3D Magic Lenses, Magic Lights, and their application for immersive building services information visualization

Seppo Äyräväinen, Researcher
BS-VE project
3.1.2003

Helsinki University of Technology
Telecommunications software and multimedia laboratory

Abbreviations and terms

Alpha channel	An extra channel in an image (texture) file, describing the opacity of the image at that pixel. For example, an RGBA image has red, green, blue, and alpha channels.
API	Application Programming Interface. Defines the interface that a programmer can use to utilize services from another software layer. For example: OpenGL, DirectX.
BS-VE	3D Visualization of Building Services in Virtual Environment.
CAVE™	CAVE Automatic Virtual Environment. A cubical construction with back-projected walls, and possibly a floor and/or ceiling.
Clipping plane	A plane cutting the 3D-space in two parts. Typically tells the graphics hardware not to draw the other half.
Culling	Reducing the amount of data to be displayed by discarding the objects that are not visible.
dPVS™	A commercial, general purpose visibility optimizer software.
EVE	Experimental Virtual Environment, the IPT system at HUT.
Frame buffer	Frame buffer is a memory region residing in the graphics hardware. In 3D graphics, there are typically at least two buffers for each display. The image is drawn in a back buffer, after which the (pointers to) back and front buffers are swapped. As only the front buffer is visible, the viewer always gets to see the final image, and not the one under construction. Using only one buffer causes the image to flicker severely.
Frustum	(Viewing) frustum defines the visible part of a scene. Objects falling outside the frustum do not need to be drawn. Frustum is typically (in perspective projection) pyramid-shaped with its top cut off (at near plane), and the viewpoint being at the peak. In other words, the frustum contains six clipping planes: left, right, bottom, top, near and far.
Geometry	A three-dimensional model, typically stored inside the computer as points in XYZ-space, or primitives (triangles, quadrangles etc.) formed from the points.
Graphics pipeline	3D graphics are drawn in multiple separate stages (including modeling and viewing transformation, lighting, clipping, projection, and scan conversion), that may each reside in different hardware. Details are out of the scope of this paper.
HUT	Helsinki University of Technology
IPT	Immersive Projection Technology

Mapping	(Texture) mapping defines the rules of wrapping the texture into a geometry. Examples: planar, cylindrical and spherical mapping.
Multipipe	A graphics hardware configuration with multiple graphics (OpenGL or other) pipelines.
Node	Scene graphs are composed of nodes, which are related to each other with parent-child-relations. Nodes can be almost anything, from 3D transformations to sounds or 3D geometry.
OpenGL	A low-level 2D/3D drawing API, supported in most operating systems.
OpenGL Performer	A scene graph API, built on top of OpenGL. Lets the programmer work on a higher level than OpenGL, managing the graphics state, display lists, and offering many utility functions not available in OpenGL.
Pipe	A single (hardware) graphics pipeline.
Rendering	In computer graphics, rendering is the process of producing a visual image from the representation used inside the computer.
Scene	The group of objects to be visualized.
Scene graph	A logical hierarchy, describing the scene objects, their relation to each other, and their functionality. The graph, traversed at each frame, keeps track of the state of the virtual world. The scene graph sends the necessary information to lower-level APIs, such as OpenGL, to do the actual drawing.
Stereo imaging	Making the image look three-dimensional by displaying a different view to the scene for each eye.
Switch node	Typically included in scene graphs, a switch node can be used to select one or more of its children to be active.
Texture	An image, which can be wrapped around a geometry to, for example, make it look more complex or realistic.
TML	Telecommunications software and Multimedia Laboratory (at HUT).
Tracker	A device for determining the position and/or rotation of one or more physical objects (typically a human). There are many different types of trackers, including magnetic, acoustic, optical, and inertia-sensing trackers.
Transformation	An operation that may change the current object or coordinate system. The most common types of translations are scaling, translating, and rotating. Transformations are typically represented as matrix operations.
VR Juggler	A suite of VR APIs that abstract the interface aspects of a program including the display surfaces, object tracking, selection and navigation, rendering, and graphical user

	interfaces. An application written with VR Juggler is essentially independent of device, computer platform, and VR system.
Wand	An input device, used commonly in VR applications. The position and orientation of a wand can typically be tracked. Most wands also include two or more buttons and possibly a joystick.
Z-buffer	A part of the frame buffer, storing information of the distance of the objects in the scene (at a pixel or subpixel level). Removes the need to order the objects to be drawn from back to front before drawing. With Z-buffer, the hardware can simply check at each pixel ("Z-test") if the current object is closer than any previously drawn object at that pixel, i.e. should it be drawn or not.

1 Introduction

This document is part of a project at the Helsinki University of Technology, called BS-VE (3D Visualization of Building Services in Virtual Environment). The purpose of the document is to introduce a concept of Magic Lenses, and how can they be applied to visualization in an Immersive Projection Technology (IPT) environment.

Previous research work and implementation methods are discussed, as well as the possible uses of Magic Lens technology in the BS-VE project.

The available implementation options are evaluated, and the preferred methods are chosen to be realized within the BS-VE project. Practical implementation details of the chosen methods are also discussed, including the changes required to the software platform used during the project. Details of the changes in code are given in the separate developer documentation.

2 Background and history

This chapter presents the history of, as well as the basic concepts related to Magic Lenses and 3D Magic Lenses – their extension in 3D space. Also Magic Lights, introduced by Hannu Napari in his master's thesis [NAPARI99], will be discussed.

2.1 The See-Through Interface

The see-through interface, including the concepts of *Toolglass* and *Magic Lens* were originally introduced by Bier et al. in 1993 [BIER93]. The see-through interface is essentially a set of (semi)transparent user interface widgets, positioned between an application and a cursor.

A Toolglass can be used to manipulate objects seen through it, while Magic Lenses are visual filter widgets that may change the appearance of the world seen through them. The effects of several Magic Lenses can be combined by stacking them on top of each other on the screen. Of course, different Magic Lens combinations may or may not make sense.

There are few restrictions in what the Magic Lenses can be like. Of course, since adequate screen refresh rates are desired in interactive work, the available processing power poses some limitations on what one can do with the lenses. Magic Lenses can for example magnify, zoom, do image processing, or hide/show some visual details from the application data.

In a multi-hand-operable system, the user may position the Toolglass and/or Magic Lenses with the other hand, while the other operates the cursor.

2.2 3D Magic Lenses

In 1996, Viega et al. extended the metaphor of the see-through interface to 3D [VIEGA96]. They presented two new concepts: *flat lenses in 3D* and *volumetric lenses*.

