Composable Visual and Temporal Lens Effects in a Scene Graph-based Visualization System

A Thesis

Presented to the

Graduate Faculty of the

University of Louisiana at Lafayette

In Partial Fulfillment of the

Requirements for the Degree

Master of Science

Jan-Phillip Tiesel

Spring 2009

Composable Visual and Temporal Lens Effects
in a Scene Graph-based Visualization System

Jan-Phillip Tiesel

APPROVED:

Christoph W. Borst, Chair
Assistant Professor of Computer Science
Center for Advanced Computer Studies

Dirk Reiners
Assistant Professor of Computer Science
Center for Advanced Computer Studies

Anthony S. Maida
Associate Professor of Computer Science
Center for Advanced Computer Studies

Gary L. Kinsland
Professor of Geology

C. E. Palmer
Dean of the Graduate School

ACKNOWLEDGMENTS

I would like to express my gratitude to the following individuals who inspired and supported my work and helped me in achieving my academic goals.

I would like to thank Dr. Christoph W. Borst for indispensable input, guidance, and trust in my work; Christopher M. Best and Vijay B. Baiyya for laying the groundwork for my research; Dr. Gary L. Kinsland for two years of fruitful and interesting collaboration as well as insight into the science of geology; Dr. Dirk Reiners for constructive feedback and for serving as a committee member; Dr. Anthony S. Maida for productive collaboration and the willingness to serve as a member of my thesis committee; and Dr. Emad Habib for insight on hydrologic models and for providing time-varying simulation data to our laboratory. Also, I would like to thank all students of the Virtual Reality Laboratory at CACS for exchange of ideas and a great learning experience.

My special appreciation goes out to my fiancée Melissa K. Berz, my parents Karin and Horst Tiesel, and my wonderful family of friends for never letting me forget that home truly is where your heart is.

Last but not least, my praise goes out to the one “from whom and through whom and to whom all things are.”

Contents

List of Tables

List of Figures

PRELIMINARY NOTE

Mathematical notation

Lower-case bold letters, such as $\mathbf{v}$, are used throughout the text to denote vectors, while upper-case bold letters, for example $\mathbf{T}$, represent matrices. Vectors are considered to be column vectors, using the superscript T on a vector denotes the corresponding row vector $\mathbf{v}^T$. Where applicable, the use of homogeneous coordinates is assumed. Consequently, a vector with a homogeneous coordinate of zero (e.g., $[1 \quad 2 \quad 3 \quad 0]^T$) represents a direction, while vectors with a homogeneous coordinate of one (e.g., $[1 \quad 2 \quad 3 \quad 1]^T$) denote positions.

When describing transformations between coordinate systems, the notation ${}^{eye}_{lens}\mathbf{T}$ is used for a matrix that transforms vectors or points from the *lens* coordinate system to the *eye* coordinate system. When expressed in words, this transform is referred to as *lens w.r.t. eye* (lens with respect to eye). Likewise, ${}^{lens}\mathbf{v}$ refers to a vector whose coordinates are expressed w.r.t. a coordinate system called *lens*. This follows the notation used by Craig (1989).

Chapter 1

Introduction and Related Work

1.1 Introduction

This work describes the author's research on extending the established concept of volumetric lenses as interactive tools in real-time computer graphics applications. The lens metaphor was introduced to the computer graphics community in 1993 by Bier et al. under the name of *Magic Lenses*™*. Their Magic Lenses were integrated into a 2D graphics system and offered to the user an alternative view within a spatially bounded region of interest (*focus*) while maintaining an overview of the surrounding features (*context*).

In the past two decades, both the academic field of computer graphics and the related industry have seen enormous growth and maturing. The domain of suitable interactive applications for tools like the Magic Lenses now includes medical imaging, geological interpretation, architectural visualization, virtual reality, and many others. Advances in semiconductor technology and computer science have led to the propagation of real-time 3D graphics applications from highly specialized workstations to a variety of devices capable of creating synthesized 3D imagery at interactive frame rates (ranging from inexpensive handheld computers to multimillion-dollar immersive virtual reality venues).

A comparison of the computing time necessary for integrating Magic Lens-like tools into a graphical application exemplifies the enormous performance boost made possible

*Magic Lens is a trademark of the Xerox Corporation

by dedicated graphics hardware and efficient computer graphics algorithms. Whereas a screen update for the 2D application of Bier et al. took about 300 ms for a scene containing a single lens, we show in Chapter 3 how modern graphics hardware can be used to render a high-resolution image of a complex 3D scene containing multiple overlapping volumetric lenses in less than 10 ms.

The concept of Magic Lenses has been adapted and extended by several researchers, and various publications describe applications and techniques related to the interactive tool. Section 1.4 provides an overview of the relevant work. However, several aspects of the powerful concept are not fully explored in previous research. Among others, these aspects include the efficient composition of complex surface shading effects using multiple lenses or the introduction of spatiotemporal lenses that can be used to examine time-varying data. This thesis investigates the mentioned prospects and extends previously published research by Best & Borst (2008).

The contributions of the author's work as described in this text are:

- Detailed description of integrating efficient rendering techniques for volumetric lenses into a scene graph-based visualization system;
- Introduction of shade tree concepts for composition of lens effects;
- Introduction of single-pass rendering technique for composable volumetric lenses;
- Quantitative evaluation of the presented rendering technique; and
- Introduction of spatiotemporal lenses as tools for interpretation of space-time features in time-varying geometric datasets.

1.2 Overview

In the remainder of this chapter, we state the main motivations behind the continued research on volumetric lenses and summarize previously published work on the topic.

Chapter 2 highlights advantages of using a scene graph-based architecture for complex visualization systems. It also provides an introduction to the underlying concepts and data structures of such an architecture as they are used frequently throughout the following chapters.

Chapter 3 is concerned with an efficient integration of composable volumetric lenses into a scene graph application. We present an implementation of volumetric lens rendering using dynamically-generated GPU programs and show how it can be extended to support composition of complex shading effects. We introduce a novel single-pass rendering technique and give qualitative and quantitative results of our approach.

We introduce the concept of *spatiotemporal lenses* in Chapter 4. After describing possible scenarios for their application, we propose a flexible software design for integrating this type of tool into a scene graph-based rendering system.

A high-level conclusion of the presented work and the obtained results can be found in Chapter 5. Finally, suggestions for future research are presented.

1.3 Motivation

Over the past years, many academic and commercial fields have seen strong progress in the resolution and availability of sophisticated imaging techniques. The sampling domain for these techniques now ranges from human brain tissue to the topology of the earth's surface. Geometric data of high complexity are readily available and can be visualized using a variety of graphics algorithms and dedicated graphics processing hardware. This development is accompanied by a growing demand for interactive tools that help users to find correlations in the massive amounts of data, develop hypotheses using the generated imagery, or adjust its parameters to create a desired appearance.

Volumetric lenses represent a class of interactive tools that allows users to apply the perceptual concept of *focus* and *context* to a virtual environment. These tools enable users to establish multiple views of the same data (using different levels of complexity or deviating modes of visual presentation). They also allow for interleaving of different data without impacting the perception of overall context.

Past research disputed the existence of a concept for semantically useful 3D lens compositions (Viega et al., 1996) and some performance results raised concerns about the ability to provide such features in an interactive real-time rendering system (Ropinski & Hinrichs, 2004). In this work, we show how modern graphics hardware can be programmed to render semantically meaningful and intuitive lens compositions at interactive frame rates. We introduce a model for efficient composition of complex surface shading effects and suggest techniques to extend the applicability of volumetric lenses to the spatiotemporal domain.

1.4 Related Work

1.4.1 The Magic Lens Metaphor

Bier et al. (1993) introduced the Magic Lens metaphor in their classic paper in 1993. Their work described the use of dynamic user interface elements that modify the presentation of objects within a graphical application in two dimensions. The user specifies the objects to be affected by positioning a 2D outline (typically a circle or rectangle) on top of them. The outline of the tool represents the scope of the visual effect that is imposed on subjacent objects. The action of moving the tool across multiple objects and simultaneously observing the established effect resembles the familiar use of a magnifying glass, which gave rise to the name that Bier et al. coined for this type of tool: Magic Lenses.

The lenses could be used to either enhance the data (e.g., by revealing previously hidden information) or to reduce its complexity (e.g., by suppressing irrelevant or distracting information). In addition, it was also possible to alter only the visual *presentation* of graphical objects. Among other applications, Bier et al. suggested this type of effect for the dynamic preview of a graphical design as it would appear in different output media (e.g., a black & white printer).

Besides introducing the tool metaphor, the work of Bier et al. also identified several challenges in adapting the concept for more advanced rendering systems. These are

- composition of multiple lenses;
- significance of lens order for the composite effect;
- ability to parameterize the underlying renderer for correct lens clipping; and

- impact of lens rendering on the performance of the rendering system.

As we highlight in subsequent chapters, the significance of these challenges persists or even increases when the the Magic Lens concept is translated to the three-dimensional domain.

1.4.2 3D Magic Lenses

A few years after the initial concept of the Magic Lenses as a tool for interactive 2D applications was established, researchers started investigating the potential of introducing the metaphor to graphics applications that render three-dimensional scenes. Viega et al. (1996) were the first to translate the idea of Magic Lenses to interactive 3D applications. They described two classes of 3D interface tools that could be derived from the original concept: flat lenses and volumetric lenses.

In their work, Viega et al. stated that when composition of lenses is allowed, the complexity of the geometrical computation increases due to the large possible number of intersection regions and their respective boundaries. The authors also doubted that composition of volumetric lenses could offer meaningful semantics. We show in Chapter 3 how the composition of volumetric lenses with complex shapes and a variety of shading effects can be achieved in a single render pass and without the necessity of maintaining complex boundary descriptions for individual lenses and their intersection regions.

Besides giving a first algorithm used for rendering of volumetric lenses (which employed clipping planes for separation of lens interior and exterior), the work of Viega

et al. suggested several interesting directions for future work:

- using volumetric lenses as transportation portals inside virtual environments;
- applying actions to enclosed objects that go beyond altering their visual presentation ("behavior lenses"); and
- using 3D lenses as "crystal balls" that alter the time-based state of objects.

To the author's knowledge, none of the proposed concepts has since been investigated in published work for the case of three-dimensional lens shapes. Itoh et al. (2006) proposed the concept of a volumetric WorldBottle that lets users view and interact with remote spaces; in addition, their tool can be used as portal to the target space. However, the WorldBottle metaphor differs substantially from volumetric lenses as it does not allow the application of lens-specific effects to objects in the context space. This work introduces the concept of *spatiotemporal lenses* in Chapter 4 and explores the implementation and applicability of this enhanced form of the Magic Lens metaphor.

1.4.3 Flat versus Volumetric Lenses

Fuhrmann & Gröller (1998) used the term *Magic Boxes* for a three-dimensional counterpart to the lenses of Bier et al. and differentiated them from the notion of a simple adaption of the "flat" Magic Lens, which they describe as a planar polygon with arbitrary boundary shape. This corresponds to the distinction made by Viega et al. Both the user's current viewpoint and the pose of the lens were needed to determine the focus region of a flat lens and to render respective effects.

The fact that the focus of a flat Magic Lens depends on a user's viewpoint limits its

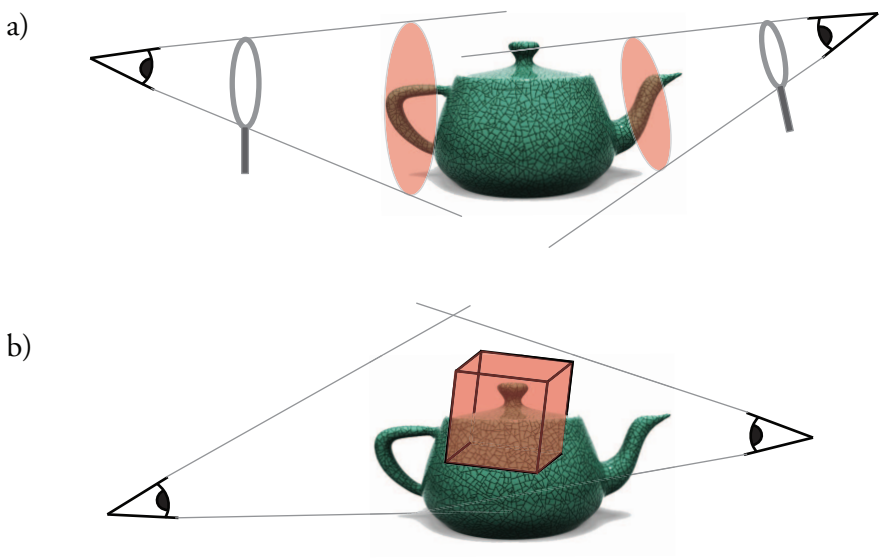

Figure 1.1: The two illustrations contrast the validity of lens focus regions for a) *flat lenses* and b) a *volumetric lens* in a three-dimensional environment shared by multiple users.

usefulness in a shared virtual environment to a single user. In addition, the lens has to be moved by the user if the viewpoint is changed in order to maintain regional focus.

Explicitly defining a *volumetric* interest region overcomes the dependency on the current viewpoint of a single user. This makes the concept suitable for collaborative virtual environments, where a common focus region can be used by multiple users independent of the viewpoint of each individual collaborator. Figure 1.1 depicts the difference between flat and volumetric lenses in the context of a collaborative environment or changing viewpoint of a single user.

1.4.4 Rendering of Lens Effects

The flat lenses of Fuhrmann & Gröller were rendered using a screen-space technique that relies on the stencil buffer, while the Magic Boxes required additional clipping

planes for correct rendering. Whereas their rendering technique required only a *single* geometry render pass (in contrast to the six rendering passes needed by Viega et al.), its applicability is very limited. This is mainly due to the fact that context geometry positioned behind the lens is not rendered at all. This behavior exhibits a strong violation of the original concept and one of its major benefits: providing an alternative presentation of the user's focus while maintaining the surrounding context.

Another major limitation of the rendering techniques of both Viega et al. and Fuhrmann & Gröller comes with their dependence on hardware clipping planes. While the clipping plane approach can be efficiently implemented on graphics processing hardware, it is limited in terms of the shape complexity of the employed lenses as well as the maximum number of lenses that can be rendered at the same time. For example, six clipping planes are required to render a single cubic lens; to approximate a spherical lens shape, a much higher number of clipping planes is necessary for a single lens. The maximum number of supported clipping planes depends on the utilized graphics hardware; the OpenGL specification merely assures the availability of six clipping planes.

Ropinski & Hinrichs (2004) presented a multi-pass rendering approach for volumetric lenses that requires a dual depth buffer–a data structure not currently directly available in graphics processing hardware. Their algorithm is inspired by the depth peeling technique, which is commonly used in real-time rendering systems to achieve order-independent transparency effects. Ropinski & Hinrichs noted the benefits of integrating lens rendering into a scene graph system for a reduction in the amount of data that

has to be processed per rendering pass. We describe in Chapters 2 and 3 how a scene graph system can be used to efficiently render composable lens effects.

Context-sensitive lenses for scene graph systems were introduced by Mendez et al. (2006). Their work mentioned *information filtering* as a potential application for volumetric lenses and notes that opposite behavior may be applied, as well, in the form of *information enrichment* (more complex rendering style, presenting additional geometry, etc.). Mendez et al. introduced a variant of the depth peeling algorithm presented by Ropinski & Hinrichs using fragment shader programs. Their technique allowed for rendering of composable volumetric lenses.

1.4.5 Interpreting Volumetric Data using Volumetric Lenses

Although most 3D lens techniques focus on real-time visualization of discrete geometric surfaces, some interesting approaches to applying volumetric lenses effects to volumetric data were presented in the past.

The work of Wang et al. (2005) takes an optical physics approach to rendering of lens effects applied to dense volumetric datasets. The authors implemented a ray casting rendering algorithm running in real-time on the graphics processing unit (GPU). Although the presented visual results are promising, their technique barely achieves interactive frame rates and is not applicable to scanline rendering techniques using the standard graphics pipeline.

Plate et al. (2007) also investigated how lenses can be used in the interpretation of volumetric datasets. Their approach allows for real-time rendering of convex

polyhedral lenses that apply various effects to the presented volumetric data. Lenses may be overlapped and the semantics of composite lens effects can be expressed using an interactive shader composer. The shader composer enables the user to establish connections and operations between input attributes (coming from the volumetric source data) that ultimately result in a single output color. Whereas the presented operations are closely related to the volumetric data being rendered, the general idea of representing the combination of multiple lens effects using a tree structure of attribute and operator nodes seems to offer an intuitive approach to the problem. We describe in Chapter 3 how we adopted the concept of *shade trees* as underlying conceptual data structure for lens effect composition.

1.4.6 Rendering of Composable Volumetric Lenses

Earlier work by Borst et al. (2007) on volumetric *windows* (an analog of the 2D windows metaphor) revealed a new approach for determining interior and exterior lens geometry during rendering. The authors introduced simple per-fragment tests to distinguish between regions affected by lens volumes (focus) versus geometry outside of any lenses (context). In this context the term *fragment* is considered to be an individually shaded (sub)pixel of the synthesized image. Borst et al. showed that expressing the three-dimensional position of a single fragment with respect to the local coordinate system of a volumetric lens makes the formulation of in-out tests for different lens boundaries very concise and intuitive. The nature of the in-out test then determines the shape of the volumetric lens.

Although Borst et al. presented interesting ideas on expanding the notion of *3D windows*, the computer graphics community has not seized the suggestion of further exploration of its potential for Virtual Reality applications.

Best & Borst (2008) presented a solution for rendering composable volumetric lenses based on the per-fragment in-out tests mentioned earlier, which allow for precise clipping of interior and exterior lens geometry. Their rendering technique requires a specialized shader program for each unique *region* that implements the composite shader effect for enclosed geometry and clips away outlying fragments. Best & Borst define the set of *regions* to be the volumes defined by lenses and their intersections. Using this approach, the number of potential render passes grows exponentially with the number of lenses (2^n for n lenses). In order to minimize computational cost, Best & Borst added a Region Analyzer module to their implementation. The Region Analyzer employs CSG (Constructive Solid Geometry) techniques to identify the minimum number of regions that need to be considered for obtaining a correct visual result. Whereas this approach guarantees correct results and improves overall performance, it causes computational overhead (and a decrease in rendering update rate) whenever the user interacts with the scene. Moving a lens, for example, may require a full update of the Region Analyzer, unless partial updates are supported by the Region Analyzer module. The computational cost then depends on the total number of lenses and the complexity of their CSG boundary representation.

We show in Chapter 3 how the rendering technique of Best & Borst can be efficiently integrated into a scene graph rendering system. In addition, we present a new rendering

technique that is capable of creating composite object-level and fragment-level lens effects in a single geometry pass. For the case of fragment-only effects, this geometry pass consists of a single render pass using a single object. For object-level effects, multiple objects need to be drawn using single-pass rendering for each respective object. It should be noted that for the special case of different object-level effects for every lens intersection region, both approaches require the same amount of GPU programs and render passes.

The work of Best & Borst described simple composition of lens effects based on per-lens blending options (e.g., replace, add, subtract). In order to achieve effect composition, subsequent calls to individual effect implementations were made to alter the value of the fragment color. For some blending behaviors (e.g., add and subtract), the previous color value was taken into account to compute the updated value. However, the expressiveness of this approach is limited when dynamic composition of complex shading effects is desired. One such example is the effect composition of a lens that modifies surface material attributes with a lens adding specular light reflectance. Our work proposes the use of shade tree concepts to allow for more tractability in the composition of lens effects.

Another interesting rendering technique for volumetric lenses defined by complex free-form shapes was presented by Trapp et al. (2008). In order to generate complex focus regions on virtual city models, the authors converted 3D geometric models to Volumetric Depth Sprites to define complex lens shapes. During rendering, the membership of a single fragment to a (composite) lens region is established using

in-out tests similar to those described by Best & Borst. In contrast to using implicit equations, Trapp et al. use layered 2D texture maps to define lens shapes. While this gives the application designer much flexibility in the creation of desired lens shapes, it suffers from aliasing artifacts due to the limited resolution of the rasterized lens shape description.

Best (2007) gave a detailed tabular overview of different lens rendering approaches, their respective capability for composition, and the number of required render passes.

Past publications have introduced different terms for the description of a class tools that implement the Magic Lens metaphor of Bier et al. In the remainder of this text, we use the term *volumetric lenses* to refer to dynamic three-dimensional focus regions that are spatially constrained and offer alternative visual presentations of enclosed geometry or display deviating content to the user within its boundaries. This concept is extended to the notion of *spatiotemporal lenses* in Chapter 4 to introduce space-time variations on time-varying datasets.

