Multi-Frame Rate Rendering

by Jan P. Springer

A dissertation submitted in conformity with
the requirements for the degree of Dr.-Ing.

Bauhaus-Universität Weimar
Fakultät Medien

Abstract

Multi-frame rate rendering is a parallel rendering technique that renders interactive parts of a scene on one graphics card while the rest of the scene is rendered asynchronously on a second graphics card. The resulting color and depth images of both render processes are composited, by optical superposition or digital composition, and displayed. The results of a user study confirm that multi-frame rate rendering can significantly improve the interaction performance.

Multi-frame rate rendering is naturally implemented on a graphics cluster. With the recent availability of multiple graphics cards in standalone systems the method can also be implemented on a single computer system where memory bandwidth is much higher compared to off-the-shelf networking technology. This decreases overall latency and further improves interactivity. Multi-frame rate rendering was also investigated on a single graphics processor by interleaving the rendering streams for the interactive elements and the rest of the scene. This approach enables the use of multi-frame rate rendering on low-end graphics systems such as laptops, mobile phones, and PDAs.

Advanced multi-frame rate rendering techniques reduce the limitations of the basic approach. The interactive manipulation of light sources and their parameters affects the entire scene. A multi-GPU deferred shading method is presented that splits the rendering task into a rasterization and lighting pass and assigns the passes to the appropriate image generators such that light manipulations at high frame rates become possible. A parallel volume rendering technique allows the manipulation of objects inside a translucent volume at high frame rates. This approach is useful for example in medical applications, where small probes need to be positioned inside a computed-tomography image. Due to the asynchronous nature of multi-frame rate rendering artifacts may occur during migration of objects from the slow to the fast graphics card, and vice versa. Proper state management allows to almost completely avoid these artifacts.

Multi-frame rate rendering significantly improves the interactive manipulation of objects and lighting effects. This leads to a considerable increase of the size for 3D scenes that can be manipulated compared to conventional methods.

Zusammenfassung

Multi-Frame Rate Rendering ist eine parallele Rendertechnik, die interaktive Teile einer Szene auf einer separaten Graphikkarte berechnet. Die Abbildung des Rests der Szene erfolgt asynchron auf einer anderen Graphikkarte. Die resultierenden Farb- und Tiefenbilder beider Darstellungsprozesse werden mittels optischer Überlagerung oder digitaler Komposition kombiniert und angezeigt. Die Ergebnisse einer Nutzerstudie zeigen, daß Multi-Frame Rate Rendering die Interaktion für große Szenen deutlich beschleunigt.

Multi-Frame Rate Rendering ist üblicherweise auf einem Graphikcluster zu implementieren. Mit der Verfügbarkeit mehrerer Graphikkarten für Einzelsysteme kann Multi-Frame Rate Rendering auch für diese realisiert werden. Dies ist von Vorteil, da die Speicherbandbreite um ein Vielfaches höher ist als mit üblichen Netzwerktechnologien. Dadurch verringern sich Latenzen, was zu verbesserter Interaktivität führt.

Multi-Frame Rate Rendering wurde auch auf Systemen mit einer Graphikkarte untersucht. Die Bildberechnung für den Rest der Szene muss dazu in kleine Portionen aufgeteilt werden. Die Darstellung erfolgt dann alternierend zu den interaktiven Elementen über mehrere Bilder verteilt. Dieser Ansatz erlaubt die Benutzung von Multi-Frame Rate Rendering auf einfachen Graphiksystemen wie Laptops, Mobiltelefonen and PDAs.

Fortgeschrittene Multi-Frame Rate Rendering Techniken erweitern die Anwendbarkeit des Ansatzes erheblich. Die interaktive Manipulation von Lichtquellen beeinflußt die ganze Szene. Um diese Art der Interaktion zu unterstützen, wurde eine Multi-GPU Deferred Shading Methode entwickelt. Der Darstellungsvorgang wird dazu in einen Rasterisierungs- und Beleuchtungsschritt zerlegt, die parallel auf den entsprechenden Grafikkarten erfolgen können. Dadurch kann die Beleuchtung mit hohen Bildwiederholraten unabhängig von der geometrischen Komplexität der Szene erfolgen. Außerdem wurde eine parallele Darstellungstechnik für die interaktive Manipulation von Objekten in hochaufgelösten Volumendaten entwickelt. Dadurch lassen sich zum Beispiel virtuelle Instrumente in hochqualitativ dargestellten Computertomographieaufnahmen interaktiv positionieren.

Aufgrund der inhärenten Asynchronität der beiden Darstellungsprozesse des Multi-Frame Rate Rendering Ansatzes können Artefakte während der Objektmigration zwischen den Graphikkarten auftreten. Eine intelligente Zustandsverwaltung in Kombination mit Prediktionstechniken kann diese Artefakte fast gänzlich verhindern, so dass Benutzer diese im allgemeinen nicht bemerken.

Multi-Frame Rate Rendering beschleunigt die interaktive Manipulation von Objekten und Beleuchtungseffekten deutlich. Dadurch können deutlich umfangreichere virtuelle Szenarien bearbeitet werden als mit konventionellen Methoden.

Acknowledgments

MANY PEOPLE HELPED in the process of developing this thesis. I want to say thank you to all and mention some of them here. Bernd Fröhlich introduced me to the idea of multi-frame rate rendering and provided guidance, advise, and motivation as well as exactness throughout. Dirk Reiners agreed early to co-advise, gave insights from the odd angle, and listened patiently. Andreas Simon was a rich source for shaping, evaluating, and rejecting ideas as well as for friendship. I want to thank my students Stephan Beck and Felix Weiszig. Without their help only a fraction of this work could have been investigated and realized. The VR Systems Group at Media Faculty, especially Hans-Friedrich Pabst and Christopher Lux, must be thanked for discussions and support as well as for pulling through at the finish line of several projects. Walter Mundt-Blum, Jörg Krall, and Frank Pudlowsky at NVIDIA Professional Solutions Group helped with many small and not so small problems as well as generously donating hardware. Kai Schmitt and Lutz Höltzer at SCC Bauhaus-Universität Weimar helped to keep network infrastructure running, even on weekends. Thanks also to the Virtual Environments Group at Fraunhofer IAIS, especially Thorsten Holtkämper, Sascha Scholz, and Jürgen Wind, for sharing resources and listening. Christian Lessig and Holger Regenbrecht provided valuable comments on thesis drafts. Heike Bräuer, Birgit Kohlhaas, and Martin Kohlhaas helped with various layout aspects and the digital print processing. Finally, in the sense of last but never the least, I thank my family, Antje, Martha, and Rosa, for their love, patience, and support.

This thesis contains rendered images of:

- Seismic data from the Wytch Farm oil field courtesy of BP, Premier Oil, Kerr-McGee, ONEPM, and Talisman.
- VW New Beetle model courtesy of Volkswagen AG.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Visual Quality and Interaction Fidelity . . . 2
- 1.3 Overview of Multi-Frame Rate Techniques . . . 6
- 1.4 Thesis Statement . . . 8
- 1.5 Outline of Argument . . . 8

2 Related Work **10**
- 2.1 Rendering Methods . . . 11
- 2.2 Geometric Methods . . . 13
- 2.3 Parallel and Distributed Graphics . . . 14
- 2.4 High-Level Approaches . . . 17
- 2.5 Summary . . . 18

3 Multi-Frame Rate Composition **20**
- 3.1 Optical Superposition . . . 20
- 3.2 Digital Composition . . . 23
- 3.3 Summary . . . 25

4 Multi-Frame Rate Rendering **26**
- 4.1 Scene Management . . . 26
- 4.2 Object Migration . . . 27
- 4.3 Image Generation . . . 29
- 4.4 Summary . . . 31

5 Multi-Frame Rate System Setups **32**
- 5.1 Graphics Cluster . . . 32
- 5.2 Multi-GPU Systems . . . 35
- 5.3 Single-GPU Systems . . . 38
- 5.4 Hybrid Systems . . . 40
- 5.5 Summary . . . 40

6 Artifacts **42**
- 6.1 Migration Artifacts . . . 42
- 6.2 Rendering Artifacts . . . 48
- 6.3 Summary . . . 50

7 Advanced Techniques **51**
- 7.1 Interactive Light Manipulation . . . 51

7.2 Multi-Frame Rate Volume Ray Casting 55
7.3 Summary 61

8 Analysis and Discussion 62
8.1 Buffer-Transfer Methods 62
8.2 End-to-End Latency Analysis 65
8.3 User Study 68
8.4 Limitations 72
8.5 Summary 74

9 Conclusions 76
9.1 Summary 76
9.2 Future Work 78
9.3 Closing Remarks 81

Bibliography 82

Curriculum Vitae 93

Ehrenwörtliche Erklärung 95

Chapter 1

Introduction

The interactive and high-quality visualization of large models is still a challenging problem even though the capabilities of graphics systems have been dramatically improved over the past years. Unfortunately, the expectations on *visual quality* have increased even more, which in general affects the interactivity of applications or *interaction fidelity*. These two qualities seem to be at opposite ends of a continuum. Visual quality is mainly dependent on the scene complexity (e. g., the number of primitives), the rendering method, the illumination and shading model, and the display resolution. While all of these factors might also improve interaction fidelity, they often lead to low frame rates if excellent visual quality is desired. Interaction fidelity heavily depends on the immediate incorporation of user actions into the simulation and image generation process which demands high frame rates.

1.1 Motivation

In recent years the use of high quality visualization in interactive applications has been embraced in many industries. Especially in the automotive and oil&gas industry the requirements for these applications are growing, constantly demanding better visualization quality while at the same time the size of the data sets steadily increase. Similar observations have been also made for a variety of other application areas [Owens 2007]. In particular virtual reality (VR) technologies are used increasingly to comprehend, analyze, and often also to manipulate large amounts of data from static as well as dynamic processes ultimately influencing decision-making processes. With the recent advances in graphics hardware users of interactive applications, and VR applications in particular, are demanding high-quality high-resolution images as well as highly interactive handling of their very large data sets.

Multi-frame rate rendering is motivated by a number of observations made with different application prototypes in the automotive and the oil & gas industry, where often highly complex scenes are explored and manipulated on large projection-based displays:

- Scenes are mostly static and only small parts are typically manipulated (e. g., an oil well or an engine part).

- System-control interaction is quite often used, but menus, sliders, etc. are difficult to manipulate at low frame rates.
- Head tracking is rarely used. Even if it is used head-tracked users in large projection-based displays move around very little in most cases. Head tracking seems to work quite well at low frame rates while selection, object manipulation, and system control become increasingly difficult.
- Navigation often involves only the coarse adjustment of view-point positions, which can be achieved at relatively low frame rates.

These observations indicate that it should be beneficial to speed-up object selection, object manipulation, and system control—which require higher frame rates to work reasonably well—than navigation and head tracking. This insight is either completely ignored in current systems and thus interactivity is sacrificed or it is resolved by rendering the whole scene with the desired frame rate and reduced visual quality (e. g., by using an appropriate level-of-detail and sending fewer polygons into the graphics pipeline).

The multi-frame rate approach presented in this thesis uses multiple image generators (IGs) to render the interactive parts of a scene (e. g., menus, cursor, or scene objects) which are currently manipulated by the user with the highest possible frame rates, while the rest of the scene is rendered at regular, i. e. lower, frame rates. The result is improved interaction fidelity *and* high image quality. The price which has to be paid are artifacts which may occur under certain circumstances. The outputs of individual image generators are optically or digitally combined into a multi-frame rate image. Optical combination can be achieved by using multiple projectors displaying completely overlapped images on the same screen. Digital image composition requires either dedicated hardware or the exchange of color and depth information between different image generators. The digital composition approach of asynchronously generated images can be regarded as an unconventional case of the Sort-Last technique [Molnar et al. 1994], which commonly focuses on balancing workload between multiple image generators to improve the overall frame rate. In contrast to Sort-Last, multi-frame rate rendering however purposely generates a highly unbalanced load for the image generators to considerably improve the interactivity and responsiveness of an application.

1.2 Visual Quality and Interaction Fidelity

Visual Quality A body of work has been already presented trying to objectively classify and assess visual quality [Comes and Macq 1990; Silverstein and Farrell 1996; Winkler 2001; Rademacher 2002]. Most of this work is concerned with the classification of good vs. bad photographs. Only Rademacher [2002] is concerned with computer-generated images but attempts to classify the perceived visual realism in such images. Research with respect to visual fidelity tries to assess how good known content in an image can be recognized by human observers (e. g.,

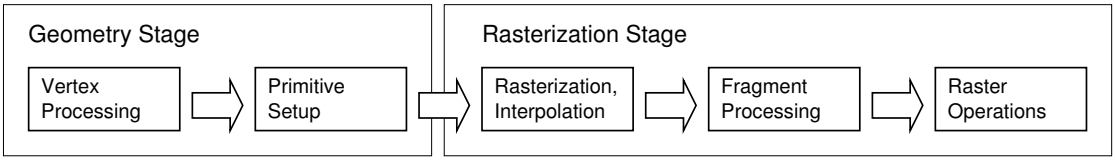


Figure 1.1: Graphics pipeline stages (simplified).

[Watson et al. 2001; Mania et al. 2006]). Visual fidelity-based approaches trade image quality for faster image updates.

The intention of an application developer might be to create visual realism, show geometric accuracy even on close-up view, or correctly simulate material and light properties. All of these factors are typically specified for an actual application scenario and contribute to visual quality. This means that the desired visual quality must be assumed as a priori defined and should not be degraded by the underlying rendering infrastructure. Generating images which match the desired visual quality might and will however often conflict with the goal of fluid user interaction. To better understand the contradicting goals, first the current raster-graphics rendering pipeline is revisited and then interaction fidelity with respect to the render process is analyzed.

Computer-generated images are usually projections of spatial data for a given view position. This means that the degree of detail of the spatial data as well as the resolution of the (discrete) image plane influence the rendering result. The common processing steps for image generation on current graphics systems are: vertex processing and primitive assembly, rasterization, fragment processing, post-fragment tests (e. g., depth, stencil), and writing color (and depth) values into the frame buffer; see figure 1.1 for a graphical depiction of the (simplified) rendering pipeline. Until the rasterization stage the data processing can be assumed to be analog, i. e. projecting primitives from world into screen space will create a discretized projected representation of a primitive. The quality of a visualization up to this stage is therefore mainly bound by the amount of data used to represent the problem domain, i. e. the model or scene. Models are usually represented by triangle sets. This means that smooth surfaces approximated with an insufficient number of elements may not only create visually unpleasing results but also potentially convey incorrect information about shape or curvature. In principle a very high amount of triangles might be necessary to approximate the desired shape. However, using large amounts of triangles may represent the shape with the desired detail but increases the processing time for these triangles and therefore also increases the time needed for creating an image frame.

After rasterization the image-generation performance depends on the number of fragments that must be processed and the work per fragment. That is the resolution of the output display and the rendering order of primitives. Too many display pixels increase frame time while not enough pixels decrease the image fidelity. Even if an optimum resolution is found changes in the light and shading models used will greatly influence the performance-to-quality ratio. The rendering order of

the primitives influences the processing of the resulting fragments. Rendering in a front-to-back order usually avoids the processing of fragments already covered by primitives nearer to the viewer. For some visual effects such as transparency the reverse primitive rendering order, i. e. back-to-front, may be necessary which degrades performance by excessive fragment processing for scenes with high depth complexity. Even imposing no primitive order at all, i. e. rendering primitives in random order, will degrade performance because of unnecessary depth comparisons and fragment processing. The decision about the primitive ordering in general cannot be made in the graphics hardware because application developers must be able to determine the desired behavior with respect to specific algorithmic requirements.

In addition to the number of fragments that must be processed the computations per fragment must also be considered. While the standard OpenGL rendering pipeline, for example, defines lighting and shading at the vertex processing stage the resulting visual quality depends on the size of the projected primitives. Computing lighting and shading at the fragment level avoids this dependency on the size of the primitive projection thereby increasing the computational effort per fragment. The trade-off is to either use many triangles and define all computations per primitive or to use lower-resolution models and compute lighting and shading per fragment. The latter approach is currently preferred because it is output sensitive.

Visual quality, as mentioned before, in the context of this thesis is assumed to be pre-defined by a particular application (software). However, if hardware-accelerated image generation is deployed, the necessary computational effort mainly depends on model complexity, image resolution and computational effort per fragment. Increasing one or all of these variables usually also increases visual quality, but also the time needed for finishing an image frame.

Interaction Fidelity The term interaction fidelity is used to describe how well users perceive their interactive input being incorporated into the image generation process. In contrast to high-quality imagery shown in movies, graphical applications used in visualization and simulation contexts are expected to be interactive. Users expect to be able to manipulate the elements of the 3D world and perceive these changes immediately on the display.

Incorporating interaction responses into the internal representation on a computer is in general computationally much less expensive than generating high-quality images. For interaction it is more important to minimize the time from real-world sensor acquisition to incorporating the resulting changes into the image. Conventional single-frame rate systems exhibit a tightly coupled visualization and interaction architecture where sensor data is evaluated as part of the visualization loop. In such systems interaction response is limited by the time it takes to render an image. Earlier research found a dependency between the frame rate of an application and the corresponding interaction fidelity. For sufficient interaction fidelity it is recommended that the frame rate should be at least 6 Hz [Airey et al. 1990], more than 7 Hz [Pausch 1991], or more than 10 Hz [Card et al. 1991;

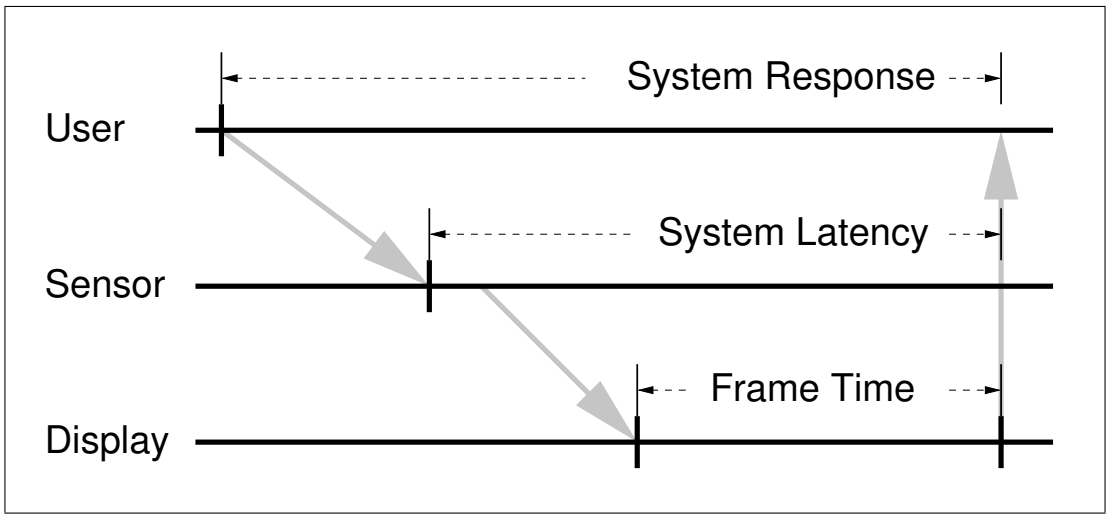


Figure 1.2: Relationship of system response, system latency, and frame time (after Watson et al. [1998]).

McKenna and Zeltzer 1992; Bryson 1993]. These values are mostly rules of thumb derived from experience or extrapolated from 2D GUI experiments.

More formal studies investigated the interaction-fidelity to frame-rate relationship for task performance in VR applications [Tharp et al. 1992; Ware and Balakrishnan 1994; Reddy 1997; Watson et al. 1998]. They all arrive at the conclusion that sufficient interaction fidelity in a single-frame rate system requires a frame rate above 10 Hz. Tharp et al. [1992] and Reddy [1997] found that for frame rates above 20 Hz and 15 Hz, respectively, no dramatic performance improvement can be seen, though both agree that the higher the frame rate the better the task performance. Watson et al. [1998] focused on studying system response for open-loop and closed-loop tasks [Wickens 1992]. They found that open-loop tasks (e. g., grasping) seem to be less sensitive to system responsiveness than closed-loop tasks (e. g., placement) because the latter require continuous visual feedback for appropriate control. System response is described here as the time elapsed from a user's action to the display of that action. In contrast frame time is the time an image is presented on the display while system latency describes the age of a measurement sample from the tracking system presented in the image. Frame time and system latency are the two main influences on system response. The relationship between these entities is depicted in figure 1.2.

The age of a sample displayed in an image consists mainly of the time it takes to process the data from a sensor and sending it to the application. This sensor data latency is typically constant for a certain setup (e. g., a tracking system). Dedicated hardware is usually used for sensor-data processing which allows for constant update rates. Once the sensor data arrives at the application it is incorporated into the application state (e. g., matrices of a scene graph). Based on the modified application state a new frame is generated. Rendering time is typically equal or similar to the time an image is displayed—the frame time. In low frame-rate

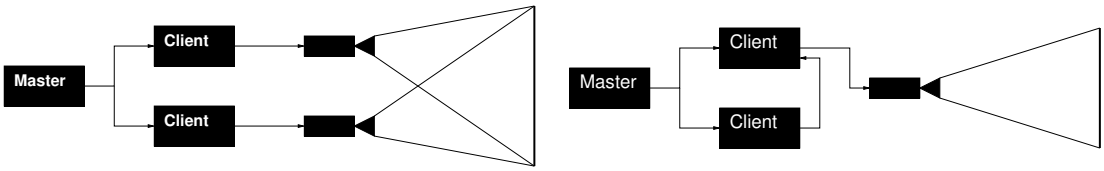
applications system response and system latency are thus dominated by frame time. System latency affects how well the response on the display is connected to the action of the user. The frame time, as the dominating part of the system latency, also contributes to this perception of action and reaction, but additionally affects the smoothness of object movements and view changes.

Conclusion From the above discussion it can be concluded that improving the frame time for the interactive parts of a scene will result in significantly improved interaction fidelity. Closed-loop tasks such as object placement are particularly affected by overall system responsiveness. Because grasping and placement are common manipulations in VR applications it seems beneficial to part from the single-frame rate approach for image generation. In a single-frame rate system interactive frame rates can only be guaranteed by controlling parameters affecting visual quality. The goal of multi-frame rate rendering is to achieve frame rates sufficient for interaction in a 3D world while also supporting high-quality visualization methods (possibly) exhibiting very low frame rates. For this approach two requirements must be fulfilled. First, it must be possible to dynamically designate parts of the scene's content as relevant to the current interaction. This might be inferred automatically (e. g., by interest prediction) or explicitly by state changes with respect to the interaction framework used (e. g., objects must be selected before being manipulated). Second, scene parts relevant for the interaction must be rendered on a dedicated image-generation resource which is sufficient to guarantee high update rates. By assuming that usually only small parts of a scene are relevant to the current interaction this should allow for sufficiently high frame rates for interaction responses to the user, i. e. minimizing the time for system response. The rest of the scene is also rendered on a separate resource, typically with much lower frame rates. Both of these graphics sub-systems are assumed to run asynchronously, i. e. they are not expected to be synchronized in any way. Finally, the partial images must be combined in a consistent way for displaying the complete scene to the user.

1.3 Overview of Multi-Frame Rate Techniques

This thesis proposes techniques which combine the output of several image generators in an asynchronous way. Each image generator runs at its own frame rate dependent on the workload, which is typically quite unbalanced with this approach. The combination of the output of multiple image generators leads to a multi-frame rate display or multi-frame rate images. These images can be displayed by either optically superimposing them using multiple projectors or digital merging in a final composition stage. The underlying technique for assigning data to the respective image generators and the image combination is called multi-frame rate rendering.

The superposition of images from multiple projectors requires precise geometric and optical calibration as discussed by Raskar [2002] and Majumder [2003]. The superimposed projectors create an *optical buffer* as shown in figure 1.3a. The alternative to optical superposition of multiple images is digital composition. This



(a) Optical superposition. **(b)** Digital composition.

Figure 1.3: Display setups for multi-frame rate rendering methods.
(a) Optical superposition of two projectors. (b) Digital composition of color and depth buffer from two render nodes using one projector.

can be achieved by using dedicated hardware such as the Lightning-2 system [Stoll et al. 2001] or the Sepia-2 system [Lombeyda et al. 2001]. These systems are not widely available and the exchange of color and depth buffers between image generators is often used. The output of asynchronously running image generators is combined in a *digital buffer* (cf. figure 1.3b), which is a variation of the Sort-Last parallel rendering technique [Molnar et al. 1994]. The image generators or the associated render clients are designated *slow client* (SC) for the non-interactive, static scene parts and *fast client* (FC) as the render client for the interactive scene elements. In addition to these two roles it is conceivable to use other image generators in a multi-frame rate setup with their own dedicated roles (e. g., for animated objects or scene-global effects such as shadows). However, this thesis will focus on the use of only two image generators—the slow and fast client.

The visual result of digitally combined or optically superimposed outputs from multiple asynchronously running image generators is called a *multi-frame rate display*. An individual image at a certain instant in time is called a *multi-frame rate image*. It is assumed that a scene is represented as a scene graph [Clark 1976]. Parts of the scene graph are distributed to different clients and the resulting outputs are digitally or optically combined. The sum of the parts of the scene graph results in the complete scene graph and the combination of the outputs from the different image generators results in the complete image. Figure 1.4 shows this process schematically. The partial scene graphs are rendered on their respective image generators and the combination of their outputs results in an image of the complete scene with the different parts potentially rendered at different instants in time which may create artifacts in the resulting images. However, for mostly static scenes where only a small subset of the objects is manipulated artifacts can be almost always avoided by appropriate state management and prediction.

User interaction with objects in the scene results in *object migration* from the slow to the fast client upon object selection and back from the fast to the slow client upon object release. View-position updates are incorporated at the speed of the slow client to avoid inconsistent image displays between the slow and fast clients. Without using any additional process refinement this is called *naïve multi-frame rate rendering*.

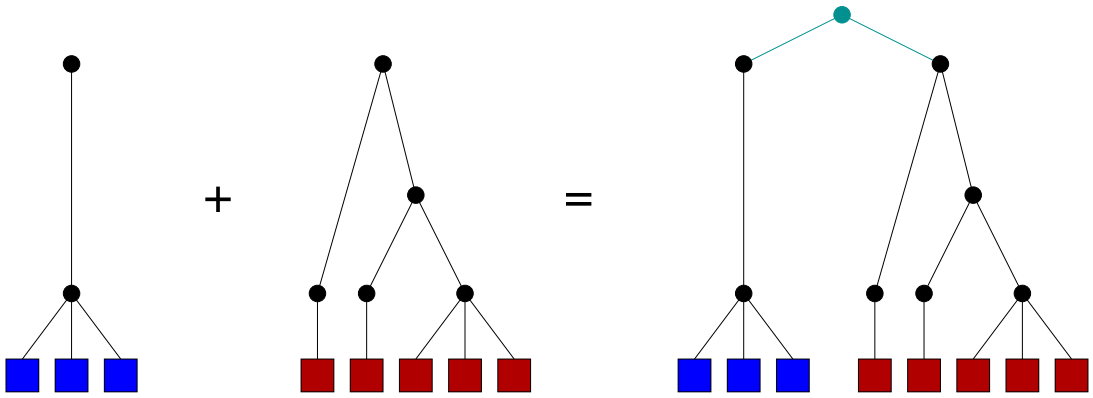


Figure 1.4: Scene superposition; the partial scene graphs (on the left and in the center) form the final scene graph (on the right) for the interacting user if rendered on a multi-frame rate display.