Flat lenses in 3D are virtual, flat objects on 3D space, through which an alternative view to the world is presented to the user. They act like magnifying glasses in the real world, with the exception that in addition to magnifying, they can do basically anything to the image. Flat 3D lenses can be used in many kinds of tasks, such as looking through walls, displaying different levels of detail, zooming and changing draw modes.

Volumetric lenses do not really have a representation in the real world. Instead, they are like magic boxes (of arbitrary shape), altering the visualization inside their volume. Volumetric lenses are well suited for gaining better insight on very large datasets. The user can, for example, easily restrict the volume that is shown in detail, while the rest of the set is displayed with fewer details – or is not displayed at all.

Since Magic Lenses in both 2D and 3D are "magical", they can, of course, display almost anything – not necessarily the view that in real life would be seen behind the lens. For example, the lens could display a view to the scene from a totally different location, i.e. act as a kind of a virtual surveillance camera. This can, however, be quite disorienting for the user, and should only be used when it really adds value to the visualization task.

One important concept of flat 3D Magic Lenses that will occur often in this document is the *lens frustum*, a term from Viega et al. [VIEGA96]. It is the subset of the viewing frustum that is covered by the (planar) lens surface, when looking from the current eye point (the gray area in Figure 1).

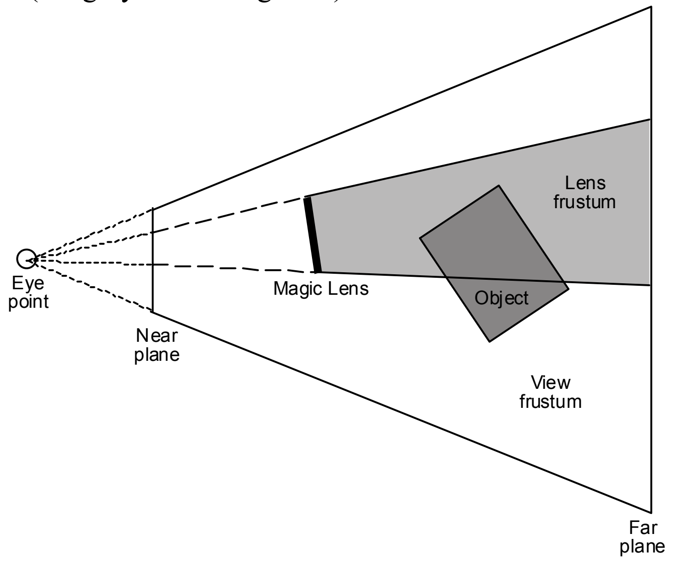


Figure 1: The lens frustum in flat 3D Magic Lenses[1]

[1] Figure adapted from Hannu Napari's thesis [NAPARI99].

2.3 Projected Magic Lenses – Magic Lights

Magic Lights, introduced by Hannu Napari [NAPARI99] is a very interesting concept, especially when used in CAVE™-like environments. Magic Light is like a Magic Lens, except that the lens frustum is not tied to the viewpoint of the user (see Figure 2). Instead, it acts like a flashlight or a headlight, highlighting parts of the virtual world. Magic Lights are well suited for viewing purposes, where manipulation of the scene is not required, especially in multi-user sessions. Of course, they can be used in addition to other types of Magic Lenses.

In addition to the method illustrated in Figure 2, the Magic Light Projection frustum can also originate from the current eye point. In this case, the Magic Light is mainly used to move the Magic Light image around the viewport [NAPARI99].

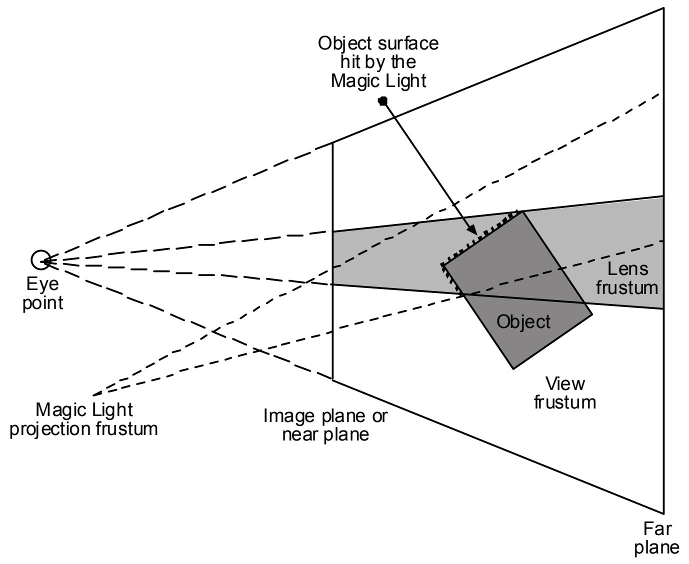

Object surface hit by the Magic Light
Eye point
Lens frustum
Object
View frustum
Magic Light projection frustum
Image plane or near plane
Far plane

Figure 2: Lens frustum in Magic Lights.[1]

3 Known implementations and methods

Magic Lenses in 2D applications have been used and researched for quite a while. Loughlin et al. [LOUGHLIN94] used Magic Lenses to create specialized views of a Computational Fluid Dynamics (CFD) dataset visualization.

Magic Lenses –like behavior has also been used in commercial software products, such as Kai's Power Tools™ 3 (one lens) and CorelDraw™ and FreeHand™ (combining several lenses). For example in FreeHand, one can set the fill of any drawing object to a "lens fill", which filters everything seen behind that object. Since FreeHand is object-oriented, the front-to-back ordering of objects also defines the order of operations. For example, in Figure 3 the lens order from back to front is: monochrome (rectangle), magnify (ellipse), and invert (rounded rectangle). The brighten lens (hexagon) on the right is on top of the magnifying lens.

[1] Figure adapted from Hannu Napari's thesis [NAPARI99].