1.4.7 Lenses as Foci in Virtual Environments

Benford & Fahlén (1993) introduced a spatial model for interaction in virtual environments that describes mutual *awareness* of objects in large virtual scenes. In terms of their model, a single volumetric lens can serve as an *adapter* object, which enables a user to define his *focus* and–if the lens is visible to collaborating users–lets others be aware of its location and extent. Accordingly, multiple scattered lenses can be used to switch between different *foci*, enabling users to direct the attention of

collaborators to certain regions of interest.

As the intersection with a volumetric lens causes objects in the environment to change their visual presentation, the lens tool can also be interpreted to be defining a *nimbus*. Benford & Fahlén use the term *nimbus* to refer to the level of awareness an observed object has of its observer.

Whereas the idea of their use as collaborative tools in virtual environments seems apparent, possible benefits of utilizing volumetric lenses for computer-supported collaborative work (CSCW) have yet to be investigated.

Besides its applications in virtual environments, research in the Augmented Reality community has also adapted the Magic Lens metaphor to create a variety of tools; examples include object selection using virtual lenses as presented in Looser et al. (2007) and lenses in concert with a MagicBook interface as described in Looser et al. (2004). A thorough overview of focus and context in Augmented Reality and related areas was given by Looser (2007).

Chapter 2

Scene Graph Architecture for Real-time Rendering Systems

2.1 Motivation

This chapter gives an overview of using scene graph structures in the context of real-time rendering systems. Basic concepts and techniques are presented to add to the understanding of our application design presented in Section 3.2. We also introduce terminology related to scene graph architecture that is used in subsequent chapters.

The early years of interactive 3D application development were characterized by a multitude of system implementations that were highly reflective of the capabilities and restrictions of the available low-level graphics programming interfaces. While programming languages like C++ made high-level abstractions of data structures and their related operations more accessible than ever before, the representations of three-dimensional scenes were mostly designed based on the drawing primitives used in the underlying graphics architecture. Using this approach might speed up the initial application development process, but its weaknesses become apparent when applications need to be adapted for different architectures or when more sophisticated user interaction with the scene is required.

For example, many users would intuitively expect an interactive application to allow for direct manipulation of individual scene objects presented to them (e.g., moving a book from a shelf to a table). However, if the design of the scene object's data structures is primarily determined by the graphics primitives needed for rendering, high-level manipulation of objects and their relationships becomes increasingly difficult

to implement. With a growing desire for more natural and intuitive forms of interaction and more independence from low-level graphics architectures, it became apparent that three-dimensional scenes need to be designed as consisting of abstract, editable objects in order to allow for direct scene manipulation and guarantee extensibility of the overall application.

2.2 Definition

An *object-oriented* approach to describing a geometric scene or environment (ranging from a single polygon to virtual worlds containing thousands of characters) was introduced by Strauss & Carey (1992). They established the concept of using a directed acyclic graph (DAG) structure for dynamic representations of three-dimensional scenes. The graph consists of *nodes*, which may represent scene object geometry (e.g., polygon meshes), specifications for object presentation (e.g., surface material properties), or structural elements defining graph traversal behavior (e.g., group nodes, camera nodes, or transform nodes). The graph has to be *acyclic* to prohibit infinite loops during traversal and to allow the identification of unique *paths* through the graph.

The graph structure establishes hierarchical relationships between nodes and allows multiple references to identical elements. This is highly desirable, as the description of duplicate scene objects with identical geometry has to be provided only once. Integrating multiple copies of the object into the scene then merely requires adding a reference to the object description node to nodes already present in the graph. This

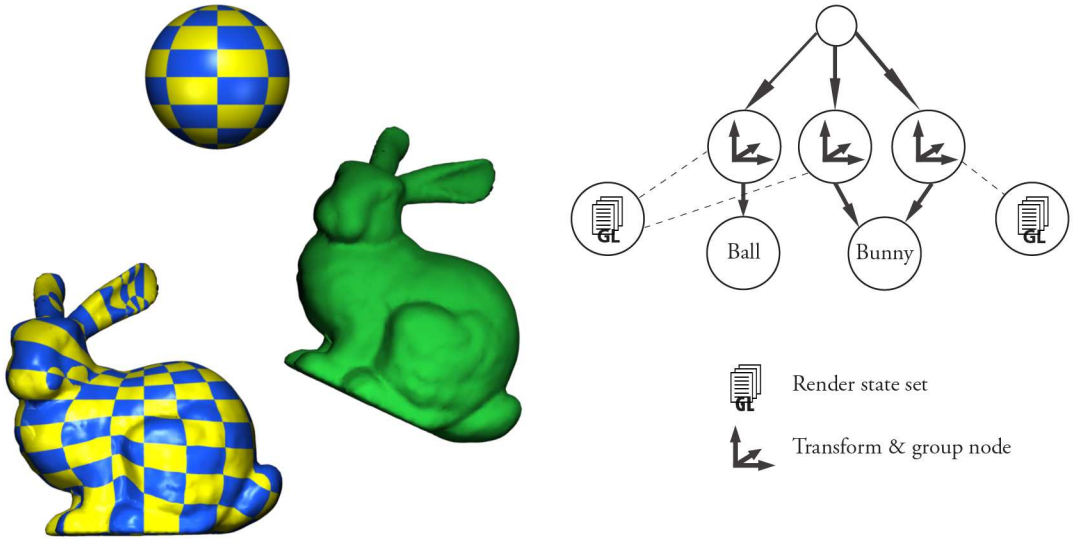

Figure 2.1: Example of a scene graph structure and a corresponding graphical rendering of the contained objects. Although both bunny objects share the same geometric data, their appearance differs due to the different render state sets connected to their parent nodes. As can be seen from the graph structure, the ball and one of the bunnies share identical render state sets. Each of the three objects is positioned in space using its parent transform node.

reference becomes a directed edge of the scene graph. The same is true for presentation properties that may be reused across multiple objects without the need for duplication (e.g., a GPU shader program implementing a certain surface shading effect that is to be applied to multiple scene objects). Figure 2.1 illustrates the concept by juxtaposing an abstract scene graph structure and a visual rendering of its elements.

2.2.1 Graph Traversal

To compute any results (e.g., visual rendering) from a given scene graph structure, a depth-first traversal of the graph is initiated at a root node (not necessarily the *global* root node, though) and a predefined *action* is performed on every visited node. Typical

actions include drawing, intersecting, culling, or computing a bounding volume. Many concrete implementations of scene graph traversals employ the *Visitor* design pattern as described in detail by Gamma et al. (1994). The use of the *Visitor* pattern allows the application programmer to keep algorithm implementations (e.g., intersection calculation) and the scene graph structure separate from each other. New actions can easily be added without alterations to the interface of the scene graph's node classes.

Most actions performed on the scene graph require the successive aggregation of node attributes during graph traversal. A prominent example is the accumulation of geometric transforms that allows three-dimensional coordinates originally expressed with respect to a *local* coordinate systems to be expressed in the *world* coordinate frame.

An update visitor traversal is generally used to allow for animated scene objects and frequent state changes. If parts of the scene are to change their transform or internal state over time, they typically realize this behavior as a callback mechanism triggered by the update visitor. The visitor object contains a time stamp that may be used by visited nodes to determine their time based state. We will describe in Chapter 4 how to exploit this mechanism for the implementation of spatiotemporal lenses.

2.2.2 Graphics System Interface

In general, many *offline* rendering systems (e.g., implementations of the RenderMan interface specification [Pixar (2009)]) also employ scene graphs to represent three-dimensional scenes, but work independently from an underlying graphics API

(application programming interface). In contrast, scene graph-based *real-time* rendering systems–which are of concern throughout this work–are built on top of a specific graphics API (e.g., OpenGL or DirectX) that interfaces with low-level graphics routines. A typical scene graph library therefore abstracts the low-level calls made to the graphics API and exposes them indirectly through manipulation of high-level objects (e.g., three-dimensional text or a geometric model imported from a file). The graphics API in turn abstracts the actual implementation of graphics routines and computations which for the most part are executed entirely on the graphics processing hardware itself.

2.3 OpenSceneGraph

Ever since the influential paper of Strauss & Carey (1992), several scene graph libraries and toolkits have been developed and used for research and engineering by the computer graphics community. Today, several open source scene graph libraries exist that allow the use of high-level scene object descriptions without compromising rendering performance. Prominent examples are OpenSG, Coin3D, OGRE, and OpenSceneGraph.

The images and performance results given in this work were generated using the author's implementation of the presented techniques; the implemented system relies on the OpenSceneGraph libraries and many of its base data types (e.g., geometry nodes, graph traversals using visitors, space partitioning using k-d trees).

The decision to implement the described lens rendering techniques using a modern

scene graph toolkit was mainly based on the fact that established techniques for hierarchical scene design, efficient space partitioning using k-d trees, culling techniques, and abstractions of GPU shader programs were available as part of the library. This allowed us to focus on the implementation of essential parts of our lens rendering system and compare different approaches in a timely manner. In addition, it enables us to present our techniques and results in a concise way using scene graph terminology, as the related concepts are established and widely known in the computer graphics community.

2.3.1 Geometry and State Sets

Similar to the original distinction made by Strauss & Carey (1992), OpenSceneGraph's data types can be categorized into those representing scene content (i.e., geometry as polygonal meshes) and those defining the way that content is rendered (render state sets). The state sets describe the instantaneous state of the underlying OpenGL graphics system by defining attribute values for common rendering settings (e.g., material properties, pixel blending mode, depth buffer activation, etc.). The distinction between geometry and state sets gives an application programmer the ability to reuse identical data structures representing scene geometry for different visual presentations. In addition, render state sets may easily be applied to multiple different geometries and the drawing process can be optimized by automatically choosing a render order that minimizes necessary state changes to the OpenGL system.

Chapter 3

Single-Pass Rendering and Advanced Composition of Volumetric Lenses

3.1 Motivation

We mentioned previously that volumetric lens tools were integrated into the rendering system used by our research group. The system was used frequently by geologists for visualization and interpretation of topological datasets. The continual observation of users working with the volumetric lenses as well as the integration of the rendering techniques with a modern scene graph system helped us to identify several implementation guidelines for volumetric lens tools. We present these principles in the following paragraphs and detail how they are reflected in our system implementation and the new single-pass rendering technique.

Make the common case(s) fast. The Magic Lens interface metaphor adapted for volumetric lenses dictates that a user has to *intersect* pieces of geometry with the lens volume in order to apply the respective lens effect. Therefore, lenses that do not intersect any scene geometry will not provide any benefit to the user and in turn we can identify the lens-geometry intersection as a very common case. In order to speed up the positive intersection test, we employ two data structures that allow for an economic implementation of the problem: k-d trees are used for efficient space partitioning of scene geometry and a specialized k-DOP (discrete orientation polytope) acts as bounding volume representation for individual lenses and their intersections.

We also found simple lens shapes (e.g., boxes, spheres) to be suitable and sufficient for our applications. Therefore, we can often achieve a reasonable approximation of

the real lens volume by employing a 6-sided k-DOP bounding volume.

In addition, best use of the tool can only be made if the process of transforming and intersecting lenses interactively does not have a distracting negative impact on rendering performance. Introducing additional lens intersections and interactively repositioning lens volumes ideally should not affect the responsiveness and update rate of the system at all. The compilation of a shader program on the GPU during run-time is known to introduce a noticeable time lag in the responsiveness of the rendering system. Therefore, our algorithm design requires that no additional shader program compilation is to be necessary for lens transformation and intersection. However, we show in Section 3.4.5 that our approach may require the compilation of a new GPU program once the order of lenses or the blending mode of a lens is changed. Whereas we expect the compilation time of an individual GPU program to be similar to the system of Best & Borst (2008), our approach lowers the frequency of creating new GPU programs when the user introduces lens intersection regions.

For generality, our algorithm should not rely heavily on optimized geometry tiling. The evaluation given in Borst et al. (2007) points out that the performance of the rendering algorithm proposed by Best & Borst depends strongly on effective space partitioning of scene geometry. The author showed that selecting an adequate tiling resolution for a given dataset becomes crucial for obtaining good performance. However, flexible tiling techniques for arbitrary geometry are not generally available and in some cases active partitioning of large geometric models might be explicitly not desired. As an example, the computational overhead introduced by the maintenance and culling

techniques for a large number of tiles might outweigh the performance decrease caused by sending redundant (invisible) geometry through the GPU pipeline. This is the case in particular if the computation related to tiling is performed entirely on the CPU and cannot be refactored to exploit parallel execution.

Our single-pass rendering approach presented in Section 3.4 reduces the sensitivity to optimized scene partitioning by avoiding redundancy with respect to fragment computations. By using a single render pass for geometry intersected by one or multiple lenses, we avoid an increase in the number of fragments to consider for a growing number of lenses. In contrast to the approach of Best & Borst, our approach keeps the number of fragment computations constant when new lenses or lens intersections are created and only fragment-level effects are considered.

Exploit the growing capabilities of graphics processing hardware. Many shader effects that rely on per-vertex attributes (e.g., surface normal or material attributes) to calculate their results can often be evaluated as a per-fragment operation. Increased performance due to highly parallel execution of shader programs on modern GPUs allows per-fragment evaluation of such effects (e.g., Phong shading) without introducing a noticeable performance penalty. In addition, the visual realism of simulated shading effects can often be drastically improved if the full shading equation is evaluated per (sub)pixel instead of using interpolated results.

The single-pass lens rendering technique introduced in this chapter exploits this recent development by delaying the complete shading computation of individual (sub)pixels until the fragment shader stage. In doing this, the membership of an

individual fragment to a lens region or a lens intersection region can be determined explicitly before any region specific effects are computed.

3.2 Application Design

We implemented the described techniques using C++ and OpenGL Shading Language (GLSL). Our system relies exclusively on libraries that support compilation and execution on a variety of operating systems and architectures. Several open source toolkits were used in the development process.

VR Juggler is used to support a wide variety of input and output devices and allows us to configure the application to run in diverse visualization environments. Besides testing the application in a single machine desktop environment, it was presented as a research demo employing stereoscopic rendering on a tiled multi-projector display (Tiesel et al., 2009).

We were able to exploit many of the data structures and operations related to scene graph computations that are part of the OpenSceneGraph library. Our system was designed with the prospect of aggregating techniques and node types related to volumetric lenses into a self-contained node kit that may be seamlessly integrated with existing visualization systems based on OpenSceneGraph. A node kit is a collection of scene graph nodes that provides interrelated functionality or abstractions. Common examples include node kits for rendering shadows, volumetric data, or terrain models.

Figure 3.1 gives an overview of the computational framework employed to create renderings of volumetric lens effects using CPU and GPU hardware. User interface

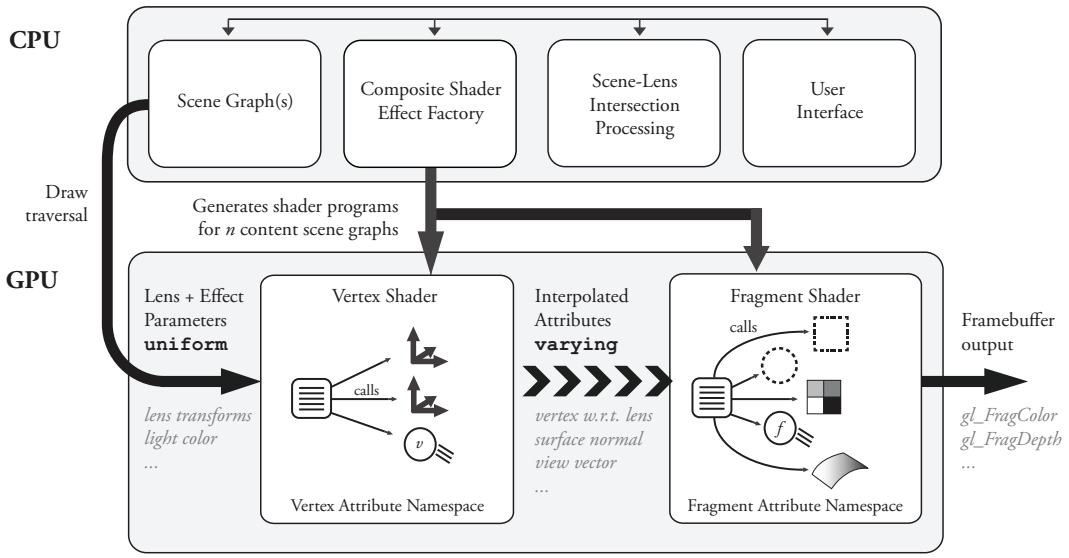

Figure 3.1: Simplified overview of our application framework showing the interaction between CPU and GPU computation.

elements alter the structure of the scene graph, for example by creating or moving volumetric lenses. Scene-lens intersections are calculated whenever the scene graph changes and they determine the scene elements that need to be rendered using a specialized GPU shader program generated by the Shader Factory module. Final visual results are computed by the GPU during the draw traversal of the scene graph initiated by the CPU. By processing scene geometry using the dynamically-generated GPU programs, proper rendering of lens effects and clipping is achieved. We describe the interaction between the different parts of our system in more detail in the following sections.

3.3 Volumetric Lens Rendering

Our implementation of volumetric lens rendering includes dynamically-generated GPU shader programs. We employ GLSL to formulate program computations.

Individual lens effects are defined in text files holding meta information (i.e., name, type, local parameters, etc.) and the GLSL source code that performs the computation necessary to achieve the respective effect. Using this approach, a large variety of shading effects may be applied to geometry intersected by a volumetric lens. Whereas geological visualizations may benefit from lens effects that use alternative color mapping schemes or show a cut-away view of the rendered topology, other applications may require lens effects to incorporate complex shading models that simulate physical light reflection behavior. By ultimately controlling the shading computation for every pixel fragment being presented to the user, any of the stated effects can be achieved using custom GPU shader programs.

The generated shader programs are executed entirely on the graphics hardware and invoke vertex and fragment level lens effects, including per-fragment clipping of scene geometry at lens boundaries. Whereas previous approaches at volumetric lens rendering used up to 2^n shader programs and render passes for a scene containing n lenses, we show in this work how a single render pass and GPU program can be used in certain cases to achieve the same visual result and better performance. We state details on the number of render passes and GPU programs required using our approach in Section 3.4.4.

3.3.1 Lens-scene Intersection

We employ optimized scene graph traversals to find geometry that is intersected by a particular lens volume and automatically apply the corresponding GPU program

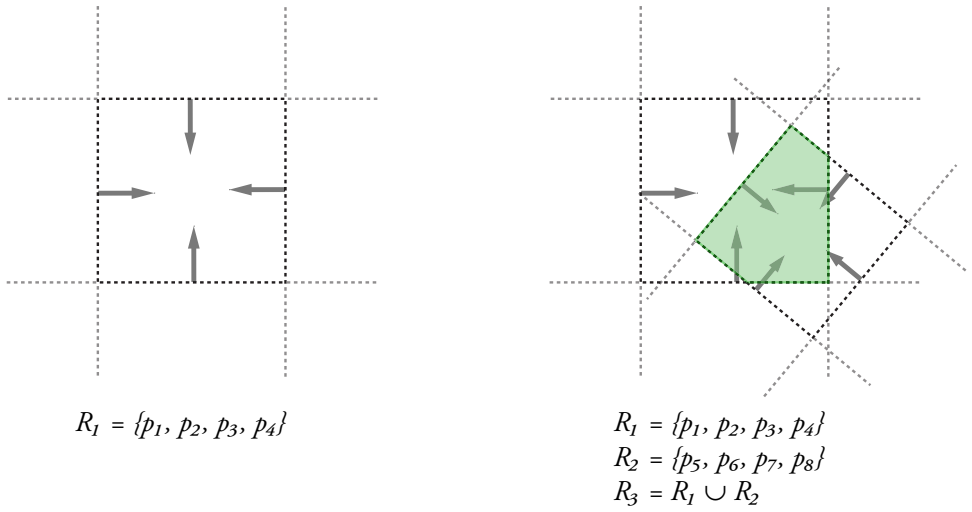

Figure 3.2: The left-hand side shows a polytope bounding volume defined by four planes projected into 2D space. Normals of polytope planes are defined to point towards the inside of the represented volume. The right-hand side shows how a polytope may be used to represent the intersection volume of two overlapping polytopes. By creating a union of the original sets of planes (R_1 and R_2), an intersection volume is defined by the set of planes denoted as R_3.

to be used for object rendering. Lens bounding volumes are represented by a 6-sided k-DOP (discrete orientation polytope). This data structure offers an efficient test for individual vertices of a mesh to determine their in-out status with respect to a lens volume. Figure 3.2 shows a 2D projection of a polytope defined by four planes. Each plane is specified using its normal direction and the plane's minimum distance to the origin. The polytopes used in our implementation are therefore represented by a set of planes. The right-hand side of Figure 3.2 shows how an intersection region can be created from two polytopes using a union operation on their set of planes.

