison-Wesley Publishing Company, Inc.

Herndon, Kenneth P.; Meyer, Tom. 1994

3D widgets for exploratory scientific visualization.

Proceedings of UIST 94, p. 69-70.

Kabbash, Paul; MacKenzie, Scott; Buxton, William. 1993

Human performance using computer input devices in the preferred and non-preferred hands.

Proceedings of InterCHI '93, p. 474-481.

Kilgard M. 1997

Realizing OpenGL: Two Implementations of One Architecture.

1997 SIGGRAPH / Eurographics Workshop on Graphics

ACM Press, NY, 1997, p. 45-56.

MetaCreations. 1997

Kai's Power Tools 3, KPT Lens f/x.

<URL:http://www.metacreations.com/products/kpt/>

Segal M., Korobkin C., van Widenfelt R., Foran J., Haeberli P. 1992

Fast shadows and lighting effects using texture mapping.

Proceedings of SIGGRAPH '92

ACM Press, NY 1992, p. 249-252

Silicon Graphics, 1996

Unified Memory Architecture

<URL:http://www.sgi.com/Technology/uma.html>

Spacetec IMC Corporation,

Spaceball

<URL:http://www.spacetec.com/>

Stoakley, Richard; Conway, Matthew; Pausch, Randy. 1995

Virtual Reality on a WIM: Interactive Worlds In Miniature.

Proceedings of the 1995 ACM SIGCHI Conference, p. 265-272.

Strauss, Paul S.; Carey, Rikk. 1992

An object-oriented 3D graphics toolkit.

Proceedings of SIGGRAPH '92

ACM Press, NY 1992, p. 341-349.

Stone, Maureen; Fishkin, Ken; Bier, Eric. 1994

The Movable Filter as User Interface Tool.

Proceedings of CHI '94, p. 306-312.

Sun Microsystems, 1991

Java™ Technology Home Page

<URL:http://java.sun.com/>

Viega, J.; Conway, M.; Williams, G.; Pausch, R. 1996

3D Magic Lenses.

Proceedings of UIST '96, p. 51-58.

VRML Architecture Group, 1997

The Virtual Reality Modeling Language,

International Standard ISO/IEC 14772-1:1997,

<URL:http://www.vrml.org/Specifications/VRML97/>

VRML Database Working Group, 1998

Metadata Node Specification,

<URL:http://www.vrml.org/WorkingGroups/dbwork/metadata.html>

Wernecke, Josie; Open Inventor Architecture Group. 1994

The Inventor Mentor - Programming Object-Oriented 3D Graphics with

Open Inventor™, Release 2.

ISBN 0-201-62495-8

Addison-Wesley Publishing Company, Inc.

Wloka, M.; Greenfield, E. 1995

The Virtual Tricorder: A Uniform Interface for Virtual Reality. Proceedings of UIST ‘95, p. 39-40.

Zhai, S.; Buxton, W.; Milgram, P. 1994

The “silk cursor”: investigating transparency for 3D target acquisition. Proceedings of CHI’94: ACM conference on Human Factors in Computing Systems, p. 459-464. Boston: ACM

A BUGS IN THE O2 OPENGL IMPLEMENTATION

A.1 Projective texturing bugs in O2 OpenGL implementation

The O2 OpenGL / DMBuffer combination did not function properly with projective textures. The bug is acknowledged by SGI, and it was promised to be fixed in IRIX 6.5.1 release. As of now, the current IRIX release is 6.5.2, and the bug still exists. Projective texturing need all of the four texture coordinates (s, t, r, q) to calculate the projection matrix. In a normal texture context everything works fine, but when the texture source is a digital media buffer, the r and q coordinates are corrupted in the OpenGL rendering pipeline.

Figure 40 displays the correct projection of a texture (see Figure 43 on page 91) using static texture matrix for projection. The projection matrix, or frustum is shown in Figure 42 on page 91. If the same matrix (and the same OpenInventor code) is used to project an image from DMBuffers, the result is uncontrollably warped, as can be seen in Figure 41 on page 90. The projection frustum seems to be distorted in many ways, for example, the bottom of the cube displays the image which it should not do even if

the projection frustum would be the one producing the image on the right hand side of the cube.