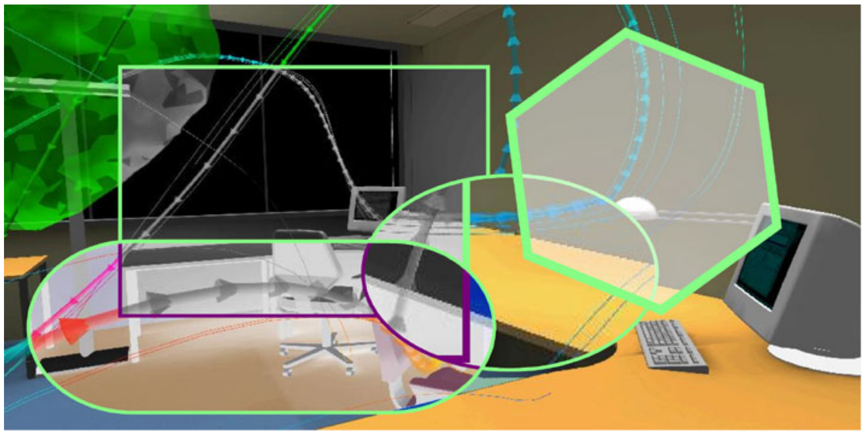


Figure 3: Objects with various lens fills in Macromedia FreeHand.[1]

3D Magic Lenses have not been very actively researched or used. This is probably due to the cost of the required hardware (trackers, fast graphics hardware) and perhaps also the complexity of the implementation of 3D Magic Lenses, especially with multiple lenses.

A recent Magic Lens-like technology is called Pliable Display Technology (PDT, see http://www.idelix.com/). It is based on configurable, magnifying lenses, and is used to view details of a large 2D image that does not fit into screen in its full resolution. Although PDT technology itself can utilize OpenGL in its internal 3D transforming functions, it is only intended for viewing 2D images. However, the company is currently developing a 3D version of the technology, called the 3D PDT. Demos of both techniques are available at the Idelix web site. Figure 4 presents a screenshot of the 3D PDT demo application. On the left, there is a 3D cube made of smaller cubes. The lens can be moved to examine one of the smaller cubes selected as the target (marked red). On the right, only the plane, intersecting the point of interest, is displayed.

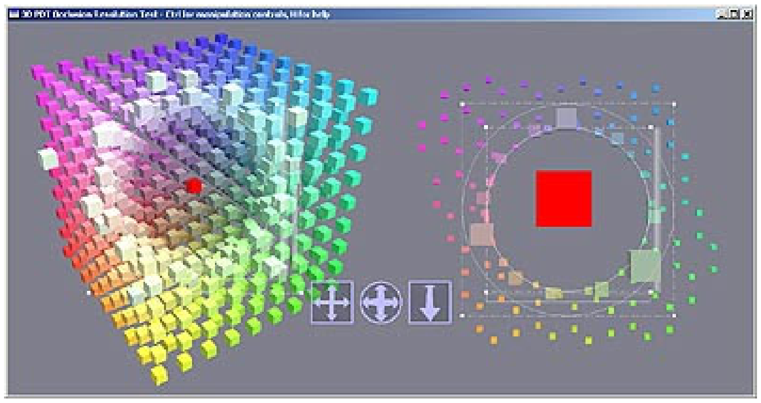


Figure 4: A screenshot of the 3D PDT demo[2].

[1] Background image courtesy of Markku Mantere, HUT/TML.
[2] Image copyright Idelix software Inc. (source: http://www.idelix.com/pdt_demos_3dpdt.shtml)

The 3D PDT implementation differs from the other techniques presented in this paper, since the lens actually moves the objects in (3D) space. This may seem a bit confusing to the user, since the actual point of interest may also suddenly jump away from the view. This may work fine with small objects forming a tight, regular grid - like the one in Figure 1, but is prone to fail in some cases. An example of that could be when a point of interest is hidden behind larger objects, that may even surround the object (a single ball in a ball bearing, for example). In those cases, moving the objects is no substitute for using a clipping plane or hiding the obstructing objects. Also, since the surroundings of the point of interest keep changing when moving the lens, the context of the actual study may change considerably. For example, a scientist studying the structure of a molecule might not like a lens that keeps changing the structure according to the viewpoint and lens location. However, since the technology is in a prototype stage, these problems may be addressed before a final version is released.

3.1 3D Magic Lenses implementation

There are several ways to implement 3D Magic Lenses in practice. However, when considering IPT environments with stereo imagery and complex models, the methods have to be very efficient in order to be usable at interactive visual update rates. The following chapters discuss the various ways to implement 3D Magic Lenses in practice.

3.1.1 3D Magic lenses with clipping planes

Viega et al. [VIEGA96] implemented 3D Magic Lenses using the hardware clipping planes in SGI Reality Engine graphics computer. Both flat and volumetric 3D Magic Lenses are rendered using essentially the same technique. The only difference is that while the lens frustum of flat lenses is partly defined by the constantly changing view frustum (and its far plane), volumetric Magic Lenses define their lens “frustums” explicitly.

The procedure when using clipping planes is as follows:

1. Render the world outside the lens frustum (once for each of the 6 sides of the lens frustum), clipping the area inside the frustum on each pass.
2. Render the world inside the lens frustum in the normal way (with a different model or rendering parameters).

As one can imagine, rendering most of the scene six times does not have a very positive effect on the frame rate. When viewed in stereo, the amount of rendering passes needed also doubles. Even if this implementation could work interactively with simple models, using models as complex as in the BS-VE project – not to mention using several lenses at the same time – is out of the question (at least with any reasonable hardware available at this time).

3.1.2 3D Magic Lenses with stencil bit planes

Another method to implement flat 3D Magic Lenses, described by Napari, is the use of stencil bit planes [NAPARI99]. The stencil buffer, found in most modern graphics hardware, is typically used to mask for example a static cockpit image in a flight simulator so that the 3D environment is not drawn on top of the cockpit, but only where there is a transparent window. The same method can be used to draw a 3D Magic Lens image separately from the rest of the scene. Napari states in his thesis, that stencil masks cannot be used to create true 3D Magic Lenses. However, after discussing with him, we agreed that it should be possible, at least in some cases.

3.1.3 3D Magic Lenses with texture mapping

Another method, also described in [NAPARI99], is to use mapped texture on the Magic Lens surface. In this method, the lens is simply some geometry (typically rectangular), into which the image displayed through the lens is mapped as a texture. This method is very effective, since after the lens texture is formed – by rendering the suitable part of an alternative scene into a texture buffer – the texture image can be further edited by traditional 2D image processing techniques. This allows very sophisticated filtering to be applied to the image displayed by the Magic Lens.

The first rendering pass, where the lens image is rendered, should be optimized in order to keep the overall refresh rate acceptable. This can be done, for example, by ordering the scene graph so that only a small part of the graph must be traversed in order to draw the lens image. This can be a very difficult task in practice, but can be simplified greatly by using advanced occlusion and/or portal culling software packages, like the dPVS by the Finnish company, Hybrid (http://www.hybrid.fi/). HUT holds an academic license to the dPVS software, and it has been tested with the EPIC software used in the BS-VE project.