In addition, we employ a k-d tree/line segment intersection test using the corners of the lens bounding box to speed up the detection of positive intersection results for scene geometry. For each 6-sided polytope representing a cubic lens bounding volume,

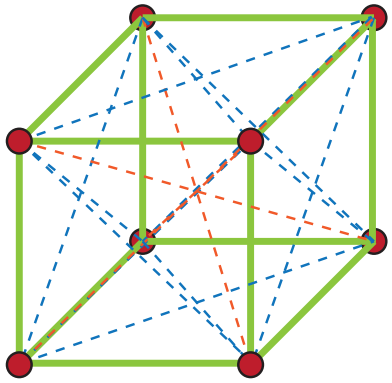

Figure 3.3: Illustration of all 28 possible line segments extracted from the cubic polytope's corner vertices. Different colors used for illustrative purposes only.

we use its 8 corner vertices to iterate through all 28 possible non-directed edges and use them as line segments for k-d tree intersection. Figure 3.3 illustrates the approach.

Subsequently, every geometry node affected by a lens is rendered using the respective GPU program, while the rest of the scene is rendered without changes. We achieve this by creating shallow copies of the intersected geometry nodes that inherit the accumulated state of the original (containing geometric transforms and render state attributes like textures or material properties). There are two reasons why duplicates rather than the original geometry nodes are used.

- Our rendering algorithm should be "non-intrusive," i.e., the changes to the *original* scene graph structure are to be kept to a minimum. Adding the activation of a GPU program to the render state set of a contained geometry node would violate this rule.
- In some cases it may be necessary to create multiple renderings of a geometry node using different GPU programs. We show in Chapter 4 that this is a requirement for correct rendering of *spatiotemporal* lens effects.

Finally, the GPU program implementing lens effects and clipping is added as an additional attribute to the render state set of the geometry node copy. During the next draw traversal of the scene graph, all geometry intersected by lens volumes will be rendered using respective lens effects and clipping as defined in the attached shader program. Duplicated geometry nodes are added to the scene graph hierarchy under a common group node. Removing the lens rendering mechanisms from the scene graphs then consists solely of removing the parent group node holding all copied geometry that is affected by lenses.

Lens-scene intersections and the subsequent tasks described above (copying of geometry nodes, creation of GPU programs, etc.) have to be performed only if relevant parts of the scene graph change. Typically, this is the case when a lens or other relevant objects are transformed or when scene objects are created or deleted. The sub scene graph holding the duplicated geometry nodes that are rendered using respective GPU programs during draw traversal serves as a simple caching mechanism for lens interactions with the scene.

It is necessary to incorporate the accumulated state information of intersected geometry into the shallow copy created for rendering in order to guarantee an identical visual presentation. We establish the accumulated state of an intersected geometry node by obtaining the complete path from the root node to the geometry leaf, iterating through the contained nodes, and successively accumulating state information. Render state attributes connected to each traversed node are successively added to a common set of attributes. Once all nodes in the path have been considered, the accumulated

state set represents the OpenGL rendering state that would be used for rendering of the geometry node during a regular draw traversal. Geometric transforms encountered during an iteration are multiplied sequentially to eventually obtain the accumulated geometric transform ${}^{root}_{geometry}\mathbf{T}$. This matrix expresses the geometry's local coordinate system with respect to the root scene graph coordinate system. The transform and render state set are then assigned to the copy of the original geometry leaf.

We implement object-level effects by applying the described intersection strategy to different sub scene graphs that hold data to be rendered inside of the lens. Although the scene-lens intersection and the geometry node duplication process are identical to the case described above, alternative GPU programs need to be generated for each distinct content scene graph. Section 3.4.6 gives details why this is necessary to allow for correct clipping of content in lens intersection regions.

3.3.2 Lens Frame Transformation and Clipping

For each lens i, we calculate the matrix ${}^{eye}_{lens_i}\mathbf{T}$, which transforms from lens coordinates to eye coordinates, and pass its inverse to the shader program. The vertex shader then uses ${}^{eye}_{lens_i}\mathbf{T}^{-1} = {}^{lens_i}_{eye}\mathbf{T}$ and the current OpenGL modelview matrix ${}^{eye}_{model}\mathbf{T}$ to transform the position ${}^{model}\mathbf{v}$ of an incoming vertex from the model coordinate system of the rendered geometry to a Cartesian lens coordinate system:

$$ {}^{lens_i}\mathbf{v} = {}^{lens_i}_{eye}\mathbf{T} \ * \ {}^{eye}_{model}\mathbf{T} \ * \ {}^{model}\mathbf{v}. \tag{3.1} $$

By calculating the partial result ${}^{eye}_{model}\mathbf{T} * {}^{model}\mathbf{v}$ in the vertex shader, we can reduce

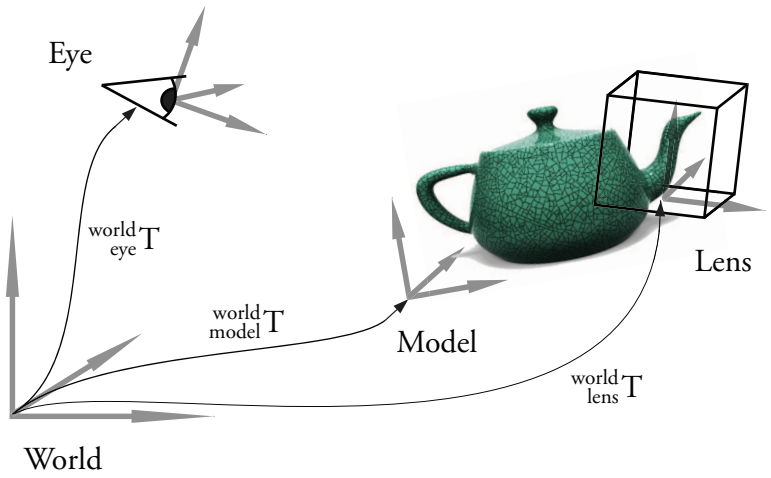

Figure 3.4: Overview of coordinate systems relevant for rendering of volumetric lens effects.

the computational cost per lens required to transform vertices into the respective lens coordinate system to a single matrix-vector multiplication. Figure 3.4 gives an overview of the coordinate systems referred to in the above transforms.

Linear interpolation of ${}^{lens_i}\mathbf{v}$ between the vertices of a primitive is performed by the graphics hardware and provides a per-fragment coordinate expressed in the lens coordinate system. This allows us to perform an efficient in-out test to determine if the fragment falls inside the boundaries of a particular lens. Individual lens shapes and their respective clipping behaviors are then defined by corresponding in-out tests evaluated within the fragment shader program.

Figure 3.5 gives the effect description for the lens transform computation of Equation 3.1. The effect definition for an in-out test resulting in a spherical lens shape is given in Figure 3.6. Note how ${}^{lens_i}\mathbf{v}$ is computed as part of the per-vertex processing (`vertex_wrt_lens`), while its interpolated value is used by the per-fragment evaluation of the in-out test performed by the code given in Figure 3.6.

```
@---HEADER---@
Volumetric Lens Transform
EFFECT_TYPE_TRANSFORM
void LensSpaceTransform_V()
NULL
@---PARAMETERS---@
// The inverse transform of lens wrt eye provided
// by the application
mat4 eye_wrt_lens
@---GLOBAL_ATTRIBUTES---@
@---LOCAL_ATTRIBUTES---@
// This local vertex position is interpolated in
// hardware for the fragment shader so that lens
// clipping can be done there wrt the lens
// coordinate system
varying vec3 vertex_wrt_lens;

@---VERTEX_SHADER---@
// This is precomputed in initialize_attributes.shader
vec4 vertex_wrt_eye;

void LensSpaceTransform_V()
{
    // Transform incoming vertex from eye coordinate system
    // into local lens coordinate system using the transform
    // provided by the application
    vertex_wrt_lens = (eye_wrt_lens * vertex_wrt_eye).xyz;
}
```

Figure 3.5: Complete effect definition for the lens space transform performed for each lens in the scene. The premultiplied vertex expressed in the eye coordinate system is transformed into the local coordinate system of the lens. By the declaring the variable to be of type `varying`, automatic hardware interpolation of its value across rendered fragments is applied.

```
@——HEADER——@
Spherical Lens Region Test
EFFECT_TYPE_REGION_TEST
NULL
bool isVertexInsideLens_Spherical_F()
@——PARAMETERS——@
float lens_size 0.05
@——GLOBAL-ATTRIBUTES——@
@——LOCAL-ATTRIBUTES——@
// This local vertex position is interpolated in
// hardware for the fragment shader so that lens
// clipping can be done there wrt the lens
// coordinate system
varying vec3 vertex_wrt_lens;
@——FRAGMENT-SHADER——@
bool isVertexInsideLens_Spherical_F()
{
    bool isVertexInside = true;

    if (length(vertex_wrt_lens) > lens_size)
        isVertexInside = false;

    return isVertexInside;
}
```

Figure 3.6: Sample effect definition for a region test of a spherical lens performed in the lens coordinate system.

For simple lens shapes (e.g., box, sphere), the in-out tests can be stated as simple range or distance tests. However, a 2D texture map with a corresponding threshold test may also be used to define the shape of a lens volume as an extruded texture. Figure 3.7 gives an example rendering of a lens shape defined by a 2D texture map. Likewise, a 3D texture may be used to define the volumetric shape of a lens. However, it has to be noted that the approaches employing textures suffer from aliasing artifacts at shape boundaries and need to incorporate techniques like texture filtering to improve visual

Figure 3.7: Two lens examples using different in-out tests to define their shape. Whereas the lens applying the red fabric effect has a spherical shape, a 2D texture is used for the marble effect lens to create an extruded "fleur-de-lis" volume.

results.

3.4 Single-pass Lens Rendering Technique

3.4.1 Note on Terminology: Single-pass versus Multi-pass

Throughout the following sections, we use the terms "single-pass" and "multi-pass" frequently to refer to technical differences between rendering approaches for volumetric lens effects. To clarify our implications when using those terms, we give a definition of "single-pass" and "multi-pass" rendering with respect to volumetric lens rendering in the following paragraphs.

In general, the term multi-pass rendering refers to repeated processing of an

object's geometric description by the graphics pipeline (including vertex and fragment shader stages) to achieve a distinct visual rendering effect for the respective object. For example, several non-photorealistic rendering (NPR) techniques require multiple render passes to each draw distinct elements of the final image. The first pass processed by the graphics pipeline could be used to draw a solid outline of the object, while subsequent passes are used to shade the object's surface. In contrast, a single-pass approach requires the geometric description of an object to be processed by the GPU only once to render the desired effect. Generally, this distinction is only made on the application programmer's side; users of a real-time rendering system are not generally aware of the rendering technique used to draw a certain object.

In the context of volumetric lens rendering, we use the term "single-pass" for a rendering technique if lens effects applied to a scene object by one or more lenses can be rendered by processing the object's geometry exactly once. The term "object" here refers to a certain instance of a geometry node for which a unique node path exists. For example, the bunnies rendered in Figure 2.1 share the same geometry node, but have different node paths (each including a unique transform node). According to our terminology, we treat them as separate objects. Therefore, we also consider lens effects that apply a transform to intersected geometry (e.g., "source box" effect described by (Borst et al., 2007)) to belong to the class of object-level effects. Likewise, we show in Chapter 4 that rendering of an object at a different time instant has to be treated as an object-level effect. In general, using a single-pass rendering approach, object-level effects are rendered by GPU processing of the geometry of each involved object exactly

once.

The single-pass technique presented in the following sections achieves rendering of fragment-level lens effects for multiple lenses by one-time GPU processing of an object's geometry.

In contrast, we call the rendering approach of Best & Borst (2008) a multi-pass technique as it requires the repeated GPU processing of an object visible in multiple lenses or lens intersection regions. However, for a case in which only object-level effects are applied inside of non-intersecting lens volumes, the technique of Best & Borst (2008) achieves lens rendering by processing each object's geometry only once. While this special case satisfies our requirement of a single-pass approach, we still consider the technique to be multi-pass because

- substantial parts of our work (including the performance comparison of both approaches) are concerned with fragment-level effects–for which the approach of Best & Borst (2008) *always* requires multiple render passes–and
- the example is a special case rather than the generally observed behavior when applying lens effects to scene geometry.

3.4.2 Motivation

The rendering technique suggested by Best & Borst (2008) requires an additional render pass for each lens and for every lens intersection region, resulting in a maximum number of 2^n passes, where n is the number of lenses in the scene. Following their approach, each pass renders only fragments that fall inside a single lens (intersection)

region; fragments found to be outside of the region are discarded after performing all in-out tests. As discussed by the authors, the technique therefore relies on efficient tiling of high resolution meshes to reduce the number of fragments being discarded in each render pass.

However, we can achieve the same visual result (and often much better rendering performance) in the case of lens effects that can be evaluated at the fragment stage. For this case, we integrate all necessary computation into a single geometry render pass. There are, however, limitations to this approach with respect to the types of lens effects that may be rendered. We illustrate the differences between effect categories in the following section.

3.4.3 Supported Lens Effect Categories

In their work on composable lens effects, Best & Borst (2008) denote three distinct types of lens effects that may be applied to intersected geometry.

- Object-level effects replace objects or data inside the focus region defined by the lens (e.g., x-ray vision);
- Vertex-level effects modify the position or other attributes of individual geometry points (e.g., magnifying lens); and
- Fragment-level effects alter the shading of individual (sub)pixels; a multitude of color, lighting, and clipping effects may be implemented using this approach.

This distinction between different effect categories is more relevant for the implementation view of volumetric lenses rather than the user's perception of different

lens effect types. Whereas for some effects the difference in effect category may be obvious to users (e.g., replacing the object inside the lens versus merely applying a different color to it), other effects like magnification may be implemented as either object-level or vertex-level effect and may not easily be categorized by a user of the system.

Most of the lens effects that we have employed for the exploration of geological datasets in the past (e.g., color map, lighting, distance tool, clipping) can be evaluated entirely at the fragment stage. To compensate for any computation necessary at the vertex level, we initialize commonly used variables (surface normal, viewing vector, etc.) at the vertex stage and provide their interpolated values to the fragment shader.

An example of a lens effect that cannot be rendered correctly using a single geometry pass is any type of vertex displacement effect (e.g., magnification lens). However, object magnification can alternatively be achieved as an object-level effect by rendering a scaled copy of the intersected geometry within the lens. Note that rendering of object-level effects generally requires more than one render pass and GPU program–even if identical geometry at a different scale is used. This is due to the fact that identical parts of the geometry might be visible both inside and outside of the lens–an effect which we cannot generally achieve by sending the respective geometry through the GPU pipeline only once. In addition, any geometric transform applied by a volumetric lens, e.g., the implementation of a "source box" as described by Borst et al. (2007), represents an object-level effect and has to be rendered using multiple GPU programs and–if the focus and context geometry are identical–multiple render

passes of identical geometry. Table 3.1 compares the lens effect categories supported by the multi-pass rendering approach of Best & Borst (2008) and the two variations of our single-pass approach, which are detailed in the following sections.

	Object	Vertex	Fragment
Single-pass (Method I)	✓		✓
Single-pass (Method II)	✓*		✓
Multi-pass (Best & Borst)	✓	✓	✓

Table 3.1: Comparison of lens effect categories supported by different rendering approaches. The distinction between the two single-pass methods is described in Section 3.4.5. (*Limited clipping capabilities; see Section 3.4.6.)

3.4.4 Comparison of Required Render Passes

In order to render all lens effects and their compositions in a single render pass, we need to consider all possible lens combinations inside of a single GPU program consisting of vertex and fragment shader code. We first generate individual lens programs that compute corresponding lens effects and that are eventually linked to the "front-end" program containing the *main()* entry point. Every linked effect exposes a function calling convention that is used in the *main()* method of the shader program to invoke individual effect implementations.

Figure 3.8 gives a comparison of the required geometry render passes for a simple scene using our rendering technique and the multi-pass approach of Best & Borst. Note that for object-level effects, multiple objects need to be drawn independently (i.e., alternating geometry inside and outside of a lens volume). However, we can draw each object using a single geometry pass regardless of the number of lenses or lens

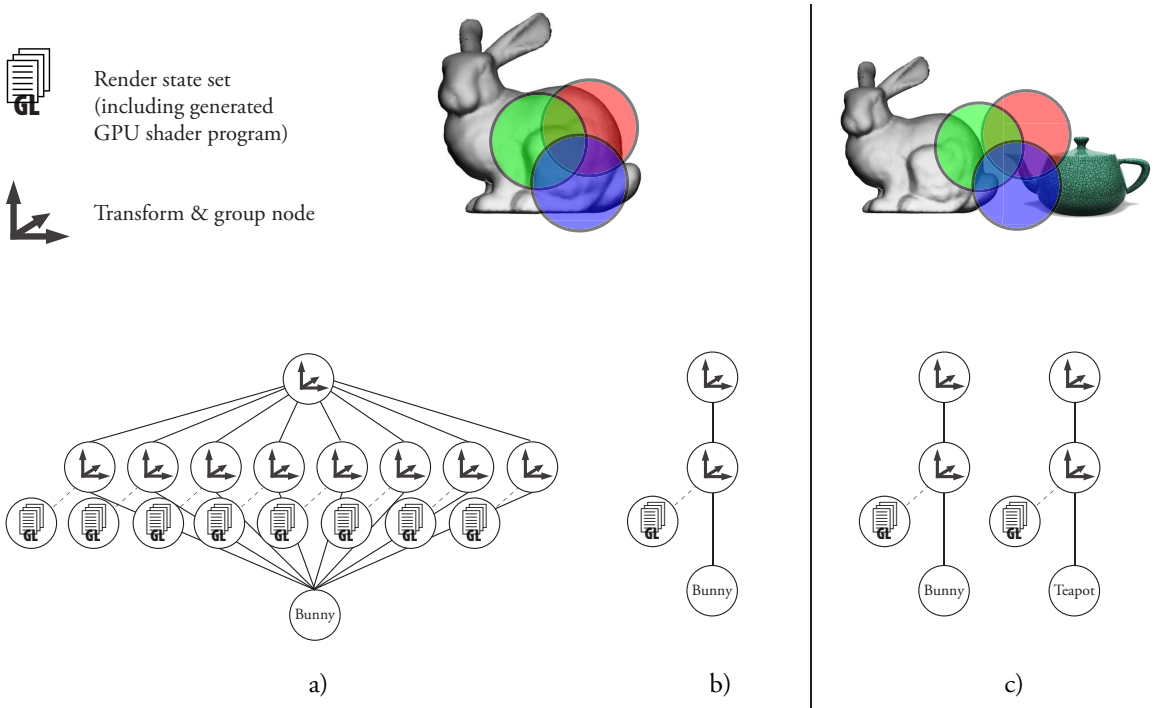

Figure 3.8: The left-hand side of the illustration compares the number of geometry render passes required to draw a scene containing a single piece of geometry (bunny) that is intersected by three volumetric lenses applying fragment-level effects. The lenses intersect each other and create a total number of eight distinct regions (including the region outside of all lens volumes). a) shows the (sub) scene graph created using a multi-pass rendering technique, while b) shows the scene graph created using our technique. The multi-pass approach requires eight different GPU programs and invokes eight draw calls of the bunny geometry. We achieve the same visual result using a single GPU program and only one geometry pass. Similarly, c) gives an example of how rendering of lenses intersecting multiple objects requires one GPU program and one draw call per object.

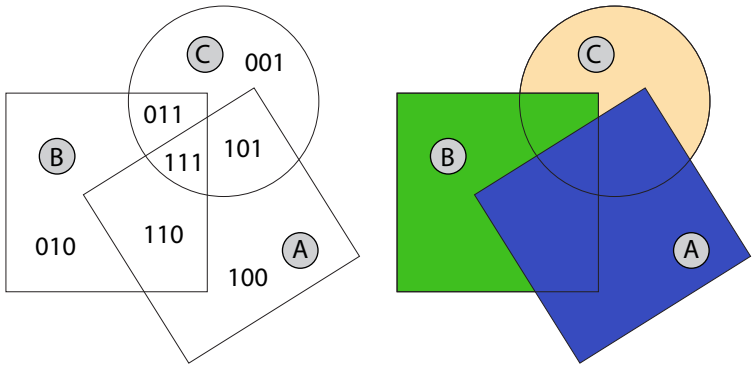

Figure 3.9: Schematic view of lens (intersection) regions and their respective region bitmasks (left). The region bitmask is established depending on the fragment's position with respect to the lenses in the scene. A fragment that is outside of all lens volumes results in a mask of 000. Lens order is A-B-C, with A being the lens effect that is applied last. Desired clipping behavior for varying lens content (depicted by the different fill colors used for the lenses) is exemplified on the right.

intersections in the scene. We show in Section 3.4.6 how clipping of object-level effects is achieved using our approach.