1.4 Thesis Statement

This thesis presents multi-frame rate rendering, a method to improve interaction fidelity while preserving desired visual quality in complex virtual environment applications. The thesis of this research is:

> *Parallel rendering is commonly used to improve graphics performance for very large models. It is aimed at an even distribution of the workload across a number of graphics nodes, which scales at most linearly with the number of involved resources. In contrast, multi-frame rate rendering is a parallel rendering technique that purposely splits the workload unevenly to improve interaction fidelity. Interactive parts of a scene are updated at the highest possible frame rates on one image generator, while the rest of the scene is rendered at lower frame rates on a separate image generator. The results of these asynchronous image generation processes are digitally or optically combined and displayed. Multi-frame rate rendering is effective in (virtual environment) applications where typically only a small subset of the objects in the scene are actively manipulated. Artifacts occurring during object migration between the asynchronous render resources can be almost always avoided by object-state management and prediction. The method does not introduce further degradation of visual quality and does not affect navigation fidelity. Multi-frame rate rendering is orthogonal to most other performance improvements and can be combined with these.*

1.5 Outline of Argument

Multi-frame rate rendering consists of several interdependent topics. In order to provide a clear argument the description of these topics has been separated into:

- **Multi-Frame Rate Composition**
  explains the composition of partial images generated by multi-frame rate rendering.
- **Multi-Frame Rate Rendering**
  describes the image generation with respect to (existing) 3D graphics APIs.
- **Multi-Frame Rate System Setups**
  analyzes system configurations suitable for multi-frame rate rendering.

Each of these sub-topics is self-contained in the sense that it applies as a whole to the other sub-topics. The ordering can be seen as the reverse of the actual image generation and composition process, i. e. a multi-frame rate display shows images generated by multi-frame rate rendering running on a multi-frame rate system setup.

The remainder of this thesis is organized as follows. In chapter 2 related and considered work is reviewed and discussed with respect to this thesis. Chapter 3 describes multi-frame rate image display by optical superposition and digital composition. Chapter 4 explains how multi-frame rate images are generated using standard 3D graphics APIs. Chapter 5 details system setups and hardware configurations for multi-frame rate rendering, while chapter 6 analyzes which artifacts in multi-frame rate images occur and explains how they can be resolved. Chapter 7 presents advanced rendering techniques for interactive light manipulation and object manipulation in translucent volume data sets. Chapter 8 analyzes buffer transfer and end-to-end latency in a multi-frame rate system. It also discusses a user study and limitations of multi-frame rate rendering. Finally, chapter 9 concludes this thesis by summarizing results, discussing conclusions, and presenting directions for future research.

Chapter 2

Related Work

Scene graphs were introduced by Clark [1976]. He motivates the use of structure in the scene representation by potentially enhanced image realism or improved performance of image synthesis algorithms. Three approaches are discussed by Clark [1976]. First, adding *information value* to the scene without significantly increasing the total amount of information in the database (e. g., [Gouraud 1971; Phong 1975]). Second, refined mathematical model descriptions to model smooth surfaces with surface patches (e. g., [Catmull 1974]). The third approach increases the amount of information in the database and employs more structured methods for handling this increased amount of information (e. g., [Newell 1975]). Clark [1976] concludes that the structured method seems to be most promising because potentially both picture quality and algorithmic performance will improve. However, there are problems with this approach. Increased scene complexity has less value if screen resolution limits are approached (e. g., using 1000 polygons for an object projected to only 20 pixels makes little to no sense). Actual implementations will have to cope with the increased memory footprint which leads to the question of how much information must be presented to appropriately convey the information content of the scene. Eventually, Clark [1976] proposes a *hierarchical approach* which describes the structure and working of what is now called a *scene graph*. Even at the time [Clark 1976] was written *structure* was already used in various ways: for defining relative placement of objects, which was also supported (to some extent) by hardware (e. g., then current systems by Evans & Sutherland), and for decreasing clipping time. Clark [1976] proposes new uses of structure:

- varying environment detail, i. e. level-of-detail (LOD) selection;
- clipping as truncated logarithmic search, i. e. view-frustum culling;
- graphical working set, i. e. in-core vs. out-of-core rendering;
- improving existing algorithms by combining LOD and view-frustum culling;
- recursive descent, visible surface algorithm, i. e. scene-graph traversal;
- and building structured databases.

Many of these techniques are in general use today. They have been refined and brought in correlation to other approaches to enhance visual quality as well as user interaction by allowing for rapid image generation. On the other hand, improved interactivity with computer generated images is still often achieved by degrading

image quality; a continuum already explained in section 1.2. The technique(s) described in this thesis provide a different approach to combine high-quality images with appropriate interaction fidelity.

Multi-frame rate rendering is related to and builds upon other work. Related work has been classified into methods providing interactive frame rates by adjusting the rendering process (section 2.1) or by simplifying the scene geometry (section 2.2). Parallel sorting as a view on graphics is related to multi-frame rate rendering using digital composition and discussed in section 2.3. Finally, several high-level approaches from the VR community are analyzed in section 2.4. The chapter concludes with a discussion of the relationships between the multi-frame rate method and the classification of related work in section 2.5.

2.1 Rendering Methods

Bergman et al. [1986] present a mechanism for adaptively refining the image presentation to the viewer. They propose to initially show the vertices only, followed by adding the edges between vertices, which in turn is followed by flat shading, shadow generation, Gouraud shading [Gouraud 1971], Phong shading [Phong 1975], and finally anti-aliasing. Bergman et al. [1986] reason that as long as the viewer does not change parameters this successive refinement provides a good combination of rendering speed, user convenience, and image quality. They also suggest the possibility of a *golden thread*, a single step that, repeated a few times, will generate a coarse image, and, when repeated further, will result in incrementally higher quality images.

Bishop et al. [1994] propose a *frameless rendering* technique that allows smooth updates of an image from a scene. Instead of using a double-buffered approach, where the new image is being generated while the previous one is shown on the display, they propose pixel computation based on the most recent user input and immediately updating that pixel on the display. The resulting images would converge to a final high-quality display when user input stops. During input changes the image display may become blurry because intermediate images will contain pixels from different temporal samplings. Watson et al. [2002] follow up on this work by introducing a visual error metric, consisting of a spatial and a temporal error, to control the image refinement. The techniques proposed by Bishop et al. [1994] and Watson et al. [2002] are however limited to ray-tracing systems. It is also worth noting that no display hardware currently exists, which is capable of efficiently updating single pixels. This makes frameless rendering at the moment a conceptual rather than a practical method [Ferwerda 2003].

Wloka et al. [1995] present a practical approach to frameless rendering. The frame buffer is split into four regions so that pixels of the top-left region are mapped from $(x, y) \rightarrow (2x, 2y)$. Similar mapping is used for the the remaining three quadrants. In each render step only a quarter-resolution image is rendered. The render process cycles through the quadrants incrementally but all quadrants will be displayed on the screen (cf. figure 2.1). Effectively, static objects, i. e.

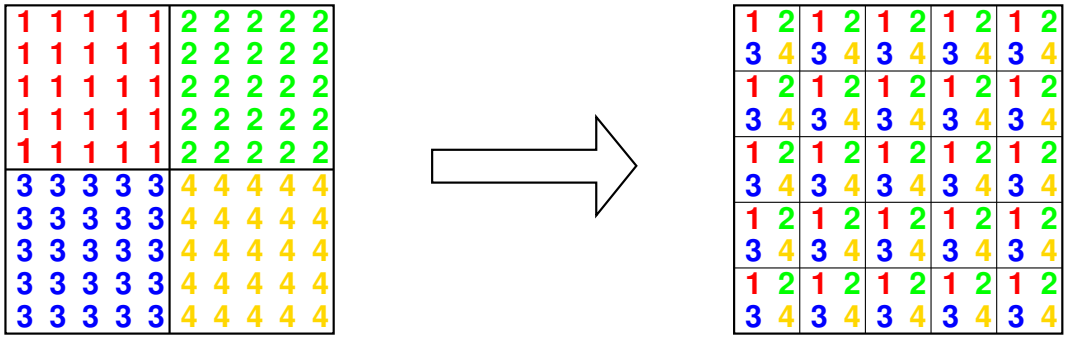


Figure 2.1: Frame-buffer split of the practical frameless rendering approach by [Wloka et al. 1995].

objects that do not change their transformation, will be displayed solid after four iterations while dynamic objects will generate pixels interspersed with background pixels leading to motion blur. To avoid coarse images, by displaying 2×2 pixels of (possibly) the same value, the camera is panned according to the quadrant currently rendered so that each pixel is constructed by 2×2 sub-pixels from the respective quadrants. Wloka et al. [1995] argue that this method is up to four times faster than rendering the full-resolution image thus improving interactivity while possibly degrading visual quality. They assume a pixel-bound rendering mechanism, i. e. performance is limited by the scan conversion or rasterization, which may not be valid for current graphics architectures.

Woolley et al. [2003] introduce the concept of *interruptible rendering*. A single image-space error measure is employed to unify the spatial error caused by rendering coarse representations and the temporal error caused by latency. A progressively refined rendering of a coarse image into the back buffer of a double-buffered frame-buffer setup is used. During this process the temporal error is monitored and once it exceeds the spatial error, further refinement is stopped and the image is displayed. Their rendering system uses LOD-based techniques combined with real-time ray tracing.

Scher Zagier [1997] proposes to incorporate the limitations of the human visual system into the rendering process to improve interaction fidelity. Together with camera input, a personalized display could be created and combined with a refined frameless rendering approach. Dumont et al. [2003] present a perceptually-driven decision theory for interactive rendering and demonstrate the practicality of the approach for various applications such as diffuse-texture caching, environment-map prioritization, and mesh simplification in radiosity systems.

Bastos [1999] considers a signal-theoretic approach to light transport and interprets the light-transport equation [Kajiya 1986] as a convolution operator. It is also shown that the light-transport problem can be linearly decomposed into simpler problems with simpler solutions. These solutions are then recombined to approximate the full solution. The central goal is to provide interactive photo-realistic rendering for walkthroughs of virtual environments.

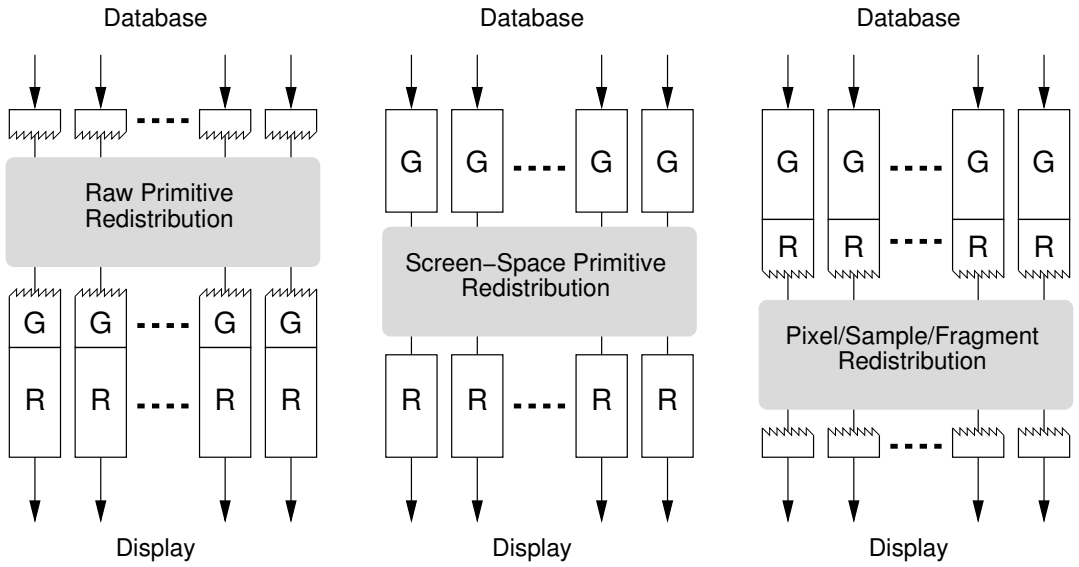
2.2 Geometric Methods

As noted before, highly detailed models do not add much to the visual quality of an image if they project only to a small number of pixels. Clark [1976] introduced the notion of varying environment detail, i. e. depending on the screen contribution of an object a simplified version of the object may be used to increase frame rates. While the generation of simplified object representations was the domain of modeling artists during the 1980s, in the 1990s interest grew in the graphics community to automate this process. First, algorithms appeared for vertex decimation [Schroeder et al. 1992] and gridded vertex clustering [Rossignac and Borrel 1993]. This was followed by optimization-based predictive scheduling for level-of-detail (LOD) selection [Funkhouser and Séquin 1993], progressive meshes for continuous LOD [Hoppe 1996], vertex hierarchies for view-dependent LOD [Hoppe 1997; Luebke and Erikson 1997], quadratic error metrics for measuring simplification error [Garland and Heckbert 1997], principled simplification of topology [He et al. 1996; El-Sana and Varshney 1999], and wavelets for subdivision [Lounsbery et al. 1997].

Discrete LOD uses multiple representations of the same object, created in an offline pre-processing step, and selects the appropriate version at run time, usually depending on the distance to the viewer. Because the offline generation cannot predict viewing directions at run time, objects are uniformly simplified. Texture data may also be employed for LOD. Mipmapping [Williams 1983] can be used to generate lower resolution versions from texture data and is available today in any 3D graphics hardware. Dumont et al. [2001], for example, describe a perceptually-based application of mipmapped texture for selecting LOD.

Hoppe [1996] presents progressive meshes, a method for storing and transmitting arbitrary triangle meshes that allow for smooth geomorphing of LOD approximations, progressive transmission, mesh compression, and selective refinement. Progressive meshes provide a continuous-resolution representation of a triangle mesh; in this respect they are an example for continuous LOD. Hoppe [1997] refines the progressive mesh approach by introducing view-dependent refinement. By exploiting view coherence frame-rate regulation is supported and, for continuous motion, the refinement can be amortized over consecutive frames. This is an example of view-dependent LOD.

In case the size of 3D models exceed the size of physical memory or even the maximum addressable memory in a computer system out-of-core simplification techniques are available to retain only the portion of the data in memory which is currently needed. This is equivalent to the graphical working set idea by Clark [1976]. Usually, the model is segmented into blocks, or tiles, which are then paged into main memory as needed. Lindstrom [2000] showed how to implement the clustering algorithm by Rossignac and Borrel [1993] using external memory, i. e. disk space. By storing the output mesh and intermediate data on disk, where out-of-core sorting is used to detect and compose the primitives associated to each grid cell, Lindstrom and Silva [2001] also removed the requirement for maintaining intermediate simplification results in main memory at the expense of slower simplification performance. Hoppe [1998] extends the view-dependent



(a) Sort-First **(b)** Sort-Middle **(c)** Sort-Last

Figure 2.2: Parallel rendering classification by primitive sorting (after [Molnar et al. 1994]). Geometry processors are denoted by **G** while **R** indicates rasterization processors.

progressive mesh framework for terrain rendering by decomposing the terrain data into a block hierarchy and simplifying each block independently. This out-of-core simplification for terrain rendering can be further extended to handle arbitrary polygonal models [Prince 2000].

A comprehensive and detailed discussion of LOD techniques can be found in the book by Luebke et al. [2002].

2.3 Parallel and Distributed Graphics

Molnar et al. [1994] introduce a classification scheme for reasoning about parallel rendering. It is based on where the graphical primitives are distributed to a particular screen, frame buffer, or image generator. This leads to the observation that rendering can be viewed as a problem of sorting primitives to a given screen, which was first noted by Sutherland et al. [1974]. This sort may happen anywhere in the rendering pipeline: during geometry processing (Sort-First), between geometry processing and rasterization (Sort-Middle), or during rasterization (Sort-Last); see figure 2.2 for a schematic presentation. Sort-First redistributes primitives before transformation into screen space. Each primitive's bounding volume is projected into screen space to determine on which processor the primitive is to be processed (cf. figure 2.2a). Sort-Middle means redistributing screen-space transformed primitives. Initially primitives are assigned arbitrarily to geometry processors. After processing the primitives are classified with respect to the screen region they belong to and (re-)distributed to the respective rasterization processor(s) (cf. figure 2.2b). Sort-Last is the redistribution of pixels, samples, or fragments.

Here each rendering node processes a subset of primitives until after rasterization and distributes the image data (cf. figure 2.2c). This distribution may contain only the actual screen rectangle where updates occurred (Sort-Last $_{\text{sparse}}$) or the complete image (Sort-Last $_{\text{full}}$). In a subsequent composition step the image data from all processors is accumulated resolving final visibility. The classification scheme allows for computational and communication costs to be analyzed; cf. [Eldridge 2001] for a comprehensive discussion.

PixelFlow [Molnar et al. 1992; Eyles et al. 1997] is an image-composition architecture based on custom chip designs. It is a Sort-Last architecture with respect to the parallel sorting classification by Molnar et al. [1994]. PixelFlow consists of a set of nodes where each node is basically a complete graphics computer. Nodes are identical but some may have additional video input or output capability, provided by add-on cards, allowing to act as frame grabbers or frame buffers. PixelFlow nodes are connected by a linear network providing dedicated pixel-level communication and built-in depth-buffer compositing. General purpose communication between nodes is provided by a message-passing network. Each PixelFlow node consists of two types of computational resources: a 128×128 SIMD array of pixel processors and a pair of general-purpose RISC processors. The pixel processors perform rasterization and shading calculations, while the general-purpose processors generate instructions for the SIMD array. These instructions are stored in board memory and fetched by an instruction sequencer that controls the array.

Lastra et al. [1995] present work on real-time programmable shading based on the PixelFlow architecture. They exploit the programmability of the general-purpose RISC processors of the PixelFlow architecture as well as the SIMD structure of the rasterization array by transforming software descriptions from high-level APIs.

Lightning-2 [Stoll et al. 2001] is a display sub-system based on image composition hardware for commodity graphics cluster. Lightning-2 implements the Sort-Last parallel sorting method [Molnar et al. 1994]. A Lightning-2 board provides DVI input from up to four graphics cards, four DVI repeater used for unmodified output of one DVI input each, and eight DVI outputs. Additionally, communication channels for frame-transfer control as well as a board configuration and programming port exist. Each input unit, a slice on the board, captures video from a single graphics device, interprets control information embedded in the video signal, and writes pixel data to its frame buffer's memory controller. The input unit repeats its DVI input to drive subsequent Lightning-2 columns. The compositing unit reads the frame buffer of the slice, compositing it with the incoming frame-buffer information from the previous slice and passes the result to the next slice. The double-buffered frame buffer allows for one frame to be stored while another is composited together with frames captured by other inputs. A Lightning-2 board is essentially a full crossbar router allowing for simple screen tiling as well as depth composition. While the number of inputs and outputs is arbitrarily scalable the cost of this architecture is proportional to the product of the desired inputs and outputs.

Sepia [Moll et al. 1999] is a Sort-Last architecture for compositing 2D and 3D images. It consists of commercial, off-the-shelf (COTS) render nodes such as standard PC systems. In addition to a graphics device each node also contains a

compositing device. The compositing device is able to merge frame-buffer data from its local graphics device as well as from the network interface and allows writing of composited data back to the network interface. Interestingly, to relieve the host system from processing high-bandwidth data via its memory and network interface the compositing device controls its own network chip set connected to a dedicated communication network.

Besides the parallel graphics platforms mentioned so far new designs are constantly evolving (e. g., [Deering and Naegle 2002; Yang et al. 2002; Ogata et al. 2003; Bethel et al. 2003]), which shows a deep interest for this topic in the research community. Still, parallel graphics hardware is not yet a consumer product. PixelFlow and Lightning-2 are research platforms generally available only to the groups who developed them. Sepia-2 is part of a system that can be bought commercially but for a price that is too high for the consumer market. Sepia-2 also includes APIs for setup and control of a clustered graphics system in terms of the OpenGL API [Segal and Akeley 2006], while PixelFlow and Lightning-2 leave this problem to the actual user. Lastra et al. [1995] hint that for PixelFlow a software API is available at the development site. Lightning-2 is probably best driven using WireGL [Humphreys et al. 2001] or Chromium [Humphreys et al. 2002], developed by the same group as the Lightning-2 platform.

Recently consumer hardware has been released embodying some of the principles researched by the aforementioned hardware architectures. A flexible pipeline architecture is used that allows developers to specify execution kernels for the vertex and fragment processing stages. On the hardware these stages are implemented as processor arrays, i. e. an SIMD architecture. This is similar to the rasterization processor array of the PixelFlow architecture though the number of processor is still smaller in current consumer products. APIs for controlling these features have been standardized for the OpenGL API in the form of the OpenGL shading language specification [Rost 2004; Kessenich et al. 2006]. Also, hardware vendors for consumer graphics are trying to exploit the power of combining several graphics devices (e. g., NVIDIA SLI [NVIDIA GPU Programming Guide 2006], ATI CrossFire [Persson 2005]; a more detailed analysis for both will be given in section 5.2). Basically, a secondary card is added to the primary graphics device which allows for Sort-First setups. Scalability here is naturally limited to the maximum number of graphics devices that can be installed in the host system.

Software realizations for the various parallel sorting classes by Molnar et al. [1994] exist for distributed scene-graph environments. OpenSG [Reiners 2002] provides implementations for Sort-First and Sort-Last methods [Roth 2005]. Compression methods have been investigated to reduce the bandwidth requirements for network transmission, especially when using the Sort-Last$_{\text{full}}$ method. Unfortunately, compression algorithms do not reduce bandwidth sufficiently to gain a performance advantage with available off-the-shelf network technology (e. g., Gigabit Ethernet) [Roth and Reiners 2006].

Recently, Sort-First parallel rendering has been combined with out-of-core techniques to handle large models at interactive frame rates for high-resolution images using off-the-shelf PCs with a small amount of memory per node [Corrêa et al. 2002].

Unfortunately, Corrêa et al. [2002] measured frame rates only for navigation within a static scene, i. e. no object manipulation was available, leaving the applicability and performance for interactive object manipulation open to speculation.

2.4 High-Level Approaches

Multi-frame rate display using optical superposition (cf. section 3.1) was inspired by Majumder and Welch [2001]. They suggest the use of completely overlapped projections from multiple projectors for creating interactive depth of field effects by optical blurring, for greater parallelism and flexibility in rendering, and for generating higher-fidelity imagery. Also, separating lighting calculations or multi-pass rendering across the projectors is suggested in reference to [Bastos 1999]. Multi-frame rate rendering also makes use of the greater flexibility when using multiple overlapping projectors, but primarily for improving user interactivity. Majumder and Welch [2001] call their technique *Computer Graphics Optique* in allusion (or deference) to Émile Reynaud's *Théâtre Optique* developed in 1889 [Auzel 1998].

Durbin et al. [1995] propose a dynamic mechanism to optimize (parts of) the scene graph at run time. Their technique consists of streamlining the computations on each node by creating an array of graphics commands, peephole optimization on the streamlined list of graphics commands, removing redundancies, flattening the tree structure of the scene graph such as that at any transformed level each node consists of a streamlined array of all its children, and peephole optimization on the streamlined array for the entire sub-tree. Several timings are measured for deciding if some sub-tree needs to be optimized again: average unaltered time, i. e. the ratio of frames an object is unaltered to the total frame count, (AUT), the time to render unoptimized (TTRU), the time to render optimized (TTRO), and the time needed for the optimization step (OT). The cost function used is $\mathrm{TTRO} \cdot \mathrm{AUT} + \mathrm{OT} < \mathrm{TTRU} \cdot \mathrm{AUT}$. If this function evaluates to true optimization will happen. The caches are invalidated per object when a mutable operation occurs (e. g., changing attributes). Effectively, only incremental array rebuild happens over time in bottom-up direction. Durbin et al. [1995] report an average speedup of 2.5 compared to running unoptimized applications. They speculate about averaging differently in the sense that AUT could be weighted over time so that recent alterations have more influence. They also propose the use of explicit application hints or control on when, where, and how (long) to optimize.

Funkhouser and Séquin [1993] describe an adaptive display algorithm providing interactive frame rates for the visualization of complex virtual environments. A scene-graph representation of the scene is used in which objects are described at multiple levels of detail and various rendering algorithms can be used for image generation. The algorithm adjusts image quality adaptively to maintain a uniform user-specified target frame rate. This is achieved by a constrained optimization to choose a level of detail and rendering algorithm for each (potentially) visible object in order to generate the *best* image possible within the target frame

time. Similar methods have been included in scene-graph implementations such as OpenGL|Performer [Rohlf and Helman 1994; Rose Clay 1996] for general use.

Godin et al. [2004b] propose a method for correctly rendering images in foveated stereoscopic displays. In a foveated display a high-resolution inset is used to display parts of the scene magnified or at a higher resolution. The inset is produced by a projector that lights part of the display with a smaller extent and another projector for the lower-resolution peripheral part of the display surrounding the inset. Perceived depth of the underlying scene may conflict with the stereoscopic cue in each eye at the visible boundary between fovea and periphery. Godin et al. [2004a] provided a solution to this problem by displacing the boundary in the images to ensure that it is always positioned over stereoscopically corresponding scene locations. Godin et al. [2004b] address the same problem by relaxing the stereo matching criteria and reformulating the problem as one of spatial partitioning. Interestingly, the necessary computations are performed locally on each node, i. e. without inter-node communication, and require only a small and fixed amount of post-rendering processing.

2.5 Summary

Multi-frame rate rendering builds upon existing ideas and develops them further. Especially frameless rendering [Bishop et al. 1994] and its successive refinements [Watson et al. 2002; Woolley et al. 2003; Dayal et al. 2005] explicitly account for user interaction to be incorporated in the image generation process. In contrast to multi-frame rate rendering these techniques allow degradation of image quality to achieve this goal. Similar problems exist for LOD-based techniques where simplified geometrical representations are used. These simplified object versions must be generated either in a pre-processing step and used according to some metric selecting the correct one at run time or they must be generated at run time. The trade-off here is such that a priori generated model simplification may not convey enough detail to the application user or the run-time generation may incur a penalty on the image update rate by competing for computing resources. In other words, the simplification is either not perfect, harming visual quality, or degrades interaction quality by reducing the frame rate.

The approaches for parallel and distributed graphics focus mostly on advanced load balancing strategies to improve the overall frame rate. This is in contrast to the multi-frame rate approach, because here the goal is only to improve the frame rate for the highly interactive parts of a scene. Parallel graphics hardware is (currently) coupled with a cost factor that inhibits widespread use. Additionally, such graphics hardware is usually build upon custom chip sets and has been shown to be slower in exploiting newly available technology. Distributed graphics solutions are usually software solutions based on standard components. Here, network bandwidth is the limiting factor. While this can be remedied by advanced load-balancing mechanisms lag is introduced in the final image composition making the frame rate as fast as the slowest client plus overhead for network transfer.

Multi-frame rate rendering also provides a trade-off between image quality and interaction performance. But by emphasizing interactive content dedicated to a separate resource the image quality remains untouched. Optical superposition is an easy way to achieve this, except for the effort of geometric and color calibration of several projectors, when system control elements or similar functionality is needed and correct depth occlusion can be neglected (cf. section 3.1). Digital composition allows for fast and accurate object interaction. Both methods exhibit object migration artifacts during interaction initialization and finalization (e. g., object selection and release). This is noticeable by the user but can be almost completely avoided by proper state management and prediction (see chapter 6).

Chapter 3

Multi-Frame Rate Composition

Multi-frame rate rendering consists of asynchronous rendering of multiple images and their combined display. In this chapter it will be explained how these multiple images can be displayed using either optical superposition or digital composition.