3.2 Projected Magic Lenses implementation

Magic Lenses suit single-user applications very well. However, when there are multiple users in an IPT system – like typically in the BS-VE project – some difficulties arise. Head tracking is not typically used with multiple users, since all but the tracked user get a distorted view of the virtual world. This forces Magic Lenses to be used view-independently, i.e. so that the image displayed through the lens is not dependent on the location of the users, but is viewed from a fixed location. Although tracking many users can be tracked at once, presenting a geometrically correct projection for both the surrounding and the Magic Lens image is not possible for multiple users. So, if a user is located far from the fixed eye point (typically the center in an installation like the EVE), the image displayed through the Magic Lens gets more and more distorted and unnatural.

Magic Lights suit multi-user situations better, since each user can see the lens image with minimal distortion. Because the lens image is projected on the actual scene geometry, there is no need to look through any geometrical lens representation. This allows a more natural way of looking at the subject covered by the “lens” (i.e. the Magic Light). However, for the reasons described earlier, projecting the lens image into the actual scene would be impractical and might decrease performance. Thus we use a rectangular lens geometry, into which a texture is mapped. Although the projection may not be correct for all users, the problem is not very severe.

A nice property of Magic Lenses/Lights is that they are naturally operable with a wand. The wand is currently used as a navigational tool in our system, and implementing the Magic Lenses/Lights control would be quite straightforward. We could, for example, use one of the wand buttons to switch the Magic Light on and off.

4 Magic Lenses in the BS-VE project

Using Magic Lenses is one of the most interesting research areas in this project. The concept has only been used or tested in virtual environments in few projects before, and it has never been implemented it in a software environment similar to ours (i.e. OpenGL Performer / VR Juggler). This means that any discoveries that we make are likely to be scientifically interesting. On the other hand, since this has not been done before, we are very much on our own regarding the actual implementation details. A step to any direction is a step to unknown, and we must prepare to face technical difficulties at all stages of the implementation.

How can we use Magic Lenses in the BS-VE project? The possibilities are endless in theory, but the limited amount of available graphics processing power makes it necessary to consider very carefully the options that really benefit from Magic Lenses and the related techniques. So, in addition to asking “what?”, we must also ask “why?”. Every new visualization option adds to the size of the scene graph, which is a significant factor, especially when dealing with large models.

4.1 Possible application areas for Magic Lenses

In the BS-VE project, there are several possible application areas for Magic Lenses. Basically all of these can be implemented by simply adding more models on top of the base model, all depicting different areas of interest. These application areas include, but are not limited to, the following:

- Air conditioning / plumbing / electrical wires visualization
- Comfortable areas visualization
- Visualization of pre-made navigation paths
- Throw distance pattern (heittopituuskuvio in Finnish) visualization
- Light distribution curves visualization
- Visualization of alternative lighting conditions
- Visualization of noise sources
- Magnetic fields visualization
- Navigation and manipulation of the objects with a space-reducing volumetric 3D Magic Lens (essentially a World in miniature, WIM [STOAKLEY95])

4.2 Implementing Magic Lenses in the BS-VE project

As discussed in the previous chapters, Magic Lenses can be implemented in various ways. Having not seen any of the implementations actually realized in practice, it is impossible to know in advance, which of the methods would work the best in our project. Since we have limited time and resources, the best way seems to be to first implement the easiest method, and then proceed with the more elaborate ones, should the concept in general work like planned. In other words, we want to see – as soon as possible – whether Magic Lenses provide enough added value (or work at all) in the BS-VE project.

Regardless of the method of creating the lens/light image on screen, we must first have an alternate representation to display. This means we must have an alternative representation of the world objects in the scene graph. Of course, the amount of duplication required in the scene graph depends on what we want the user to see through the lens (or with the light). Here are some possible cases:

1. A change in the rendering style (for example, outline, wireframe, solid, or semi-transparent).
2. A change in the geometry (for example a ventilation system behind the walls).
3. A change in the lighting conditions and/or materials on the scene.

Case 1 is the easiest in that it does not require any duplication of the scene graph. Instead, the scene is simply rendered twice, each time with different rendering parameters (like solid/wireframe etc.).

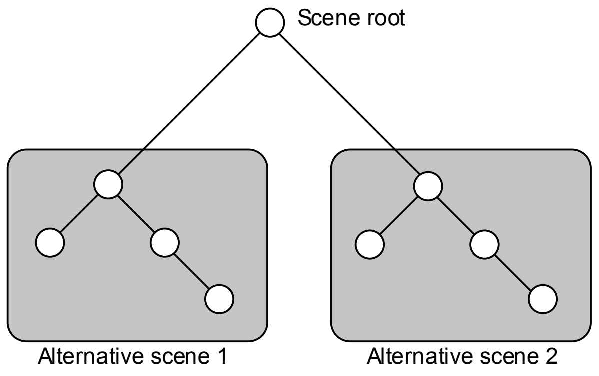


Figure 5: A duplicated scene graph.

Case 2 requires the scene to be duplicated because the geometry is different. There are basically two options: duplicating the whole scene (Figure 5), or using switch nodes in the scene graph to select between different representations (Figure 6). If the alternative scene is totally different from the original, the only reasonable way is to duplicate the whole graph, since there is no direct correlation within the structure of the two geometries. However, if the alternate scene is just a more/less detailed version of the same scene, using switch nodes may be more efficient. Since we don't have any tools for building scene graphs (like Multigen Creator), using switch nodes would bring considerable amount of overhead to the process. If we want the model path from the companies to the EVE to be as quick as possible, the only option is to load all the models required into different branches of the scene graph at the application launch. Of course, loading various models could be done also on demand at runtime.

Case 3 requires loading and assigning textures to the model at runtime. This has been investigated in our project, but has not been yet integrated with the Magic Lenses implementation. By simply switching the textures used in a model, we can avoid loading the same geometry multiple times.