3.4.5 Lens Composition and Clipping using Region Bitmasks

We determine the lens effects that need to be applied to a single fragment arriving at the fragment shader by performing the lens in-out tests as described in Section 3.3.2. From per-lens test results, we establish a composite region bitmask that determines the membership of a fragment to a certain lens (intersection) region. Figure 3.9 illustrates the concept for an example case containing three lenses. Once the region bitmask is found, its value may be used to conditionally branch to a respective sequence of effect calls that implements the (composite) effect for the specific region. Figure 3.10 contains abbreviated pseudocode for a "front-end" program generated for the example case illustrated in Figure 3.9.

```
region ← 000
if fragment is inside of lens C then
    region ← region OR 001
if fragment is inside of lens B then
    region ← region OR 010
if fragment is inside of lens A then
    region ← region OR 100

if region = 000 then
    apply the object's standard rendering effect
else if region = 001 then
    activate blend mode of lens C
    apply effect of lens C
    shade fragment
else if region = 010 then
    activate blend mode of lens B
    apply effect of lens B
    shade fragment
else if region = 011 then
    activate blend mode of lens C
    apply effect of lens C
    activate blend mode of lens B
    apply effect of lens B
    shade fragment
else if region = 100 then
    ...
```

Figure 3.10: Pseudocode (abbreviated) of fragment shader program used to render a scene containing three lenses (Method I).

Not shown in the code of Figure 3.10 is the potential for optimization of certain effect sequences in a single branch during the creation of the shader source code. For example, if a lens effect A is known to *overwrite* the state changes previously computed by effect B, we can omit the call to effect B altogether and only compute the results of effect A. We note that while we can compute the resulting scene containing n lenses in a single geometry pass for each content sub graph, the number of possible regions to be considered in the fragment shader program grows exponentially (2^n cases).

This rapidly growing number of designated branches results in a maximum number of six lenses rendered at the same time using our hardware configuration as stated in Section 3.6.2. The limiting factor is the maximum instruction count for the fragment shader program supported by the utilized graphics card.

In order to support a higher number of lenses and avoid the large number of conditional branches, we suggest an alternative variant of the "front-end" program. Instead of delaying any effect computations until the complete evaluation of the region bitmask, we compute intermediate results after a positive in-out test that determined the membership of the fragment to an individual lens volume.

After the region bitmask evaluation is completed, we can determine whether to shade the fragment using the previously computed attributes (if the fragment is inside any of the lenses) or using the object's standard shading program (e.g., OpenGL's fixed-function fragment shader). Pseudocode for this approach (labeled Method II) is given in Figure 3.11.

Using implementation Method II, we were able to render scenes containing ten lenses at interactive update rates. Whereas our approach generally supports even more lenses, the limiting factor is the available number of varying floats supported by the graphics card, as a three-dimensional float vector is used to interpolate ${}^{lens_i}\mathbf{v}$ for every lens in the scene. The employed graphics hardware specified in Section 3.6 supports up to 60 floating point variables of type "varying," which corresponds to a maximum number of 20 lenses–if no other varying floats are needed for effect computation.

```
region ← 000
if fragment is inside of lens C then
    activate blend mode of lens C
    apply effect of lens C
    region ← region OR 001
if fragment is inside of lens B then
    activate blend mode of lens B
    apply effect of lens B
    region ← region OR 010
if fragment is inside of lens A then
    activate blend mode of lens A
    apply effect of lens A
    region ← region OR 100
if region = 000 then
    apply the object's standard rendering effect
else
    shade fragment
```

Figure 3.11: Pseudocode of fragment shader program used to render a scene containing three lenses (Method II).

3.4.6 Order-based Lens Clipping

If lenses are to render geometry inside of their boundaries that is different from the surrounding context (for example, a different dataset available for the same region), an individual shader program has to be created for each unique content sub graph. This is due to the fact that clipping behavior will differ depending on what elements are to be rendered. However, our single-pass rendering approach can still be used to eliminate the need for rendering geometric elements that are shared by multiple lenses more than once. Each individual shader program then has to consider all possible 2^n branches for n lenses.

In regions created by 3D lens intersections, volumetric lenses do not reveal an inherent ordering as is often the case for their 2D counterparts, where a lens closest

to the viewer occludes parts of other intersected lenses. We would like to achieve a similar effect that allows for an unobstructed view of the content shown inside of the most recently selected lens volume.

Therefore, lenses with diverse content need to apply clipping in their intersection regions depending on their respective order. We let users define lens order based on which lenses were recently interacted with and give the option to temporarily lock a certain lens order. The right-hand side of Figure 3.9 shows desired lens clipping behavior based on order. In the example, lens A was selected most recently and is therefore "in the foreground." Note how the lens order determines the clipping behavior of individual lens content with respect to involved intersection regions. In addition to the relevance of lens order for object-level effects, we show in Section 3.5.5 how the order of lenses is used to determine the composition behavior of involved fragment-level lens effects. The user can therefore control lens clipping behavior of object-level effects and the order in which fragment-level effects are applied by repeatedly selecting different lenses. In addition, our system allows users to lock a certain lens order.

For a correct implementation of the desired lens clipping, we need to be able to define clipping behavior per region. This can easily be achieved by discarding individual fragments inside of respective conditional branches using implementation Method I (given in Figure 3.10). In order to render the example scene shown in Figure 3.9, three distinct shader programs would have to be created for a correct implementation of the desired clipping based on lens order. For example, the shader program used to render the content of lens A would discard all fragments with a bitmask of 000, 001, 010,

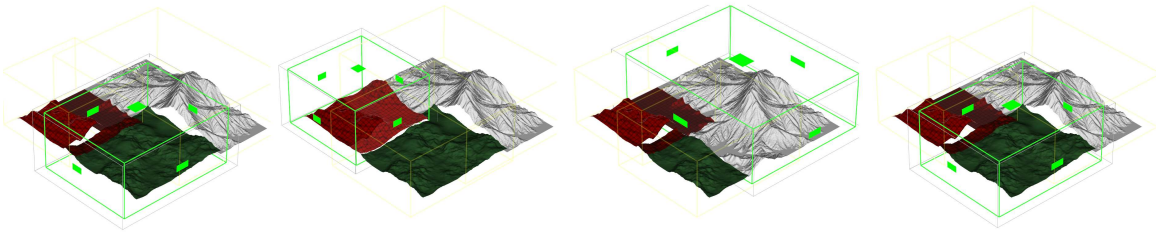

Figure 3.12: Example scene showing order-based clipping of lens content rendered using Method I. The currently selected lens is shown with a green outline. Note how the clipping of individual lenses changes as the user successively selects lenses that show different content.

and 011. The shader program generated for rendering the alternative content shown in lens C would only render fragments with a bitmask of 001 and discard all other cases. Figure 3.12 gives an example of order-based lens clipping behavior for lenses with varying content graphs as rendered by our system using Method I.

Despite the fact that multiple shader programs might be necessary for rendering of intersecting lenses, we still consider our technique a single-pass rendering approach. This is due to the fact that geometry spanning over multiple regions will be sent to the graphics pipeline only once. This is in contrast to earlier techniques that require a geometry render pass and an individual GPU program for every intersection region.

However, as individual region bitmasks are not tested in Method II (Figure 3.11), the desired clipping behavior cannot be achieved using this approach. Instead, alternating content of intersecting lenses will be "merged," i.e., multiple content sub graphs will be visible across multiple lens volumes. Alternatively, *all* fragments falling into intersection regions of lenses with different content graphs will be discarded. However, one of these behaviors is acceptable or even desired depending on the context of the application. Figure 3.13 gives example renderings generated by our system to

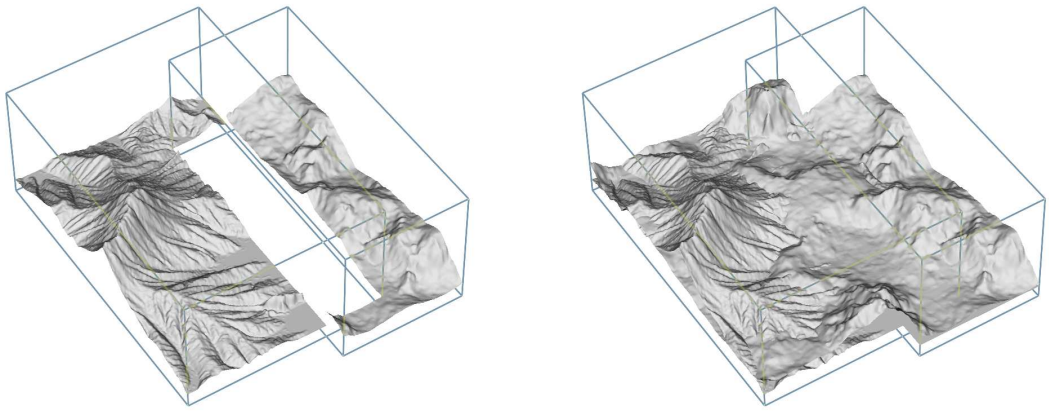

Figure 3.13: The renderings illustrate the limited abilities to implement proper clipping of intersecting lenses with different content graphs using Method II. The left-hand image shows the clipping result if fragments in intersection regions are discarded, while the image on the right shows the "merged" result that is generated if those fragments are not discarded.

illustrate the described visual result.

3.5 Shader Composition Framework

Previous rendering approaches that supported composable lenses either did not consider the composition of fragment-level lens effects in intersection regions or offered only limited flexibility in the definition of effect composition behavior. The rendering approach of Best & Borst (2008) supports composition of per-fragment shading effects, but is limited regarding the number of modifiable attributes in each effect instance. Only the predefined output variables of each respective shading unit (i.e., vertex position, fragment color, etc.) are accessible to individual effects in the composition process. While this approach allowed for basic composition of vertex displacement effects and successive blending of output colors, it could not be used to model more complex lens effect compositions and their respective blending options. One example

is the preservation of diverse light reflectance behaviors introduced by multiple lenses: using only a single color channel, we will not be able to alter the diffusely reflected material color without affecting a specular highlight added by a previous effect.

We overcome this restriction in our newer approach by introducing global attributes such as different channels of light intensity (ambient, diffuse, specular) that can be accessed and modified by shader effects during the execution of the shader program. A directional light effect would, for example, modify per-fragment light intensity according to light parameters and lens blending options. These intensity channels can then be used by the surface shader (the last effect to be applied to a fragment) to compute the final fragment color. The operation is called "shade fragment" in the pseudocode examples of Figures 3.10 and 3.11. Using this approach, a Phong surface shader could be implemented by introducing the surface normal as a varying attribute in the vertex shader code and using its interpolated value along with global attributes like material properties and light intensity channels in the fragment shader to compute the final pixel color.

3.5.1 Lens Effect Definition

As mentioned earlier, individual lens effects are defined by vertex and fragment shader source code (as they might require computation in both pipeline stages), the declaration of variables necessary for their execution and an indication of effect type (e.g., clipping, lighting, surface shader). Knowledge of the effect type is crucial for reliable and optimized composition of multiple shader effects as described in Section

3.5.5. An example of a complete effect description for a simple color effect that alters the diffuse reflectance properties of the surface is given in Figure 3.14.

3.5.2 Variable Naming Convention

Our naming convention for different types of variables accessible to the GPU effect shaders is based on McCool et al. (2004).

Parameters map to read-only uniform variables accessible in both vertex and fragment shaders and are constant per draw call. These typically describe per-lens parameters (e.g., lens transform, light direction) and are set during the application stage. This can be done either by statically binding their values to application data (e.g., modifying the lens transform matrix using an interaction device) or by dynamically exposing control over their values using introspection of the effect source code (e.g., by letting the user change the value of a parameter of type float by dragging a slider that is labeled with the respective name of the parameter).

Global attributes are used across shader effects to allow for progressive evaluation of the shaded pixel color. Examples are diffuse light intensity, surface material properties, and the currently active effect blending mode. Attributes are declared within GLSL shader code at global scope as non-qualified variables. This allows shared read-write access across shaders of the same type within the same linked program. As vertex and fragment shaders each have their own global namespace, attribute values cannot be shared between the shader units for a specific effect. This limitation can be overcome by introducing variables of type varying. These may be written to in the vertex shader

```
@---HEADER---@
Solid Color
EFFECT_TYPE_COLOR
NULL
void SolidColor_F()
@---PARAMETERS---@
float3 solidInputColor 0 0.3 0
@---GLOBAL-ATTRIBUTES---@
vec4 color_base;
vec4 material_emission, material_ambient, material_diffuse,
   material_specular;
vec4 light_emission, light_ambient, light_diffuse,
   light_specular;
int light_enabled;
int blend_mode;
@---LOCAL-ATTRIBUTES---@
@---FRAGMENT-SHADER---@
#define BLEND_MODE_REPLACE 0
#define BLEND_MODE_ADD 1
#define BLEND_MODE_SUBTRACT 2
#define BLEND_MODE_MULTIPLY 3

void SolidColor_F() {
    vec4 color = vec4(solidInputColor, 1.0);

    if (blend_mode == BLEND_MODE_REPLACE)    {
        color_base = color;
        material_diffuse = color;
    } else if (blend_mode == BLEND_MODE_ADD) {
        color_base += color;
        material_diffuse += color;
    } else if (blend_mode == BLEND_MODE_SUBTRACT) {
        color_base -= color;
        material_diffuse -= color;
    } else if (blend_mode == BLEND_MODE_MULTIPLY)    {
        color_base *= color;
        material_diffuse *= color;
    }
}
```

Figure 3.14: Definition of simple color lens effect including meta information, the declaration of shader parameters and attributes, and the GLSL fragment shader code executed on the graphics processing unit.

code and the graphics hardware will interpolate their values for each fragment. The interpolated value is then accessible inside the fragment shader. Prominent examples include the geometry's surface normal and the current view vector.

Local attributes may be used to introduce per-lens varying variables. The difference to the previously described *global* attributes is the necessity for automatic renaming of these variables to avoid naming conflicts if multiple instances of the same effect need to be linked to a single GPU program. The need for local attributes becomes apparent in the definition of the lens transform effect: although each lens requires a unique variable of type "varying" to express the rendered fragment's position with respect to the lens coordinate system, a universal effect description of the lens transform should be sufficient.

3.5.3 Resolving Naming Conflicts

Figure 3.5 showed the effect definition for the lens transform computation. Once multiple lenses are introduced to the scene, more than one instance of this effect will be necessary to provide independent lens transforms and fragment positions that are used for the per-lens in-out tests. However, simply duplicating the code specified in the effect definition will result in compilation errors as multiple variables with the same name (in this case `vertex_wrt_lens`) are found. We use automatic variable renaming where necessary to avoid such naming conflicts of shader variables.

In the example, the local attribute `vertex_wrt_lens` is automatically renamed by our system once an instance of the effect is created for a specific lens. This is

accomplished by adding the unique identifier of the corresponding lens to its name, allowing us to reuse local attributes in related effect definitions as they will be renamed following the same scheme. For example, if both the lens transform effect of Figure 3.5 and the fragment region test effect shown in Figure 3.6 are instanced for a lens with the unique identifier "2," the local attribute `vertex_wrt_lens` will be renamed to `vertex_wrt_lens_2` for both effects before compilation and linking.

3.5.4 Shade Trees

Cook (1984) introduced the concept of shade trees to overcome lacking flexibility in defining light reflection behavior for arbitrary surfaces. While many closed-form equations exist that may be used to simulate certain materials or lighting effects (e.g., copper, wood, or diffuse Lambertian reflection), most of them rely on a rigid set of input parameters and output values. Often, the single evaluated output is the outgoing radiance from an infinitesimal point on the surface toward the observer.

Cook showed how much flexibility and artistic expressiveness in defining surface shading characteristics can be gained by arranging individual shading or lighting effects into a graph structure and combining partial results using simple operators like *multiply* or *add*. A shade tree is then evaluated for every sampled surface point by traversing its nodes in postorder and obtaining the reflected surface color as the output of the tree's root node.

Cook uses the term *appearance parameters* for the set of properties that ultimately determine the observed color (i.e., the light reflection behavior) of a surface. These

appearance parameters (which also include basic geometric information like the surface normal) serve as leafs of the tree. From a mathematical point of view, the presented shade tree representation is a directed acyclic graph (DAG), as leaf nodes in the graph may serve as input to *multiple* shader nodes (e.g., the surface normal is necessary for the evaluation of many different shading models and therefore has to be connected to multiple nodes in the graph).

3.5.5 Using Shade Trees for Lens Effect Composition

We use shade trees to define effect signatures for every lens in the system which fully describe the vertex- and fragment-level effects applied by a particular lens. We incorporate the lens transform and clipping strategy described in Section 3.3.2 by adding two effects which perform the necessary per-vertex transform and the subsequent in-out clipping test. See Figure 3.15 for an example of lens signatures and their composite shade tree representation generated by our system.

Using our conceptual shade tree model, we can easily define rules for composition of effects. Every lens signature and its respective shade tree inherently define effect composition as a sequential order of effects: the shade trees in Figure 3.15 each respectively translate to a sequential order of four effects, beginning with the lens frame transform and ending with the surface shader effect that computes the final fragment color.

If lenses are intersected and a composed shader effect tree needs to be generated, there may only be one final surface shader at the top of the shade tree. A developer

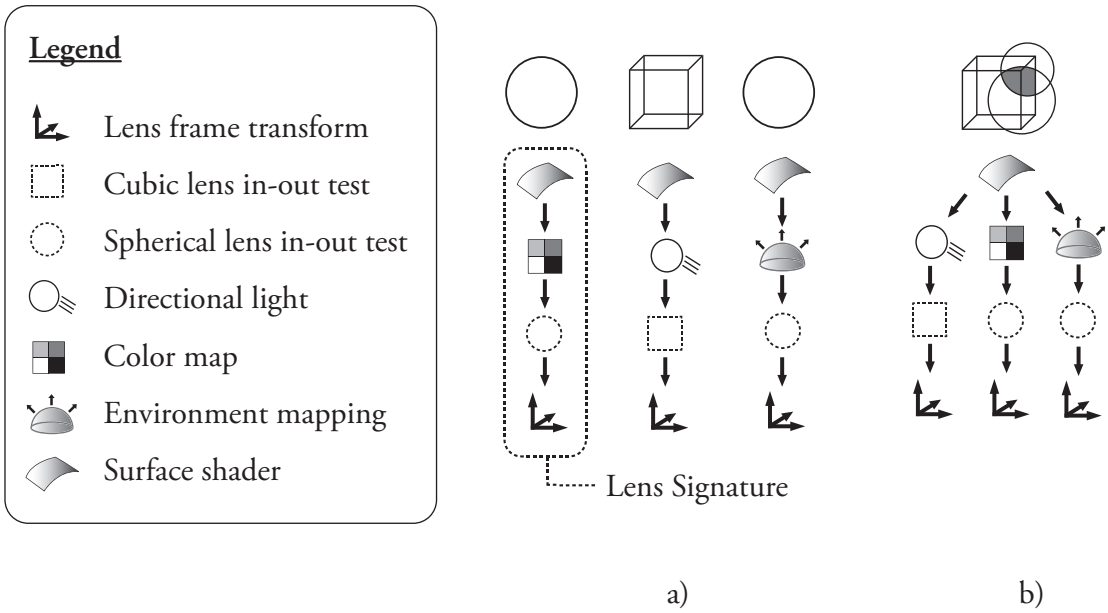

Figure 3.15: Examples of lens signatures for three lenses applying different effects are shown in a), while a respective shade tree generated by our system for the lens intersection region accentuated in gray is illustrated in b).

either writes a universal surface shader that is applied at the end of each individual lens program (e.g., general Phong shading model) or creates different surface shaders and the system chooses which one should be applied in the composite case based on lens order (e.g., Phong shading inside one lens and cartoon-style shading in another).

After the root node has been established as described, the tree is composed by successively adding the nodes of the remaining lenses' shade trees from left to right according to the global lens ordering. Figure 3.15 b) gives the resulting shade tree for a composition of three lenses with different signatures. Once the desired tree representation of the composite effect is known, the sequential order of effects can be established using a postorder traversal of the composed tree. After this transformation is performed, the sequential effect list is passed to a factory class that generates a GPU

program implementing the desired effect chain.

This approach may be used with our single-pass rendering technique as well as other multi-pass techniques that employ GPU programs for computation of lens effects. However, only Method I presented in Figure 3.10 is capable of incorporating the full effect chain generated by our shade tree traversal into the GPU program as it defines individual branches that each fully describe a composite lens effect. Method II merely provides fixed-order execution of lens effect computations.