3.1 Optical Superposition

Multi-frame rate display by optical superposition uses the additive nature of light to combine the output of two projectors (cf. figure 3.1). The projection area of both projectors is assumed to be completely overlapped. This requires precise geometric as well as optical calibration as discussed by Raskar [2002] and Majumder [2003]. Each of the projectors is connected to a separate graphics card and no synchronization between the graphics boards is required. One of the graphics devices is designated as the fast client, responsible for rendering the interactive scene parts, and the other as the slow client, responsible for rendering the static rest of the scene. The outputs of both projectors create an optical output buffer merging the partial images from the slow and fast client. Objects selected for interaction migrate from the slow client to the fast client for the duration of the interaction. When the interaction stops these objects are migrated back to the slow client. The actual migration can be accomplished in various ways and is discussed in detail in section 4.2.

Figure 3.2 shows digital photographs taken from a multi-frame rate display using optical superposition for image composition. Figure 3.2c shows the final

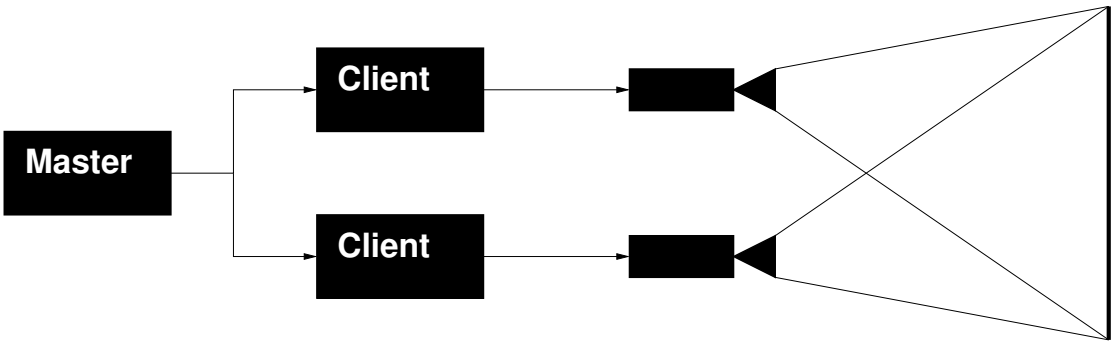
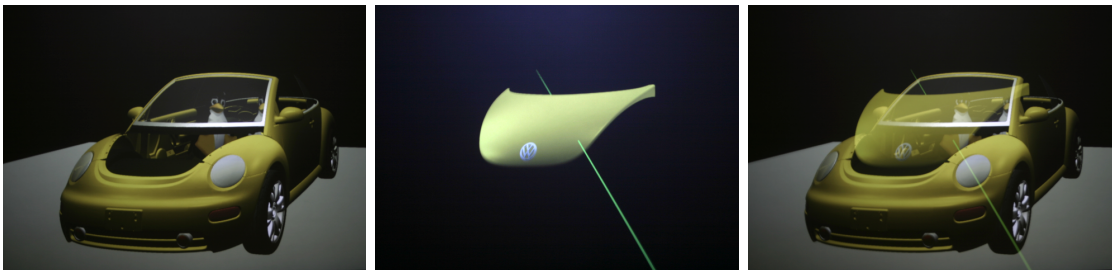


Figure 3.1: Optical superposition of two projectors creating an optical (output) buffer.



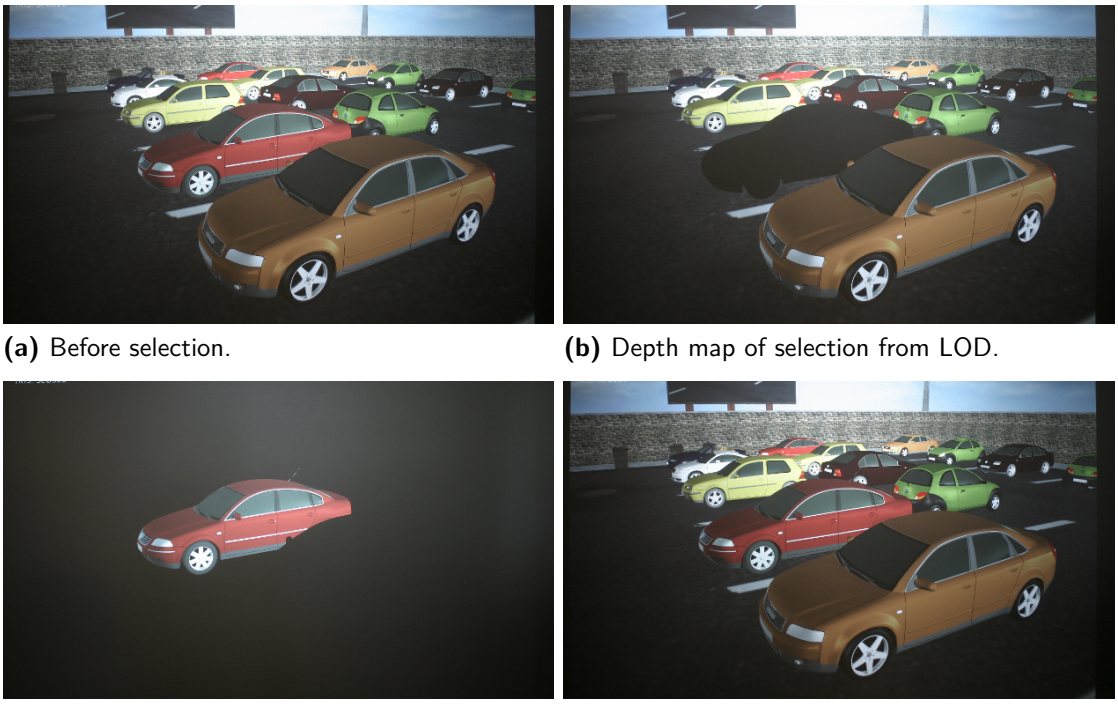
(a) Slow part. **(b)** Fast part. **(c)** Optical superposition.

Figure 3.2: Multi-frame rate rendering by optical superposition. (a) Scene part rendered by SC. (b) Scene part rendered by FC. (c) Optical superposition of (a) and (b) creating the final image as seen by the user.

image produced by two fully overlapped projectors while figures 3.2a and 3.2b show the images from the projectors attached to the slow and fast client, respectively. Figure 3.2c clearly shows that optical superposition does not allow for correct depth occlusion leading to half-transparency effects. While this may not be a problem for system control elements such as menus, objects in the scene are difficult to manipulate without correct occlusion. Nevertheless, for simple interaction schemes that do not rely on depth-correct occlusion optical superposition is easy to accomplish on a current PC system using a dual-GPU setup (cf. section 5.2). This method is also related and was inspired by Computer Graphics Optique [Majumder and Welch 2001].

LOD-based depth testing was developed as a variant to the basic optical superposition technique to avoid the half-transparency effects and to generate correct depth relations in the optical buffer. The idea is that both the slow and fast client render the whole scene, but the respective parts from the other client will be rendered into the depth buffer only. The content of the depth buffer is then used for depth testing the regularly rendered scene elements on each client. In other words, the slow client will render all currently interactive objects to its depth buffer only before rendering its assigned scene parts. The fast client accordingly renders the static scene part to its depth buffer only and its assigned interactive objects afterward. Processing the static scene part on the fast client might reduce its performance advantage. To ameliorate this performance hit simplified versions of the objects in the non-interactive scene part will be rendered to the depth buffer. Additionally, rendering only to the depth buffer can be considerably faster on current graphics hardware as long as the rendering process is not limited by transformation computations.

Figure 3.3 shows digital photographs taken from a multi-frame rate display using the LOD-based depth testing mode for optical superposition. Figure 3.3a shows the combined view before object selection, i. e. the image is generated solely by the slow client. Figure 3.3b shows the part rendered by the slow client after selection occurred; a black shape can be seen, from rendering into the depth buffer only, where the now selected object was located. Figure 3.3c shows the image from the fast client at the same instant as figure 3.3b; only the selected object is visible but



(a) Before selection.

(b) Depth map of selection from LOD.

(c) Selection occluded by other object.

(d) Combined view.

Figure 3.3: LOD-based depth testing and image composition in the optical buffer. (b) shows the projection from SC while (c) shows the part projected by FC; (a) and (d) show the combined view of both projectors.

partly covered by black pixels for the depth values of an object located nearer to the viewer; the final rendering of this occluder will be accomplished by the slow client. Finally, figure 3.3d shows the optical superposition of both figure 3.3b and 3.3c as seen by the user.

As noted before, LOD-based depth testing was primarily developed to avoid incorrect depth occlusion. This is traded for a *dragging depth shadow* if the frame rates of the slow and fast client differ substantially. In figure 3.4 a detail from a digital photograph of a multi-frame rate display configured with LOD-based depth testing is shown where the difference in frame rates is large enough, resulting in a delayed dragging of the depth mask for the selected object in the depth buffer of the slow client (marked outline in figure 3.4). This shape-boundary discrepancy is resolved when user interaction is stopped or becomes very slow, which enables fine grained positioning with correct depth relations while coarser interaction patterns will exhibit a dragging depth shadow. Additional discrepancies between an object and its depth shadow may exist depending on the quality of the simplified object version selected for rendering into the depth buffer. The use of silhouette-preserving LODs (e. g., [Luebke and Erikson 1997]) can correct this situation at the expense of additional computations per image frame.

An alternative to LOD-based depth testing is the use of a restricted viewport for rendering the fast client's content. The screen projection of the selected object is used to determine the visible extent on the fast client and everything inside this

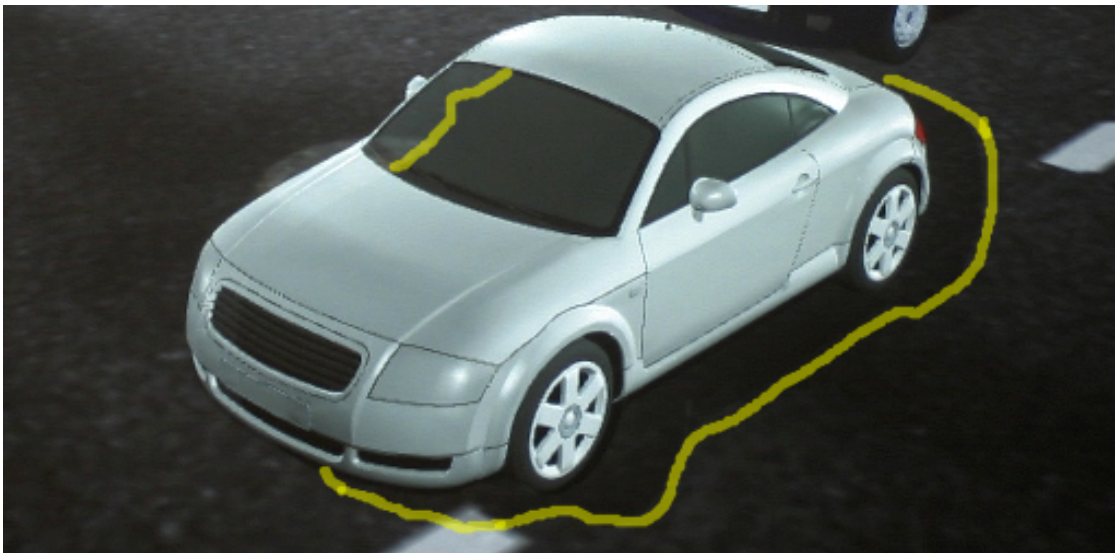


Figure 3.4: LOD-based depth testing artifact due to frame rate differences being too large between SC and FC. The shadow-like outline (marked-up) is the location of the car in the depth map of SC not yet updated to the final position.

viewport is rendered. On the slow client the same screen-space extent is used to exclude scene content from rendering. The restricted viewport is updated whenever the interactive content changes. To avoid discrepancies between the slow and fast client similar to the dragging depth shadow for LOD-based depth testing, a heuristic may be used that enlarges the extent on the fast client to account for delayed updates from the slow client. To avoid conflicting cues for depth occlusion and stereopsis the solution by Godin et al. [2004b] can be used. This method will decrease the fast client's frame rate as the viewport reaches the limit of the full-screen projection leaving the slow client with no work to do while the fast client renders the whole scene.

3.2 Digital Composition

In a multi-frame rate display configured with digital composition the slow client's depth and color buffers are transferred to the fast client. The fast client uses this information to initialize its own depth and color buffer before rendering the interactive objects. The schematic setup is shown in figure 3.5. This approach can be seen as a variation of Sort-Last parallel rendering [Molnar et al. 1994], which gathers images of a subset of the scene using an image composition node or a dedicated hardware compositor. Traditional Sort-Last has to wait until all the images arrive, so the slowest rendering node determines the frame rate. Instead, multi-frame rate rendering using digital composition does not wait for buffers to arrive from the slow client. It always renders at the frame rate of the fast client and incorporates new color and depth information from the slow client as it becomes available.

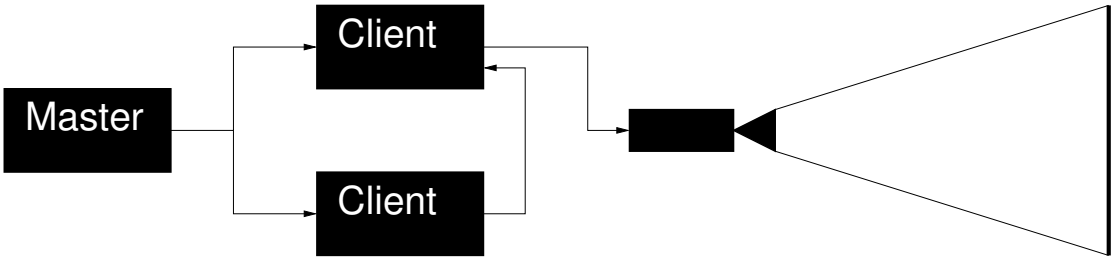
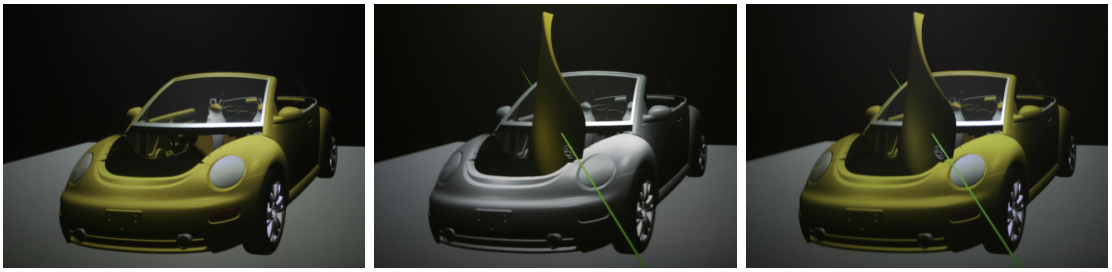


Figure 3.5: Digital composition of color and depth buffer from two render nodes creating a digital (output) buffer.

Figure 3.6 shows digital photographs of an application prototype using multi-frame rate rendering by digital composition. Figure 3.6a shows the part rendered by the slow client. Figure 3.6b shows the output of the fast client, where the frame buffer image transferred from the slow client is shown in gray scale and the part actually rendered by the fast client in regular colors. Figure 3.6c shows the digital composition as seen by the user on the display.

The slow client may need to produce new buffers only when changes in the static part of the scene occur (e. g., on selecting and releasing objects or changing the view point). The extreme case would be an application where only a fixed view point is used. Here, only object migration between the slow and fast client would trigger buffer transfers.

Transfer of color and depth information introduces lag in the system in addition to the actual render time of the slow client. This can be reduced by sending only screen portions where pixel updates actually occurred. This is feasible because on average only a small portion of the scene will be used for interaction. On the other hand, user interaction with an object may lead to close-up inspection, creating a situation where the object's screen projection approaches the size of the whole screen. Sending only screen portions would also introduces jitter in the transport medium because buffers of different sizes must be transferred. A detailed analysis of bandwidth requirements can be found in section 8.1 and an end-to-end latency analysis in section 8.2.



(a) Slow part. **(b)** Fast part. **(c)** Digital composition.

Figure 3.6: Multi-frame rate rendering by digital composition. (a) Scene part rendered by SC. (b) Scene part rendered by FC, frame buffer image from SC in gray scale. (c) Final image on FC as seen by the user.

3.3 Summary

Optical superposition uses two completely overlapped projectors to create an optical buffer for displaying a multi-frame rate image. Because the partial images are merged in the optical buffer no correct depth occlusion can be produced. While this may be sufficient in some situations, such as system-control menus, object interaction without correct depth relations is very difficult. The LOD-based depth testing approach can ameliorate the situation but may introduce other artifacts when the frame rates of the slow and fast clients differ significantly, i. e. a lagging depth shadow. Two overlapping projectors are able to produce only a monoscopic display. Stereoscopic displays are supported by doubling the number of projectors, i. e. two projectors for the left eye and two for the right eye display. However, using an active stereo setup where left and right eye are displayed time multiplexed by each projector can only be supported if the projectors are by synchronized.

Digital composition involves merging the slow client's image and the fast client's image and displaying the result on a single graphics output. The use of depth values from the slow client ensures correct occlusion between objects. In contrast to optical superposition this approach supports not only projection-based systems but also conventional monitor displays or helmet-mounted displays (HMDs). The transfer of full-screen-sized buffers from the slow to the fast client introduces additional lag into the system for non-interactive content. This lag depends primarily on the bandwidth of the underlying transport medium and is discussed in more detail in section 8.1. Optical superposition does not exhibit increased lag because each render client computes its image locally and no information has to be transferred between the clients.

The digital composition method solves the incorrect depth occlusion problem by sending the color and depth values of the slow client to the fast client to be used as the base of the interactive render process. Because the incorrect depth occlusion of the optical superposition method will severely affect user interaction of object manipulation, a result from a user study described in section 8.3, the remainder of this thesis will primarily focus on the digital composition method.

Chapter 4

Multi-Frame Rate Rendering

Multi-frame rate rendering is the process of producing images for a multi-frame rate display. Independently of the chosen display method, optical superposition or digital composition, this requires to designate parts of a scene for interaction, i. e. which parts of the scene are rendered on the fast client and which parts of the scene are rendered on the slow client. This division is however not static. User interaction may affect any object in the scene, i. e. it must be possible to migrate objects from one rendering process to the other upon request. When the images from the slow and fast client are combined the result should be as similar as possible to rendering the whole scene by a conventional single-frame rate process.

Similar to multi-frame rate display by optical superposition or digital composition (cf. chapter 3) the superposition of the scene parts assigned to the slow and fast client results in the whole scene as if it would be rendered by a single-frame rate approach. Figure 4.1 graphically shows the basic idea. To accomplish this it must be possible to split the scene and assign one part to each of the clients as well as to allow for object migration between those parts upon user interaction.

4.1 Scene Management

Scene management for multi-frame rate rendering requires the consistent handling of object migration in addition to conventional tasks such as describing object relationships, material properties, or maintaining geometric object representations. A scene graph [Clark 1976] provides a suitable abstraction for these tasks. Multi-frame rate rendering can be mapped very naturally onto existing scene-graph APIs which have proved to be invaluable for complex 3D graphics applications.

Software prototypes have been developed based on two scene-graph APIs: Avango [Tramberend 1999, 2003], a framework for building distributed virtual reality applications based on OpenGL|Performer [Rohlf and Helman 1994], and the OpenSG scene-graph API [Reiners et al. 2002; Reiners 2002]. In addition to conventional scene-graph functionality both provide facilities for controlling scene distribution, object replication, and incremental change management for distributed graphics applications in a cluster-of-workstations environment. They are also capable of handling multiple graphics devices on a single computer which allows different strategies for hardware setups that support multi-frame rate rendering. This is described in detail in chapter 5.

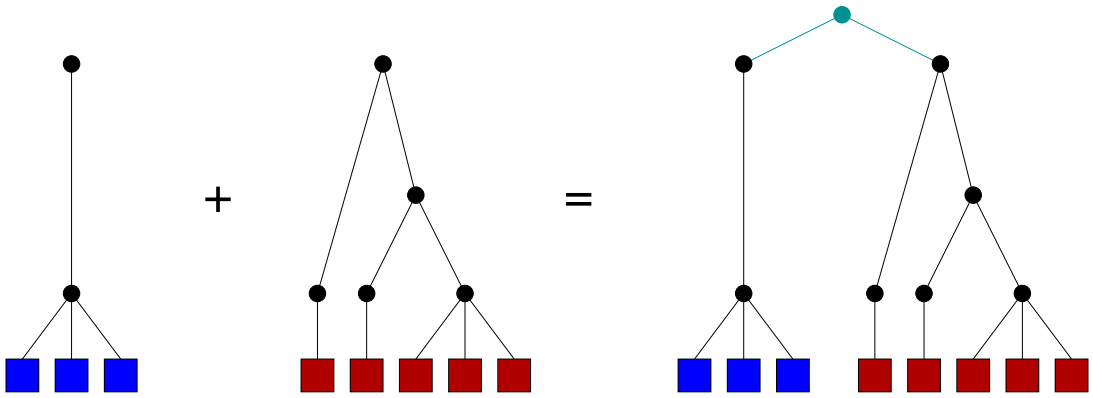
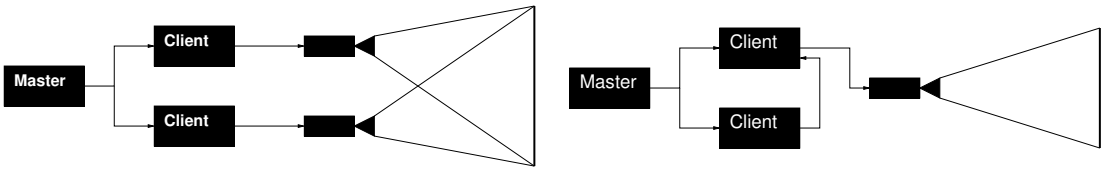


Figure 4.1: Superposition of scene parts; the partial scene graphs (on the left and in the center) form the final scene graph (on the right) for the interacting user when rendered on a multi-frame rate display.

For multi-frame rate rendering using single-system multi-GPU setups other scene-graph APIs may be used that do not support scene replication and distributed updates (e. g., Inventor [Strauss and Carey 1992; Strauss 1993] or OpenSceneGraph [OpenSceneGraph]). However, two features are required for multi-frame rate rendering in this case. First, it must be possible to handle multiple graphics contexts assigned to physically separated GPUs within one application. Second, it is necessary for these graphics contexts to run at independent render speeds. The second requirement is not trivial to implement and control because it may break conventional assumptions made by system designers for scene-graph APIs targeting single-frame rate rendering systems with support for multiple graphics devices. From the above mentioned scene-graph APIs only OpenSG is currently flexible enough to support concurrent traversal contexts as well as controlling their graphics contexts in an asynchronous way. Still, other scene-graph APIs can be employed by designing an application using a multi-process strategy and having the separate rendering processes communicate via dedicated system resources; this is described in detail in section 5.2 and analyzed in section 8.1.

4.2 Object Migration

Object migration between the slow and fast client occurs at the beginning and the end of a user interaction with an object. It is assumed that object migration from the slow to the fast client is part of the selection process. The reverse migration of objects, from the fast to the slow client, is part of the release process. No particular interaction scheme is imposed here. A simple interaction scheme may invoke the intersection test of a ray from the interaction device into the scene. The intersection test will find the first or an appropriate object by traversing the scene graph bottom-up. More sophisticated methods along with a comprehensive treatment of 3D user interaction can be found in the book by Bowman et al. [2004].



(a) Optical superposition. **(b)** Digital composition.

Figure 4.2: Role setup for multi-frame rate rendering.

An easy way to divide the scene for the participating render processes would be the use of object lists. User input may affect any object by selection and release. Once an object is selected it has to migrate to the object list relevant for interaction and upon release it is migrated back to the object list containing the rest of the scene. This allows for implementations running in a single process on the same host machine as well as in distributed setups. However, most of the current infrastructure for visualization and VR systems is based on a scene-graph API. By using such a scene-graph API for managing object migration for multi-frame rate techniques existing infrastructure can be used that has been proved to be beneficial for structuring, maintaining, and processing 3D content efficiently. While a simple object list approach does provide a solution to migrating objects back and forth it also introduces management overhead that can be avoided by using a scene graph.

In a scene-graph environment an object can be assigned to a render process. This is achieved by comparing node masks on objects and traversal masks on the render-traversal process. Assuming a generic setup consisting of a master node and client nodes for the slow and fast part (cf. figure 4.2) object migration works as follows. The master node loads the complete scene and distributes the resulting scene graph to all clients. Thus, each client holds the complete scene graph. *Traversal masks* are then used to select a certain part of the scene graph for rendering on each individual client. Two types of masks are used. Nodes in the scene graph have bit masks assigned, which are called node masks. Scene graph traversal processes, such as the rendering process, have also bit masks assigned, which are commonly called traversal masks. During traversal of the scene graph the current node mask of the node visited is evaluated against the traversal mask of the traversal process using a bit-wise `AND` operation. If the result is zero the sub-graph below the current node is not further considered for traversal by this process. Each render client has a different traversal mask assigned for its rendering traversal as shown in figure 4.3. The master node assigns different node masks to the respective parts of the scene graph, such that these parts are rendered only on the corresponding clients. These node mask changes as well as any other changes inside the scene graph are communicated to the clients using the cluster support of the scene-graph toolkit. Note that distributing the complete scene to all clients greatly simplifies the process of activating and deactivating parts of the scene graph on individual clients.

By changing node masks objects consisting of whole sub-graphs in the scene graph are assigned to and taken out of consideration at individual rendering clients.

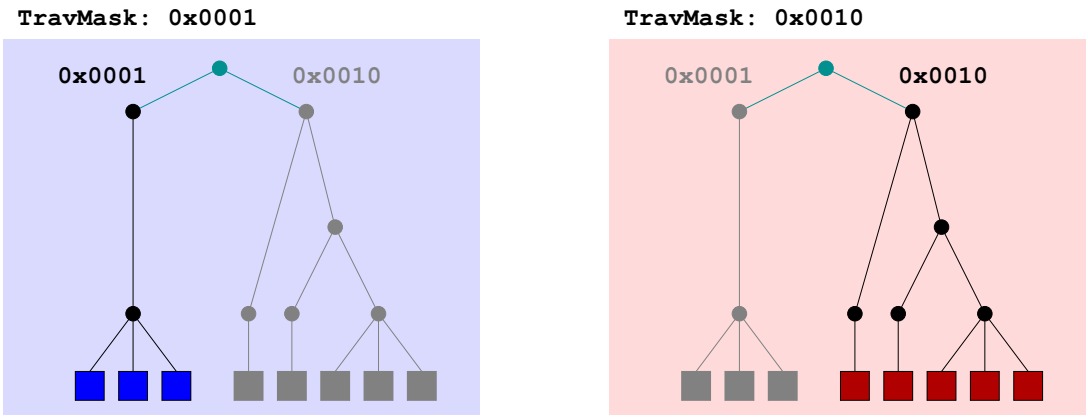


Figure 4.3: Traversal-mask update on different multi-frame rate clients. On the left only the node mask of the left partial graph is compatible with the traversal mask; the partial graph on the right is excluded from the traversal. On the right the reverse situation is shown for another multi-frame rate client.

This process may be used for object interaction or animation. In case of interaction the object selection will change the node mask so that the fast client will include the object in its traversal while the slow client will exclude the same object. Upon object release the node mask will be reversed so that the fast client excludes the object while the slow client includes the object again. Animation can be supported by deliberately using different node mask for object parts. For example a clock's second hand may be rendered on the fast client while the minute and hour hands are rendered on the slow client.