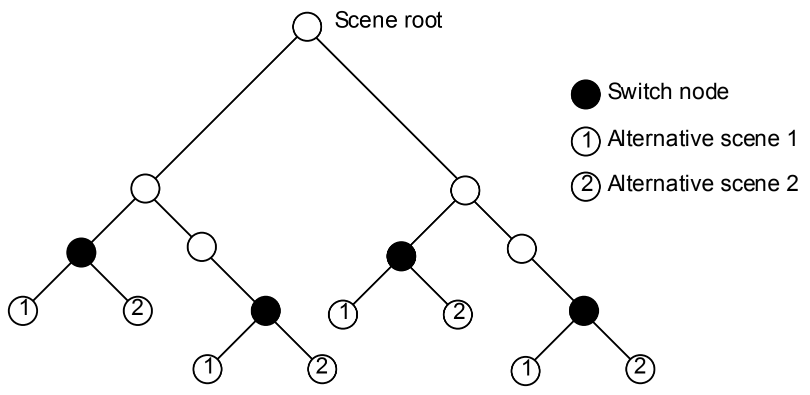


Figure 6: Duplicating the scene graph using switch nodes.

4.2.1 "Cursor" Magic Lights – the first phase

The easiest method to implement is Magic Lights with the cursor metaphor. In this method, the Magic Light image is simply mapped to a rectangular area on the viewport, possibly filtered by an alpha map for "spotlight" or some other kind of effect. The wand points the location of this sub-viewport, like it was a 2D cursor. A Magic Lens frustum is formed from the viewing location, very much like with flat 3D Magic Lenses (see Figure 1 on page 6). The only exception is that there is no physical "lens" in the scene, but the view is simply replacing a rectangular area on the viewport.

With a simple implementation, we can test most of the possible uses of Magic Lenses, and see if they add value to the visualization, and also if they reduce performance considerably. The preparation work (scene graph duplicating) can also be fully tested and implemented using this method. When this phase is working, additional methods can be developed if needed.

4.3 Implementing the Lenses / Lights in practice

This chapter discusses the various possibilities of implementing the Magic Lenses / Lights in this project. The chosen methods and the reasons for choosing them are also explained.

4.3.1 Switch nodes vs. scene graph duplication

In either case, the first thing to do was to render the alternate scene (as seen by the Magic Lens / Light) to an off-screen buffer. Scene graph duplication was chosen to be a more suitable method for different representations than switch nodes, since we need to display various, complex models with the software. There would simply be too many switch nodes to add to each model.

In any case, changes must be made to the VR Juggler framework itself, since there is no other way to render two scenes within one frame. This is very unfortunate, because it means that with every new version of VR Juggler, we must migrate the changes to the new version. At some point, this may stop working because of some fundamental changes in VR Juggler itself. It is, however, unlikely that the OpenGL Performer Draw manager part of the VR Juggler would be considerably changed in the near future.

4.3.2 Texture-mapping vs. frame buffer manipulation

First we had to decide whether to use texture-mapping techniques, i.e. to use the lens images as a texture in the scene – or to directly access the frame buffer(s).

At first, the idea of manipulating the frame buffers seemed fine, since there would be no need to add any physical objects to the virtual world, nor would there be need to map the textures projectively into the scene, wasting precious graphics resources. The idea was to simply draw the alternative scene to the off-screen buffer, then draw the scene as normal – and finally composite these two scenes in the frame buffer. This seemed like the most efficient method.

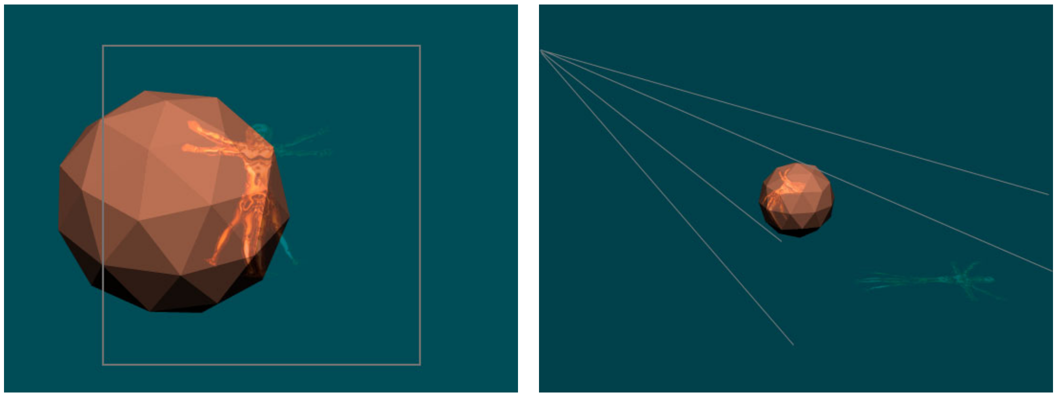
However, since we are dealing with a 2-pipe system with 4 physical stereo displays, as well as VR Juggler controlling the drawing, frame buffer manipulation would have required an extensive rewrite of VR Juggler to implement. It could work with one display, but the cases with Magic Lens / Light image overlapping several displays at once, would be very tedious to implement, especially in stereo. So, the decision was to leave the frame buffers alone and let VR Juggler handle the multipipe/multiwall issues. In other words, we should use the off-screen buffer image as a texture.

4.3.3 Projective texturing vs. physical lenses

As we were going to implement the Lenses / Lights using texture mapping, there were still two options to choose from. We could project the texture straight to the scene geometry using a technique called projective texturing – or we could add a physical lens object to the scene and map the off-screen rendered texture into it.

In projective texturing, the texture is mapped in a slideshow-like fashion into the existing scene geometry. There is no need for a separate projection surface. The texture coordinates required for the projection are calculated in graphics hardware in the InfiniteReality, so performance should be acceptable. Since we could use Magic Lights much like real flashlights, the issues with multiple walls would be completely handled by VR Juggler and OpenGL Performer.

Figure 7 illustrates the concept of projective texturing. The polygonal ball represents the current scene, in which a picture of a man (the texture) is projected. In the left image the texture is seen undistorted, since the eye location and the projection origin are the same. On the right image, viewed from outside the projection origin, the image on the ball (scene) seems distorted.



Looking from the projection origin Looking from a different position

Figure 7: Projective texturing – an example.

However, when using the scene itself as a projection surface, there would be issues with visibility when there is nothing in the scene in the direction the user is looking at. For example, if one is outside a house, there is no way to display any information in the area not covered by the house model. Placing some geometry outside the model (like a sky dome and/or a ground) could solve this, but projective texturing works best when the far plane of the projection frustum is approximately in the distance of the projection surface (i.e. the scene). This is very hard to accomplish with a sky dome placed far from the scene itself. So, in the end we decided to go with physical lens geometry and mapping the texture to that geometry.