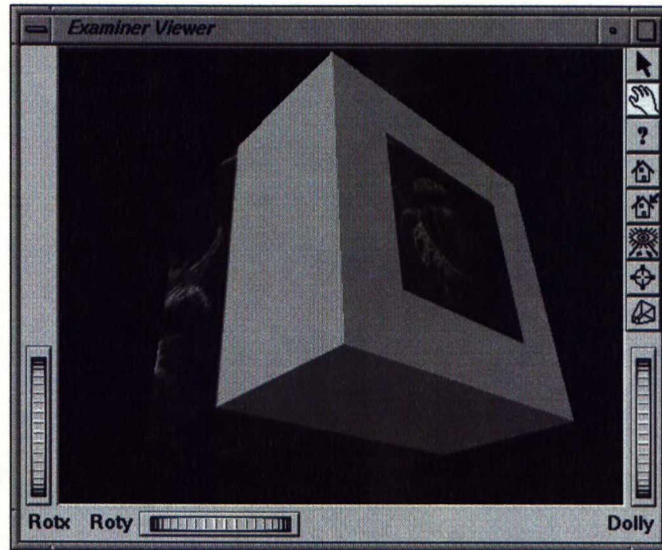


Figure 40 **Correct projection using normal texture**

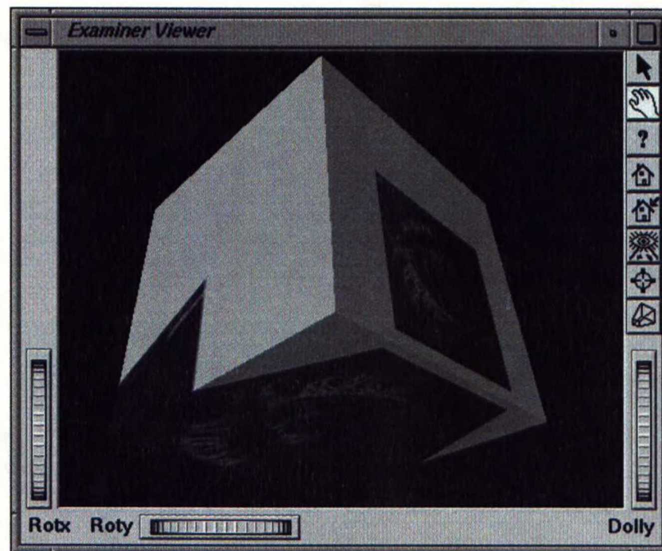


Figure 41 **Incorrect projection from a DMbuffer texture source**

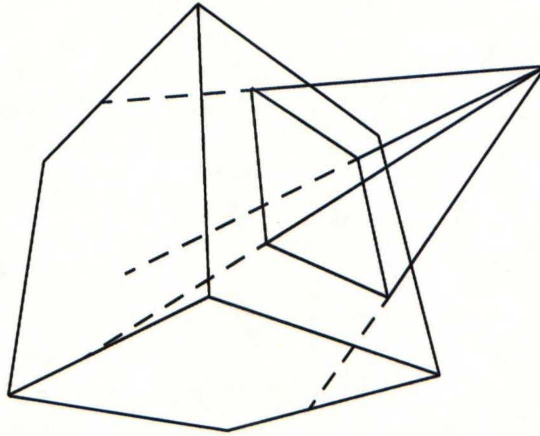


Figure 42 Projection frustum for image

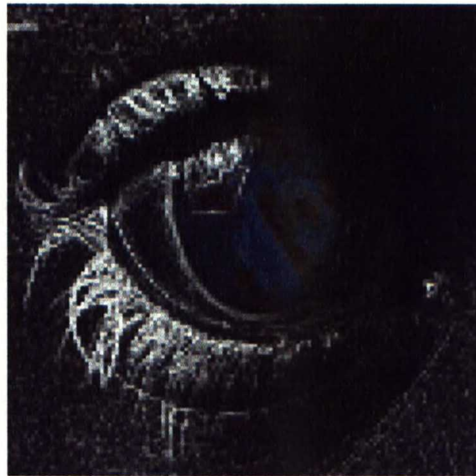


Figure 43 Projected image