Alternatively, object migration may be achieved by using different sub-graphs for the interactive and static parts of a scene. Object selection would then remove the selected object, i. e. its top-level node and therefore its entire sub-graph from the graph containing the static scene parts, and insert it at a predefined position in the graph holding the interactive scene parts. This top-level node must also maintain a transformation that ensures object movement between both graphs does not change object relations with respect to the scene. This process involves repeated computation of inverse path matrices and their multiplication with the object's top-level node for successive object interaction and may be a source of numerical inaccuracies over time.

4.3 Image Generation

Multi-frame rate images are created from partial images by the participating rendering processes, i. e. the slow and fast client. For multi-frame rate display using optical superposition these images are combined in the optical path of the output devices. This is directly equivalent to the superimposing of parts of the scene as shown in figure 4.1. In a multi-frame rate display using digital composition a similar process is used. Here, the color and depth data from the slow client are

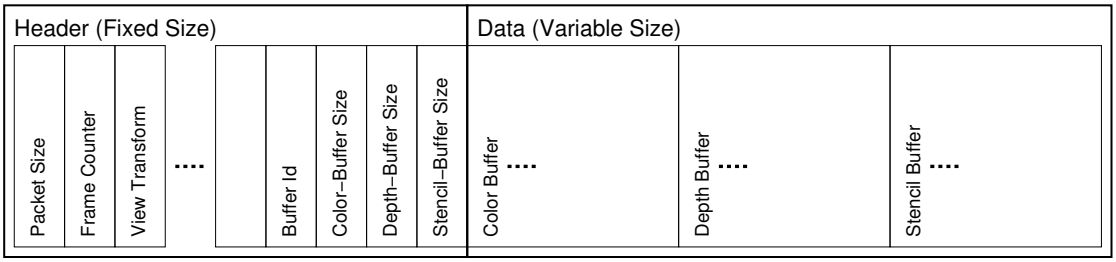


Figure 4.4: Transfer-buffer abstraction used for transferring buffers from the slow to the fast client.

used to initialize the render buffers on the fast client before the interactive parts of the scene are rendered. This means that the fast client first fills its frame buffer with the most current color and depth values from the slow client before rendering the interactive scene parts. Because slow and fast client run at very different frame rates the fast client will reuse the last color and depth information from the slow client over several successive frames.

The color and depth information from the slow client must be transferred to the fast client. In addition to the color and depth data from the slow client also management data is necessary to allow for data processing on the fast client. Figure 4.4 shows the transfer buffer (for multi-frame rate rendering using digital composition) schematically. A fixed size header describes the current content of the transfer buffer by packet size, i. e. the overall size in bytes of the actual buffer, the frame counter, the view transform used for generating the image contained in the buffer, and the current size in bytes of the individual sub-buffers holding color and depth data. The fixed size of the header allows for easy decoding and analysis of its items. After the header is analyzed the sizes of the individual color and depth buffers are known and target buffers can be created dynamically, if necessary. In figure 4.4 also a buffer id is included in the header. This item is used to encode the buffer association such as left-eye or right-eye buffer. It allows for supporting stereo-buffer setups as well as setups for multiple render targets (MRT). In both cases the receiving process, or thread, on the fast client will have to ensure that, in case of a stereo setup, all buffers from the same frame of the slow client are received before they are applied to their respective buffers on the local graphics device. The view transform item, containing the view transform with which the buffer content was generated on the slow client, will be used to update the fast client's view transform. This conservative view point update allows for image generation on the fast client to be consistent with the current buffer from the slow client.

As mentioned above the sub-buffers for color and depth values of a transfer buffer have to be filled by downloading the respective buffers from the slow client's graphic device and will be used for upload to the fast client's graphics device. While the transfer buffer allows for flexible buffer sizes (e. g., a stencil buffer size of zero in the header simply means that no stencil values are contained) the contiguous layout of the sub-buffers must be separated at run time to be used as input to the actual

download interface of the 3D graphics API. To avoid performance hits as well as increased memory footprint this separation should avoid copying data buffers as much as possible (cf. section 8.1).

4.4 Summary

Multi-frame rate rendering can be expressed in terms of existing infrastructure. With a scene-graph API partitioning the scene into parts relevant to the current interaction and a static rest multi-frame rate images are created by rendering these scene parts on the assigned render processes and combining the partial images afterward. This can be thought of as a superpositioning of the partial scenes assigned to the individual traversal processes to recreate the scene as a whole. Efficient object migration between the participating render processes for interaction purposes is also supported by existing scene-graph APIs using different traversal masks for the slow and fast client and changing node masks for objects at run time.

Chapter 5

Multi-Frame Rate System Setups

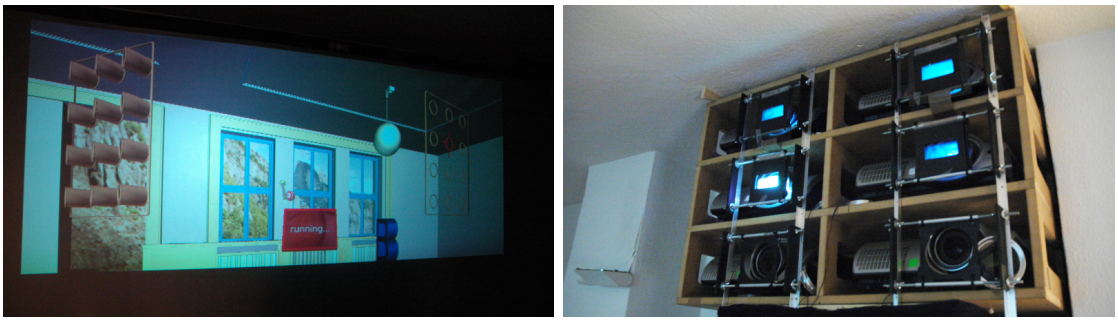
Multi-frame rate techniques can be realized with a variety of system setups. The requirement for asynchronously running image generators can be satisfied by a cluster of workstations, where each node provides its own graphics device, or by a single workstation with multiple separate graphics cards. The information exchange between the graphics devices can be performed a variety of transport media depending on the hardware platform used. It is also possible to implement multi-frame rate rendering with a single graphics card on a single computer by interleaving the command execution for the participating graphics contexts, thus virtualizing the graphics hardware. Finally, multi-frame rate rendering can be employed in existing cluster-system setups creating hybrid systems.

5.1 Graphics Cluster

A *cluster of workstations* (COW) uses multiple computers connected by a shared network to accomplish a task. The idea is that if a single computer is unable to generate results fast enough or cannot handle all the data using more computation resources will decrease processing time or allow for the handling of parts of the data on each computer. The specific setup and control usually depends on the application problem and can only be partially generalized. Workstation clusters have been in use for some time especially in the high-performance computing community. Here, the data size of the problems usually exceeds the memory capacity of a single workstation. Examples are weather forecasting, protein folding, or finding very large prime numbers. This can be extended to the point where the compute resources are distributed all over the world with each node running a program tailored for the purpose and using system resources only when the regular computer usage falls under a threshold; prime examples for this *grid computing* are clients for distributed computing running as screen savers [Folding@home; SETI@home]. While specialized machines have been designed and build to allow the investigation of such problems their costs do exceed the cost of a cluster of workstation by orders of magnitudes. This cost comes primarily from to the use of specialized bus systems for data transport from and to the CPUs and the system memory as well as from the production process of a low-volume market. Workstation clusters, on the other hand, are build from off-the-shelf PCs and standard networking technology. Two factors are essential here. First, exchanging and upgrading parts only involves the

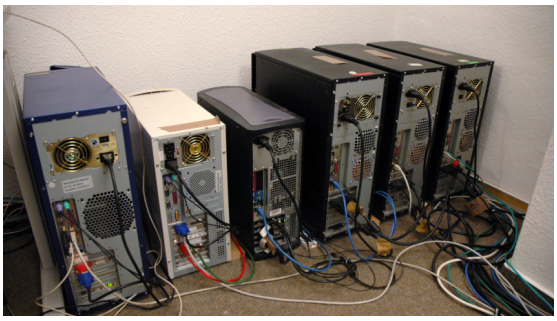
replacement of standardized components. Second, compute power usually can be increased by simply adding more or newer components to the cluster nodes. The price to be paid, with respect to specialized super computers, is that communication between cluster nodes depends on the networking technology used and is in general at least an order of magnitude slower. Recently, manufacturers of super computers are also building their products as clusters of workstations but employ specialized technologies that provide improvements over standard parts (e. g., high-speed networking interconnects, minimizing form factor or power consumption to justify higher cost). This leads to a convergence of low cost workstation clusters and specialized super computers where different financial budgets and computational requirements can be met.

A special case of workstation clusters are *graphics cluster*. As mentioned before, in general clusters of workstations are used for compute-intensive problems or handling of data too large for a single machine. In addition, graphics cluster are used for generating visual output on time. Use cases include tiled walls, where multiple projectors, each driven by a cluster node, are used to create very high resolution images (e. g., [Li et al. 2000]), or single projection screens using Sort-Last/Sort-First load balancing to speed up the image generation process of very large or complex data. Examples also include the deployment of workstation clusters for ray tracing very large scenes (e. g., [Wald et al. 2001, 2003; DeMarle et al. 2003]). Recently, support for multiple users emerged for these display types [Fröhlich et al. 2005; Blach et al. 2005]. In contrast to a general cluster of workstations, the purpose of a graphics cluster is to create a final image from distributed image creation processes without introducing artifacts in the display output. This means the image generation processes are usually synchronized, i. e. swap-locked, as well as the graphics devices, i. e. frame-locked. These techniques ensure that physically separated graphics devices begin a new frame at the same time by being swap-locked. Frame locking is used to guarantee that each projector shows its image content at the same time as any other participating projector. This is especially important for time-multiplexed stereoscopic displays but also for high-resolution monoscopic displays consisting of many tiles to ensure image consistency. The amount of render nodes in a graphics cluster is primarily determined by the amount of projection devices. This is unlike general compute clusters where adding additional nodes will improve performance until network bandwidth is saturated. Performance in a graphics cluster can be improved by using an additional compute cluster to pre-compute aspects of the output image and (re-)distributing the results to the actual graphics cluster for display. In both cases it must be noted that system setup, control, and maintenance require considerable efforts. On the other hand, graphics cluster exhibit a very good update path with respect to improvements in graphics hardware. It normally takes only the exchange of the graphics devices for a newer generation to achieve substantial improvements in graphics performance or feature additions or both. This also implies that the base hardware for graphics-cluster workstations should be specified with balanced CPU and memory requirements in mind.



(a) Projection screen.

(b) Projector array.



(c) Cluster nodes.

(d) Network switch.

Figure 5.1: Components of a graphics cluster for projection-based displays.

Figure 5.1 shows a typical setup for a graphics cluster build from off-the-shelf components. It consists of a projection screen, a set of projectors, a set of workstations, and a network switch connecting the workstations. The workstations are divided into a master node, a tracking system control node, an audio processing node, and three render nodes (from left to right in figure 5.1c; the setup shown was used for combining multi-viewer stereo display and wave field synthesis audio [Springer et al. 2006]). Each render node, in this particular setup, is connected via its graphics card outputs to two projectors to achieve passive-stereo display (horizontal rows of projectors in figure 5.1b). The projectors can be calibrated to create a tiled high-resolution image or to create a single image, i. e. all the projector images are completely overlapped. The completely overlapped setup in turn can be used for a Sort-First rendering setup or a multi-user setup. The master node processes the update events from the tracking control node which is connected to the actual interaction devices. While any of the rendering nodes may also be used for this role experience from experimentation demonstrates that a master node not taking part in the actual image creation process for the screen has the advantage of providing continuous processing of update events originated by the interaction devices. This is unlike an actual render node because these may use up most of their processing power for computation and image generation thereby exhibiting considerably more latency when processing interaction events. This is discussed in detail in section 8.2.

Specialized software frameworks are used for driving a graphics cluster. While general APIs for distributed communication, such as MPI [Snir et al. 1994] from the

high-performance computing community, could be used graphics clusters are usually driven by software toolkits providing a distributed scene-graph API [MacIntyre and Feiner 1998]. This is achieved by accumulating changes in the scene graph of one node (each frame), usually the master node, and sending these changes to all other participating nodes. On the receiver side(s) the changes are then incorporated so that the scene-graph states for the sending node at this frame and the corresponding frame at the receiving nodes are the same. The changes must be communicated in an ordered and reliable way [Tramberend 2003]. The scene-graph API may provide this functionality by itself such as OpenSG [Voß et al. 2002] or it can be build on top of a scene-graph API using a distributed communication API (e. g., Avango uses Ensemble [Ensemble] on top of OpenGL|Performer [Tramberend 1999]).

Multi-frame rate rendering is naturally implemented using a graphics cluster. For a setup supporting optical superposition two render nodes are necessary each connected to a projector, or a pair of projectors for passive stereo. These nodes will be assigned the roles of slow and fast client, as described in section 3.1, and will use the object migration mechanism described in section 4.2. Similar requirements exist for the digital composition approach. The node assigned to the role of the fast client is connected to its projector or pair of projectors, as described in section 3.2, while the slow client node technically does not need a graphical output capability. The color and depth values from the slow client are read from the slow client's graphic device, send to the fast client, written to the fast client's graphics device, and used to initialize the fast client's frame buffer (cf. section 8.1). This means that for a digital composition setup considerably more bandwidth is necessary for the cluster node's interconnect because the slow client will send full-screen-sized image data to the fast client. Alternatively, a dedicated network connection between the slow and fast client can be used at the expense of increased computational processing at the operating-system level on the sending and receiving nodes. In contrast to nodes in a single-frame rate graphics cluster no synchronization of the render processes or graphics devices is required because the goal of multi-frame rate rendering is to improve interaction fidelity by creating a purposely unbalanced load ratio on the participating image generators.

5.2 Multi-GPU Systems

Support for multiple graphics cards in a single computer system once was the domain of specialized high-end graphics systems (e. g., SGI's RealityEngine [Akeley 1993] or InfiniteReality [Montrym et al. 1997] platforms). With the introduction of the PCIe bus system [PCI Express] this capability became recently also available for standard PCs. Besides using multiple graphics boards in such a PC system for dedicated tasks, vendor-specific solutions exist which allow the combined usage of multiple graphics devices for the generation of a single image (e. g., NVIDIA SLI [NVIDIA GPU Programming Guide 2006; NVIDIA SLI] or ATI CrossFire [Persson 2005; ATI CrossFire™ Technology Whitepaper 2005]; cf. figure 5.2). To simplify

(a) NVIDIA SLI using 2 × NVIDIA GeForce 8800.

(b) ATI CrossFire using 2 × ATI X850 XT.

Figure 5.2: Multi-GPU system hardware.
(a) from http://i.dell.com/images/global/products/superview/sv_nvidia_sli_8800.jpg and (b) from http://www.pcper.com/images/reviews/168/board_installed.jpg.

the usage these solutions expose the coupled graphics devices as a single virtual graphics device and provide several rendering-management strategies. Alternate frame rendering (AFR) processes frame n on one card and frame $n+1$ on the other card, transferring the result from the secondary to the primary card using a vendor-specific hardware channel, and only the primary card displays the resulting images. In split frame rendering (SFR) mode both cards process every frame but for different screen regions or tiles. The screen regions are variable and a scheduling mechanism attempts to size these regions so that both cards take the same amount of time for rendering one frame. The screen regions are overlapping and, when combined, result in the full resolution of the screen. The secondary card transfers its result to the primary card where the final image is composited and displayed. Image enhancements are supported by a third mode that uses full-screen anti-aliasing on both cards and combines the singular results on the primary card. With these vendor-specific solutions the image generators are tightly coupled and they are presented as a single graphics device to an application. Unfortunately, there is currently no support for application-specific Sort-Last composition using the high-bandwidth low-latency hardware channel available to the vendor-provided modes, although this largely seems to be a problem of a lacking software interface. While this may change in future versions, currently this can only be achieved by transferring data from one card to another via host-system memory (cf. sections 8.1 and 8.2).

A multi-GPU system setup for multi-frame rate rendering is displayed in figure 5.3. The particular setup shown in figure 5.3 differs in two ways from a standard single-PC setup. First, the PC systems acted as part of a graphics cluster driving a front-projection display system with two projectors (PC in the middle of figure 5.3). Second, instead of a dual-GPU setup using standard consumer graphics, as shown in figure 5.2a, an external graphics sub-system was used containing two professional graphics cards, a dedicated PCIe bus system, and its own power supply (smaller box on the right in figure 5.3; close-up views shown in figure 5.4). One graphics

Figure 5.3: Multi-GPU Hardware Setup. PC in the middle is the render client. The graphics sub-system, smaller box right beside the render PC, is provided by an NVIDIA Quadro Plex (cf. figure 5.4) with four outputs: two for the passive-stereo projection system (fast client output) and two for the output devices in the upper right corner of the image (slow client output). In the image the output displays in the upper right show the slow client's left eye image on the left monitor and the fast client's left eye image on the right monitor for demonstration purposes.

card from the external graphics sub-system connects to the projectors while the other graphics card is connected to standard desktop-display devices for monitoring purposes only. The graphics card connected to the projectors is assigned to the role of the fast client while the other graphics card acts as the slow client (cf. output displays in the upper right corner of figure 5.3).

Software control of multi-frame rate techniques in a multi-GPU setup is similar to the software control in a graphics cluster as discussed in section 5.1. This means that for the purpose of driving such a setup the software infrastructure from an existing cluster implementation can be reused by treating the single system as two graphics nodes distinguished only by the context ids of the physical graphics devices. Such an implementation approach may exhibit sub-optimal performance because concurrent processes would compete for unique system resources such as network bandwidth or host memory. With a multi-GPU setup it is also possible

Figure 5.4: NVIDIA Quadro Plex using 2 × NVIDIA Quadro FX 5600 plus NVIDIA G-Sync (second slot from the left in the rear view).

to use a single-process approach where only necessary functionality such as the asynchronous rendering is dedicated to lightweight processes or threads. Section 8.1 discusses this topic more detailed.

5.3 Single-GPU Systems

Implementing multi-frame rate rendering on a single PC with multiple GPUs can provide performance improvements since it avoids the need for a cluster of workstations. Even a single-GPU single-machine setup as it is available to most computer users today can be used for multi-frame rate techniques. The basic idea here is to virtualize the GPU resource and use it quasi-simultaneously as slow and fast client. Such an interleaved rendering allows even users of portable computers or other standard graphics devices to benefit from the improved interaction fidelity of multi-frame rate techniques.

Interleaved rendering uses the time left before buffer swap of the fast client part to render a certain amount of geometry from the slow client part. In the following it must be distinguished between frame rate and sync rate. Frame rate is the frequency with which frame buffer updates are available. Sync rate is the rate with which frame buffer content is *scanned-out* to the output device. While the sync rate is constant (e. g., 60 Hz or 75 Hz) the frame rate usually is not. For example for a desired frame rate of 60 Hz the actual frame time might be anywhere between 8 ms, or even lower, and 16.6 ms.[1] In a graphics cluster setup or a multi-GPU configuration this time can be used to update frame buffer data from the slow

[1] A frame time of 8 ms corresponds to a frequency of ≈120 Hz while a frame time of 16.6 ms corresponds to 60 Hz. Because a fixed rate of 60 Hz is assumed for updating the image display the actual render time may be lower and is thus given as a time period rather than a frequency.

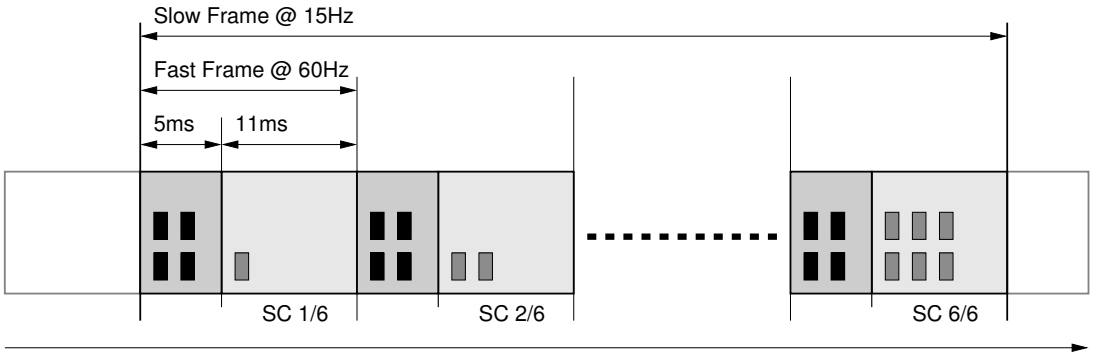


Figure 5.5: Multi-frame rate rendering by digital composition on a single computer system and a single image generator using interleaved rendering streams.

client on the fast client. In a single-GPU setup this leftover time until buffer swap for the fast client is used to perform partial work for the slow client by rendering *chunks* of geometry (cf. figure 5.5).

Rendering of arbitrary sized spatial parts of a scene is a non-trivial problem. In an experimental prototype a simple mechanism was used to split up the scene in chunks of geometry with an equal amount of triangles. Chunk size was chosen so that its graphics processing time was small enough to create batches of several such chunks that would fit into the remaining frame time of the current fast client's image frame. A special render context is used for the slow client. This graphics context uses a double-buffered frame-buffer object (front-FBO and back-FBO, similar to double-buffered rendering) as a render target to reuse the results from earlier passes. The fast client always reads from the slow client's front-FBO. The slow client always renders the parts assigned from a scheduler into its back-FBO. Once the slow client is finished with all of the assigned parts its front-FBO and back-FBO are swapped, which is just a pointer swap and therefore inexpensive. The new front-FBO is used as input for the fast client while the slow client starts with the next frame cycle rendering to its newly assigned back-FBO until all relevant parts of the scene are finished again. Figure 5.6 shows screenshots from an interleaved rendering application prototype. The intermediate results from the slow client's

(a) SC at 25%. **(b)** SC at 50%. **(c)** SC at 75%. **(d)** SC at 100%.

Figure 5.6: Screenshots of a multi-frame rate application in interleaved rendering mode on a single GPU where the array of cars is rendered by the slow client context only. (a) to (d) show the completion of the slow client's frame content at 25%, 50%, 75%, and 100%, respectively.

frame completion at 25%, 50%, 75%, and 100% are used for emphasis here in the digital composition process.

The interleaved rendering approach has limitations. Most importantly the fast client's draw time directly affects the draw time of the slow client. This results in a theoretically infinite draw time for the slow client if there is no time left between the end of the fast client's frame and display sync. Even if this theoretic limit cannot be reached, the fast client's update rate may become very low for realistic setups. Multi-frame rate display by optical superposition is not supported because only one output image is generated. Successive rendering of parts of a scene requires a spatial segmentation of the scene similar to out-of-core algorithms (e. g., [Isenburg and Gumhold 2003; Lindstrom 2003]). Most of these algorithms, while solving the segmentation problem, may exhibit an excessive computation time; pre-processing in an off-line stage is still state of the art for out-of-core rendering (e. g., the `OOCProxyBuilder` tool included in OpenSG 1.8 [OpenSG]). In a single-GPU multi-frame rate rendering setup re-computation of the spatial segmentation would be necessary because it is not possible to determine chunk sizes a priori that correlate well with the current application scenario as well as graphics hardware and features used. Such on-the-fly re-computations are currently subject of active research in the graphics community.

5.4 Hybrid Systems

Hybrid system setups are conceivable in several forms. First, either the slow client or the fast client do not represent a node in a graphics cluster but are itself a cluster of graphics nodes. Especially in the case of the slow client this would allow for scalability of the graphics performance by only increasing the latency in the slow client's image. Second, if both the slow and the fast client are a graphics cluster than these two clusters can be combined and nodes in the cluster can be arbitrarily assigned to be either part of the slow or the fast client, or even being idle. This would allow for a dynamic load-balancing scheme in case either the logical slow or fast client drops below a certain frame-rate threshold by reassigning nodes to the part with insufficient performance.

5.5 Summary

Multi-frame rate rendering is supported by a variety of systems setups. It is intuitively implemented on a graphics cluster or a single-system multi-GPU setup. In a multi-GPU setup the increased bandwidth of the graphics devices as well as the system memory allows for lower transfer latency of the slow client's frame-buffer data (cf. section 8.2 for a detailed discussion). Multi-frame rate rendering can also be implemented on single-GPU systems using an interleaved rendering strategy on a conceptually virtualized graphics device where the work of the slow client is performed in successive chunks in the time left between the fast client's frame

generation and buffer swap. It completely avoids buffer transfer at the expense of application-specific scene partitioning but allows the use of low-end graphics devices such as mobile phones or PDAs. Finally, hybrid systems based on larger graphics clusters can be used to allow for dynamic load balancing of resources to deal with performance variations of either the fast or the slow client.

Chapter 6

Artifacts

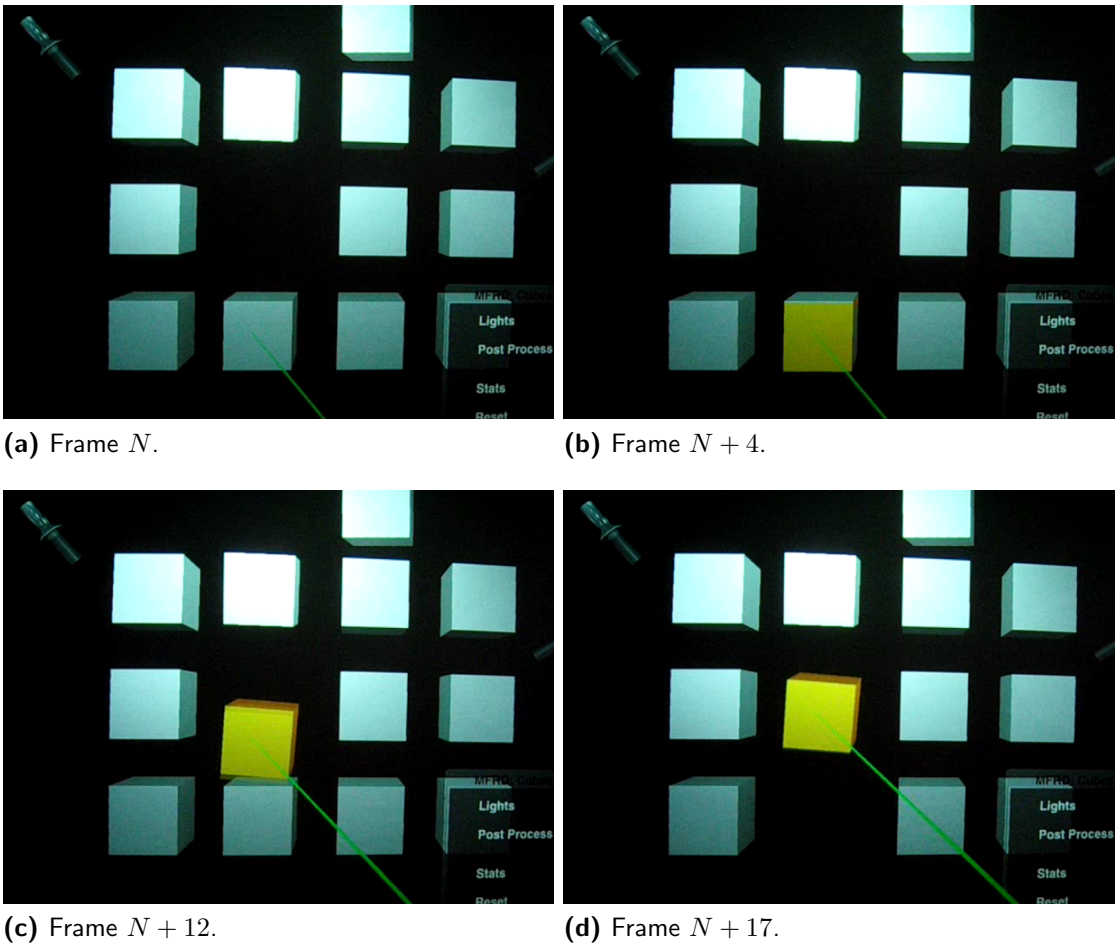
Multi-frame rate rendering introduces artifacts into the final image because partial images with temporally different sampling from asynchronously running image generators are composited. Migration artifacts may occur in the process of object selection or release, i. e. object migration. Scene-global rendering effects are also affected by asynchronously running rendering processes such as light and material manipulation or transparency.