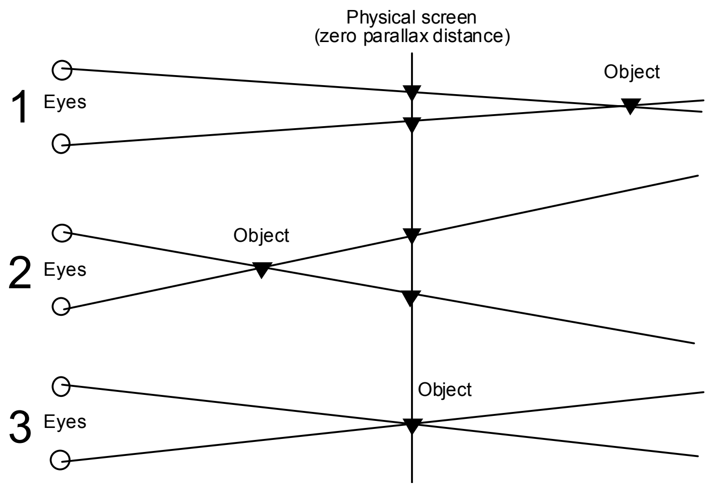


Figure 8: Three types of stereo parallax: positive (1), negative (2), and zero parallax (3).

When using physical lens geometry, the lens should theoretically be located as close to the near plane as possible, so that the scene objects would not obstruct the lens view. However, especially in stereo mode, this can cause eye strain because of the large distance between left and right eye's image on the physical projection surfaces. Also the risk of clipping (because of the near plane being close) of the lens partly or completely is high, if the lens is not exactly perpendicular to the eye.

A more suitable method, used in our implementation, is to place the lens into or near the zero parallax distance (in practice, the distance of the physical projection surface) and ignore Z-buffer values while drawing the lens geometry. Objects at the zero-parallax distance are drawn at the same location on the physical projection surface, making the viewing of the lens easy to the eye. Figure 8 illustrates the three types of parallax (excluding divergent parallax, where the eye vectors diverge, and which should be avoided at all times in stereo projection). In positive parallax, the object is behind the image plane, and the object is projected to the left for left eye and to the right for right eye. In negative parallax, the object is between the viewer and the screen, and the object is projected to the right for left eye and vice versa. In zero parallax, the object is at the distance of the image plane, and the projection images for both eyes are at the same location.

Ignoring the Z values cause the lens to be drawn regardless of whether there are scene objects between the user and the lens. However, when using a lens with alpha-faded edges, this could lead into artifacts with scenes containing other alpha-blended textures. The lens should be rotated along a spherical surface, depending on the user's eye position and the direction indicated with the wand.

4.3.4 Pbuffers vs. the back buffer

As we were going to render the scene twice – and thus suffering a performance loss, we wanted to be able to read the texture, rendered in the first pass, as fast as possible to the texture memory. SGI O2 machines have the best support for this (digital media pbuffers, an OpenGL extension called SGIX_dm_pbuffer). However, since this application should run on multipipe configurations (and perhaps Linux), SGI proprietary digital media buffers were out of the question.

Unfortunately, we could not get the regular OpenGL pbuffer extension to work in either the O2 or the Onyx2, so we had to find another way to render the texture off-screen. The next option was to render the first pass to the back buffer and to copy it into a texture for the actual rendering. This approach has its drawbacks with regard to performance. However, the Magic Lens image does not always have to be drawn (and read) at full frame resolution (1024x1024), for example, if the lens area is small in the scene. The physical dimensions of the Magic Lens buffer can be dynamically changed to optimize bandwidth usage.

The backbuffer rendering method is implemented as follows (for each eye):

1. Clear the back buffer.
2. Draw the Magic Lens scene to the back buffer.
3. Copy the back buffer into the lens texture object.
4. Clear the back buffer.
5. Draw the actual scene, using the previously read texture on the Lens geometry.
6. Swap buffers.

Currently, the lens image resolution can be manually changed using keyboard commands. This is not a very important feature, since the lens size on the display surface is constant. It is technically possible to create an adjustable-sized lens, but it was not considered to be a high priority feature and was not implemented in our project. If user could change the lens size, the resolution should also change dynamically to avoid blurring of the lens image with large lens sizes.

4.3.5 Projection issues: tracker data, matrices

Calculating the correct lens projection unexpectedly turned out to be the hardest part of the Magic Lens implementation. Many issues arose that we couldn't see before actually trying to implement the projections.

The lens geometry needed to be translated according to the positions of the head of the user, and the wand. The projection was supposed to follow that movement. More precisely, the lens was chosen to be located at a constant distance from, with the textured side always facing the viewer. This means that the lens rotates along a sphere, the radius of which is fixed, and is always facing in the direction of a vector calculated from head to wand position. In practice, it was found that the hand of the user often obstructed the Magic Lens image. This was corrected by adding a vertical offset to the wand position, which makes the Magic Lens image appear above the wand instead of being directly behind it.

VR Juggler defines a special kind of a display surface that can be tracked and can be used with, for example, head-mounted displays. This was chosen to be used with Magic Lenses, since the lens is basically also a tracked display surface. However, the surface class had to be modified since there was no single tracker sensor with which the movement would be controlled. Instead, the surface position had to be calculated using both head and wand sensors. The lens geometry translation and rotation was calculated in the application code (EPIC), while the lens projection was calculated in VR Juggler, using modified tracked display surfaces.

Since the same calculations were done in two places, using tracker data from different points in time, many issues arose. In the application code, OpenGL Performer matrices (class pfMatrix) were used, whereas in VR Juggler we had to use Juggler matrices (class vjMatrix). The possible operations with these two types of matrices differ, and thus making two pieces of code doing exactly the same thing was not straightforward. This also made debugging the code quite hard. Fortunately, we found some (poorly documented) utility functions from VR Juggler that allowed us to convert these two types of matrices into each other. This way we could make sure that the calculations at both places were correct – or at least produced similar results.

The fact that we used tracker data from different times for calculating the physical lens position and the projection, caused the lens image to be out of sync with the surrounding image (one frame delay). This was corrected by using real-time tracker data in the application code, instead of the values previously fetched and saved in application data update loop.

4.3.6 Projection issues: frustums and stereo

Creating the projections correctly was not an easy task, since there were so many parameters that affected the result. These include the physical lens geometry (the corner points), texture mapping coordinates for the lens, the viewport width and height used in drawing the back buffer image, the width and height used to read the backbuffer image into a texture, and the initial position and coordinates of the tracked surface used to represent the lens.