3.6 Results

For better comparability of our performance results, we kept some variables of our machine and rendering configuration constant throughout tests. These constants are listed in Table 3.2.

Machine	Intel Core2 2.4GHz, 2 GB RAM
Graphics card	NVIDIA 9800 GX2
Operating system	Windows XP Professional (32-bit)
Rendering viewport resolution	1280x1024 pixels
Rendering options	16x Anisotropic filtering, 16x Antialiasing

Table 3.2: Constant configuration used for all performance tests presented in this work.

3.6.1 Lens-scene Intersection using *k*-d Trees

We evaluated the performance of the lens-scene intersection necessary to find geometry in the scene that needs to be rendered using the specialized shader program. For this purpose, we compared the performance of the rendering system using a naive linear search intersection technique and the k-d tree intersection test. In both cases,

a scene containing a single mesh of 720,000 primitives was rendered from an identical point of view using the single-pass rendering technique following Method II. We do not state results obtained using Method I separately, as no difference in performance was detected for the given cases. All lenses had identical dimensions and applied a height-based color map to geometry enclosed by the cubic lens volume. Both test cases employed identical optimization strategies: initial intersection test using an axis-aligned bounding box; termination of lens-object intersection test after first vertex (for linear search) or primitive (for k-d tree) was tested "positive"; no repeated lens intersection tests for geometry that was found to be intersecting any of the considered lens volumes. The results are given in Table 3.3.

			Number of lenses					
			0	1	2	3	4	5
k-d tree	No interaction		187	169	143	122	110	94
	Moving a single lens			169	142	121	110	94
Linear search	No interaction		187	169	145	122	110	93
	Moving a single lens	worst		21	18	18	19	20
		best		121	120	117	110	93

Table 3.3: Performance results for lens-scene intersection comparing rendering update rates for different intersection strategies. The update rate is given in frames per second.

As expected, the update rate hardly varies at all between the two strategies for the "No interaction" case, as no lens-scene intersection is performed. If one or multiple lenses are introduced to the scene and interactively moved by the user, the additional overhead introduced by the per-frame intersection test becomes apparent. We state the best and worst update rates for linear search, as the computational cost of the intersection varies depending on the position of the lenses with respect to the rendered

mesh. As can be seen from the results, the variance in update rate can be as large as 100 frames per second, in this particular case indicating a computational overhead of 41.7 ms for the worst case.

In most cases, the minimal overhead of the k-d tree intersection performed for each frame could not even be detected. The maximum measured cost of the k-d tree intersection was 0.07 ms per frame for a scene containing three lenses. Our results validate the decision to employ k-d trees to achieve fast lens-scene intersection tests.

It should be noted that–for a wide variety of applications–linear search is not an efficient search technique. However, we chose it for comparison to the k-d tree approach because an implementation of the linear search technique is readily available in many scene graph systems (including OpenSceneGraph). In addition, it represents a search technique with a high variance in computation time, which we wanted to contrast with the steady computational cost of the k-d tree intersection.

3.6.2 Performance Evaluation of Single-pass Rendering Technique

For the performance evaluation of the single-pass rendering technique presented in this work, we used two different test scenes. Both test scenes contained a digital elevation model with 720,000 primitives and 366,000 vertices (scene 1) or 2,883,000 primitives and 1,448,000 vertices (scene 2). The whole dataset was visible inside the chosen view frustum and all lenses introduced to the scene intersected the mesh. Identical lens shapes (cube) and applied effects (color map) were used for all lenses in the scene to allow for comparison of added rendering cost per lens. Scene geometry was

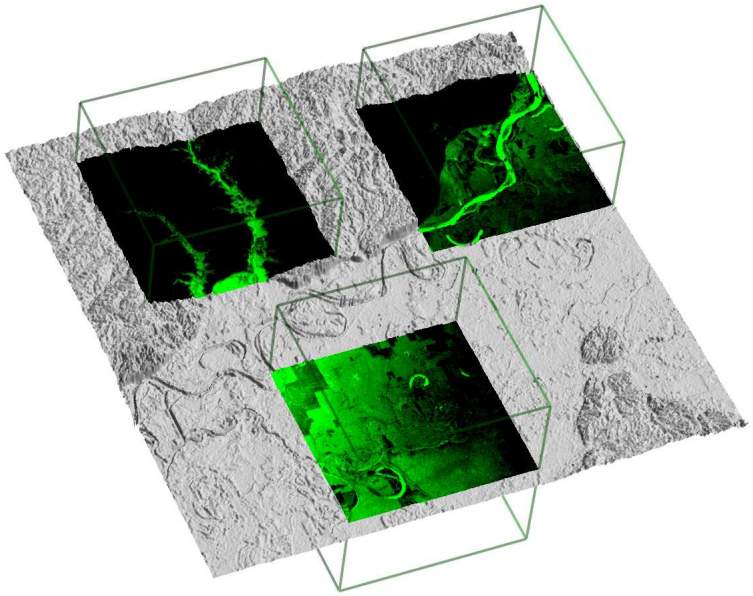

Figure 3.16: Example rendering of scene used for performance evaluation. Shown here is Scene 1 containing three cubic color map lenses. Elevation data is property of the Shuttle Radar Topography Mission (SRTM), which is headed by the National Geospatial-Intelligence Agency (NGA) and the National Aeronautics and Space Administration (NASA).

stored in GPU memory and rendered as a Vertex Buffer Object (VBO). In addition, a full copy of the scene geometry and a k-d tree representation thereof was stored in main memory and used by the CPU to perform lens-scene intersection calculations.

Our evaluation only considers fragment-level lens effects; the model shown inside of all lenses (focus) and outside of lens boundaries (context) is identical for all presented cases. A performance evaluation of object-level effect rendering was not carried out, but capabilities and limitations of our approach with respect to object-level clipping are discussed in Section 3.4.6.

No tiling of the dataset's geometry was performed to emphasize the good performance of our algorithm despite the lack of optimized spatial partitioning. We

chose a high antialiasing setting to emphasize the cost of additional lenses as each lens increases the computational complexity per fragment.

Tables 3.4 and 3.5 state the results of our performance evaluation of the single-pass lens rendering technique (using Method II) and a variation of the multi-pass rendering technique described by Best & Borst (2008). Contrary to their approach, we do not employ CSG data structures and operations for the Region Analyzer module, but use polytopes as described in Section 3.3.1 to approximate volumes of lenses and their intersections. Instead of using a custom lens-scene intersection technique like the one mentioned by Best & Borst (2008) for the multi-pass approach, we employ identical intersection calculations using k-d trees for both approaches. This is done to emphasize the difference in performance that can be attributed to GPU computations. As mentioned previously, the dataset was not tiled for either one of the compared rendering approaches. However, the number of render passes and GPU programs required for the multi-pass approach is identical to the technique described by Best & Borst (2008).

In addition to the tabular results, a graphical plot of the rendering update rate for both scenes is given in Figure 3.17. The results stated for "lens interaction" reflect the worst performance measured while moving a single lens through the mesh and thereby creating multiple lens intersection regions. We do not separately state results for Method I, as the measured performance was equivalent for all cases that included up to six lenses. The identical performance of both methods is due to a similar number of per-fragment operations (equal number of in-out tests, equal number of

hardware-interpolated floats, equal instructions for individual effect implementations) performed by the GPU. For more than six lenses, rendering Method I could not be used, as the GPU program length exceeded the maximum instruction count of the utilized graphics card.

Table 3.4: Performance results for single-pass lens rendering (using Method II).

		Number of lenses										
		0	1	2	3	4	5	6	7	8	9	10
Scene 1	no interaction (fps)	187	169	143	122	110	94	77	65	52	37	20
	lens interaction (fps)		169	142	121	110	94	77	65	52	37	20
	additional lens cost (ms)		0.57	1.08	1.20	0.89	1.55	2.35	2.4	3.85	7.8	22.97
Scene 2	no interaction (fps)	85	58	47	38	34	29	11	5	4	4	3
	lens interaction (fps)		57	47	38	34	29	11	5	4	4	3
	additional lens cost (ms)		5.48	4.04	5.04	3.1	5.07	56.43	109.1	50	0	83.33

Table 3.5: Performance results for multi-pass lens rendering.

		Number of lenses										
		0	1	2	3	4	5	6	7	8	9	10
Scene 1	no interaction (fps)	187	88	58	43	33	29	24	21	18	16	14
	lens interaction (fps)		88	58	43	33	29	24	21	18	16	14
	additional lens cost (ms)		6.02	5.88	6.01	7.05	4.18	7.18	5.95	7.94	6.94	8.93
Scene 2	no interaction (fps)	85	31	18	13	10	8	7	6	5	4	3
	lens interaction (fps)		31	18	13	10	8	7	6	5	4	3
	additional lens cost (ms)		20.49	23.30	21.37	23.08	25	17.86	23.81	33.33	50	83.33

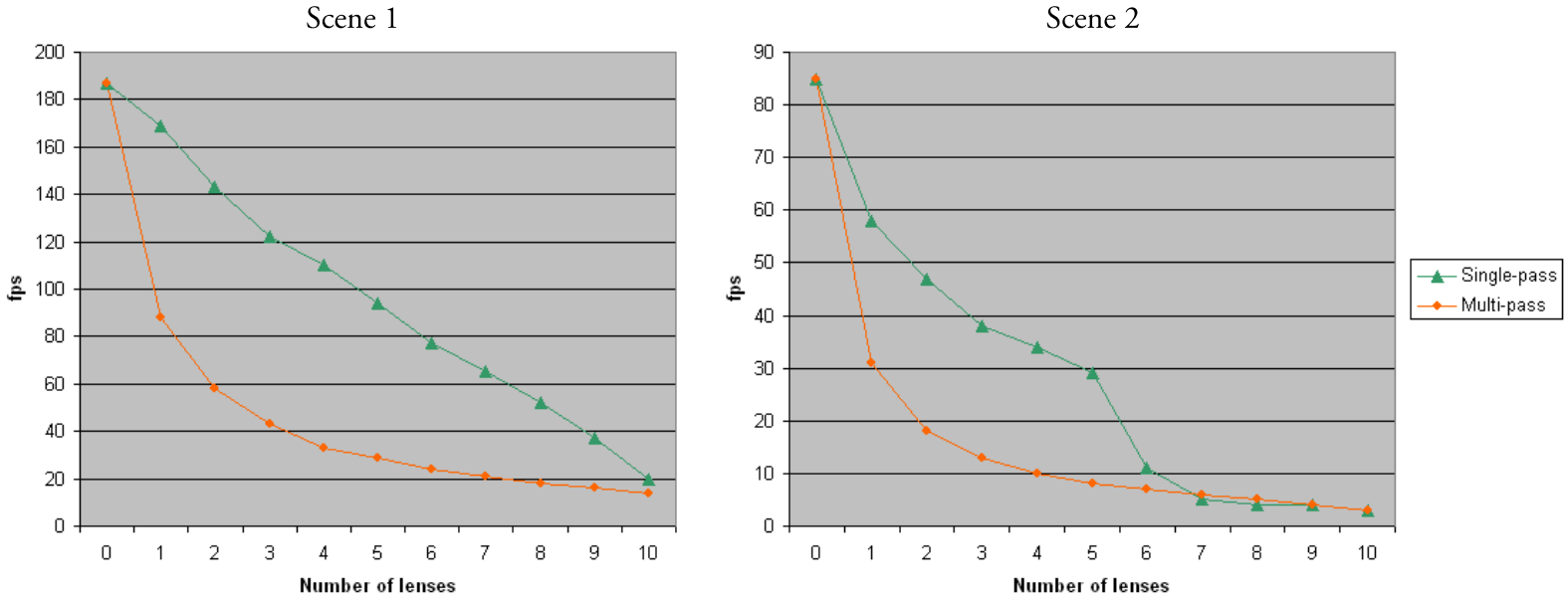

Figure 3.17: Performance plots comparing single-pass and multi-pass rendering approach.

No relevant performance differences between Method I and II were measured for cases supported by both approaches (1-6 lenses). This suggests using Method I for a relatively small number of lenses being rendered simultaneously, as we can easily achieve desired clipping behavior for lenses showing different content as described in Section 3.4.6. As soon as a greater number of lenses (more than six) is required for a certain application, a trade-off between multi-pass approaches like the one described by Best & Borst (2008) and our single-pass Method II has to be made: while the multi-pass approach requires optimized tiling of scene geometry to achieve acceptable performance for a high number of lenses, our approach may not be able to implement desired clipping behavior.

For comparison, we implemented a multi-pass rendering approach using the technique described by Best & Borst (2008). The performance results for rendering of the same test scenes using this approach are given in Table 3.5. Note that no lens intersections were created for the sample cases. However, we measured the performance for both single-pass approaches and no change in update rate was detected when lens intersections were introduced compared to the results stated in Table 3.4. This is not surprising, as the number and complexity of fragment computations are independent of the number of lens intersections. In contrast, Best (2007) gives detailed results that show how the performance of their multi-pass approach deteriorates with an increasing number of lens intersections. Also note that for the evaluated cases, we measured identical performance for the interaction and static scene using the multi-pass approach. These results highlight the benefit of fast k-d tree intersection tests and the

low overhead of the polytope region composition operations.

As expected, our algorithm outperforms the multi-pass rendering approach for the given test scenes. This is not surprising, as no tiling of the large polygonal mesh is performed to improve performance of the multi-pass technique. Comparable performance of the two approaches was only measured for rendering of scene 2 with a high number of lenses. The slowdown in the update rate apparent for the single-pass approach is due to the fact that the computational complexity of per-vertex and per-fragment operations increases with each additional lens as follows:

- Vertex stage: one additional matrix multiplication; and
- Fragment stage: three additional hardware-interpolated floats, one additional region test, and potentially additional effect computation.

Comparing the additional computational cost introduced by a new lens brought into the scene shows that for the multi-pass technique, the cost was between four and five times higher than for the single-pass approach. However, this is only true if the number of lenses is between one and five. For a high number of lenses, the additional cost per lens for our approach suddenly increases drastically. Currently, we do not have a comprehensive explanation for this effect. However, our results show that the effect appears for a lower number of lenses in the scene 2 test case (six lenses) than for scene 1 (ten lenses). This indicates that the effect might be emphasized if the number of rendered primitives is higher. In addition, we tested the performance of rendering scene 2 using a lower image resolution (1024x768 pixels) and lower image quality settings (no anisotropic filtering, lowest possible antialiasing setting). The sudden performance

drop could not be reproduced using these settings. This suggests that the number of fragments–which is considerably higher for our initial test–might have a large impact on whether the performance drop effect occurs at a certain lens count.

Using scene 1, we achieve 90% of the original "no lens" rendering update rate after the first lens is introduced to the scene. This number decreases to 68% for the more complex scene 2. For comparison, the depth peeling technique of Ropinski & Hinrichs (2004) caused a performance drop to 40% of the initial rendering update after the introduction of the first lens. Depending on scene complexity, we show that our technique is able to render four to six *intersecting* lenses at 40% of the original frame rate.

3.6.3 Performance Comparison to Rendering Approach of Best & Borst

Best (2007) gives a comprehensive performance evaluation of their multi-pass rendering approach using a dataset similar to the one used in our tests. However, Best employed optimized mesh tiling to accentuate the performance benefits of the presented approach. A direct comparison to their results is difficult due to different system configuration and datasets used. However, we measured the performance of the system implementation created by Best, which was also used in the evaluation given in Borst et al. (2007) and readily available to the author. Again, results were obtained using the system configuration described in Table 3.2.

A rendering of the test scene used in the evaluation of Best & Borst's approach is given in Figure 3.18. A tiling resolution of 1024 was used. The depicted scene

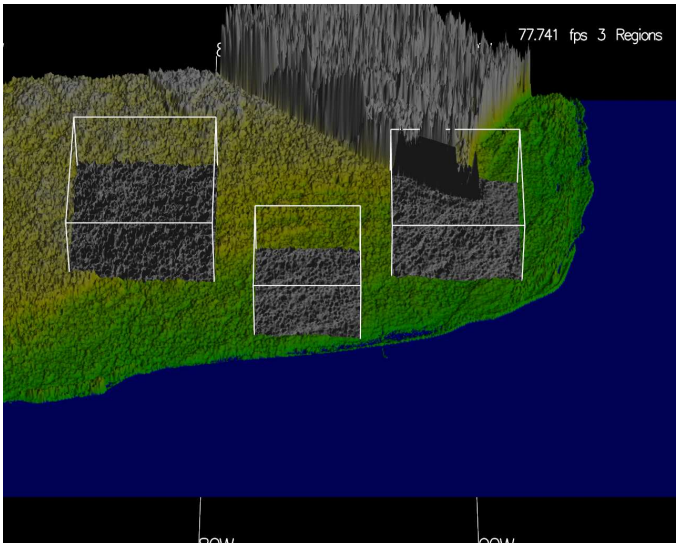

Figure 3.18: Example rendering of scene used for performance evaluation of the rendering approach of Best & Borst. Shown here is the fixed view of the scene containing three cubic color map lenses. Elevation data is property of the Shuttle Radar Topography Mission.

contains 2.16 million vertices and was rendered using a fixed viewpoint. In contrast to our application, the system of Best & Borst uses OpenGL display lists to render geometry. Our application renders the terrain data of scene 2 using Vertex Buffer Objects. While the scene and application setup is not identical for both systems, we find it reasonable to compare the performance results in terms of additional cost per lens and lens intersection. Therefore, instead of comparing the rendering update rates of the results to previous numbers, we calculated the additional computational cost required to complete the rendering of a single frame. Figure 3.19 contrasts the cost per additional lens and lens intersection region for our approach and the multi-pass approach of Best & Borst. For completeness, the rendering update rates are given in Figure 3.20. Scene 2 was used to obtain the performance results for the single-pass approach.

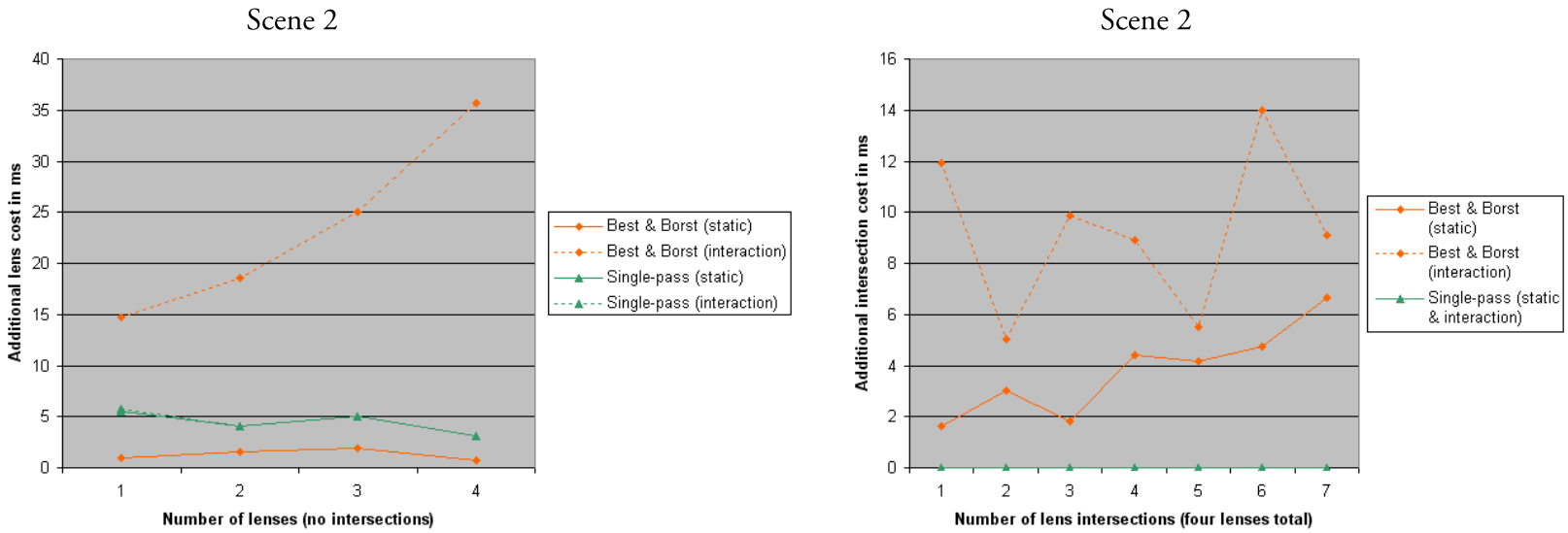

Figure 3.19: Performance plots comparing our single-pass technique and multi-pass rendering approach of Best & Borst in terms of added cost per additional lens in the scene (a). Higher numbers of lenses are not supported by the multi-pass implementation used by the author. Plot b) compares results in terms of added cost per lens intersection region in the scene. A total number of four lenses was used for all intersection cases. Note that in a), plots for the single-pass results overlap for most values shown.