6.1 Migration Artifacts

In multi-frame rate rendering objects are migrated from the slow client to the fast client at the beginning of an interaction. The objects are migrated back to the slow client once they are no longer manipulated. With a replicated scene graph in each rendering process only a bit mask has to be manipulated to migrate an object between clients (cf. section 4.2). However, because of the parallel and asynchronous rendering processes on the fast and slow client in combination with object migration during user interaction the selection and release artifacts may occur.

Selection Artifact Once an object is selected it is activated on the fast client and deactivated on the slow client. Because it takes typically a couple of frames until the updated frame buffer from the slow client arrives on the fast client, the object is actually rendered twice into the multi-frame rate image: once by the slow client and once by the fast client. However, this will become apparent only if the user starts to manipulate the object immediately before the updated frame buffer from the slow client arrives. In this case both the image of just moved object and the image of the object in its original location will be shown. Otherwise depth comparison takes care of the doubly rendered object and this artifact will not be visible to the user. Figure 6.1 shows this process exemplary with the slow client's image content in gray scale while figure 6.2 shows the process as seen by the user.

Figure 6.2 also conveys the perception problem of the user. If the difference between the frame rates of the slow and fast client is large enough, the user, currently manipulating an object, will perceive the sudden change in brightness and contrast upon arrival of the new frame buffer from the slow client. This *popping-out artifact* can be perceptible enough to distract the user from his task. Also, it degrades visual fidelity in the sense that objects vanish from the image



(a) Frame N. **(b)** Frame $N + 4$.

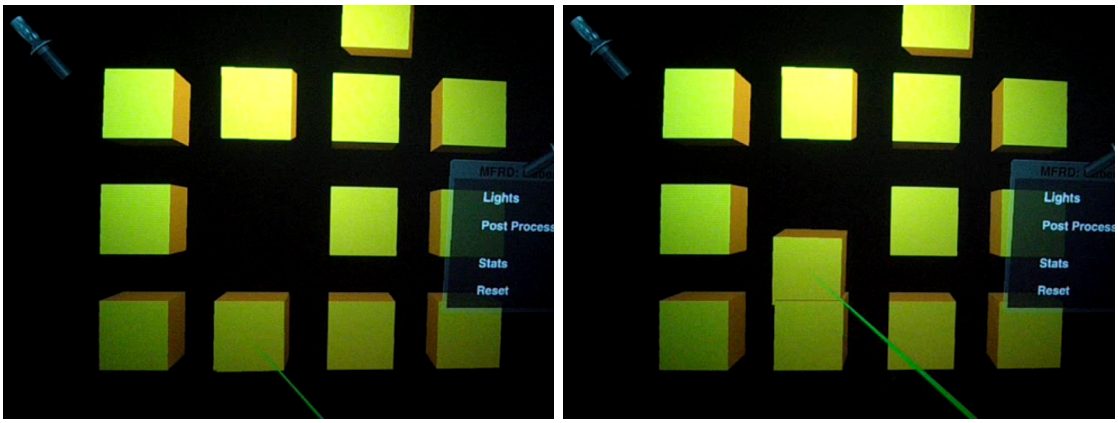
(c) Frame $N + 12$. **(d)** Frame $N + 17$.

Figure 6.1: Selection artifact. Images show (successive) still frames from a digital video of a prototype application running in naïve multi-frame rate rendering mode; color buffer from SC drawn in gray scale for emphasis. (a) situation before selection, (b) user selected object, (c) selected object moved but original object still in last buffer from SC, (d) buffer update from SC arrived excluding the selected object.

without the user being (always) sure of the correlation between his actions and object appearance in the fast client's image.

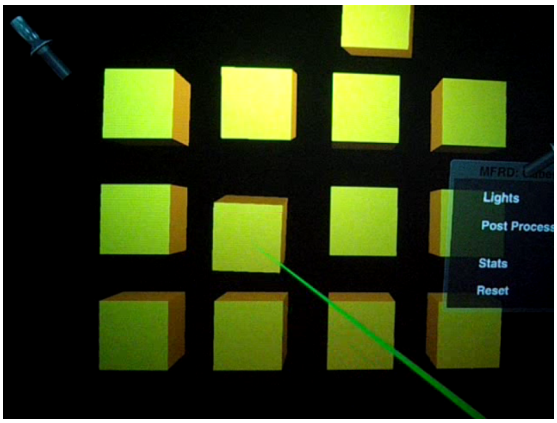
Release Artifact Once an object is no longer needed for manipulation it is immediately deactivated on the fast client and, also immediately, activated on the slow client. Thus, the fast client does not render this object anymore, but it takes again some frames of the fast client until an updated frame buffer from the slow client arrives which incorporates this change. During this intermediate period the object is not displayed at all and the user perceives it as a *popping-in artifact*. Figure 6.3 shows this migration artifact exemplary with the slow client's color-buffer content in gray scale for emphasis. Figure 6.4 displays the images for a release artifact as seen by the user.

Arguments similar to those for the selection artifact also hold for the release artifact with respect to its user perception. A user releasing an object will see

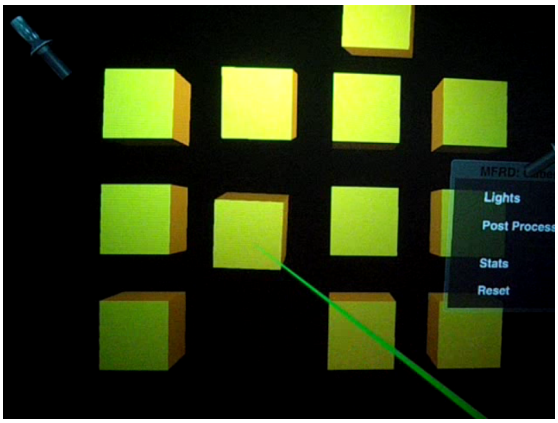


(a) Frame N.

(b) Frame $N + 5$.



(c) Frame $N + 9$.

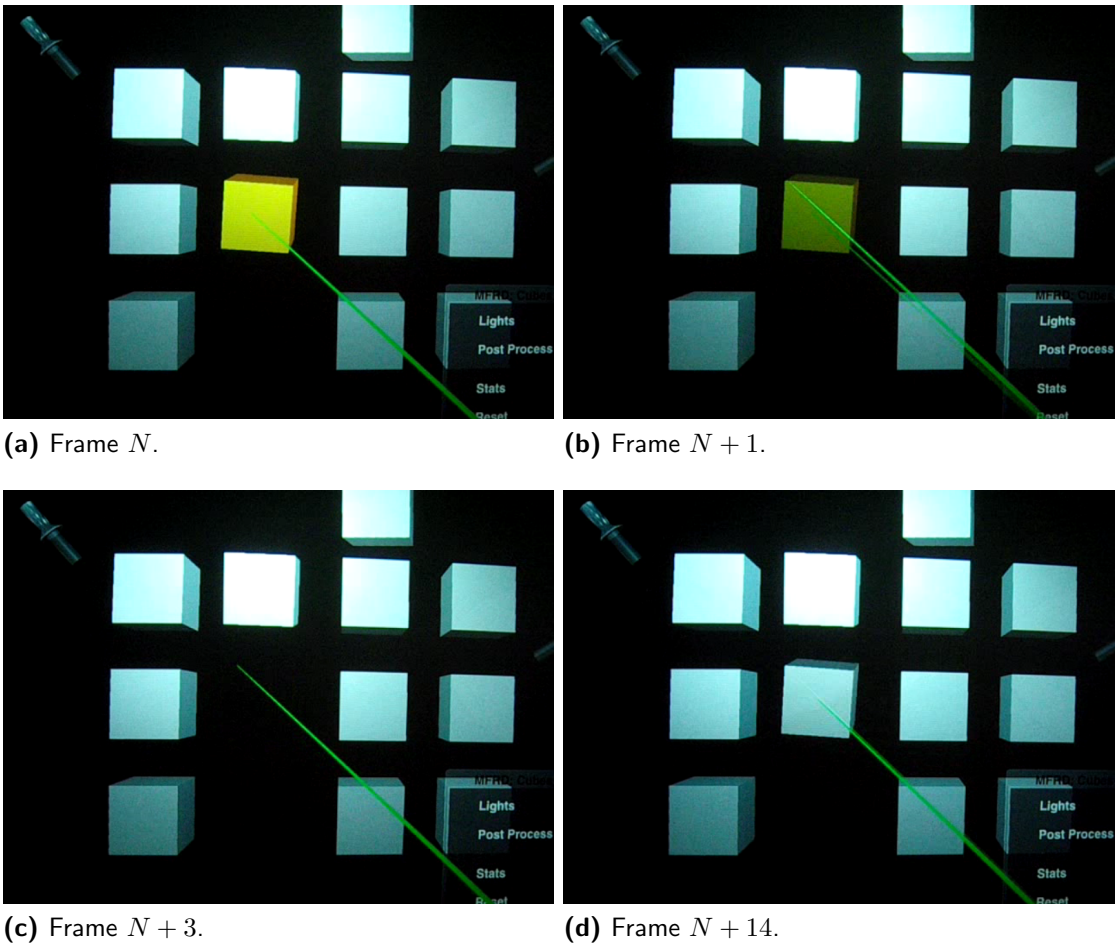


(d) Frame $N + 10$.

Figure 6.2: Selection artifact (user visual). Images show (successive) still frames from a digital video of an prototype application running in naïve multi-frame rate rendering mode. See figure 6.1 for process description.

the object vanish. After some time the object will reappear as part of the slow client's image causing a sudden change in brightness and contrast in the whole image. Assuming the user just started the next manipulation the reappearance of the object will then distract from the current task. Here, as above, visual fidelity can be degraded and which may also influence interaction fidelity.

Solution The selection and release artifacts can be ameliorated by using prediction and appropriate bookkeeping. For predicting the selection of an object an *over* status similar to 2D interfaces is introduced, i. e. a mouse-over event is triggered as soon as the pointing device is over the object. Assuming ray-based selection the object is activated on the fast client as soon as the ray intersects the object even if the user has not yet explicitly selected that object. At the same time it is deactivated on the slow client. Experience shows that this heuristic works well because in most cases users need some time from entering the over status until the actual selection. Fixing the release artifact seems simple at first glance: it should be enough to just keep the object active on the fast client until it is incorporated



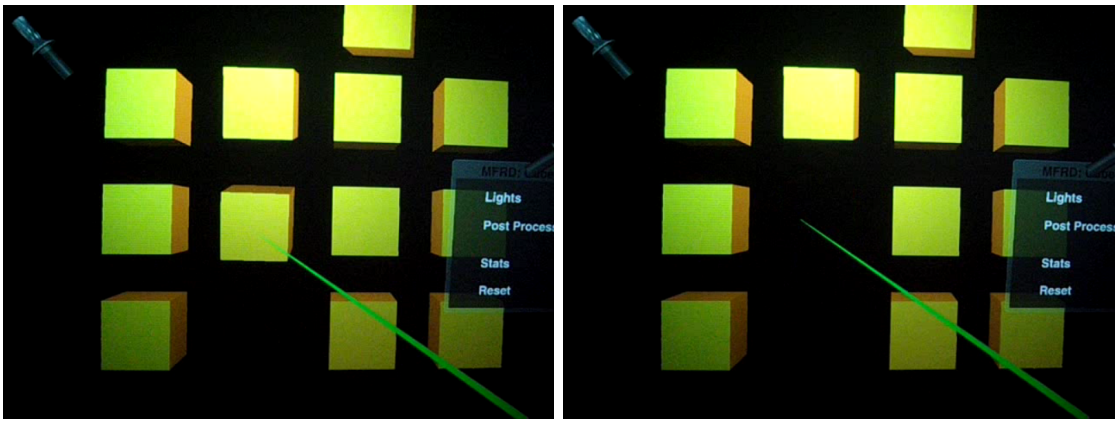
(a) Frame N. **(b)** Frame $N + 1$.

(c) Frame $N + 3$. **(d)** Frame $N + 14$.

Figure 6.3: Release artifact. Images show (successive) still frames from a digital video of a prototype application running in naïve multi-frame rate rendering mode; color buffer from SC drawn in gray scale for emphasis. (a) object release, (b) intermediate (video) frame showing object vanishing from FC's image, (c) object not in FC's image and no update arrived from SC, (d) buffer update arrived from SC including the released object.

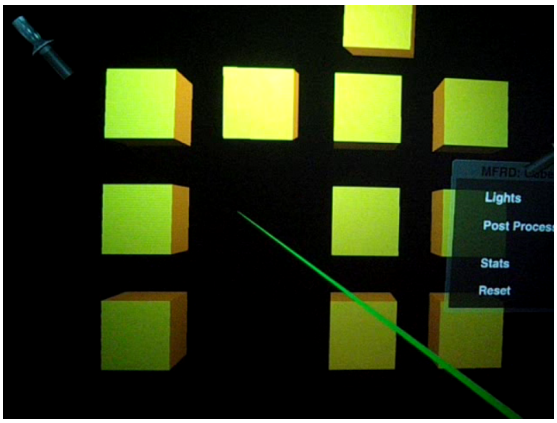
into the frame buffer of the slow client. However, it is slightly more complicated. The two render processes are running fully asynchronous and it is not known which update from the slow client will contain the just released object. In addition, the user might have activated the object again in the meantime, which further complicates the situation.

For dealing with these problems it is necessary to know which objects were actually rendered into the buffers of the slow client that are currently used on the fast client. Thus, the slow client does not need to pass only the frame buffer to the fast client, it also has to provide the list of rendered objects (or the list of missing objects) for that frame buffer. This is easy to achieve by extending the buffer-transfer abstraction described in section 4.3 to include this information. Although this additional information introduces some overhead the buffer sizes for the color and depth values will still determine the data size for buffer transfer

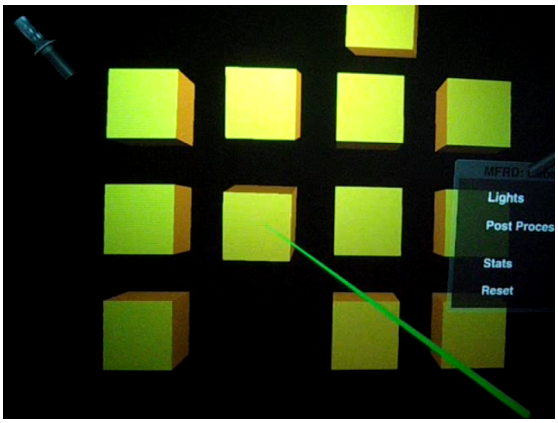


(a) Frame N.

(b) Frame $N + 1$.



(c) Frame $N + 16$.



(d) Frame $N + 26$.

Figure 6.4: Release artifact (user visual). Images show (successive) still frames from a digital video of an prototype application running in naïve multi-frame rate rendering mode; See figure 6.3 for process description.

because for the object ids only a single integer per object must be transmitted. On the fast client the selection and release process for an object then can be described by the state diagram shown in figure 6.5. An object in the scene graph on the fast client can be in one of the three states: SELECTED, OVER, or INACTIVE. In addition, it contains the information if the object in question is contained in the currently used frame buffer from the slow client or not. Thus, there is only a total of six state combinations which may occur.

Table 6.1 considers all possible state combinations and determines if the object needs to be rendered on the fast client or not as indicated by the variable FC. State combinations 1 and 2 represent an object that is in state SELECTED or OVER but no longer contained in the frame buffer of the slow client. Thus, it needs to be rendered on the fast client. State combination 3 defines an INACTIVE object, which is not yet contained in the frame buffer of the slow client. Such an object needs to be rendered on the fast client as well. State combinations 5 and 6 are the most interesting ones. State combination 6 represents a selected object, which

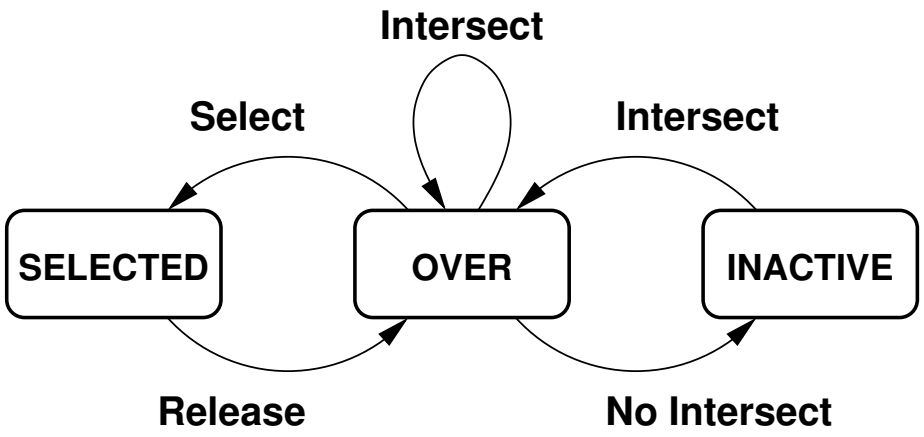


Figure 6.5: State transition diagram for possible object states during an interaction.

is still contained in the frame buffer of the slow client, but the user may have already started to manipulate the object. Thus, it needs to be displayed on the fast client—possibly as a *shadow object* until it is no longer contained in the frame buffer of the slow client. State combination 5 may have several reasons. The user may have just moved the interaction representation over an object, but it is not yet selected and still contained in the frame buffer of the slow client. The fast client would not have to render it, but it would do no harm if it does. The other possibility is that a user just released an object and entered the OVER status while the object was not yet removed from the frame buffer of the slow client due to a very short interaction time or the very slow frame rate of the slow client. In this case the position of the object in the frame buffer of the slow client and the current position on the fast client might be different. Thus, the object needs to be rendered on the fast client—possibly as a shadow object as well. State combination 4 represents an INACTIVE object that is contained in the frame buffer of the slow client, so the fast client does not need to consider that object at all.

Shadow objects are also used in distributed collaborative applications (e. g., [Benford and Mariani 1993]) to show users that a lock for an object has not yet been acquired, but the interaction may have already started. Here, the object is also drawn twice—once at its original location with its regular appearance and once as a shadow object (e. g., as a line drawing). Once the lock is acquired the original object vanishes and the shadow object turns into the object's regular appearance. If the lock cannot be obtained the shadow object vanishes. However, experimentation with shadow objects do not encourage their use for multi-frame rate rendering. Depending on the actual frame-rate ratios between the slow and fast client the above described state handling may generate these problematic cases only sporadically. It might thus be surprising for a user when shadow objects actually appear for a split second while doubly drawn objects remain mostly unnoticed. Shadow objects should be used only cautiously and with an appearance that does not itself further distract the user by creating similar visual effects as the above described migration artifacts in the final image.

#	SC	INACTIVE	OVER	SELECTED	FC
1	0	1	0	0	1
2	0	0	1	0	1
3	0	0	0	1	1
4	1	1	0	0	0
5	1	0	1	0	1*
6	1	0	0	1	1*

Table 6.1: Possible state combinations of an object on the fast client with respect to the state transitions in figure 6.5. The state variable SC denotes that the object is contained in the current frame buffer of the slow client. The resulting state variable FC is set to 1 if the object needs to be rendered on the fast client. Results in the FC column denoted with an asterisk indicate that a shadow object might be drawn on the fast client instead of the regular object appearance.

6.2 Rendering Artifacts

Rendering artifacts for scene-global rendering effects occur in a multi-frame rate setup because of the asynchronous processing of the rendering clients.

Transparency The interaction between transparent and solid objects can be problematic in a multi-frame rate setup. In the case of an optical superposition display this might be negligible because the image composition takes place in the optical path of the output devices. Here, transparent as well as opaque objects will exhibit a transparent appearance because of the incorrect depth occlusion as already discussed in section 3.1. With digital composition correct transparency blending between objects in the slow client's image and the fast client's scene part is only possible if transparent objects are rendered by the fast client. Current graphics hardware requires that objects are rendered back-to-front if correct transparency is to be obtained. The reason for rendering transparent objects on the fast client is that transparent objects blended into the slow client's frame buffer are associated with the depth values of the transparent object. Depth comparison on the fast client will reject fragments for objects on the fast client further away from the viewer without being able to consider the transparent-composite nature of the already existing fragment (cf. figure 6.6). Rendering all transparent objects on the fast client avoids this problem but bears the potential of also creating a performance hit on the fast client. The worst case in this scenario are scenes consisting only of transparent objects such as translucent volumes. In this special case a parallel volume rendering approach is presented in section 7.2 that allows the manipulation of opaque objects within a translucent volume with correct depth occlusion as well as very high interaction fidelity. Nonetheless, transparency remains in general a problem on current graphics hardware.

Scene-Global Effects Multi-frame rate rendering uses a conservative approach for applying head-tracking sensor data to change view transformation. This is achieved by incorporating the information in the slow client only while the fast

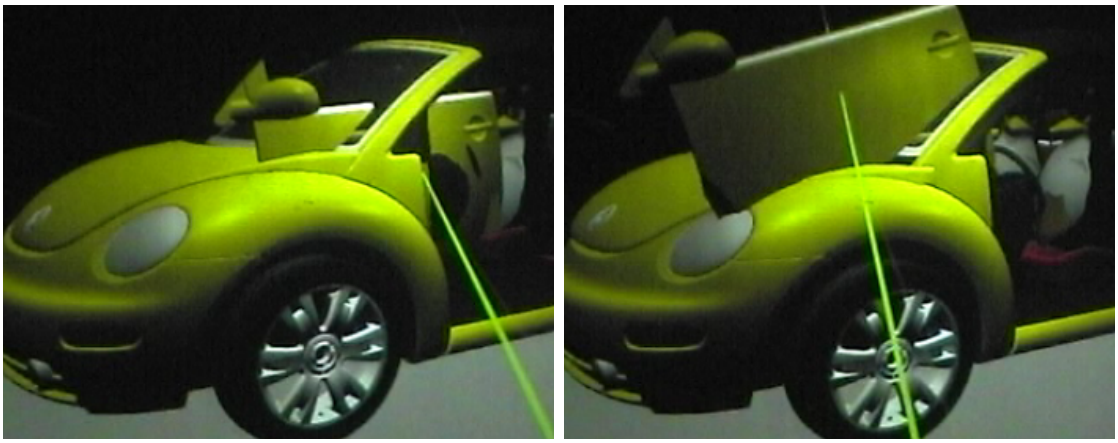


Figure 6.6: Still frames from digital video showing transparency artifacts in a multi-frame rate display using digital composition. The windscreen is a transparent object rendered on the slow client while the door is rendered on the fast client. Note that while the door exhibits correct occlusion with respect to the windscreen it is not correctly transparency-blended.

client receives the view transformation updates from the slow client along with the buffer updates (cf. section 1.3). This means, while the interaction fidelity is improved on the fast client for manipulated objects, the *navigation fidelity* is determined by the update rate of the slow client. However, this approach exhibits a considerable draw back: any scene-global effect that must be recomputed with respect to the viewer's position, such as lighting computations, can be updated only at the same rate as the view position, i. e. at the frame rate of the slow client. Moreover, any changes which affect the whole scene even with a fixed view point must be recomputed at the slow client's frame rate and sent as image and depth information to the fast client (e. g., shadow-map updates caused by moving objects).

A simple way of manipulating a light source is to use a proxy geometry and to apply any manipulation on the light representation to the light abstraction provided by the 3D graphics API. Additional controls such as sliders or dials can be used to manipulate light properties not representable by the geometry proxy (e. g., color values or attenuation function). In the case of multi-frame rate rendering this means that while the user is interacting sufficiently fast with the light representation the actual updates of the changes in light parameters are only incorporated at the slow client's frame rate. This results in two conflicting cues: a fast moving light geometry and a slow moving lighting effect on the whole scene. For this particular problem a technique based on deferred shading is presented in section 7.1. Similar problems such as shadow-map generation can be solved by the same approach. In section 9.2 a combined approach is discussed for improving navigation fidelity which would be suitable to also allow for fast shadow updates and similar applications.

6.3 Summary

Object migration artifacts almost always can be avoided by using prediction and proper state handling. The release artifact can be avoided completely by including the list of object ids for objects contained in the slow client's current frame buffer using the state transition evaluation from table 6.1. The selection artifact is handled by the same state transition evaluation but not completely avoidable. However, distinguishing between an OVER status upon object intersection and a SELECTED status for the actual start of a manipulation operation will on average provide the system with enough time to update the slow client's frame buffer. Selection artifacts will then appear only if the frame-rate ratios between the slow and fast client differ significantly and the user starts interacting with an object immediately after selection. Experience shows that frame-rate ratios need to be in the order of 1:30 or greater to become apparent to the user.

Rendering artifacts in multi-frame rate images result from the asynchronous nature of the image generation process. Transparent objects must be rendered on the fast client to achieve depth-correct transparency blending at the expense of increased load on the fast client. Certain scenarios such as manipulating geometry inside translucent volumes can be handled by specialized techniques as discussed in section 7.2. Scene-global effects need to be handled by the slow client as well as the fast client because the scene geometry resides on both systems. Thus, these effects have to be executed with the slow client's frame rate to be consistent and they may lag behind the cause of the effect, which for example might be a torch moved around by the hand of the user. How these problems can be avoided is discussed in sections 7.1 and 9.2.

Simulation-driven objects, such as animations or objects controlled by dynamics simulations, might also produce artifacts. Managing such entities is often application dependent and in the context of multi-frame rate rendering needs to be analyzed on a case-by-case basis. This is further discussed in section 8.4.

Chapter 7

Advanced Techniques

Basic or naïve multi-frame rate rendering may exhibit rendering artifacts in the presence of scene-global rendering effects. In this chapter two specialized but important techniques are presented: the interactive manipulation of lights and light properties and the interactive manipulation of opaque geometry within large volume data sets.

7.1 Interactive Light Manipulation

In addition to interacting with geometry users often also want to manipulate lights or more specifically objects representing a light to change the lighting in the scene. With multi-frame rate rendering this will however cause problems as already discussed in section 6.2. While interacting with the light, the geometry representing the light is rendered on the fast client providing sufficient interaction speed but the underlying light abstraction of the graphics API is updated only on the slow client. This signals the user two disconnected visual cues: the proxy geometry is updated appropriately but the light's influence on the scene is lagging behind, i. e. the position of the proxy geometry for the light and the light's effect on the scene are inconsistent. With digital composition the final image contains the color and depth values generated by the slow client complemented by the fast client's additional rendering of the interactive scene parts. Because the color base image already contains lit pixel values the lighting stage cannot be moved to the the fast client. However, if the frame buffer image from the slow client would contain the information necessary for shading the pixels, the lighting could be shifted to the fast client. This can be implemented by using a multi-frame rate specific adaptation of deferred shading [Whitted and Weimer 1981; Deering et al. 1988].