After all these parameters were correctly defined and tested in mono mode, we began to test the stereo mode. Unfortunately, the results were not quite as expected: The images for each eye seemed to switched positions in the lens image, and the distance between the images were different in the lens image and its surroundings, although the scenes were the same. What we wanted was a lens image that would match the background seamlessly.

Finally we found that the cause of the problem was in different zero parallax distances between the lens and the surrounding image. The problem results from the fact that the lens image is first created using the projection of the current display surface – in this case, a wall of the EVE – and then mapped to a smaller surface (the lens geometry). What we should do is to use the projection of the lens to the current display surface and calculate the frustum from those projected corner coordinates. This yields in general case, a quadrilateral, but not a quadrangle. From Figure 9 it is clearly seen, that the lens projected on the display surface does not generally yield a quadrangle into the display surface. With OpenGL, generating a frustum from such a general quadrilateral is not possible, since the frustum is only given one value for each the top, bottom, left and right sides.

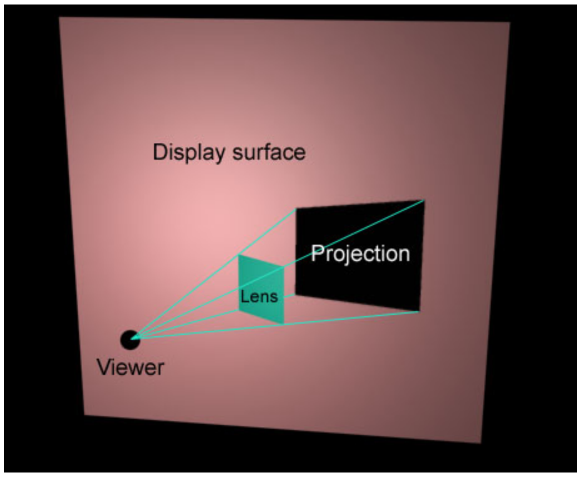


Figure 9: The projection of a lens on a display surface.

The only solution we could come up with was to use exactly the same frustum with the Magic Lens as with the projection surface. In other words, we must draw the Magic Lens scene as if we were drawing it to fill up the whole projection surface, and then only use the portion defined by the projection of the lens into the projection plane. So, for example, in the Figure 9 we should first draw the whole display surface area (pink in figure) and then clip the texture falling into the black area ("Projection") and map it into the lens geometry. In practice, this should be easiest to implement using stencil bit planes, discussed in chapter 3.1.2. With stencil bit planes we only need to draw the lens seen from the current view point into the stencil buffer, and then draw the lens image into the display surface as we would draw the normal scene. No texture mapping is required, since the stencil buffer already defines the borders of the lens image. This implies that using texture mapping is a not a feasible approach to the Magic Lens problem, at least when viewing in stereo.

Since there was no time left in the project to make such a fundamental change to the code, we – as a temporary solution – made the lens image monoscopic. The lens itself is viewed in stereo, but the images for both eyes in the lens are the same. This causes some discomfort when viewing the lens image in context, but much less so than when trying to draw the incorrect stereoscopic lens image. In fact, most viewers do not seem to notice the lens image not being stereoscopic.

4.3.7 Fading the lens borders using the alpha channel

Using a simple, rectangular lens is not a very elegant solution, because the lens borders do not necessarily quite fit into the scene. The larger the lens, the more noticeable the distortion. Even if the images matched, due to the matrix operations used, the lens rotates itself when moved towards the sides, which may seem distracting to the user. The easiest solution for this is to use a circular-shaped lens, which fades into the background near the edges. This causes some pixels near the corners to be lost, but eliminates the rotation problem, as well as the hides the possible discontinuities near the lens edges. Our implementation allows any 512x512-pixel RGBA texture to be used as the alpha map. This texture can be defined in the configuration file, so changing the texture does not require recompiling.

Different options for implementing the blending were evaluated. One option was to write the alpha channel directly into the texture read from the framebuffer, but that would have taken longer to implement and could have been slower. The approach we took was to create another "lens" to the Magic Lens scene. The sole purpose of this lens – with geometry similar to the actual lens – was to fill the entire field of view, and to provide an alpha channel for the back buffer before it is read into the Magic Lens texture.

Figure 10 illustrates the usage of an alpha texture to blend the lens image with the background (in the figure, the background is white for clarity). The "lens" in the Magic Lens scene is textured with a texture including an alpha (opacity) channel. The actual colors in the texture do not matter, since they are not written into the image. This is implemented by using the OpenGL command `glColorMask(0,0,0,1)`, which disables writes to any other channel than alpha, in a OpenGL Performer pre-draw callback. As a result, the image written into the back buffer now contains the alpha channel from the texture. When the actual scene including the Magic Lens is drawn with blending enabled, the lens is the more opaque, the more intensity there is in the original alpha channel. In the picture, the center is drawn fully opaque (white in the alpha texture), while the lens corners are not drawn (black in the alpha texture).

The gray values cause the image to be drawn with opacity corresponding the alpha intensity.

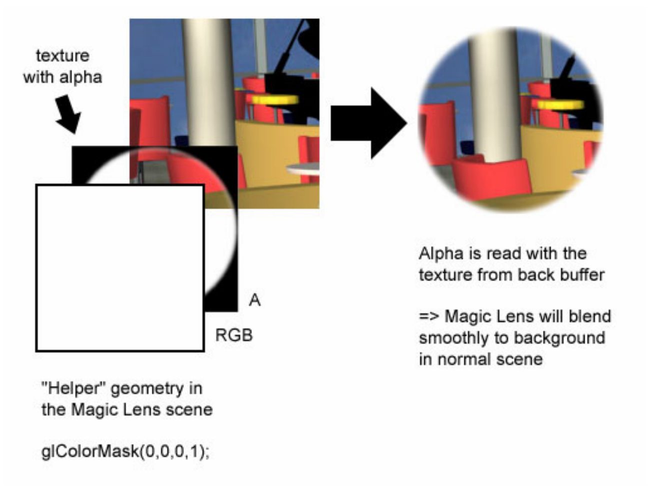
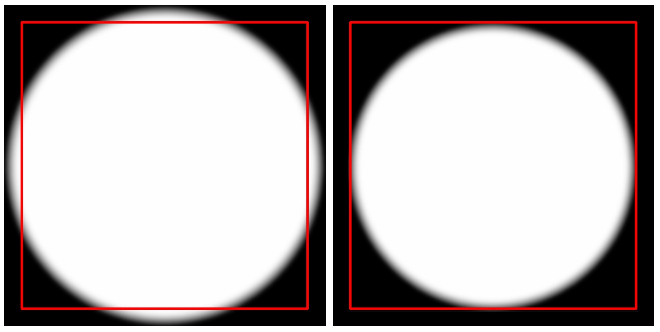

texture with alpha
A
RGB
"Helper" geometry in the Magic Lens scene
glColorMask(0,0,0,1);
Alpha is read with the texture from back buffer
=> Magic Lens will blend smoothly to background in normal scene

Figure 10: Using the alpha texture with Magic Lenses.