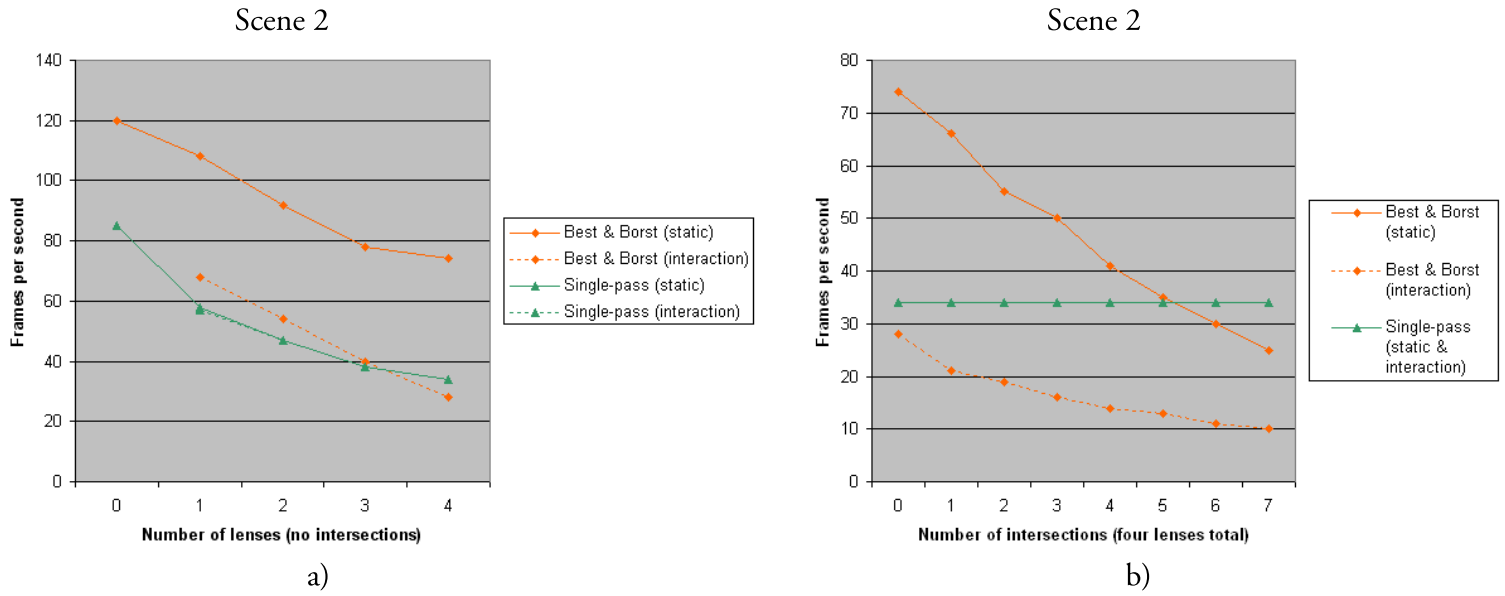

Figure 3.20: Performance plots comparing our single-pass technique and multi-pass rendering approach of Best & Borst in terms of render update rate. Note that different datasets and system implementations were used. Therefore, a direct comparison of absolute frame rates is not meaningful, whereas relevant performance trends may still be observed. While a) shows the impact of additional non-intersecting lenses on the render performance, b) compares results in terms of performance drop per additional lens intersection region in the scene. A total number of four lenses was used for all intersection cases. Note that in a), plots for the single-pass results overlap for most values shown.

Figure 3.19 a) shows that there is hardly any difference in performance if the single-pass approach is used and a user interacts with a lens volume compared to the static lens case. While the additional lens cost is smaller for the static case of the multi-pass approach (compared to single-pass), the benefit of the single-pass rendering becomes apparent for the lens interaction case. Here, the additional lens cost steadily increases (up to 35 ms) for a higher number of lenses if multi-pass rendering is used, while it oscillates between three and six ms for the single-pass case.

Figure 3.19 b) highlights another performance advantage of our single-pass approach. While the additional computational cost of lens intersections is very irregular for the multi-pass case and might be as high as 14 ms if a lens is moved, the performance of the single-pass rendering is not affected by additional lens intersections at all.

3.6.4 High-level Comparison

We see several advantages of the presented single-pass rendering approach over comparable rendering techniques:

- supports high number of lenses rendered at interactive update rates;
- hardly any performance drop during lens interaction (move, scale, rotate); and
- no performance penalty due to increased number of lens intersection regions.

Previous rendering techniques experienced a deterioration of rendering performance once users moved lenses or altered their size. Our results show that this performance decrease can be overcome using our methods. Neither lens translation nor the introduction of additional lens intersection regions caused the rendering performance

to decrease in our tests.

Our single-pass rendering technique for volumetric lenses in general does not require the *a priori* knowledge of intersecting lens volumes in the scene. Therefore, additional computation on the CPU side for what Best and Borst called "Region Analysis" and respective techniques using CSG (constructive solid geometry) operations can be omitted in cases where lenses perform fragment-level effects.

Optimized partitioning of meshes with a high polygon count can potentially increase the rendering performance of both our single-pass approach and other multi-pass techniques. Besides the speedup gained by CPU optimizations before rendering (e.g., view frustum culling), tiling may increase render performance on some GPUs; Borst et al. (2007) showed that using a higher number of small display lists rather than a single large display list can result in a substantial performance increase. However, our approach does not explicitly rely on a mesh partitioning that is optimized for lens rendering, as the number of times the mesh has to be processed by the graphics card is minimized. One reason why optimized partitioning of large meshes might not generally be desired is the computational overhead imposed on the CPU by creating and maintaining corresponding data structures. This applies in particular to deformable mesh structures that would require a frequent update of the employed space partitioning data structures. Our new rendering approach therefore reduces the dependence on optimized mesh tiling for efficient rendering of composable volumetric lenses. However, a deforming mesh would also require an update of k-d tree structures used by our approach for lens-scene intersection tests. Therefore, our technique cannot

entirely avoid additional overhead for lens effects applied to deformable meshes. While a full k-d tree update is generally an expensive operation, Zhou et al. (2008) showed that per-frame updates of the tree structure are more feasible if the GPU is leveraged for k-d tree construction. Zhou et al. (2008) report a speedup between three and six of their GPU algorithm when compared to common k-d tree generation techniques performed on the CPU.

A hybrid approach that automatically determines whether our single-pass technique or previous multi-pass approaches should be used may be employed to add the option of incorporating vertex-level lens effects.

3.6.5 Limitations and Scalability

One obvious limitation of the described rendering technique is the fact that only effects that can be rendered using a single pass are supported. Several shading effects have been presented in the past that require identical geometry to be rendered multiple times using different shader programs or alternating render state settings (e.g., disabling the depth buffer test in order to achieve glass-like transparency effects). However, our techniques for effect composition are not tied to single-pass rendering approaches, but may also be employed for lens effects that are rendered using multiple passes.

Using our technique, the number of lenses that may be rendered at the same time is limited by the number of automatically interpolated floating point variables supported by the employed graphics card. We require three floating point variables per lens to

interpolate ${}^{lens_i}\mathbf{v}$, the position of a shaded fragment with respect to the local coordinate system of $lens_i$.

In addition, the maximum number of fragment program instructions supported by the graphics card becomes a limiting factor if shader programs are generated using Method I. As 2^n distinct cases for the value of the created region bitmask have to be considered, the generated GPU program exceeds the maximum instruction count for a large number n of lenses in the scene. More efficient representations of the desired clipping behavior may be employed to reduce the complexity of the generated fragment program in the future.

In general, view frustum culling of lens volumes can be employed to limit the amount of in-out tests and possible region bitmask cases to a minimum. Unnecessary tests can be omitted for lenses that are outside of the user's field of view as no fragments that will have to be considered for rendering can possibly be affected by them.

We showed that the fast *positive* intersection test using k-d trees resulted in stable render update rates for the case of a user interacting with a lens volume that intersects relevant geometry. However, rendering performance currently decreases for the following case: the bounding box test used initially for the lens-geometry intersection test signals a potential intersection and the k-d tree intersection test does not determine a positive intersection result (compare Figure 3.21). In this case, the fallback intersection method that potentially tests *every* primitive of the respective geometry for membership in the k-DOP bounding volume has to be used. In our case, a simple linear search is used, which results in an unstable performance decrease

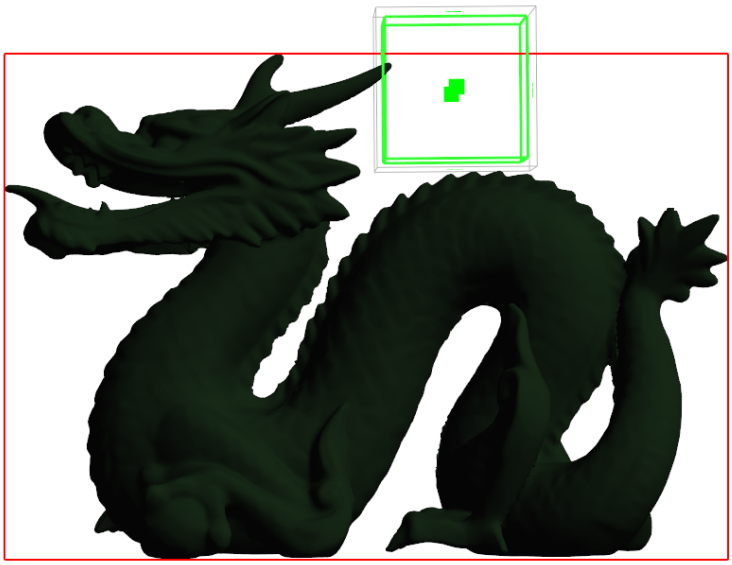

Figure 3.21: Example case for which lens-geometry intersection has to be determined by a fallback method. Shown in red is the axis-aligned bounding box of the geometry, the green outline shows the extent of the lens volume. For this case, the bounding box test indicates a potential for intersection, while the line segment/k-d tree intersection test may not detect an actual intersection. Therefore, a fallback method has to be used for intersection that ultimately determines the result of the intersection test.

depending on the relative positioning of the geometry and the internal ordering of its primitives. More efficient search techniques could be employed as fallback method in order to obtain a more predictable and stable update rate.

3.6.6 Lens Effect Composition

The new composition behavior enables us to create lens effects that alter certain surface properties (like the diffusely reflected color) without affecting related attributes (e.g., the surface's shininess). Using this approach, we can easily generate combinations of different light reflection behavior for lens intersection regions, e.g., the strong specular highlights of a marble surface and the subtle highlights of a satin-like fabric

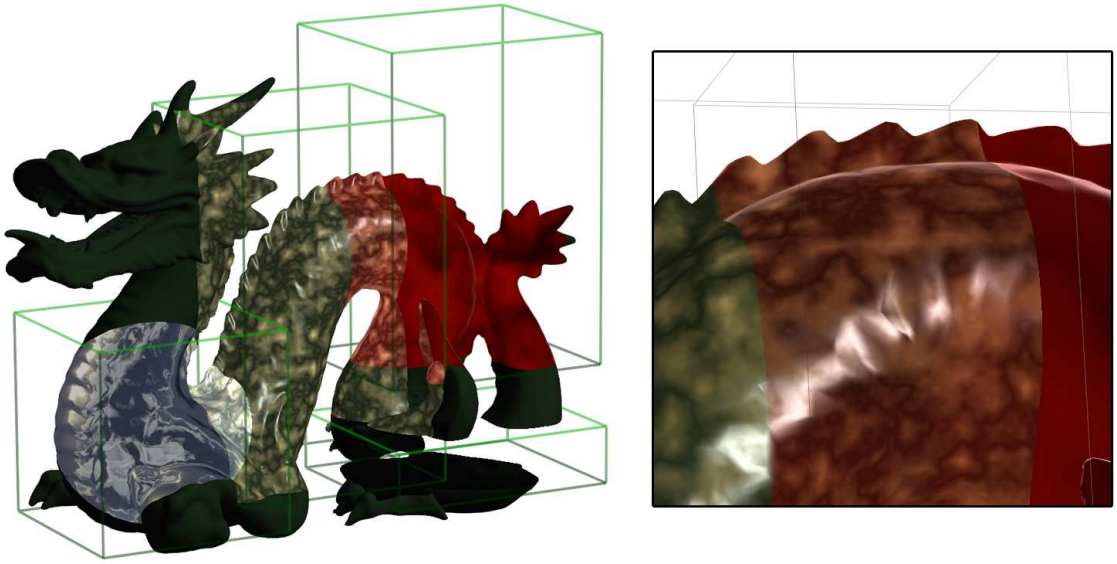

Figure 3.22: Example of lens effect composition using marble and fabric shader effect. Note how the specular highlights of the marble as well as the highlights of the red fabric at glancing angles are preserved in the lens intersection region. Dragon model is property of Stanford Computer Graphics Laboratory.

visible only at glancing angles. See Figure 3.22 for a rendering of the described composite effect.

Using our approach, the user is given the ability to change the visual qualities of composite effects by simply changing lens order and respective lens blend modes. Whereas advanced effect composition behavior of multiple lenses might not be widely required in the context of scientific visualization, we see applications for our technique in domains such as video game development and CG movie production. Shader artists in those fields are often required to develop new shader models for realistic appearance of three-dimensional objects. These shading models typically have to be formulated in a "technical" shader definition language such as GLSL (real-time rendering) or RenderMan Shading Language (offline rendering).

Our system allows users to generate composite shader effects from a library of basic effect types without the need for learning the employed shader definition language. Instead, direct visual evaluation and refinement of the created shading effect inside of volumetric lens volumes may now be employed to create a desired object appearance. As the composite effect is generated as GLSL source code by our application, the created artifact (a shading model expressed in shader definition language) may be passed on by the artist as before without the need for altering the production process.

3.6.7 Exchange of Techniques between Rendering Approaches

As described in previous sections, we were able to reuse and extend several of the approaches to lens rendering presented by Best & Borst (2008). Similar to their system, we employ a Shader Effect Factory that is capable of selectively generating the required GPU programs for the multi-pass and single-pass rendering techniques. We extended their work to allow for automatic renaming of variables to avoid naming collisions and to minimize the effort for writing a reusable shader effect definition. The concept of using in-out tests evaluated in local lens space was inherited from Best & Borst (2008); our work shows how test results may be combined into a bitmask to allow for single-pass rendering of multiple lenses. In addition, we show that in-out tests for lens volumes may be defined using extruded 2D texture shapes (see Figure 3.7).

The idea of using a single GPU program to consider all possible lens intersection regions may be used in the rendering approach of Best & Borst (2008) to avoid the creation of individual GPU programs for each lens intersection region. In this case, the

same GPU program may be used to render all regions containing identical geometry. A uniform variable may be passed to the program to determine which region branch is to be evaluated for lens effect composition. In addition, vertex-level effects may be implemented by creating a front-end vertex shader that contains branches for each possible region similar to the fragment shader pseudocode given in Figure 3.10.

While this approach would result in a simpler interface between CPU and GPU implementation of lens rendering (single GPU program instead of multiple programs per object), it would also limit the maximum number of lenses that can be rendered at the same time using the technique of Best & Borst (2008), as it does with the new technique.

The lens-scene intersection techniques presented in Section 3.3.1 may be used together with the rendering approach of Best & Borst (2008) and could potentially speed up the broad-phase culling process. In addition, using polytopes instead of CSG objects to represent lens intersection regions may help reduce the CPU overhead imposed by their approach.

The described techniques for lens effect composition are not exclusive to our single-pass rendering approach, but may be used in any related lens rendering system that employs per-fragment evaluation of lens effects. One such example is the rendering technique of Best & Borst (2008), which can easily be extended to support the described lens effect composition using shade tree concepts.

3.6.8 Scene Graph Integration

The presented rendering techniques may be integrated with any scene graph system that supports the following mechanisms particular to our approach.

- Lens-scene intersection test (e.g., using bounding volumes and k-d trees);
- Accumulation of render state and geometric transforms required for shallow copies of intersected scene geometry;
- Abstractions for GPU shader programs and their uniform variables for implementation of the shader effect factory;
- Access to the view matrix used during draw traversal for updates of ${}^{eye}_{lens_i}\mathbf{T}$;
- Extensibility of update traversal to include the setup stages required before lens rendering (intersection, copying of geometry, GPU shader generation); and
- User interface capabilities for interaction with lens volumes.

Many modern scene graph libraries already provide some of the required data types and related algorithms or are flexible enough so that missing functionality can easily be added. For example, OpenSceneGraph supports the generation of k-d trees for imported meshes, but the visitor class that implements the required polytope/geometry intersection had to be extended to make use of the fast k-d tree intersection test. Other classes had to be added but could derive some of their functionality from existing ones, e.g., a visitor class that accumulates the render state for a given list of scene graph nodes. Newly developed classes that abstract composite shader effects as well as the factory class that creates those effects from individual effect definition files had to be entirely designed and implemented by the author.

The required classes may be grouped as follows:

- Widgets and menus for user interaction with lens objects (for transforming and adjusting individual lenses);
- Factory and composite classes for shader effect generation and composition;
- Scene graph nodes for high-level representation of volumetric lenses (including visual boundary representation);
- Scene graph analysis class triggered by scene updates (initiates lens-scene intersection and creates sub scene graphs for lens rendering); and
- Builder, shape, and visitor classes for actual lens-scene intersection calculation (automatic generation of k-d trees, polytope volume representation, intersection visitors).

Chapter 4

Extending Volumetric Lenses to Spatiotemporal Visualization Tools

4.1 Motivation and Applications

One of the major innovations of interactive tools designed after the Magic Lens metaphor is the ability to provide an alternative presentation of a user's spatial focus region without loss of the surrounding context. It is the author's belief that this perceptual concept can also be applied if a *temporal* dimension is added to the presented environment. We propose the concept of a spatiotemporal lens tool that allows users to explore time-varying data through direct manipulation of time in a spatially constrained area.

Many conceivable scenarios for computer graphics applications and visualization systems involve a notion of time in addition to the spatial information presented to a user of the system. These scenarios range from large virtual environments resembling the "real world" (in which avatars over time create memorable experiences by interacting with each other) to dynamic physical simulations that are presented in an abstract fashion to allow for efficient human interpretation. Regardless of the character of those visualizations, their instantaneous internal state will be determined by a mechanism modeled after the naturally perceived passage of time.

In contrast to the real world, active manipulation of the time instant that determines the state of such systems is possible and sometimes essential to their usefulness.

4.1.1 Time Navigation Techniques

Several visualization techniques have been suggested that support the simultaneous or condensed sequential presentation of imagery ranging over multiple time instants. Rapid Serial Visual Presentation Techniques (RSVP) represent a prominent class among these. However, many of these techniques focus on 2D imagery and may not be generally applicable to 3D graphics applications, where users generally have to rely on the visual presentation related to a single time instant. Many application designers choose to expose interface elements for time control to the user that resemble related functionality for linear media playback (e.g., *play*, *pause*, and *rewind* buttons on video cassette recorders (VCR)). If the time span of interest is constrained, these controls are often accompanied by a linear time slider that depicts the current playback position. Although this approach offers intuitive control over the progression of time as presented by the visualization system, it is typically limited to controlling a global "clock" that determines the overall state of the visible environment.

Recently, Wolter et al. (2009) suggested an interesting time navigation technique for scientific visualization that uses direct manipulation of scene objects to control time. Their system lets users drag objects along their three-dimensional trajectory, thereby controlling the overall simulation time. Wolter et al. showed that for certain applications, this type of time navigation is superior in terms of navigation speed and user satisfaction to the prevalent time slider technique. However, their approach relies on the existence of spatial trajectories for scene objects of interest. If an observed object remains stationary over time (but exhibits, for example, changes in surface

properties), this navigation technique is not applicable.

Ryall et al. (2005) introduced the concept of a Temporal Magic Lens that may be used for a combined temporal and spatial query of video data. The tool described by the authors serves as a "window in time" and is defined by a *region* of interest and a *time period* of interest. The Temporal Magic Lens then presents a visual summary of the spatial region over the given time period. Ryall et al. achieve this for their particular application by blending multiple video frames into a composite image. Their system allows for control of the stacking order and weighting of individual frames that are considered in the composition of the presented image.