Deferred shading is a rendering technique which separates the computation of per-fragment information necessary for shading from the actual shading by storing this information as intermediate results. Thus, deferred shading exhibits only screen-space complexity. While hardware implementations existed for some time (e. g., PixelFlow [Molnar et al. 1992; Lastra et al. 1995; Eyles et al. 1997]), consumer-product GPUs which allow for implementations that provide interactive frame rates became available only recently (e. g., NVIDIA NV40) [Hargreaves and Harris 2004]. Deferred shading is a multi-pass rendering method. The first pass collects relevant per-fragment shading information and writes it to pre-configured

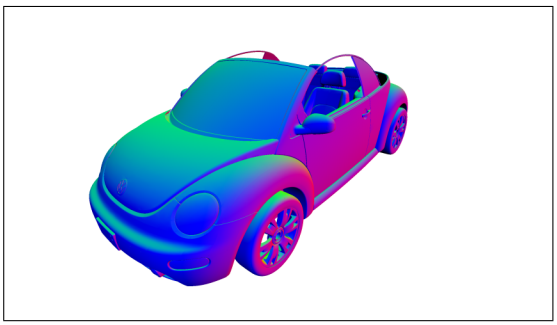
buffers such as *G*-buffers [Saito and Takahashi 1990] or off-screen render targets on current GPUs. The second pass performs the shading on the actually visible fragments.

The separation into two rendering passes using a single-frame rate approach is accomplished by first rendering the scene into multiple off-screen buffers storing per-fragment parameters needed for lighting calculations: position, normal as well as diffuse and specular reflectance values. The second pass performs the shading by evaluating the lighting model for each fragment using the stored per-fragment parameters. For this pass only a full-screen quad is rendered executing the fragment program responsible for the lighting calculations. As a result the lighting calculations are completely decoupled from the geometric complexity of the scene, this yields a linear computational complexity depending on the number of active light sources and the number of pixels. Storing multiple per-fragment attributes generated during the first rendering pass is accomplished by using the `ARB_draw_buffers` OpenGL extension, which provides a mechanism for rendering to multiple off-screen render targets. For deferred shading four-component buffers are used. While 8 bits precision per component suffices for the specification of color and material parameters, using 16 bits per component for the normal vectors yields better results.

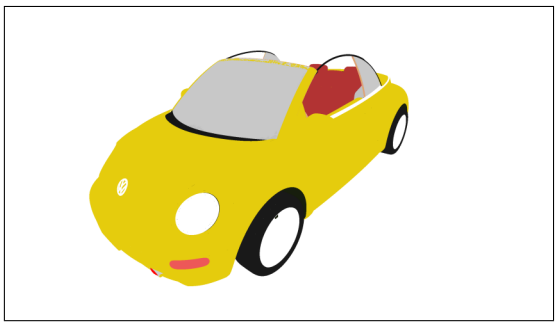
It is easy to see that the two-pass algorithm for deferred shading can be used within a multi-frame rate rendering setup. The geometry pass for the slow clients geometry is run on the slow client while the shading pass is entirely performed on the fast client. Along with the depth values the slow client will also write the interpolated normal as well as diffuse and specular reflectance coefficients of the current material for each fragment. A floating-point buffer with 16 bits per component is used for normal data and for material parameters floating-point buffers with 8 bits per component are employed. These buffers are transferred together with the depth buffer to the fast client (cf. figure 7.1a to 7.1d).

The fast client executes the shading pass using the buffers from the slow client as input textures in a fragment program applied to a full-screen rendered quad. To reconstruct the 3D position of a fragment its window position and depth value are used. Interactive geometries are then rendered afterward on top of this base image (cf. figure 7.1e). Alternatively, the interactive geometry may also be rendered into the render-target buffers from the slow client. This allows for a unified lighting shader at the end of the fast client's render process.

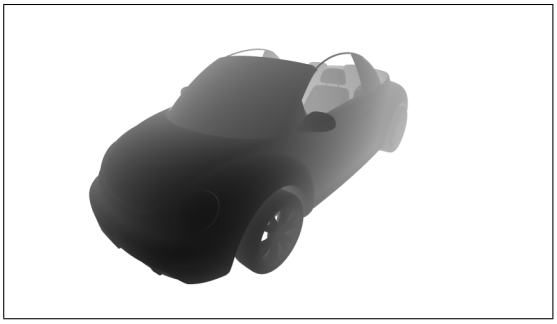
Multi-frame rate deferred shading allows for visually correct user interaction with light representations and lighting parameters at interactive frame rates even though these manipulations affect the entire scene. However, there is also a cost involved: more buffers of larger size have to be transferred from the slow client to the fast client. For the original multi-frame rate rendering method using digital composition 64 bits per pixel are necessary: 32 bits color and 32 bits depth information. The minimum bit size per pixel for deferred shading is rather 128 bits: 32 bits diffuse color, 32 bits depth, and 64 bits normal (i. e. 16 bits per component). To increase the numeric precision the normal values might be even stored as 32 bit values per



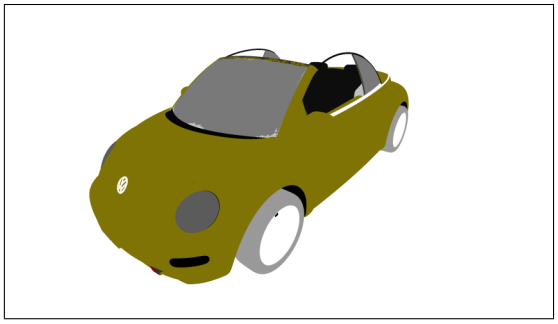
(a) Normal buffer on SC.



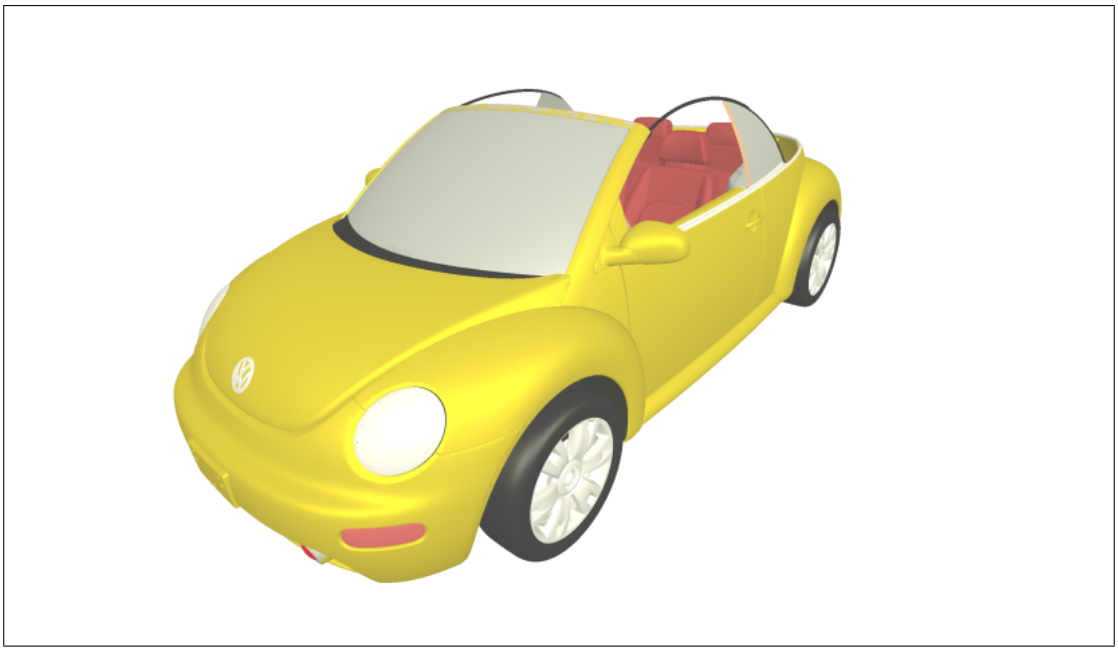
(b) Diffuse buffer on SC.



(c) Depth buffer on SC.



(d) Specular buffer on SC.



(e) Final shading on FC.

Figure 7.1: Deferred shading in a multi-frame rate rendering setup. (a) to (d) show the content of the slow client's render targets holding normal, diffuse, depth, and specular values, respectively. (e) shows the final shading using the content of the buffers from the slow client together with the currently active lights as computed on the fast client. The render clients were each running on an NVIDIA Quadro FX 5600 using a resolution of 1600×1200 pixels. Frame times were 80 ms for the slow and 6 ms for the fast client. Four point lights were enabled on the fast client.

component instead of 16 bit which increases the 64 bits per pixel buffer size for the normals to 128 bits.

Beside the depth and normal buffer generated by the slow client the remaining buffers contain material descriptions. Instead of actually storing the material parameters, it is much more efficient to only store material indices on a per fragment basis on the slow client. The material palette and the stored indices are then used on the fast client to look up the actual material parameters during the shading pass. This reduces the buffer transfer overhead to only depth, normal and material index information.

In addition to the increased buffer sizes of multi-frame rate deferred shading approach also computational load is shifted from the slow client to the fast client. In the naïve multi-frame rate rendering approach the slow client would not only have to transform and rasterize potentially large sets of geometry but also compute shading for the resulting fragments. With deferred shading these calculations are moved completely to the fast client. Even though the complexity is limited by the screen resolution, the actual shading programs might implement an expensive lighting model. This can decrease the frame rate on the fast client more than desirable. It is however possible to balance the cost for the lighting model by decomposing the lighting process into currently manipulated active lights and static lights, which are not manipulated by the user at the moment. The shading for non-interactive static lights can then again be computed on the slow client leaving only the active lights for processing on the fast client. This approach bears the cost of requiring an additional render target buffer, containing a pre-shaded image of the non-interactive lights, on the slow client which must also be transferred to the fast client.

For scenes that do not cover the screen completely special care is required for non-covered background regions. The simplest way to avoid evaluation of all fragments in the full-screen pass on the fast client is to create stencils for fragments that pass the fragment stage on the the slow client. This stencil buffer can be combined with the depth buffer using the `EXT_packed_depth_stencil` OpenGL extension; resulting in no further increase in buffer size. On the fast client only screen fragments passing the stencil test will then be subject to the desired shading in the fragment processor stage. While this reduces computational load for medium filled screens it does not improve situations where the screen is filled completely with pixels from geometry (e. g., interior scenes). Alternatively, the same effect can be achieved using the buffer containing the normal values. On the slow client the buffers for multiple render targets are (usually) cleared to zero before being written to. On the fast client any fragment for which the look-up into the normal buffer produces a vector whose norm is less than or equal to zero can be skipped.

Figure 7.2 shows digital photographs of an application prototype using multi-frame rate deferred shading. In this particular application the buffers from the slow client are used to (re-)create the static scene parts while for the interactive parts a conventional Phong [Phong 1975] shader was used. Note that no difference can be detected between the shading methods used on the buffers from the slow client and the interactive scene parts.

(a) (b)

Figure 7.2: Digital photograph of a software prototype using multi-frame rate deferred shading. (b) slow client parts emphasized in gray scale.

7.2 Multi-Frame Rate Volume Ray Casting

Many application areas require high-quality visualization of large volume data sets. Especially in the oil & gas industry interaction fidelity for the manipulation of geometry such as a well path inside a semi-transparent volume is necessary. However, single-frame rate approaches are currently not capable to support high-quality volume rendering for interactive applications because only limited interaction fidelity is provided.

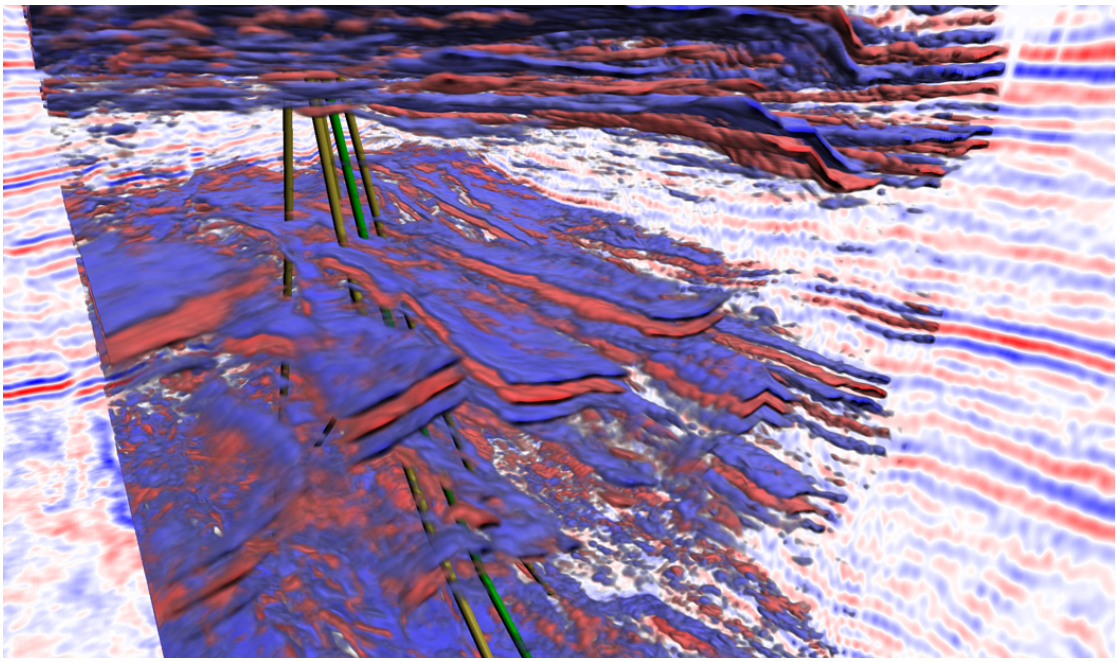
With multi-frame rate rendering such a scenario could be trivially decomposed into non-interactive parts containing the volume representation and interactive parts containing the well geometry. While this would accomplish the desired behavior of the slow client carrying the volume rendering load and the fast client providing high frame rates for the interaction, the final image would suffer from incorrect occlusions which is unacceptable in most scenarios. With the digital composition of the slow client's image and the fast client's render process transparency information cannot be correctly incorporated. The color buffer of the slow client contains an already finished image while transparency requires strict back-to-front rendering. Executing a rendering pass on the slow client's color buffer using the also available depth information does not provide any information about intermediate transparency values before accumulation in the final composition stage (cf. section 6.2). This means that interactive geometries on the fast client can appear correctly only in front of but never inside or behind the translucent volume. To ensure correct occlusion the volume would have to be drawn after all opaque geometries, thus shifting the computational load of the volume rendering to the fast client and diminishing all performance advantages.

The solution presented here allows for correct blending of the interactive geometry with the translucent volume by detecting screen-space portions where the volume potentially overlays this geometry and redrawing only these portions on the fast

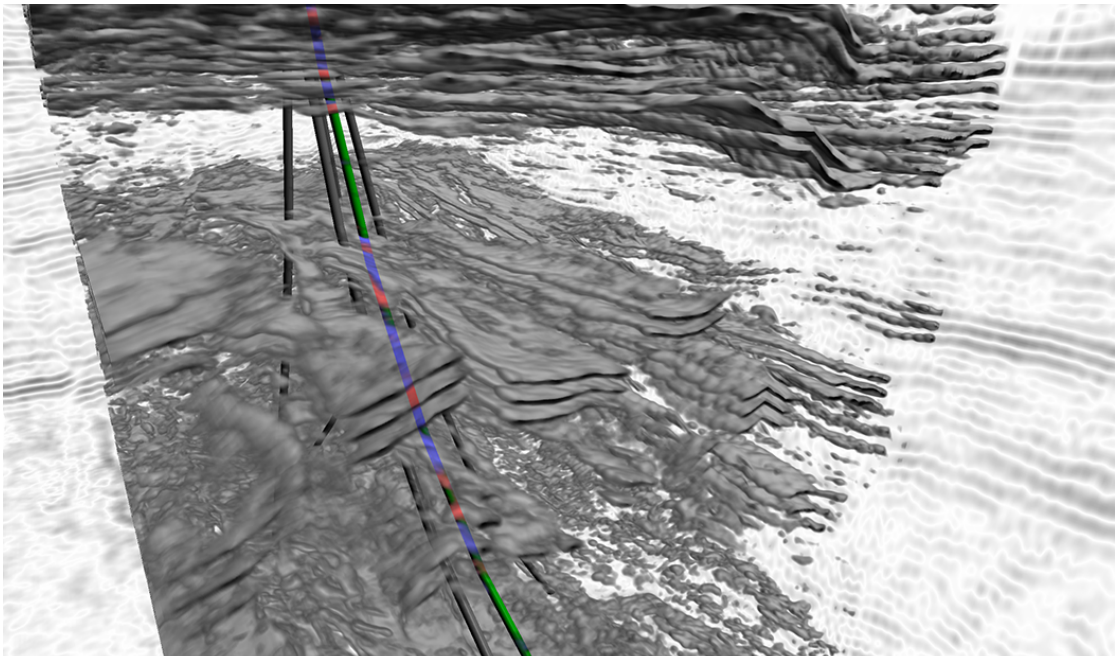
client. Usually only small parts of the volume actually do overlay manipulated objects (e. g., geometry in a well planning task consists only of thin tube structures as can be seen in figure 7.3). For this reason the volume rendering load on the fast client is significantly lower compared to rendering the full volume on the slow client. The approach guarantees that exactly the screen-space fragments covered by the interactive geometry trigger the volume-rendering algorithm, thus preventing unnecessary work on the fast client.

Volume rendering [Drebin et al. 1988] is an field of active research for some time now. 3D texture-based direct volume visualization methods such as those by Cullip and Neumann [1993] sample the volume data by using a stack of typically viewport-aligned slice planes as proxy geometry. These planes are then blended into the frame buffer in front-to-back or back-to-front order. As an object-order method the density and therefore the amount of slice planes that must be generated and rendered depends directly on the data complexity as well as the desired output quality. Volume ray casting [Levoy 1990] samples the volume data at discrete locations along rays originating from the viewer position. For each pixel a ray is generated and traversed through the volume allowing direct evaluation of the volume rendering integral. Implementations of volume ray casting techniques on the GPU emerged in recent years. Early GPU-based volume ray casting approaches [Roettger et al. 2003; Krüger and Westermann 2003] were dependent on multiple render passes and temporary image buffers for storing intermediate results. Later methods [Stegmaier et al. 2005] show how to implement the complete volume ray casting algorithm in a single fragment program by employing enhanced programmability features of recent GPU architectures (e. g., NVIDIA NV40). These approaches are not directly aware of prior frame-buffer content created from regular geometry, which poses problems for proper image composition. Correct volume clipping at geometry boundaries using depth information of previously rendered geometry has been demonstrated only by [NVIDIA SDK; Engel et al. 2006]. The main advantage of GPU-based volume ray-casting techniques over slice-based volume rendering is the independent ray traversal, allowing the implementation of advanced acceleration techniques such as early ray termination in a straightforward way. For reasonably small data sets slice-based volume rendering as well as volume ray-casting approaches achieve interactive frame rates with reasonable output quality using recent graphics hardware, i. e. NVIDIA G80 or newer. However, multi-gigabyte data sets as used in the oil & gas industry still suffer from low frame rates making the precise placement of objects inside the volume difficult.

Stencil testing is used to prevent screen-space fragments from being processed. OpenGL [Segal and Akeley 2006] specifies stencil tests to be performed after the fragment processing stage. However, current graphics hardware does perform an early stencil test prior to this stage, rejecting fragments before entering the fragment processor and thus circumventing potentially expensive calculations. The required stencil mask is generated while rendering the active geometry on the fast client. This rendering pass is executed with the depth buffer content received from the slow client, so the mask spans precisely the visible fragments of the interactive geometry. While the frame-buffer image from the slow client contains the color base

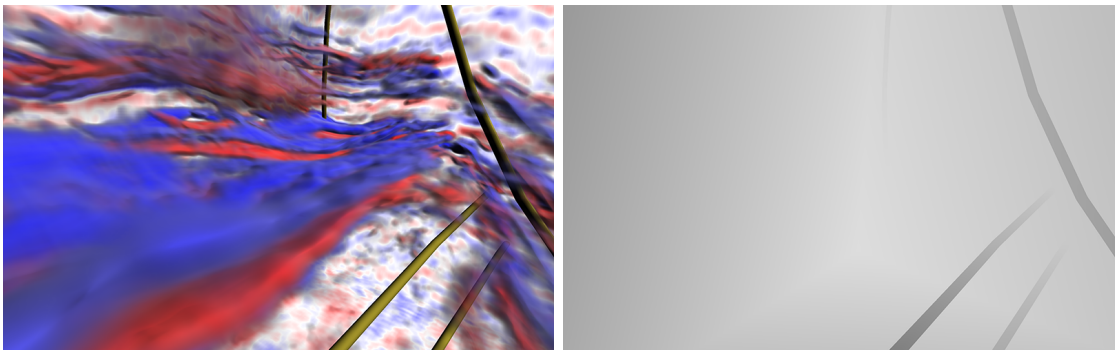


(a) Final image with manipulated well (green).



(b) Contribution of the slow (gray scale) and the fast client (color).

Figure 7.3: Volume rendering application using multi-frame rate rendering running on a single multi-GPU system. (a) is a screenshot of the fast client showing the result of the digital composition at a resolution of 1600×1200 pixels. The $1024 \times 439 \times 734$ volume data set was rendered with volume ray casting using 2048 samples per ray. The fast client renders only the screen area in front of the manipulated well (green), while the slow client renders the whole volume. (b) shows the contribution of slow (gray scale) and fast client (color). Each client was running on an NVIDIA Quadro FX 5600 at a frame rate of 1 Hz and 30 Hz, respectively.



(a) Color buffer on SC. **(b)** Depth buffer on SC.

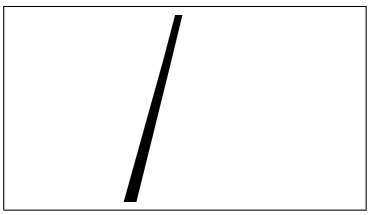
Figure 7.4: Volume rendering in a multi-frame rate rendering setup, slow client part.

image of the non-interactive geometry combined with the fully rendered volume (cf. figure 7.4a), the depth image from the slow client must contain only values generated by non-interactive geometry to allow correct clipping with geometry rendered on the fast client (cf. figure 7.4b). Subsequently executing the volume rendering algorithm on the fast client with stencil testing enabled will perform image updates in exactly those screen regions where the volume is actually covering interactive geometry.

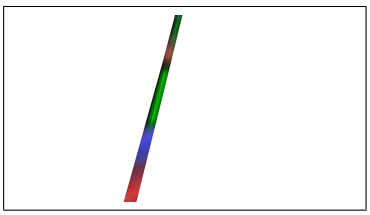
The volume visualization technique employed in this work is similar to the ray casting by Stegmaier et al. [2005]. By rendering only the bounding box of the volume data set a single fragment-shader program is executed which implements the complete algorithm. Thus, the stencil test discards complete rays instead of just slice fragments as it would be the case for slice-based volume rendering, which renders a large stack of viewport-aligned polygons, making the stencil test much less efficient. The basic volume ray-casting algorithm determines the ray entry and exit positions for the volume analytically. It is unaware of the current frame-buffer content and thus prevents correct interaction of the volume with contained geometry. To achieve correct clipping behavior the shader program needs to also access depth information of previously rendered geometry. Therefore, a two-pass approach is applied as described by Engel et al. [2006, 11.4]. The first pass renders the entire scene geometry into off-screen buffers holding depth and color information. On the fast client a stencil buffer is added in this pass simultaneously generating the stencil mask. The subsequent volume rendering pass uses the depth buffer as an input texture in order to terminate ray traversal if scene geometry is hit.

The work flow for multi-frame rate volume rendering on the slow client and the fast client is quite similar. Special care must to be taken on the slow client when to read back the frame-buffer image components. The depth values must only contain the depth information of the geometry rendering pass. Thus, the depth buffer content is read directly after processing all scene geometry, preventing the volume rendering from modifying depth values. After receiving the frame-buffer image on the fast client the geometry pass on the fast client is used to generate the stencil

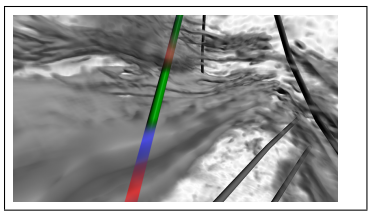
mask for the interactive geometry with respect to the depth values from the slow client. Afterward the volume rendering is updated for a subset of the screen-space fragments, i. e. for fragments created by rendering the interactive geometry and which passed the stencil test. Figure 7.5a shows a stencil mask generated on the fast client for the interactive geometry of the selected well object located inside the semi-transparent volume. As illustrated in figure 7.5b the actual scene contribution from the fast client is generated based on this mask. Figure 7.5c shows the digital composition of the slow and the fast client's contributions, emphasizing the scene part generated on the slow client using gray scale. The final image composite as seen by the user is shown in figure 7.5d.



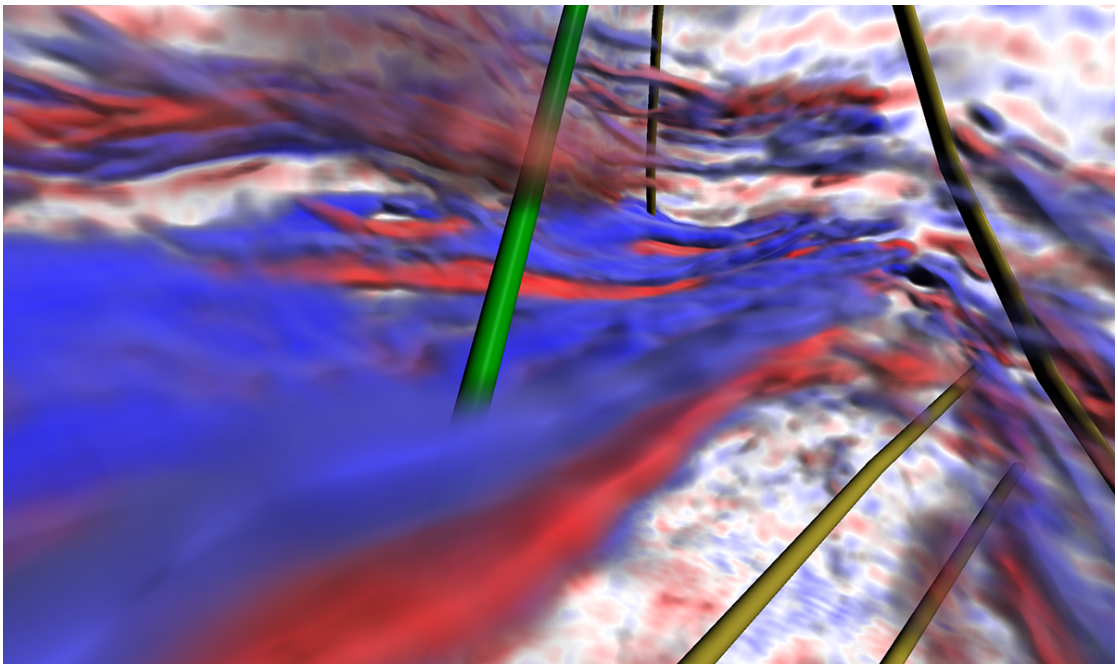
(a) Stencil buffer on FC.



(b) Color buffer on FC.



(c) Digital composition.



(d) Final rendering on FC.