Due to the imperfections of the projector settings in the EVE, some display surfaces, especially the floor, are not exactly 1024 x 1024 pixels in size. This results in some artifacts at the lens image borders, since some non-image pixels are read from the edge areas of the back buffer. This was partly resolved by mapping the texture into the lens geometry so that the alpha texture with transparent borders is "stretched" over some of the lens edges by approximately 5 %. There should therefore be a wide enough black area around the alpha texture to avoid clipping the transition area (from black to white). Figure 11 illustrates selecting the alpha channel image in practice. The image on the left, with white area exceeding the safe area marked in red, will show clipping artifacts (i.e. the transition from lens to background will not be smooth near the edges), while using the image on the right results in a smooth lens image.



Clipping artifacts No clipping artifacts

Figure 11: Choosing a suitable alpha channel shape. Safe area marked red.

4.4 Changes to VR Juggler

In order to create the channels for rendering Magic Lens images, as well as to render them differently from normal channels, we had to modify the VR Juggler software itself. In order to make as few changes to VR Juggler as possible, we created a new application class, vjMlApp (see Figure 12 below), derived from vjApp. This class is essentially the same as vjPfApp (the OpenGL Performer application class of VR Juggler), but with a few changes to the channel initialization and rendering.

By deriving the class directly from vjApp, we are not depending on changes made to the OpenGL Performer API of VR Juggler in the future. We could even change the rendering API from Performer to something else. However, this is very unlikely since our eveNavigator application depends heavily on Performer and its scene graph architecture.

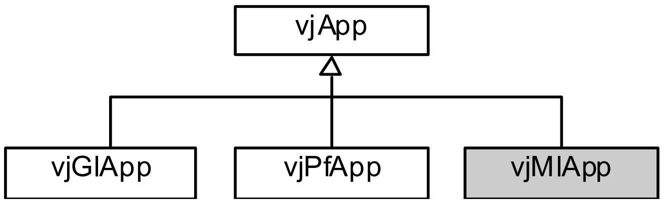


Figure 12: Modified VR Juggler application classes.

Similarly, a new draw manager (vjMLDrawManager) has been written, based on vjPfDrawManager. A new projection type, vjMLWallProjection, is based on vjTrackedWallProjection. Details of the changes in these new classes are discussed in the developer documentation.

Implementing Magic Lenses required quite a few changes to VR Juggler, but fortunately most of these are defined as new classes, making the migration to the next version of VR Juggler easier. However, as VR Juggler is currently going through major revision with version 2.0 soon released, the code will require more work to fit into the new Juggler than into another 1.x version.

5 Future work

The Magic Lens development is still at a very early stage, and there is a lot to be done to make it a really useful and generic tool for visualization. As mentioned, the implementation should probably be done using the stencil buffer to make the stereo work properly. The performance should be given more thought from the beginning, perhaps utilizing the dPVS visibility culling software already in use at the TML laboratory. The features should be integrated into our user interface implementation, replacing the current keyboard commands.

The roles of VR Juggler and the application should be redesigned so that the application programmer would not need to rewrite the code used to move the lens, but merely define the lens properties and functions in a configuration file. Configuring the Magic Lens should be integrated into VR Juggler, so that it would have a configuration chunk of its own. The lens could be made adjustable in size and distance from the user. Magic Lenses could even be included in the VR Juggler distribution, so that developers around the world could participate in their development – perhaps even porting them into other scene graph systems like OpenSG or Open Scene Graph.

6 Conclusions

Magic Lenses is a concept with a lot of potential. At the same time, the implementation with current software and hardware is quite difficult. What is needed is more developers to devote their time to the concept, and make the integration with VR Juggler as smooth as possible.

In our project, one problem was that although there were many potential uses for Magic Lenses, they did not seem to be as important to the client companies as other, more practical, features. This is easy to understand since 3D Magic Lenses is a challenging subject, both technically, and in relation to the user interface design. This is not a research project that will produce results quickly and easily. Instead, it requires substantial resources. If we had devoted two or more researchers to Magic Lenses development from the start, we would probably now have more functional software with a proper user interface. However, this is a good start, and although Magic Lenses were abandoned in the BS-VE continuation project plan, we should not let the work already done fade away.

7 Trademarks and patents

Toolglass and Magic Lens are trademarks of Xerox Corporation. Kai's Power Tools and CorelDraw are trademarks of Corel Corporation. FreeHand is a trademark of Macromedia, Inc. CAVE is a trademark of University of Illinois Board of Trustees. dPVS is a trademark of Criterion Software Limited.

8 Acknowledgements

I would like to thank Hannu Napari for giving valuable advice on several issues concerning the Magic Lens implementation, and SGI hardware. I also thank Markku Mantere for, as the project manager, pushing me forward and maintaining his interest in the Magic Lenses during the long development phase. Thanks also go to others in the EVE group and the project partners for constructive comments and critic during the development and demos. Last, but definitely not the least, I would like to thank Iikka Olli for devoting his time to help me in both coding and design issues, especially with the projection implementation at the final stages of the development.

9 References

[BIER93] Eric A. Bier , Maureen C. Stone , Ken Pier , William Buxton , Tony D. DeRose, Toolglass and magic lenses, Proceedings of the 1993 ACM SIGGRAPH conference, p.73-80.

[NAPARI99] Hannu Napari, Magic Lens User Interface in Virtual Reality, Master's Thesis, Helsinki University of Technology, 1999.

[LOUGHLIN94] Loughlin, M., and Hughes, J. An Annotation System for 3D Fluid Flow Visualization. In Proceedings of IEEE Visualization, 1994, 273-279.

[STOAKLEY95] R. Stoakley, M.J. Conway, R. Pausch. Virtual Reality on a WIM: Interactive Worlds in Miniature. In Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems, p. 265–272, 1995.

[VIEGA96] J. Viega, M.J. Conway, G. Williams, and R. Pausch. 3D magic lenses. ACM UIST'96 Proceedings, p. 51-58. ACM, 1996.