The Temporal Magic Lens of Ryall et al. is defined in terms of a combined spatial and temporal query. Whereas the *spatial* query is equivalent to other "flat" Magic Lens tools (i.e., the user positions a two-dimensional lens shape over the region of interest), the *temporal* query is specified by a user-defined time range. Unfortunately, this specification limits the applicability of the approach to those visualization techniques for which an aggregation of multiple time instants in a single representation is available and desired. While this representation can easily be achieved and interpreted for the domain of *video*, the concept of Ryall et al. lacks generality for other visualization domains.

Recently, Adar et al. (2008) presented an information system that employed a 2D *temporal* lens used for visual, structural, and textual queries of archived web content. Each lens exposes a time slider that allows control of the time used to query and render its respective web site content. Their Zoetrope system also allows the user

to create *bind groups* of lenses that simultaneously update their time sliders if one of them is changed by the user. The authors pointed out that users found it easier to track changes in a small focus region rather than having to scan the entire screen to detect changes in the observed web site. To the author's knowledge, the work of Adar et al. is the only example of lens tools that offer temporal and textual queries in addition to the common visual and structural filters. Similarly, most published work on *volumetric* lenses has focused exclusively on *visual* (vertex or fragment level) and *structural* (object level) filters.

4.1.2 Spatiotemporal Lens Tools

We propose the concept of "spatiotemporal lenses" (or "time warp lenses") as versatile visualization tools that combine spatial and temporal queries within user-defined focus regions. The tool enables users to introduce spatial and temporal focus regions to create simultaneous views of the visualized data at different time instants. Analogous to the Magic Lens metaphor, the temporal context may be preserved as the lenses do not affect global time.

Figure 4.1 gives an example of a typical scenario in which spatiotemporal lenses may be used to simplify time navigation. Imagine a user studying population density data that is being rendered inside of a spatial focus region defined by a volumetric lens. To find prominent local features in the time-varying samples, the user might want to continuously move the global time back and forth while observing changes in the visualization. This task is typically separated from the additional scene navigation and

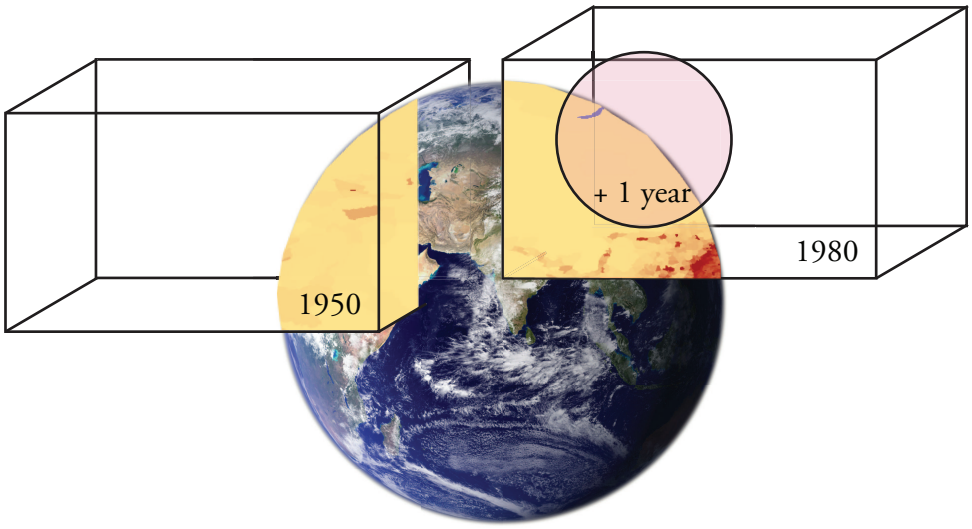

Figure 4.1: Example application for composition of spatiotemporal lenses (constructed). By combining *absolute* time referencing with a *relative* time offset, a user can study changes in the visualized population density data between 1950 and 1951 as well as between 1980 and 1981 in different parts of the world by simply moving the respective lenses. Images are property of NASA–Visible Earth Project.

lens translation necessary to refine the individual's focus region.

Using our spatiotemporal lens metaphor, we can combine these tasks into a single translation task. Initially, the user creates a spatiotemporal lens and defines an *absolute* time reference (e.g., the year 1950) or a *relative* time offset (e.g., one year) to be used. Intersecting the lens with the visualized data now causes the data's internal time-based state to be altered for rendering inside of the lens volume. Overlapping a lens with a fixed time reference of 1950 with a lens applying a relative time offset of one year results in population data of the year 1951 to be shown inside of the intersection region. By moving the lenses in and out of the region of interest, the user can easily observe time-based changes in the data without the need to constantly switch between time navigation and spatial navigation.

The direct manipulation of time using a spatial lens structure may be applied for developing and testing of hypotheses regarding spatiotemporal correlations in time-varying data. For example, a scientist interpreting population density maps may be interested in finding spatial regions with a large annual increase in the measured density value. To accomplish this task in a visualization system, one would typically have to switch repetitively between spatial navigation (translation, rotation, scale) and time navigation (change global time in increments of one year) to alter both the spatial *and* temporal focus. Using a spatiotemporal lens, this task can be simplified to a single spatial manipulation task. If the lens applies a time increment of one year to an enclosed data fragment, it may be moved through a spatial interest region to observe any changes in the data that occur over the course of a single year.

In the following paragraphs, we describe potential applications of the presented tool metaphor to underline the validity and usefulness of our approach.

As high-resolution medical imaging techniques like magnetic resonance imaging (MRI) are now widely available and affordable, it becomes feasible and desirable for medical professionals to interpret and compare patient data obtained at different time instants (e.g., during an extended period of chemotherapy treatment). The visualized data could be used for monitoring of tumor growth or to observe the body's response to medical treatment. While examining the time-varying data, the observer then typically varies both his *spatial* and *temporal* focus continuously (e.g., by first rotating the regarded 3D model and subsequently moving forward in time to watch its progression). Multiple spatiotemporal lenses may be used in this context to simultaneously observe

and discuss progression at different time instants. ("While it took two months for metastasis A to disappear, we can see that metastasis B is still apparent after four months of treatment.")

The Temporal Magic Lens described by Ryall et al. can be regarded as a special case of our concept. Instead of rendering a single time instant within the volume of a spatiotemporal lens, multiple representations obtained at different time instants may be aggregated and combined into a single representation. Besides its applicability to the domain of video sequences, this aggregation technique is commonly used in computer animation systems and is often referred to as "ghosting." Figure 4.2 shows an example of how the technique–which is typically applied to whole objects in the scene hierarchy–may be combined with a spatiotemporal lens to reduce the added visual complexity of the overall presentation to a minimum.

Another application of the presented technique is the introduction of an independently controlled timeline for a time warp lens. If a user wants to observe changes in a spatial focus region over a certain period of time, the lens may be configured to automatically create a looped playback of the enclosed region using time constraints defined by the user. Moving the lens through the scene will allow the user to observe an animated view of data inside the lens volume. The visual context is preserved and the scene remains "uncluttered" as the looped animation is only applied to data within lens boundaries.

Our *spatiotemporal lens* interface metaphor extends the original Magic Lens metaphor by providing focus and context in both *spatial* and *temporal* dimensions.

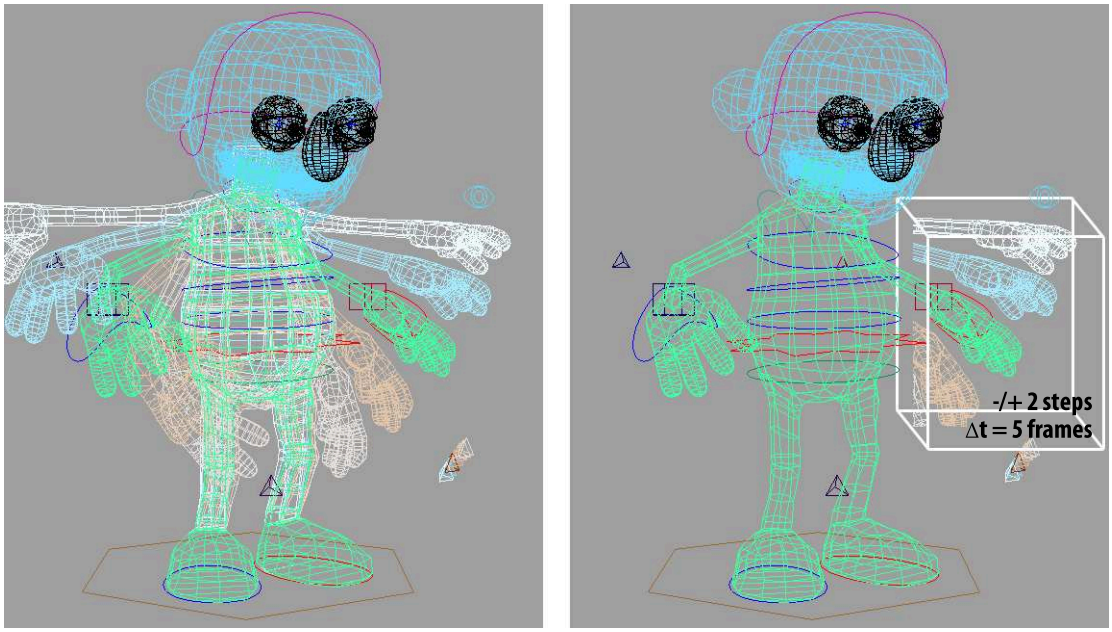

Figure 4.2: The left image shows a typical use of the *ghosting* effect in modern 3D animation systems. The right image gives an example of using an aggregating spatiotemporal lens to achieve the same effect in a user-defined region of interest without affecting the rest of the scene (constructed). Image generated using Generi character rig by Silke (2009).

We believe that the tool can provide an intuitive interface for time navigation tasks, especially if changes in the visualized data over the course of a fixed time span are to be observed.

4.1.3 Spatiotemporal Lenses in Virtual Environments

In the context of immersive virtual environments, many applications of spatiotemporal lenses are imaginable. As suggested by Viega et al., the lenses might serve as "crystal balls"–windows into time and space. Enabling a user to "step into" a spatiotemporal lens and use it as a portal would allow users to virtually travel through space *and* time.

Spatiotemporal lenses can be combined with a concept for wayfinding in virtual worlds that was presented by Elvins et al. (2001). In their work, three-dimensional thumbnails called *worldlets* serve as easily recognizable landmarks in large virtual environments. Worldlets may be created by a user within the environment as a spatial snapshot of his immediate surroundings. Once created, individual worldlets typically provide enough spatial context to use them for future wayfinding within the virtual world.

The idea of using self-contained fragments of the virtual environment for navigation and wayfinding readily extends to *spatiotemporal snapshots*, which allow users to capture their immediate environment at a certain point in time. If the appearance of the virtual world is affected by the passage of time (e.g., simulation of different environmental conditions like sunlight, weather, traffic, etc.), the ability to capture the user's surrounding at the time of visit might be a useful addition to the worldlets technique. The temporal context of such a snapshot may help users to identify and remember a previously visited location (e.g., the user first visited the place of interest during night and heavy rainfall) despite the fact that it might look significantly different once the user returns to it.

4.2 Design and Implementation

4.2.1 Absolute and Relative Time Offset

We identify two distinct types of time warp effects applied by our lenses: relative and absolute time offsets. A time warp lens with the absolute time t_{abs} will show a

rendering of the visualized data at time t_{abs} within its boundaries, while the unaffected parts of the scene are shown at global time t_{global}. Using a relative time offset of t_{rel} for a time warp lens will take into account the global time and add the corresponding time offset to this value.

We use the following notation to describe composition behavior of time warp lenses below: $R_{abs} < t_{abs} >$ denotes a time warp region of *absolute* time t_i, while $R_{rel} < t_{rel} >$ is a time warp region with *relative* time offset t_{rel}. The absolute time inside of $R_{rel} < t_{rel} >$ depends on the global time t_{global} and is computed as the sum $t_{global} + t_{rel}$.

The time warp effects described above only incorporate individual, non-intersecting lenses. We describe composition behavior of time warp lenses in Section 4.2.3.

4.2.2 Interface Design

The interface design of a spatiotemporal lens has to unambiguously communicate the time instant shown within its boundaries. Otherwise, users are unable to rely on their interpretation of the additional information presented within the lens volume. For many applications, a textual description of the depicted time will be sufficient. Depending on the temporal granularity of the available data, certain time formats or units will be more suitable than others. In the case of a physical simulation that uses very small time steps to update its results, the description might have to reflect time differences as small as a microsecond. For a user interpreting population density data that is collected annually, stating only the year in which the data were collected

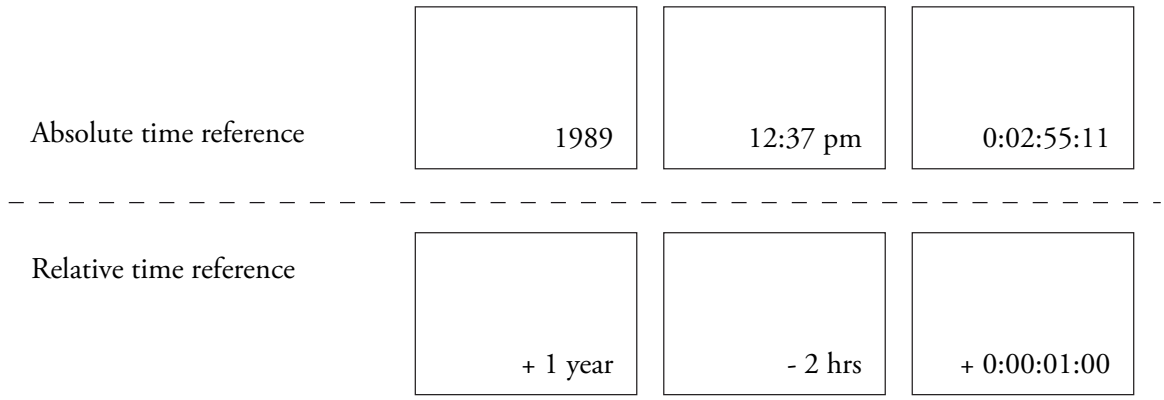

Figure 4.3: Conceptual user interface for spatiotemporal lenses using either absolute or relative time offset to be applied to intersected objects. Different time formats may be used depending on the domain of the visualization and the temporal granularity of the observed data.

might be sufficient. If the concept were integrated into a character animation system, established time code formats like SMPTE may be the best choice to help domain experts in understanding the functionality of the tool. Figure 4.3 shows several conceptual examples of user interface designs for a spatiotemporal lens that reflect the discussed considerations.

Depending on the chosen time format, suitable user interface elements should be chosen to let the user define the time offset of a particular lens. For example, a calendar tool may be used to pick a month or an individual day of the year as absolute time if the temporal granularity of the visualized data matches this. Figure 4.4 gives several examples of how user interface elements can be used to communicate the time warp effects created by multiple lenses.

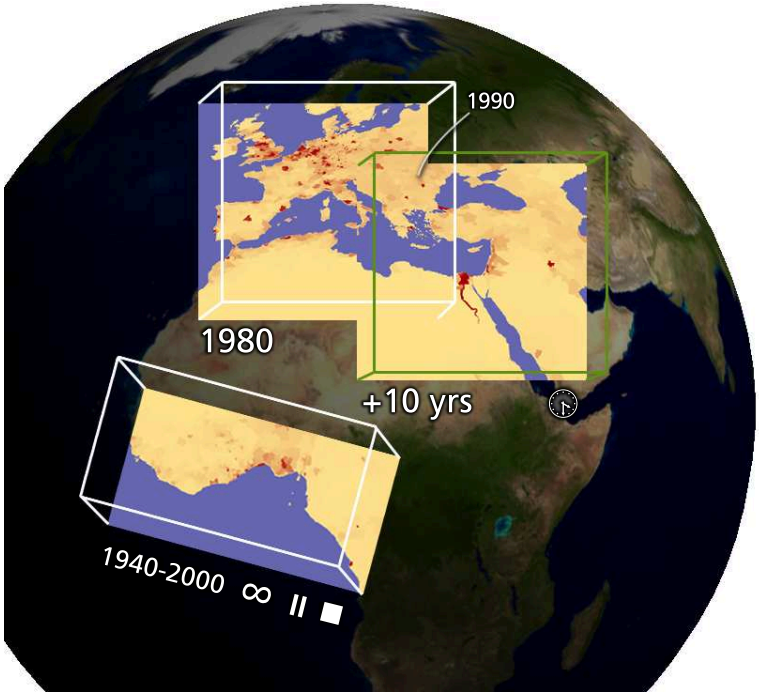

Figure 4.4: Conceptual user interface for different spatiotemporal lenses. The lens labeled "1980" applies an absolute time to the rendered population data ($R_{abs} < 1980 >$). As it is intersected with a lens applying a relative time offset ($R_{rel} < 10 >$), the intersection region $R_{abs} < 1990 >$ is created. The lens in the bottom left shows a special case of absolute time: instead of a single *fixed* time instant, it uses a time *range* and continuously updates its time to create a looped animation of the population data between the years 1940 and 2000.

4.2.3 Lens Composition

The necessity of designing composition behavior for volumetric lenses arises naturally if the user is allowed to spatially overlap multiple lenses. Therefore, we considered the existence of semantically meaningful composition of lenses that operate in both the spatial and temporal dimension. As described in Chapter 3, using shade tree concepts for the composition of surface shading effects offers a flexible and powerful design to create a large variety of visual effects. However, the composition of temporal lenses has not been considered previously. Due to the lack of previous research known to

us, we present a first attempt at a design for meaningful composition of spatiotemporal lenses.

While users might find it harder to intuitively predict and interpret composition behavior for spatiotemporal lenses, we believe that their composition can be meaningful and aid in the task of time navigation. Especially the composition of an absolute time and a relative time offset may be used to easily compare changes over a fixed period of time for different spatial or temporal references. An example of how this concept can reduce the complexity of required time navigation to a simple translation task involving multiple spatiotemporal lenses is given in Figure 4.1.

Also, the composition of time warp lenses and other 3D lens types (e.g., lighting, clipping, visual filtering) is supported by our technique. As described for the composition of visual effects, we let the user establish lens order by interacting with different lens volumes. Depending on their order, we define the composition behavior of multiple time lenses as follows.

$$
\begin{aligned}
R_{abs} < t_1 > \cap^* R_{abs} < t_2 > &= R_{abs} < t_2 > \\
R_{abs} < t_1 > \cap^* R_{rel} < t_2 > &= R_{abs} < t_1 + t_2 > \\
R_{rel} < t_1 > \cap^* R_{abs} < t_2 > &= R_{abs} < t_2 > \\
R_{rel} < t_1 > \cap^* R_{rel} < t_2 > &= R_{rel} < t_1 + t_2 >
\end{aligned}
$$

The right-hand operand of the intersection operations denotes the lens region that was more recently interacted with and that is therefore closer to the front of the ordered list of lenses. As can be seen from the equations, absolute time references *overwrite*

previous time changes, while relative time offsets are *added* to the existing time. As stated before, it may be necessary to communicate the semantics of lens composition through additional user interface elements. Figure 4.4 gives suggestion of how this could be achieved in a visualization scenario by introducing additional annotating text elements.

In addition to time warp lenses that apply a *fixed* relative or absolute time (offset), one can think of scenarios in which a user might want to create a time lens that warps the time inside of its boundaries to midnight (or any other time of day). Alternatively, a lens may warp the time used for visualization of intersected objects to the next full hour. Both cases can not be modeled (or composed) using a fixed relative time offset or absolute time only.

However, both described cases can be implemented by introducing semantics and intersection rules for the new time warp type. In some cases, the introduction of additional composite time warp types might be required to create the desired time warp effect. We exemplify the possibility of extending the class of time warp regions by a lens type that sets the time inside of its boundaries to the next full hour.

For the following examples, we assume that time values t are given in minutes and the "origin" $t = 0$ can be classified as a "full hour." Let $R_{fullhour} < t_i >$ be the new time warp region, where t_i is used as a relative time offset for region composition. A top-level lens region of this type would therefore be written as $R_{fullhour} < 0 >$. A

possible definition of the intersection operation for the new class is given below.

$$
\begin{aligned}
R_{abs} < t_1 > \ \cap^* \ R_{fullhour} < t_2 > &= R_{abs} < t_3 >, t_3 \leq t_1 + t_2 + 59 \wedge t_3 \% 60 = 0 \\
R_{fullhour} < t_1 > \ \cap^* \ R_{abs} < t_2 > &= R_{abs} < t_2 > \\
R_{rel} < t_1 > \ \cap^* \ R_{fullhour} < t_2 > &= R_{fullhour} < t_1 + t_2 > \\
R_{fullhour} < t_1 > \ \cap^* \ R_{rel} < t_2 > &= R_{fullhour} < t_1 + t_2 >
\end{aligned}
$$

Note that the results of the last two lines, which only differ in the order of region operands, are identical. However, a composition of a "full hour" lens followed by a relative time offset lens should first add the number of minutes missing to the next full hour and subsequently add the relative time offset. To support this case, a composite "helper" region would need to be constructed that supports this behavior. Consequently, semantics and a set of intersection rules would have to be created for this region type, too.