Figure 7.5: Volume rendering in a multi-frame rate rendering setup, fast client part. (a) shows the stencil mask generated by the active geometry on the fast client while (b) shows the final image contribution from the fast client. (c) shows the composition of the contributions from the slow (gray scale) and fast client (color). (d) shows the final image as seen by the user. The images show a section of a $1024 \times 439 \times 734$ data set rendered with volume ray casting using 2048 samples per ray. The clients were each running on an NVIDIA Quadro FX 5600 with a frame rate of 1 Hz and 30 Hz, respectively, at a screen resolution of 1600×1200 pixels.

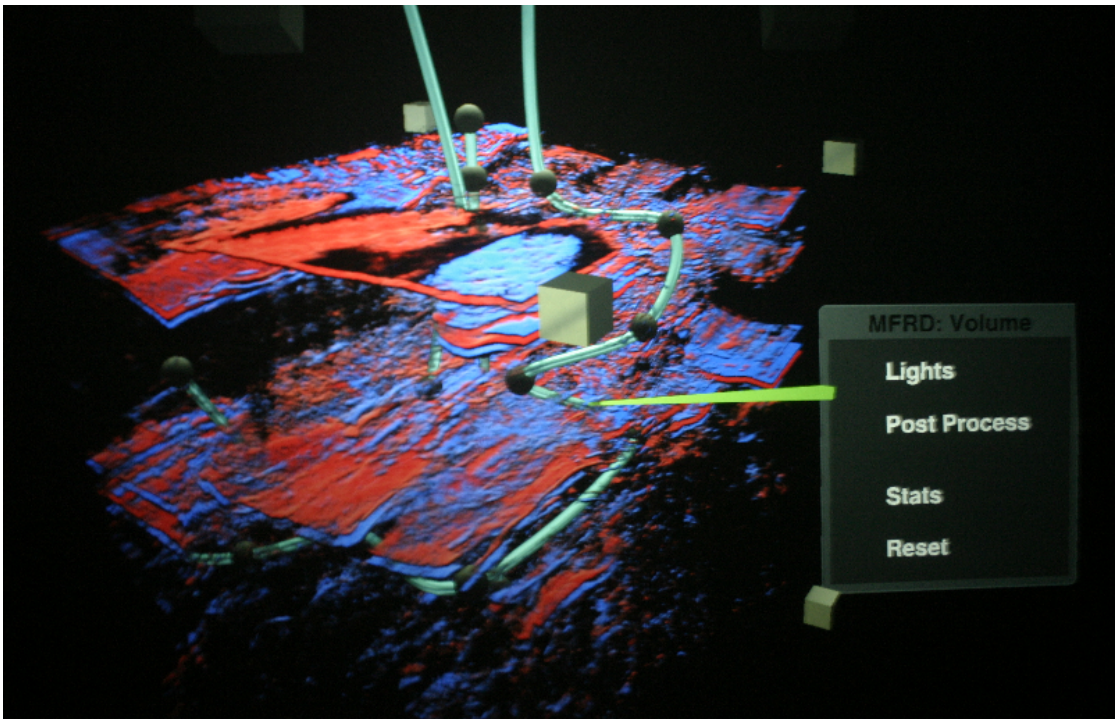


Figure 7.6: Digital photograph of a prototype using multi-frame rate volume ray casting for interactive manipulation of opaque geometry.

Combining GPU-based volume ray casting with stencil testing on the fast client permits smooth interaction with scene geometry placed inside semi-transparent volume data. However, the efficiency of this approach depends on the size of the screen projection and the penetration depth of the interactive geometry within the volume. The projection size defines the number of rays generated and traversed through the volume. The penetration depth determines the maximum number of samples taken along the ray. The computational cost for traversing a ray through the volume is at most linear. Optimizations such as early ray termination may reduce the number of samples that must be processed.

The presented multi-frame rate volume rendering approach is based upon the same frame-buffer image as the original digital composition technique (cf. section 3.2). However, depth and color values are read back at different points in time on the slow client. An additional pipeline stall is introduced here for reading the content of the depth buffer after the geometry rendering pass. This stall can be prevented by using an asynchronous read-back mechanism (e. g., OpenGL's `ARB_pixel_buffer_object` extension). This way the volume rendering would not be delayed while the depth values of the geometry-rendering pass are transferred to host memory.

Figure 7.6 shows a digital photograph of a multi-frame rate software prototype featuring parallel volume ray casting and interactive manipulation of well-path geometries.

7.3 Summary

Multi-frame rate rendering using a parallel deferred shading approach allows to change light and material properties at interactive frame rates. This enables the direct and interactive manipulation of scene properties influencing the whole scene. A parallel volume rendering technique combined with multi-frame rate rendering allows for the interactive handling and manipulation of opaque geometry within very high-quality volume rendering with depth-correct occlusion. These advanced techniques show the potential of multi-frame rate rendering in combination with high-quality visualization techniques.

Chapter 8

Analysis and Discussion

In this chapter observations from prototype implementations and experimental results are presented. Buffer-transfer methods used by the digital composition method are discussed and end-to-end latency for relevant multi-frame rate rendering setups are analyzed. The results of a user study comparing conventional single-frame rate rendering with multi-frame rate methods are presented. Finally, limitations of the multi-frame rate rendering approach are discussed.

8.1 Buffer-Transfer Methods

Buffer transfer is an integral part of multi-frame rate rendering using digital composition. If a graphics cluster is used the natural choice of the buffer-transfer medium is a network connection (cf. section 5.1). In a single-system multi-GPU setup more choices are available.

There are several ways to transfer frame-buffer content from the slow to the fast client in a single-system multi-GPU setup. First, existing infrastructure for multi-frame rate rendering in a graphics cluster can be reused. Instead of assigning the slow client and the fast client to different cluster machines both client processes are executed on the same machine but use different graphics devices. Frame-buffer transfer can then be implemented via the local loopback device. Second, instead of using the local loopback device process-shared memory can be employed. Third, depending on the scene-graph API involved it is also possible to use process-local memory together with a multi-threaded application architecture.

Software prototypes based on multi-frame rate techniques were developed using three different scene-graph APIs. Avango [Tramberend 1999, 2003], which is based on OpenGL|Performer [Rohlf and Helman 1994], OpenSG [Reiners 2002], and a prototype development of a minimal scene-graph implementation. For the purpose of this discussion the structural difference between these APIs is their support for asynchronous draw traversals. While OpenGL|Performer supports multiple image generators using a multi-process aware buffer-transfer approach the actual scene-graph traversal cannot be separated in an asynchronous way; there is only one application traversal per application instance which internally synchronizes the available cull and draw processes [Rose Clay 1996]. In this case separate application instances must be used in a single-system multi-GPU configuration that communicate via external resources (e. g., network or process-shared memory). OpenSG does support distributed rendering in a PC cluster as well as concurrent

render traversals of the scene using multiple threads. However, the implementation of various render strategies for multi-GPU multi-frame rate rendering (e. g., naïve multi-frame rate rendering as well as multi-frame rate deferred shading) turned out to be non-trivial with OpenSG because it is an uncommon use case for a scene-graph API. Thus, a minimal scene-graph prototype allowing for asynchronously running graphics contexts was developed. For both OpenSG and the scene-graph prototype process-local memory was used to share resources and communicate results between the participating graphics contexts.

Network API Sending frame buffer content from the slow to the fast client via the local loopback device requires the use of a networking API usually based on sockets [Stevens 1997]. A raw byte buffer representation of the frame buffer is created that additionally contains management information such as overall buffer size, frame counter, and the logical graphics buffer (cf. section 4.3). This buffer will be passed to the network API for sending. The network API will first copy the buffer internally and then split it into smaller parts that are actually send over the physical network. Because the actual send operation takes some time, it is important to decouple reading from the graphics device and sending with the network API. Otherwise the send and flush operations of frame-buffer-sized buffers would stall the draw process and further delay the slow client. Transfer buffer reception from the slow client must also be decoupled from buffer insertion in the draw process on the fast client. In the case of receiving multiple buffers comprising left and right view in a stereo setup, all buffers for a certain frame number must be available before they can be uploaded to the graphics device. An additional increase of transfer latency can be observed for sending multiple buffers on the same network.

Because the local loopback device does not employ networking hardware a speedup compared to using a Gigabit Ethernet device was expected. However, the network API's internal buffer copy for full-screen-sized frame buffers is still a limiting factor. With the available hardware and operating system, i. e. standard PC hardware and GNU Linux kernel series 2.6.20, only a bandwidth of 2 Gb/s was experienced for the local loopback device. Theoretically, the bandwidth should be limited only by the memory bandwidth of the PC in use, approximately 6 Gb/s for the actual hardware platform, because loopback device transfer is essentially the same as copying buffers from one location in local host memory to another. This unexpected limiting behavior maybe lifted with newer hardware or software version.

Shared-Memory API To reduce the latency between reading the frame buffer on the slow client and uploading it on the fast client process-shared memory was used. The implementation uses a copy mechanism by reusing the infrastructure from the networking API and adding custom copy functionality. Unfortunately the achieved bandwidth was only slightly better than the bandwidth obtained via local loopback device. This indicates that bandwidth sharing took place between the participating processes, limiting the copy mechanism in the same way as the loopback device.

The copying of frame buffers is not an option even on shared-memory machines. For this reason a zero-copy mechanism was employed—at least as far as main memory is concerned. There is currently no implementation of true shared memory between GPUs available and there is also no API available for direct buffer transfers from one GPU to another using a vendor-specific hardware channel; such inter-GPU transfer is only available for special vendor-specific modes (e. g., NVIDIA SLI [NVIDIA GPU Programming Guide 2006; NVIDIA SLI]; cf. section 5.2). Instead, process-shared memory was used, which is a unique location in machine-local memory that is mapped to a virtual address within the processes attaching to the shared-memory segment. To realize buffer updates only pointer values to the current buffer in the shared-memory segment must be adjusted. The implementation uses a triple-buffer approach so as to minimize the impact on the render processes involved. One buffer is assigned to the slow client for writing its rendering results. Another buffer is used by the fast client for reading into its frame buffer. The third buffer is a transfer buffer, which is either outdated after a swap of the fast client or it contains the most recently read frame buffer from the slow client. Once the slow client has read its frame-buffer content, a lock is entered and pointers to the slow client's buffer and the currently unused transfer buffer are swapped. A flag in shared memory is used to indicate that a new buffer is available. The fast client is polling this flag and, on finding it set, will enter the same lock to swap the pointer to the transfer buffer with its own buffer used for holding frame-buffer content to be uploaded into its GPU. Before exiting the lock the flag is reset. The prototype implementation was realized using the Boost.Interprocess API [Boost.Interprocess Library] which allows for the management of shared-memory segments, the placement of C++ container abstractions inside it, fully supporting standard container operations, and the placement of synchronization mechanisms (e. g., locks and condition variables) within the shared-memory segment itself. The overhead for the described triple-buffer scheme is nearly unnoticeable, lock retention usually less than 0.1 ms; so the latency for sending buffers from the slow to the fast client practically drops to the actual render time of the slow client plus the time needed for download and upload of the buffers from and to the respective GPU. For scene-graph APIs capable of asynchronous render traversals, such as OpenSG and the minimal scene-graph implementation, the same mechanism can be used. However, because only one process-local address space must be managed the implementation becomes simpler. Buffer management here uses heap memory instead of shared memory avoiding the use of a separate memory-management API. As expected the latency for buffer transfer from the slow to the fast client in this case also drops to the actual render time of the slow client (plus the time needed for download and upload of the buffers from and to the respective GPU). Note that the triple-buffer mechanism for zero-copy buffer transfer does not depend on any particular 3D rendering API because only basic operating-system functionality is required.

Summary It has been shown that there are several ways to implement multi-frame rate rendering using digital composition on a single-system multi-GPU setup. The

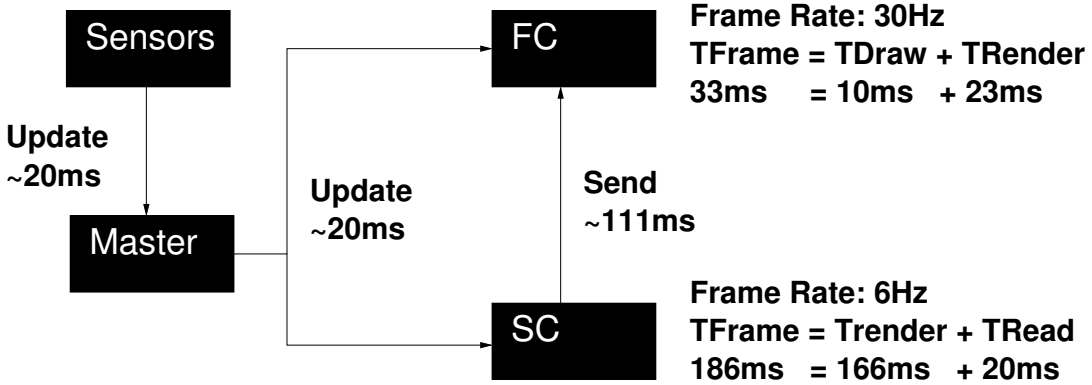


Figure 8.1: End-to-end latency for multi-frame rate rendering using digital composition. **TDraw** is the time for uploading color and depth values on the fast client while **TRead** is the time for reading color and depth data from the graphics device on the slow client. **TRender** is the time for rendering the scene on either client and **TFrame** is the final frame time.

methods vary in their efficiency. However, the zero-copy mechanism is clearly the method of choice because swapping pointers to buffers provides the lowest latency for buffer transfer. Practical experiences with the prototype implementations also showed that multi-frame rate rendering not only works on a multi-GPU setup but that software infrastructure can be build that allows for selecting the best communication and transfer path for specific application scenarios as well as a variety of hardware configurations.

8.2 End-to-End Latency Analysis

System response is crucial for interaction fidelity as discussed in section 1.2. Multi-frame rate rendering improves interaction fidelity by separating frame rates for interaction-relevant parts of a scene and the non-interactive rest. However, this separation also changes the composition of the system-response time. Because the actual frame rate of the slow client does not affect the interaction fidelity of the fast client it is necessary to investigate the end-to-end latency of an interaction event and relate this to the case of conventional single-frame rate rendering.

In the following end-to-end latency is analyzed for conventional single-frame rate rendering and multi-frame rate rendering using digital composition. A screen resolution of 1280×1024 pixels is assumed. Also assumed are a frame rate of 6 Hz for the slow client and 30 Hz for the fast client (cf. figure 8.1). In all cases sensor data and event updates are received with an assumed latency of 40 ms by participating clients from an externally running tracking system sending new values at a rate of 50 Hz, i. e. every 20 ms.

Conventional Single-Frame Rate Rendering The end-to-end latency in a conventional single-frame rate rendering setup consists of the sensor-update latency and the frame time of the render client, which is roughly the same as the render

		1280×1024		1600×1200	
		MB/s	ms	MB/s	ms
Read	RGBA	521	10.1	684	11.6
	BGRA_EXT	997	5.3	939	8.4
	DEPTH	565	9.3	733	10.8
Draw	RGBA	1298	4.0	1412	5.6
	BGRA_EXT	2081	2.5	2166	3.6
	DEPTH	1213	4.3	1372	5.7

Table 8.1: Timings for reading from and writing to the graphics device (NVIDIA GeForce 8800 GTX, driver rev. 97.46, ASUS P5N32-E SLI based system). Note that read/write performance of color data is heavily format dependent.

time of the slow client (TRender in figure 8.1). In the example this is 206 ms if event updates arrive at frame start which is assumed for the following discussion. In the worst case the time for one tracking frame (20 ms) needs to be added.

Cluster-Based Multi-Frame Rate Rendering In a cluster-based multi-frame rate rendering setup using digital composition sensor data is also received with a latency of 40 ms. On the fast client the frame time is split into the time needed for uploading the frame buffer from the slow client (TDraw in figure 8.1) and the time for rendering the relevant scene parts for the current interaction (TRender in figure 8.1). TDraw is the accumulated time for uploading color and depth values at a certain resolution. Table 8.1 indicates TDraw to be ≤ 10 ms for an image size of 1280×1024 pixels with a current graphics device using the PCIe bus system. This amounts to an end-to-end latency of 73 ms for the fast client. On the slow client the frame time consists of the time needed for rendering the relevant scene parts (TRender in figure 8.1) and the time for downloading the frame-buffer image (TRead in figure 8.1). TRead, like TDraw, is the accumulated time for reading color and depth data from the graphics device as shown in the upper part of table 8.1. TRead is then ≤ 20 ms. Additionally, the just read frame-buffer image must be transferred over the network to the fast client. For a resolution of 1280×1024 pixels this takes $\approx$111 ms as can be seen in table 8.2. The final end-to-end latency for an event to be incorporated into the frame-buffer image of the slow client, transferring the frame buffer to the fast client, and uploading it to the fast client's graphics card is

Resolution 64 Bits Color/Depth	Buffer Size	Transfer Time	Transfer Rate
1024×768	6 MB	66 ms	15 Hz
1280×1024	10 MB	111 ms	9 Hz
1600×1200	15 MB	166 ms	6 Hz

Table 8.2: Network transfer times at different buffer sizes for Gigabit Ethernet (observed for application level end-to-end buffer send and receive on a Cisco Catalyst 3560G switch).

	Conventional		FC		$SC_{Cluster}$		$SC_{multi\text{-}GPU}$	
	best	worst	best	worst	best	worst	best	worst
Sensor Update	40	60	40	60	40	60	40	60
$TFrame_{Conv.}$	166	166						
$TFrame_{SC}$					186	186	186	186
Network Transfer					111	111		
$TFrame_{FC}$			33	33		33		33
$\sum$	206	226	73	93	337	390	226	279

Table 8.3: End-to-end latency values for conventional single-frame rate rendering and multi-frame rate rendering using digital composition in a graphics cluster and a multi-GPU setup; all times given with respect to figure 8.1 and expressed in milliseconds.

then 337 ms, or 390 ms at most considering an additional lag of 20 ms for sensor data as well as an additional render frame for the fast client of 33 ms. However, the end-to-end latency relevant for interaction is the end-to-end latency on the fast client. This is only 73 ms compared to 206 ms for the conventional single-frame rate rendering case.

Multi-GPU Multi-Frame Rate Rendering End-to-end latency for event updates on the fast client remains the same for multi-frame rate rendering using digital composition in a multi-GPU setup compared to a cluster-based setup, i. e. 73 ms, by assuming that a master application is still being executed on a separate machine. Only the fast client and the slow client are running on a single machine. Because buffer transfer from the slow client to the fast client is realized by a zero-copy approach, as explained in section 8.1, network-transfer latency is avoided. In the example the event-update latency on the slow client decreases in a multi-GPU setup to 226 ms for the ideal case, 279 ms for the worst case, from 337 ms and 390 ms for best case and worst case, respectively, in a cluster-based setup.

Comparison In table 8.3 the best and worst case values for end-to-end latency for analyzed cases of the above example are summarized. It can be seen that end-to-end latency differs at least with a factor of three between conventional single-frame rate rendering (206/226 ms) and the fast client in a multi-frame rate rendering setup (73/93 ms). This means that the fast client responsible for rendering interaction responses does not only run at a higher frame rate, which improves interaction fidelity, it also incorporates interaction events with less latency than single-frame rate rendering. This double advantage provides a clear benefit for multi-frame rate rendering in practice.

The bandwidth that can be achieved on current graphics systems and network setups differs by at least one order of magnitude. Gigabit Ethernet provides an approximate maximal bandwidth of 90 MB/s in practical experience for application-level usage. Thus, sending a buffer of 10 MB, i. e. $1280 \times 1024 \times 8$ bytes (4 bytes color and 4 bytes depth), takes $\approx$111 ms while reading and writing the same buffer from and to graphics hardware only takes 20 ms and 10 ms, respectively. In a

stereoscopic setup, where two complete frame buffers must be sent per frame, these times would double. Clearly network transfer is the limiting factor here. A multi-GPU system taking the roles of both the slow and the fast client reduces the end-to-end latency of frame-buffer image updates from the slow client to the fast client from 337/390 ms to 226/279 ms because only reading and writing of image data on the same host memory is necessary. Including the sensor and event-update processes into the same machine will reduce latency even further. It is noteworthy that the host system used for a multi-GPU setup should nonetheless provide enough compute power. Early experiments on systems with a dual-core CPU showed an excessive amount of compute contention between the processes or threads participating in the setup. Using a machine with a quad-core CPU setup exhibited a significantly better balanced system load.

8.3 User Study

Demonstrations of multi-frame rate rendering to colleagues and visitors generally produced a positive feedback. In order to gather more objective data in contrast to subjective observations a user study was carried out to assess the method's influence on user performance.

Hypothesis The hypothesis of the user study was that the improved interaction fidelity of multi-frame rate rendering techniques facilitates common selection and object manipulation tasks in virtual environments and thus leads to an improved task performance. While there are many parameters that can be studied with this setup, the focus was a head-tracked 3D docking task running at 10 Hz on the the slow client and at 30 Hz on the the fast client. Four methods were compared: basic multi-frame rate rendering by optical superposition ($\mathsf{MF}_{\mathsf{opt}}$), basic multi-frame rate rendering by digital composition ($\mathsf{MF}_{\mathsf{dig}}$), conventional single-frame rate rendering for the whole scene at 10 Hz (SF_{10}), and at 30 Hz (SF_{30}). The SF_{10} scenario served as a lower baseline. Because interaction was to be performed at 30 Hz, while head tracking and rendering the rest of the scene at 10 Hz in the multi-frame rate configurations ($\mathsf{MF}_{\mathsf{opt}}$ and $\mathsf{MF}_{\mathsf{dig}}$), the upper baseline is provided by SF_{30}. The optical superposition technique supporting LOD-based depth testing was excluded from the study because the fast object manipulation during a docking task would strongly pronounce the artifacts of this technique. A scene was selected that could be rendered at 30 Hz on a single graphics card and the frame rate was limited appropriately for the different techniques.

Experimental Setup The study was comprised of 16 individuals, who were all daily users of computer technology and most of them had worked with 3D graphics before. All participants had stereo vision capabilities and could interact with stereoscopically displayed objects which were positioned in front of the screen.

The 3D docking task required participants to select a small ball-shaped object (3 cm diameter) from a random location on one side of the screen and move it to the other side, where the object had to be dropped off in a ring-shaped target

Figure 8.2: Display setup used during the user study.

location, as can be seen in figures 8.2 and 8.3. The pick-up and drop-off location were randomly chosen from a set of predefined pairs of positions such that the distance between the positions in any pair was always equal. Additionally, the ball-shaped object at the pick-up location was occluded if viewed from the center position in front of the screen to enforce the use of head tracking. Selection required some head movement toward the appropriate side of the screen, followed by further head movement toward the drop-off location. The task was performed with an optically tracked input device using a ray-casting metaphor and required three degrees of freedom. The orientation of the manipulated object was not considered. The displayed image was restricted to 1080×440 pixels for all tests to allow a conservative 10 Hz network transfer rate for the frame-buffer updates in the $\mathsf{MF}_{\mathsf{dig}}$ method (cf. table 8.2). Stereoscopic images were presented using passive stereo and linear polarization.

All participants performed the docking task with each of the techniques. The order of techniques (SF_{10}, $\mathsf{MF}_{\mathsf{opt}}$, $\mathsf{MF}_{\mathsf{dig}}$, and SF_{30}) was balanced across participants using a Latin Square design. The task was explained to each participant followed by a practice run with each of the four techniques in the same order as the subsequent trials. A trial was started by pointing at a start button in the middle of the screen and pressing a button on the input device. The trial was finished once the ball-shaped object was dropped off at the correct target location. The duration for each trial was measured and recorded for later evaluation. Figure 8.3 provides screenshots from the running application showing the start and finish of one trial. The required precision for task completion was 1.5 cm, which was half the diameter of the manipulated ball-shaped object. Each participant performed 15 trials with each method. At the end of each method participants filled out a written questionnaire asking about problems and observations with each method. After the participants were finished with all the techniques they reported about their preference with respect to each rendering technique on a scale from one to five.

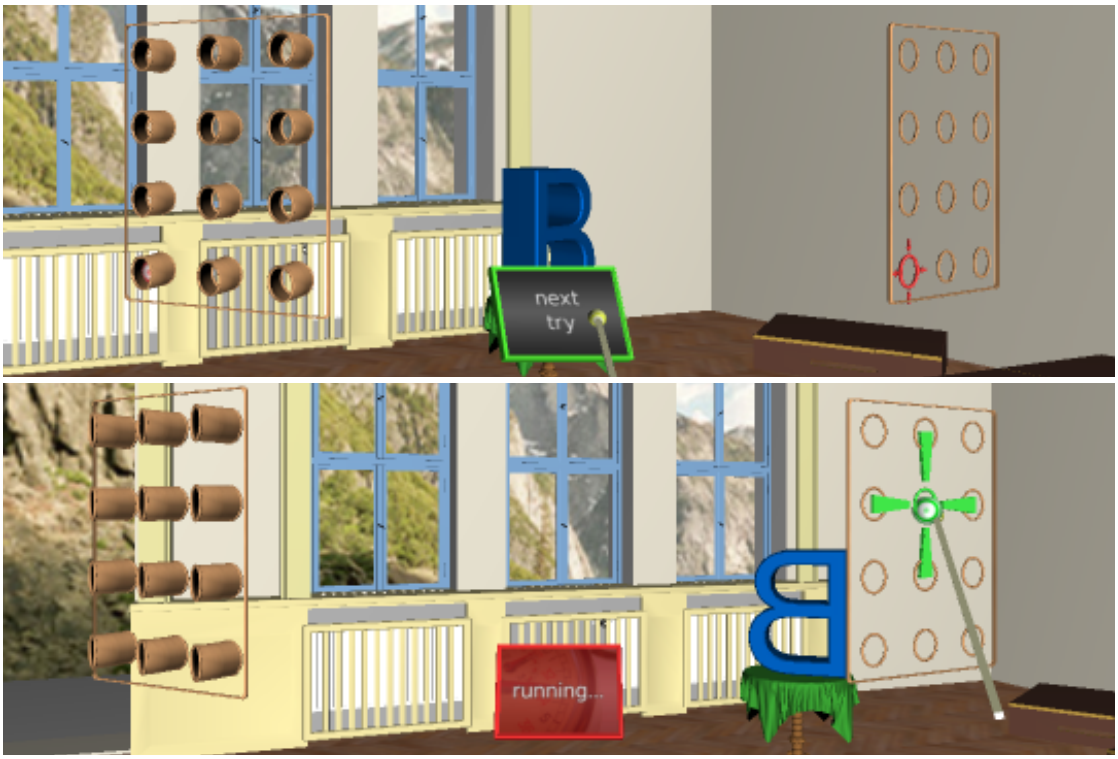
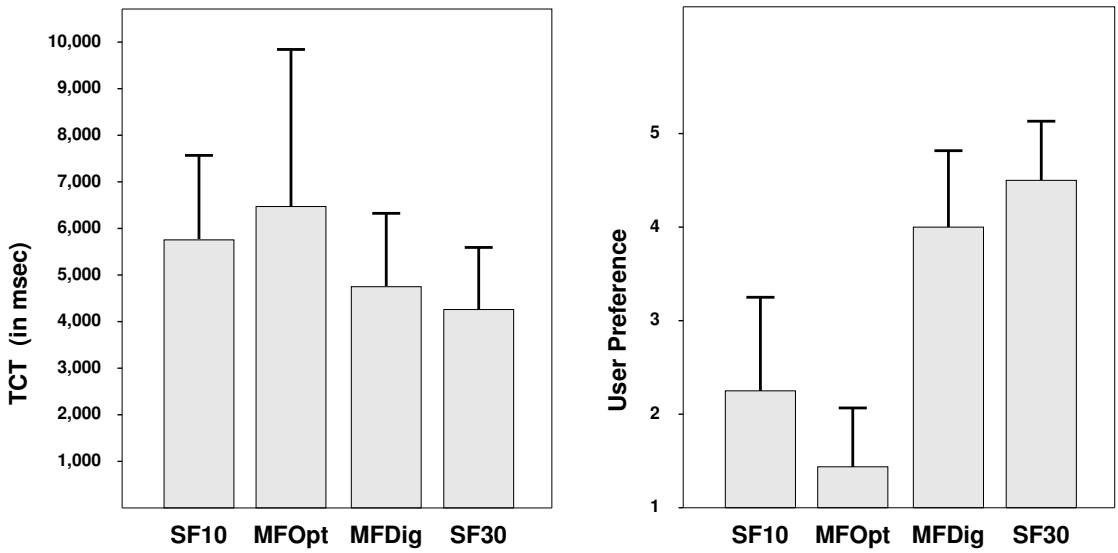


Figure 8.3: Screenshots of the application prototype used in the user study; upper image shows the start of a trial and the lower image trial finish.