As seen in the example, the complexity of implementing consistent composition behavior of spatiotemporal lenses rises substantially if more sophisticated time warp effects are required. For the described case, a trade-off has to be made between creating a full set of composition rules for all involved time warp regions and (partially) abandoning the lens-order based semantics for composition of time warp effects.

Alternatively, composite spatiotemporal lens behaviors could be represented as a chain of time warp effects with each element applying a specific operation to an input time description to produce a resulting time value (e.g., adjusting the given input time to the next full hour). In this case, composite effect chains would be evaluated to an

absolute time value after spatial intersection regions have been determined.

4.2.4 Rendering

For best rendering performance, it is desirable to render a scene containing spatiotemporal lenses in a single render pass. However, if scene geometry is transformed or its hierarchical structure changed over time, this is not generally possible. Figure 4.5 illustrates a typical case in which scene geometry needs to be rendered multiple times to create a correct visual result inside of a spatiotemporal lens. However, we can still employ the techniques described earlier to optimize rendering performance by minimizing the number of times a geometry element has to be processed by the graphics hardware.

Before a scene containing spatiotemporal lenses may be rendered, all unique time stamps apparent inside of lens volumes and their intersection regions need to be identified. The resulting list of time stamps is then used to iteratively update the rendered scene graph to respective time instants. Only after the update can we correctly compute the scene objects that intersect the lens volumes, as the intersection result may differ due to time-based changes in position, size, shape, etc. of relevant objects in the scene. For each time instant, lens-scene intersection is performed as described in Section 3.3.1 to identify geometry being affected by lenses, and prepare it for rendering using one of our dynamically generated GPU programs.

Applying the above rules for intersection of multiple spatiotemporal lens regions, we employ a "region subdivision" algorithm (see Figure 4.6) to identify all unique, non-

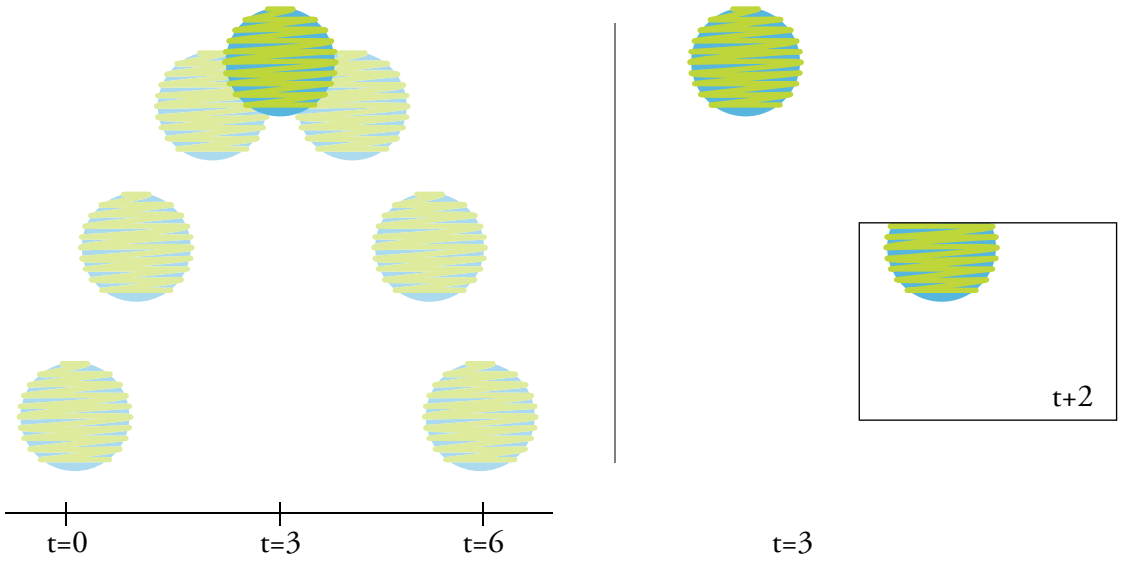

Figure 4.5: Example scene illustrating the need for handling time warping as object-level effect. On the left, a time composition of a scene containing a single animated sphere is shown. While the solid rendering of the sphere depicts its position at time $t = 3$, the non-opaque spheres show its position at previous and successive time steps. On the right, a conceptual rendering of the same scene at time instant $t = 3$ is shown. In addition, the scene now contains a spatiotemporal lens that renders the scene at time $t = 3 + 2 = 5$ inside its boundaries. As the sphere has moved and parts of it are visible inside *and* outside of the lens volume, its visual representations have to be treated as two distinct scene objects (compare Section 3.4.1). The depicted lens effect therefore has to be categorized as object-level effect.

overlapping regions created as combinations of lens volumes. The region subtraction operator used in the algorithm affects only the region's bounding volume and lens clipping behavior; time offsets are not altered due to subtraction. The obtained list of regions is then used to extract all unique time instants that are to be shown inside of the lens (intersection) volumes.

Figure 4.7 presents a high-level pseudocode summary of the steps required to render a complete scene containing time warp lenses. The algorithm is simplified as it only shows the rendering of lenses sharing a single scene graph S at multiple time-based

Input : R^{in}: list of n top-level regions defined by the volume descriptions of the n lenses in the scene; R_i^{in} denotes the i-th element of the list

Result: R^{out}: a complete list of unique, non-congruent, non-overlapping regions created as combinations of R^{in}'s elements

Data: R_m denotes a specific region, while R^n represents a denumerable list of regions of which R_m^n is the m-th element. An empty list is denoted by $\emptyset$ and $R_\emptyset$ represents an empty region, i.e., its spatial extent is 0

Operators: List operations: $\setminus$ – subtract, $\cup$ – append.
Region operations: $\cap^*$ – intersect, $-^*$ – subtract

$R^{out} \leftarrow \emptyset$
foreach R_i^{in} *in* R^{in} **do**
  $R_{current} \leftarrow R_i^{in}$
  $R^{temp} \leftarrow R^{out}$
  $R^{out} \leftarrow \emptyset$
  foreach R_i^{temp} *in* R^{temp} **do**
    $R_\alpha \leftarrow R_{current} \cap^* R_i^{temp}$
    if $R_\alpha \neq R_\emptyset$ **then**
      $R^{out} \leftarrow R^{out} \cup \{R_i^{temp} -^* R_{current}, R_\alpha\}$
      $R_{current} \leftarrow R_{current} -^* R_i^{temp}$
    else
      $R^{out} \leftarrow R^{out} \cup \{R_i^{temp}\}$
  $R^{out} \leftarrow R^{out} \cup \{R_{current}\}$

Figure 4.6: Region subdivision algorithm used to identify all non-congruent, non-overlapping regions from a list of lens volumes.

states $S(t_i)$. In a full implementation, the algorithm has to be performed multiple times for all distinct content sub graphs S_i used for lenses in the scene.

Our single-pass rendering technique can be employed to assure that each geometry element in $S_i(t_j)$ that intersects multiple regions needs to be processed by the graphics pipeline only once. In some cases, the number of geometry elements to be rendered can be decreased further. For the example shown in Figure 4.4, the geometry inside of all spatiotemporal lenses is identical, only the texture map used for surface shading varies between lenses. Therefore, complete rendering of the scene could be achieved using a

```
S ← scene graph to be rendered inside of lenses
L ← create regions from ordered list of lenses
R ← region subdivision (L)
T ← extract list of unique time stamps from R
foreach t ∈ T do
    establish S(t)
    foreach r ∈ R : warpedTime(r) = t do
        intersect r with S(t)
        foreach intersecting scene object o do
            add copy of o to S
            apply GPU program to copy of o
establish S(t_global)
render S
```

Figure 4.7: Algorithm for rendering of a scene graph structure containing spatiotemporal lenses.

single geometry pass of the mesh used for the earth's shape and a single GPU program modeled after Method II.

A combination of "regular" volumetric lens rendering and the described time warp lenses is straightforward. By simply representing each volumetric lens as a spatiotemporal lens with an offset of $R_{rel} < 0 >$, we can use the presented rendering techniques to support both types of lenses. This makes it easy to integrate visual lens filters that work uniformly when the respective lens is intersected with a spatiotemporal lens or other volumetric lenses. For example, imagine a lens that highlights all sample values in a population map that are above a certain threshold level. Using our rendering approach, we can apply this effect uniformly to any intersected object regardless of whether the object's time instant was previously altered by a time warp lens.

4.3 Results

While we are convinced of the usefulness of tools similar to the spatiotemporal lenses introduced in this paper, a formal user evaluation will be necessary to test for a positive effect of our tool metaphor on task performance. An evaluation should focus on tasks that require frequent and effective time navigation. Also, many extensions of the technique are possible. For example. the time offset of several lenses could be linked together to allow the investigation of time-delayed effects in a dataset. Adar et al. (2008) showed how this mechanism could be used in the context of 2D lenses; further exploration of the potential of combining multiple timelines in 3D visualizations is required to show its general applicability.

Chapter 5

Conclusion and Closing Remarks

5.1 Conclusion

We present an efficient single-pass technique for rendering of composite volumetric lens effects on the object and fragment level. Our approach decreases the dependence on optimized tiling of high-resolution meshes that is imposed by other multi-pass rendering algorithms. Real-time visualizations that cannot afford to maintain a mesh partitioning optimized for lens rendering (e.g., due to a deformable mesh structure) will benefit from our single-pass technique.

While the above is true for fragment-level effects, we show that our approach is limited with respect to its ability to provide correct clipping behavior for object-level effects involving a large number of lenses. However, in contrast to earlier approaches, the performance of our technique does not decrease as lenses are moved or intersected.

We expect that details of the presented rendering technique may be improved in the future. As described earlier, further optimizations for certain visualization needs may be combined with our technique. For example, view frustum culling of lens volumes can be used to minimize the complexity of generated GPU programs and therefore the per-frame computational complexity.

We show that complex surface shading effects may be combined efficiently for real-time rendering and can be used as dynamic lens effects. This allows the creation of interactive previews of future surface shading/material property changes to a certain object. As more sophisticated rendering effects can be computed at interactive frame

rates today, our technique enables users to preview different shading effects side by side applied to partial regions of a 3D object. This technique can be used for the creation of composite shading effects by artists to decrease the time spent on finding the desired effect that can then be applied to the whole mesh.

We imagine a variety of future additions to the user interface of the described tools. One such example is the application of the toolglass metaphor introduced along with the Magic Lens tool by Bier et al. (1993). Using a toolglass attached to the non-dominant hand may be a beneficial addition to a VR application. For example, the toolglass may be used to easily choose point colors while laying down points onto a topological surface rendered inside of a volumetric lens. Alternatively, it may be used to help in selecting a desired shader effect for a volumetric lens by providing a preview of the effect on the surface of the toolglass.

We present a focus and context time navigation technique for real-time visualization systems that builds on the Magic Lens metaphor. We state details on an efficient implementation of the tool using our previously described rendering techniques. While our work presents the semantic and computational fundamentals of the spatiotemporal lens metaphor, more work is necessary to evaluate the general usefulness of the approach and to identify concrete application scenarios. Several challenges arise with the composition of spatiotemporal lenses that have to be addressed by future work. Examples include the question of how to avoid ambiguity in communicating the selected time instant per lens for general applications and the effective design of a user interface that gives consideration to the added complexity of the tool.

5.2 Closing Remarks

The presented research is a continuation of previous work on volumetric lenses by Christopher M. Best, Christoph W. Borst, and Vijay B. Baiyya conducted at the Virtual Reality Laboratory at The Center for Advanced Computer Studies (CACS), University of Louisiana at Lafayette. Former research assistants working for Christoph W. Borst in the Virtual Reality Laboratory developed a visualization system that included rendering support for composable volumetric lenses.

The lenses were used commonly in the interpretation of topological datasets by Gary L. Kinsland, who is an Endowed Professor of Geology at the University of Louisiana at Lafayette. This ongoing collaboration allowed the author to discuss and evaluate many of the presented innovations with a user who actively utilized the volumetric lenses on a regular basis.

Results of the geological interpretation of LIDAR data using volumetric lenses were published at the Annual Convention of the Gulf Coast Association of Geological Societies in 2008 (Kinsland et al., 2008). A research demo highlighting the use of composable volumetric lenses for surface exploration was presented by the author at an international conference on Virtual Reality (Tiesel et al., 2009). A poster abstract introducing the single-pass rendering technique described in Chapter 3 was accepted for publication at an international conference on Computer Graphics (Tiesel & Borst, 2009). In addition, a paper that proposes spatiotemporal lenses for time navigation was submitted to an international conference on visualization.

All results described in this work are based on a self-contained implementation of

the respective algorithms performed by the author. All figures containing renderings of volumetric lenses (except for Figure 3.18) were generated exclusively using the author's own implementation. The bunny, dragon, and buddha model used to create some of the images in this text are property of Stanford University Computer Graphics Laboratory. Other data sources are identified in figure captions.

Bibliography

Adar, E., Dontcheva, M., Fogarty, J., & Weld, D. S. (2008). Zoetrope: interacting with the ephemeral web. *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, 239–248.

Benford, S., & Fahlén, L. (1993). A spatial model of interaction in large virtual environments. *ECSCW'93: Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, 109–124.

Best, C. M. (2007). *A Complete Solution for Composable Volumetric Lenses*. Master's thesis, University of Louisiana at Lafayette.

Best, C. M., & Borst, C. W. (2008). New rendering approach for composable volumetric lenses. *Proceedings of the IEEE Virtual Reality Conference 2008*, 189–192.

Bier, E. A., Stone, M. C., Pier, K., Buxton, W., & DeRose, T. D. (1993). Toolglass and Magic Lenses: the see-through interface. *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, 73–80.

Borst, C. W., Baiyya, V. B., Best, C. M., & Kinsland, G. (2007). Volumetric windows: application to interpretation of scientific data, shader-based rendering method, and performance evaluation. *Proceedings of the International Conference on Computer Graphics and Virtual Reality 2007*, 72–80.

Cook, R. L. (1984). Shade trees. *SIGGRAPH Computer Graphics*, *18*(3), 223–231.

Craig, J. J. (1989). *Introduction to Robotics. Mechanics and Control. Second Edition*. Reading, MA: Addison-Wesley.

Elvins, T. T., Nadeau, D. R., Schul, R., & Kirsh, D. (2001). Worldlets: 3-D

thumbnails for wayfinding in large virtual worlds. *Presence: Teleoperators and Virtual Environments*, *10*(6), 565–582.

Fuhrmann, A., & Gröller, E. (1998). Real-time techniques for 3D flow visualization. *VIS '98: Proceedings of the conference on Visualization '98*, 305–312.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Professional.

Itoh, M., Ohigashi, M., & Tanaka, Y. (2006). WorldMirror and WorldBottle: Components for interaction between multiple spaces in a 3D virtual environment. *Proceedings of the Tenth International Conference on Information Visualization*, 53–61.

Kinsland, G. L., Borst, C. W., Tiesel, J.-P., & Das, K. (2008). Interpretation and mapping in 3-D virtual reality of pleistocene Red River distributaries on the prairie surface near Lafayette, Louisiana. *Gulf Coast Association of Geological Societies Transactions*, vol. 58, 525–533.

Looser, J. (2007). *AR Magic Lenses: Addressing the Challenge of Focus and Context in Augmented Reality*. Doctoral dissertation, University of Canterbury.

Looser, J., Billinghurst, M., & Cockburn, A. (2004). Through the looking glass: the use of lenses as an interface tool for augmented reality interfaces. *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, 204–211.

Looser, J., Billinghurst, M., Grasset, R., & Cockburn, A. (2007). An evaluation of

virtual lenses for object selection in augmented reality. *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, 203–210.

McCool, M., Du Toit, S., Popa, T., Chan, B., & Moule, K. (2004). Shader algebra. *Proceedings of the ACM SIGGRAPH 2004 Conference*, 787–795.

Mendez, E., Kalkofen, D., & Schmalstieg, D. (2006). Interactive context-driven visualization tools for augmented reality. *Proceedings of the IEEE/ACM International Symposium on Mixed and Augmented Reality 2006*, 209–218.

Pixar (2009). RenderMan - Developers Corner - RI Specs. Retrieved February 23, 2009, from https://renderman.pixar.com/products/rispec/.

Plate, J., Holtkaemper, T., & Froehlich, B. (2007). A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics*, *13*(6), 1584–1591.

Ropinski, T., & Hinrichs, K. (2004). Real-time rendering of 3D magic lenses having arbitrary convex shapes. *Journal of the International Winter School of Computer Graphics (WSCG04)*, 379–386.

Ryall, K., Li, Q., & Esenther, A. (2005). Temporal magic lens: Combined spatial and temporal query and presentation. *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*, 809–822.

Silke, A. (2009). Generi - free Maya rig. Retrieved February 23, 2009, from http://www.andrewsilke.com/generi_rig/generi_rig.html.

Strauss, P. S., & Carey, R. (1992). An object-oriented 3D graphics toolkit. *SIGGRAPH*

Computer Graphics, *26*(2), 341–349.

Tiesel, J.-P., & Borst, C. W. (2009). Single-pass rendering of composable volumetric lens effects. *SIGGRAPH '09: ACM SIGGRAPH 2009 posters*, 1.

Tiesel, J.-P., Borst, C. W., Das, K., Kinsland, G. L., Best, C. M., & Baiyya, V. B. (2009). Composable volumetric lenses for surface exploration. *Proceedings of IEEE Virtual Reality 2009*, 291–292.

Trapp, M., Glander, T., Buchholz, H., & Döllner, J. (2008). 3D generalization lenses for interactive focus + context visualization of virtual city models. *Proceedings of the 12th International Conference on Information Visualisation, IV 2008*, 356–361.

Viega, J., Conway, M. J., Williams, G., & Pausch, R. (1996). 3D Magic Lenses. *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, 51–58.

Wang, L., Zhao, Y., Mueller, K., & Kaufman, A. E. (2005). The magic volume lens: An interactive focus+context technique for volume rendering. *Proceedings of IEEE Visualization 2005*, 367–374.

Wolter, M., Hentschel, B., Tedjo-Palczynski, I., & Kuhlen, T. (2009). A direct manipulation interface for time navigation in scientific visualizations. *Proceedings of IEEE Symposium on 3D User Interfaces 2009*, 11–18.

Zhou, K., Hou, Q., Wang, R., & Guo, B. (2008). Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, *27*(5), 1–11.

Tiesel, Jan-Phillip. Bachelor of Science, University of Bremen, Summer 2006;
Master of Science, University of Louisiana at Lafayette, Spring 2009
Major: Computer Science
Title of Thesis: Composable Visual and Temporal Lens Effects in a Scene Graph-based Visualization System
Thesis Director: Dr. Christoph W. Borst
Pages in Thesis: 123; Words in Abstract: 212

ABSTRACT

Volumetric lenses have been studied as tools for interactive visualization in recent years. However, their potential to be used for intuitive time navigation by extending their underlying focus and context metaphor to the temporal domain has not been addressed before. This work introduces new rendering techniques and proposes "spatiotemporal lenses" as versatile visualization tools that combine spatial and temporal queries within user-defined focus regions. In order to establish the techniques required for an efficient implementation of the new tool, we state design principles and describe the integration of volumetric lens rendering into a scene graph-based visualization system. Our work introduces new techniques for single-pass rendering of composable volumetric lenses as well as a flexible framework for lens effect composition.

We evaluate the computational performance and visual quality of our rendering approaches, compare our results to previous research, and provide suggestions on future implementations of visualization systems incorporating volumetric lens rendering. Our results show that the presented algorithms can reduce the dependence on optimized scene partitioning imposed by earlier approaches. In contrast to previous research, our technique achieves consistent rendering performance for user interaction and lens intersection cases.

We present semantic and computational fundamentals of the spatiotemporal lens metaphor, suggest composition behavior for time lenses, and present application scenarios for their usage.

BIOGRAPHICAL SKETCH

Jan-Phillip Tiesel was born in Herford, Germany, on February 20, 1982, to Karin and Horst Tiesel. Following the completion of a study abroad program at Indiana University Purdue University Indianapolis (IUPUI) in 2005, he obtained the degree Bachelor of Science in Digital Media from the University of Bremen, Germany, in the year of 2006. After working as a self-employed multimedia developer for one year, he went on to pursue his Master of Science in Computer Science at the University of Louisiana at Lafayette. He was initiated as a member of the Phi Kappa Phi honor society in 2009.