Results The task-completion times (TCT) of all participants were entered in a 4×1 analysis of variance for repeated measures with the within-factors rendering techniques (SF_{10}, $\mathsf{MF}_{\mathsf{opt}}$, $\mathsf{MF}_{\mathsf{dig}}$, and SF_{30}) as well as the between-factor order of techniques. The order of the techniques did not produce a main effect nor did it interact with the rendering technique indicating that there was no transfer between the techniques. This probably results from the simplicity of the chosen task and the extended practice runs.

The performance of the four techniques differed significantly: $F_{3,188} = 10.01, p < .001$. Docking with SF_{10} took 5.75 seconds (standard error $se = .26$), while $\mathsf{MF}_{\mathsf{opt}}$ produced even longer TCTs of 6.47 seconds ($se = .49$). The shortest TCTs were obtained using SF_{30} with 4.26 seconds ($se = .19$) followed by the $\mathsf{MF}_{\mathsf{dig}}$ technique with 4.75 seconds ($se = .23$). Figure 8.4a shows these results graphically. Post-hoc comparisons revealed that all techniques differed from each other ($p < .05$), except the pair SF_{10} and $\mathsf{MF}_{\mathsf{opt}}$ as well as the pair $\mathsf{MF}_{\mathsf{dig}}$ and SF_{30} did not produce significant differences. User preference ratings (cf. figure 8.4b) of the techniques differed significantly as well: $F_{3,60} = 16.91, p < 0.05$. Post-hoc comparisons here showed the same results as for the TCTs. This means that the subjective measures, i. e. user preference, are exactly coincident with the results of the performance measures.

In summary the digital image composition approach $\mathsf{MF}_{\mathsf{dig}}$ performs significantly better than simply rendering at 10 Hz frame rate (SF_{10}). Even more, the $\mathsf{MF}_{\mathsf{dig}}$ TCTs



(a) Task-completion times. **(b)** User preference.

Figure 8.4: User-study results. (a) Task-completion times (mean values and standard deviation, in milliseconds) for single-frame rate rendering at 10 Hz (SF_{10}) and 30 Hz (SF_{30}) as well as multi-frame rate rendering using optical superposition (MF_{opt}) and digital composition (MF_{dig}) at a frame-rate ratio of 10/30 Hz for slow and fast client, respectively; (b) User preference for the tested methods (mean values and standard deviation, from $1 =$ dislike to $5 =$ prefer).

and the user preferences are almost at the level of rendering the whole scene at 30 Hz (SF_{30}), which shows the potential of this approach because rendering at 30 Hz is not possible for many real-world scenarios using a single-frame rate rendering approach. At first sight the performance of the optical superposition technique MF_{opt} seems to be disappointing, because it is the overall slowest technique and the participants also did not like the technique. Further investigation of the questionnaires revealed that the depth perception during selection and drop-off of the manipulated object was problematic. This is most likely because of the two conflicting depth cues: occlusion and stereopsis. The simple optical superposition does not provide any occlusion information. This information was particularly important during the selection process, because a selection ray of only 1.5 m length was used. The end of the ray had to pierce the ball-shaped object, which was difficult to judge just from the stereoscopic parallax. Thus, the basic optical superposition technique is not well suited for these types of tasks where objects intersect and precise manipulation is required. However, it's advantage is that it is very easy to implement.

Participants were usually able to distinguish between all methods tested in the experiment. This is especially interesting for the SF_{30} and MF_{dig} methods. Here the difference lay only in the head-tracking update rate, 30 Hz for SF_{30} and 10 Hz for MF_{dig}, and the fact that MF_{dig} used naïve multi-frame rate rendering, i. e. migration artifacts could be seen by the user. Despite this, the analytical results of the TCTs produced no statistically significant differences. This suggests that multi-frame rate

rendering at a frame-rate ratio of 10/30 Hz performs as well as using a conventional single-frame rate approach at 30 Hz but enables the use of scenes with greater geometric complexity or more compute-intensive rendering methods because the slow client needs to maintain only a frame rate of 10 Hz while the fast client will always exhibit a steady 30 Hz frame rate. It is also worth noting that the initial argument that users are able to *cope* with low(er) navigation update rates (cf. section 1.1) was indirectly confirmed. While SF_{30} was running at a 30 Hz frame rate it also provided navigation updates at the same rate. $\mathsf{MF}_{\mathsf{dig}}$ on the other hand maintained only a 10 Hz update rate for navigation but the post-hoc comparisons show that both methods do not significantly differ. Moreover, $\mathsf{MF}_{\mathsf{dig}}$ will still provide sufficient interaction fidelity even if the slow client's frame rate drops to lower values.

8.4 Limitations

Multi-frame rate rendering has a number of limitations. While the uneven workload distribution between the slow and fast client allows for improved interaction fidelity, it assumes a static scene and the manipulation of only a small subset of the objects in the scene. Graphics-related requirements such as high geometry complexity of interactive objects, transparency blending, or scene-global effects may make the multi-frame rate rendering method inapplicable. Buffer transfer and end-to-end latency affect the overall responsiveness of the application. Simulation-driven objects, such as animations, while not yet considered, may limit performance in a multi-frame rate rendering system. In the following these limitations are discussed in more detail.

Graphics Related The workload of the fast client ultimately defines interaction fidelity in a multi-frame rate system. As already mentioned the fast client needs to run with the highest possible frame rates to provide sufficient interaction fidelity. High geometry complexity of interactive objects as well as rendering many transparent objects on the fast client will increase the frame time and contribute to limited interaction fidelity. While current graphics architectures provide very high bandwidth and rasterization performance, complex operations per fragment also increase the time for creating an image which reduces update rates and limits interaction fidelity.

Scene-global effects must be resolved with the update rate of the slow client in a naïve multi-frame rate rendering setup (cf. section 6.2). This also leads to limited interaction fidelity if manipulation of objects related to such effects is required. However, using multi-frame rate deferred shading this limitation can be resolved at the cost of additional management and buffer-transfer overhead.

Often mostly static scenes are used in high-quality visualization scenarios and VR applications (cf. section 1.1). The use of dynamic scenes (e. g., animated objects or dynamics simulation for object behavior) would also introduce additional workload for the fast client.

Bandwidth and Latency The screen resolution and the number of buffers that must be transferred from the slow to the fast client directly influence the required bandwidth. Because buffer transfer must be accomplished via a physical medium (cf. section 8.1) the transfer performance is bound to the bandwidth of that physical medium (cf. section 8.2). The delay in buffer updates from the slow to the fast client can be minimized by using a single image generator technique such as the interleaved rendering method (cf. section 5.3). Here no actual buffer transfer to or from the graphics device would be necessary because all buffers reside on the same physical graphics memory. However, memory requirements are then further increased.

As analyzed in section 8.2 multi-frame rate rendering reduces the end-to-end latency for incorporating user input events on the fast client. On the other hand the same process on the slow client is at best as fast as conventional single-frame rate rendering. For multi-frame rate rendering, even with object-migration artifact fixes enabled, this determines the update rate for navigation-related input events (e. g., view transform changes).

Simulation-Driven Objects To enhance realism designers of VR applications increasingly employ simulation-driven entities, such as animations or objects controlled by dynamics or physics engines. In the context of multi-frame rate rendering this raises the question of where in the system these simulated objects are to be processed: on the slow client, on the fast client, or on both? Because the slow client potentially produces new images at a much lower rate than the fast client it seems preferable to process the simulation results for such objects on the fast client. Even if the slow client generates new buffers at a satisfying rate the transfer latency from the slow to the fast client might be too large for a given simulation type. On the other hand, executing all simulations on the fast client might increase the workload on the fast client to the point where performance drops to non-interactive frame rates. Separating a simulation into parts running on the slow client and into parts running on the fast client may only possible for a limited class of simulations. For example simulating an analog wall clock that allows no direct interaction by the user might be a good candidate for such a separation. Here the static parts of the clock as well as the hour and minute hands are rendered on the slow client while the second hand is rendered on the fast client. The result is a display of continuously smooth movements of the second hand.

More difficult to realize are simulation-driven objects that also allow direct user interaction. A simple example is a gear box containing two gear wheels where one wheel drives the other. The gear box and its wheels are assumed to be rendered on the slow client but the gear wheels can be independently manipulated. User manipulation of either gear wheel will most likely lead to artifacts similar to the selection and release artifacts discussed in section 6.1. Because the objects change their position and rotation from frame to frame the object migration process will show the same artifacts as the object selection process where user manipulation immediately started. A simple solution would be to migrate the interactive object and all its dependent objects, i. e. to extend the object migration to the whole gear

box in the example. However, this will not be perfect. Only always rendering the whole gear box on the fast client will avoid artifacts. In this case the fast client's workload might be increased to the point of non-interactive update rates.

In case of a multi-frame rate setup where buffer transfer and end-to-end latency play a secondary role the use of a dedicated render client or process might be considered for generating partial images containing only those parts of the scene that are dependent on simulation results. The buffers containing the partial images are then used on the fast client for composition. If the update rate of such a dedicated resource is high enough it would also allow to incorporate interaction-related events relevant to the simulated objects without the need for object migration. This scheme can be generalized by defining a set of slow clients, or rather simulation clients, where each client has a certain target update rate. Objects influenced by a simulation where a certain time budget must be met are then assigned to a simulation client that fits the schedule best. This is similar to the hybrid multi-frame rate system discussion in section 5.4 but may also be a viable option for emerging PC systems featuring many-core multi-CPU setups and multiple GPUs.

Object simulation driven by a dynamics or physics engine can be seen as a generalization of object animation. In addition to defining the object behavior (e. g., movement or spin) object interaction such as collision or penetration can be modeled as well as environment influences such as gravity. Because of the high frequency of the simulation steps and the varying simulation rates object updates should be performed on the fast client. However, large parts of the scene may be involved because of inter-object behavior resolution. This raises the potential for decreasing the update rate on the fast client and degrading interaction fidelity. Using a set of simulation clients with defined target update rates, as mentioned before, may be a possible solution. Generally, integrating object simulation in a multi-frame rate system will require increased resource-management overhead.

Finally, simulation-driven objects have many constraints that can only be completely determined for specific application scenarios, so only the general problem space could be outlined here.

8.5 Summary

Buffer transfer in a multi-frame rate setup using digital composition primarily influences the latency with which buffers from the slow client are available at the fast client for incorporation. Methods for buffer transfer in a general graphics cluster environment as well as for single PC setups have been discussed. End-to-end latency analysis shows that latency can be substantially reduced in a single PC setup as compared to a graphics-cluster environment or conventional single-frame rate rendering. More importantly the interactivity of the fast client is not affected by the end-to-end latency for transferring buffers from the slow to the fast client.

A user study comparing conventional single-frame rate rendering at low and high frame rates with multi-frame rate rendering using optical superposition as well as digital composition revealed that task performance of the digital composition

method was nearly as good as conventional single-frame rate rendering at high frame rates. This is complemented by user preference, rating multi-frame rate rendering by digital composition nearly as high as rendering everything with highly interactive frame rates using the conventional approach.

Multi-frame rate rendering exhibits some limitations. Interaction fidelity is influenced by the workload of the fast client. Geometric complexity of interactive objects as well as resolving transparency are contributing factors here. By using multi-frame rate deferred shading scene-global effects can be updated on the fast client while naïve multi-frame rate rendering must resolve them on the slow client. Buffer transfer and end-to-end latency in the system greatly influence the responsiveness of an application. Together with the slow client's update rate this also defines the event-update rate for navigation and head tracking. Simulation-driven objects, such as animations, require additional effort for integration into a multi-frame rate system and are heavily dependent on specific application scenarios.

Chapter 9

Conclusions

This thesis is concluded by a summarizing discussion of multi-frame rate rendering as well as ideas for future work. The summary recapitulates the aspects of multi-frame rate rendering as well as results presented and provides a discussion of their relationship in the general context of interactive computer graphics. Future work is discussed to round off the multi-frame rate method and touching on ideas for improvements in several ways.

9.1 Summary

In this thesis multi-frame rate rendering was presented. It has been shown that by using dedicated asynchronously running image generators for the static and interactive parts of a 3D scene it is possible to separate the tight coupling of sensor-data evaluation for interaction and the visualization loop. This separation does improve interaction performance while not degrading the visual quality targeted by the application developer.

Optical superposition is the simplest method to combine images created by multi-frame rate rendering. By using completely overlapping projectors the partial images from the slow and the fast client are optically combined. While this composition method is easy to realize it bears the disadvantage of not providing correct depth occlusion. It is however useful for separating system control elements such as menus.

Digital composition allows for correct depth occlusion in a multi-frame rate image by using the slow client's color and depth values as the base for rendering on the fast client. The method can be seen as a special case of Sort-Last parallel rendering with intentional unbalanced load. The transfer of buffers from the slow to the fast client introduces additional lag into the system which heavily depends on available capabilities of the transfer medium. However, this additional lag does not degrade the improved interaction fidelity realized by the fast client. Digital composition can be implemented with a variety of system setups such as clusters of graphics workstations, multi-GPU systems, or on a single virtualized graphics device using an interleaved rendering strategy.

A user study, comparing single-frame rate rendering at low and high frame rates with multi-frame rate rendering using optical superposition as well as digital composition, showed that for 3D placement tasks multi-frame rate rendering using optical superposition compares worse than using a single-frame rate method at

low frame rates for both task completion times and user preference. This means that optical superposition may be used only for simple tasks, such as displaying a menu, or for coarse interaction with scene objects where correct depth occlusion of objects may be negligible. In the same user study multi-frame rate rendering using digital composition produced no statistically significant difference in performance compared to single-frame rate rendering at high frame rates while providing the possibility of higher interaction fidelity for very large and complex scenes.

Migration artifacts introduced into multi-frame rate images can almost always be avoided by prediction and proper state management of objects in the selection and release process. The asynchronous nature of the participating rendering processes requires a state transition method to avoid doubly drawn objects or objects that temporarily disappear. The presented state transition approach completely avoids migration artifacts for released objects by including a list of object ids for objects already contained in the slow client's image. The fast client can thus preserve visual appearance for objects released from interaction but not yet included in the image of the slow client. Depth-value comparison between the slow client's buffer and fragments generated by the fast client avoids doubly drawn objects in the selection phase. However, because the slow client updates its buffers at a much lower rate immediate interaction after selection on the fast client may show the object twice. Separating the entering of an object or its enclosure (e. g., its bounding box) with the virtual pointer from the actual selection event provides in general sufficient time for the update on the slow client and thus for object migration from the slow to the fast client for most situations.

Advanced multi-frame rate techniques were presented that allow the interactive manipulation of lights and light parameters and the correct occlusion of opaque geometry within a translucent volume. The interactive manipulation of lights is based on an adaption of the deferred shading method to multi-frame rate techniques and provides a variety of design choices to trade the shift of computational load among the slow and the fast client. A parallel volume rendering technique was presented that enables the correct occlusion of opaque objects within a volume data set by re-computing the volume rendering on the fast client for only those parts of the final image where interactive objects are actually covered by the volume.

Multi-frame rate rendering can be implemented with a variety of hardware setups such as graphics clusters, multi-GPU PCs, or even single-GPU systems. Because current GPUs exhibit more bandwidth than current off-the-shelf network technology buffer-transfer latency mainly depends on the network bandwidth if employing a graphics cluster. This latency can be reduced by using a multi-GPU setup on a single machine together with a zero-copy approach for transferring frame-buffer content between the participating image generators. Single-GPU systems are supported by a virtualization of the image generator and interleaving the rendering streams of the slow and fast client. This single-GPU support enables the use of multi-frame rate techniques on low-end graphics systems such as laptops, mobile phones, and PDAs. Hybrid multi-frame rate configurations can be build that allow for dynamic load balancing of system resources in a graphics cluster.

In addition to the improved interaction fidelity of multi-frame rate rendering the method is orthogonal to most conventional (single-frame rate) rendering approaches. Multi-frame rate rendering describes a high-level concept that can be used together with a variety of image generation and rendering methods. Especially its use within existing or standardized graphics APIs should create interest in communities traditionally struggling with the interactive visualization of large data sets and high-quality image requirements. In contrast to other acceleration methods for image generation multi-frame rate rendering does not impose any restrictions on the underlying rendering technique(s). The image generation process is decomposed into parts relevant for interaction and the static rest of a scene. Each part is processed using traditional rendering techniques on dedicated asynchronously running rendering resources. This decomposition does not affect the actual requirements of an application scenario and application developers are therefore free to define their own terms of visual fidelity. Interaction fidelity is then provided by the asynchronous processing of interaction-relevant scene elements using multi-frame rate techniques. Multi-frame rate rendering is also available for a variety of hardware system setups which enables users of multi-frame rate applications to define and to control their trade-offs according to their actual needs.

9.2 Future Work

Navigation Fidelity Multi-frame rate rendering improves interaction fidelity, but does not affect *navigation fidelity*. To generate consistent images the view transform from the slow client must be also used on the fast client. This leads to a view point update rate with the same frequency as the slow client's update rate and hence defines navigation fidelity in the system. Although the initial observations from section 1.1 are still valid, it is nevertheless important to improve the situation also for navigation fidelity. This would increase user acceptance as well as the usability of multi-frame rate rendering, especially in head-mounted display setups where fluid view-point updates are essential.

Depth-image warping is used in image-based rendering to generate novel views from given reference images considering per-pixel color and depth information. Post-rendering 3D warping [Mark et al. 1997] is a particular warping technique, which focuses on increasing the overall frame rate of interactive systems by generating new views between the current view point and a predicted view point. This approach has proved to be quite effective in exploiting view-point coherence. Multi-frame rate rendering using digital composition also provides a depth image from the slow client. By using this depth image and the current view transform on the fast client a new view can be created that is consistent with the current sensor input for the fast client. It should also be possible to account for the considerable lag introduced by buffer transfer from the slow to the fast client. Effectively, this will increase the navigation fidelity to the performance level of the interaction fidelity on the fast client. However, warping can introduce visual artifacts, such as holes and potentially also blurriness.

Shadow Support The depth-image warping idea for improving navigation fidelity can also be adapted to allow for interactive shadow updates in a multi-frame rate deferred shading setup. On the slow client, or another dedicated system, depth images per light would be generated and sent to the fast client to be used for incorporation in the shadow generation process. These shadow buffers need only be generated for static lights and upon object migration. On the fast client these shadow buffers are then used to compute shadow contributions in the final image. Depth-image warping can be used to compute intermediate updates for new view points. The quality of such warped shadow maps however has to be assessed. The shadow map creation may be executed with potentially different frame rates on a dedicated render client or render context. This would create true multi-frame rate rendering.

Multi-GPU Direct-Data Transfer Single-system multi-GPU setups greatly reduce system-response time in contrast to cluster-based systems or conventional single-frame rate rendering (cf. section 8.2). However, reading from and writing to a graphics device still introduces considerable lag into the system. This could be avoided if frame-buffer data were directly transferable from one graphics device to another. The idea is to create a shared render target known by all graphics boards in the system. If such a render target is updated on one of the GPUs the graphics driver would schedule a buffer transfer to all other GPUs without using host-memory bandwidth or CPU resources. Such an API would require the support from hardware vendors; preliminary work has been carried out (e. g., NVIDIA's `WGL_NV_share_object` OpenGL extension).

Single-GPU Vendor Support Multi-frame rate rendering techniques can also be applied to systems with only a single GPU by interleaving the rendering process of the fast and slow client as described in section 5.3. The entire scene needs to be partitioned such that each part requires approximately the same amount of rendering time. Rendering of these scene partitions is then distributed over multiple frames of the fast client. An equal workload distribution is usually difficult to achieve. However, in case deferred shading is used, the partitioning depends mainly on the geometry complexity, which is easier to manage. An explicit scene partitioning would not be required for this approach if GPUs would support an efficient render task scheduling mechanism. Interleaved rendering enables users of single GPU and low-end graphics systems to benefit from the improved interaction fidelity of multi-frame rate rendering. In addition, because buffer transfers have also a significant influence on single-system implementations using multiple GPUs, it might be sometimes more efficient to use interleaved rendering on a single GPU without the need for any transfer operations through host memory.

Multi-Frame Rate Ray Tracing Real-time ray tracing has recently become a feasible rendering method for VR applications. While ray tracing is able to generate excellent image quality by considering a number of optical effects (e. g., reflections, refractions, and shadows), the interaction fidelity is often quite limited. However, VR applications often require the interactive manipulation of objects, which may change their position and orientation from frame to frame depending on user input.

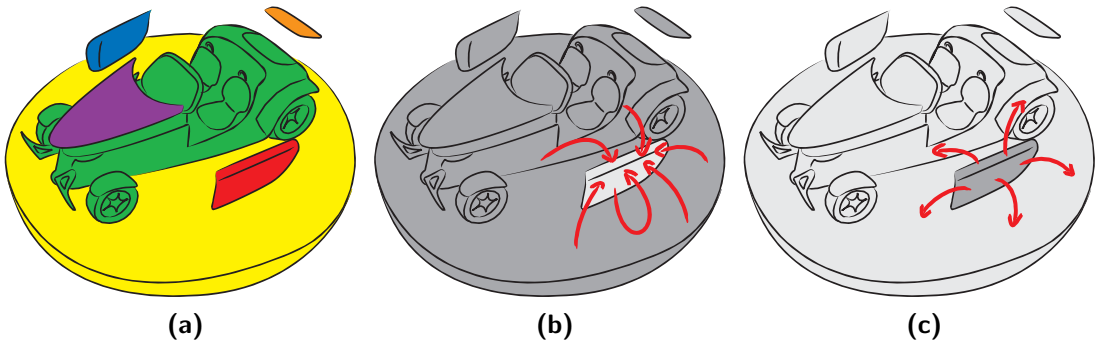


Figure 9.1: Multi-frame rate ray tracing example (after [Kurz et al. 2008]). The left door of the car is currently manipulated (a). Reflections are decomposed into reflection of the scene in the left door (b) and the reflection of the left door in the rest of the scene (c).

This is still a challenge for ray-tracing systems—in particular if they are supposed to run on a single computer.

Initial work has been conducted for using a multi-frame rate approach within a ray-tracing system [Kurz et al. 2008]. The original multi-frame rate approach was developed for rasterization-based rendering. It extends also to ray casting, but for ray tracing it is not immediately applicable because of the light interaction between interactive objects and the rest of the scene. For ray tracing the idea is to use an approach similar to multi-frame rate rendering for the first order reflections in a limited ray-tracing system. The scene is also divided into two sets: the set of currently manipulated objects and the set containing the rest of the scene (cf. figure 9.1). The specular reflections between these two sets can be decomposed into reflections from the static scene parts in the manipulated object and vice versa (cf. figure 9.1b and 9.1c). The reflections within the objects of the static scene remainder are recomputed whenever the view point changes. As long as no view-point change occurs, these pre-computed reflections are used as the basis for interactively ray tracing the scene. Because typically only a small subset of the scene is manipulated, the interactivity of the ray-tracing system is significantly improved during object manipulation.

The extension of this method to refractions and shadows introduces a large memory overhead. With further generalization to higher ray-recursion depths the advantage of the pre-computed effects diminishes because of the additional computational overhead. Reduction of both memory and computational costs must be explored to allow for a more general application of the algorithm.

User Tests It would be beneficial to perform the user study described in section 8.3 again because neither the migration artifact fixes nor the additional navigation support using depth-image warping were used in the experimental setup. It can be speculated that by including these techniques users will no longer be able to discriminate between single-frame rate rendering at high frame rates and multi-frame rate rendering using digital composition.

An additional point of interest in this context is the question of a lower threshold for the head-tracking update frequency. Currently the slow client is assumed to render as fast as possible but being potentially much slower than the fast client. Incorporation of head-tracking updates will only be possible with this frame rate for naïve multi-frame rate methods. On the other hand, the slow client is not required to run at a steady frame rate at all. For multi-frame rate techniques it is sufficient if the slow client reacts to object-migration events. It must also act on view-point changes. It should be instructive to investigate if these two update events, probably in combination with the additional support for improved navigation fidelity, are sufficient to generate new images on the slow client to be used on the fast client as the base for creating images exhibiting high fidelity for interaction, navigation, and visualization.

So far frame-rate ratios between the slow and fast clients were used from empirical data. It should be insightful to determine if there are frame-rate ratios that work better than others, especially with respect to interaction and navigation fidelity, i. e. if the additional navigation fidelity support is available or not.

9.3 Closing Remarks

Adapting, refining, and newly developing advanced rendering techniques in the context of multi-frame rate rendering considerably extends the potential of current graphics systems and enables the interactive manipulation of extremely large and complex virtual environments. The combination of all advanced multi-frame rate techniques certainly leads to an increase of the manageable size of 3D scenes in VR applications by an order of magnitude. This is such a substantial improvement that multi-frame rate techniques should be an integral part of any future interactive 3D graphics application or VR system.

Bibliography

J. Airey, J. Rohlf, and P. Frederick. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *SI3D '90: Proceedings of the 1990 Symposium on Interactive 3D Graphics*, pages 41–50. ACM, 1990. DOI 10.1145/91385.91416.

K. Akeley. Reality Engine Graphics. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 109–116. ACM, 1993. DOI 10.1145/166117.166131.

ATI CrossFire™ Technology Whitepaper. 2005. URL http://ati.amd.com/technology/crossfire/CrossFireWhitePaperweb2.pdf.

D. Auzel. *Émile Reynaud et l'image s'anima*. Dreamland, 1998.

R. Bastos. *Superposition Rendering: Increased Realism for Interactive Walkthrough*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 1999.

S. Benford and L. Mariani. Requirements and Metaphors of Shared Interaction. COMIC Project, Esprit Basic Research Project 6225, Deliverable D4.1, Lancaster University, October 1993.

L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image Rendering by Adaptive Refinement. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, volume 20(4), pages 29–37. ACM, 1986. DOI 10.1145/15922.15889.

E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. Sort-First, Distributed Memory Parallel Visualization and Rendering. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 41–50. IEEE Computer Society, 2003. DOI 10.1109/PVGS.2003.1249041.

G. Bishop, H. Fuchs, L. McMillan, and E. J. Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. In *Proceedings of ACM SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 175–176. ACM, 1994. DOI 10.1145/192161.192195.

R. Blach, M. Bues, J. Hochstrate, J. Springer, and B. Fröhlich. Experiences with Multi-Viewer Stereo Displays Based on LC-Shutters and Polarization. In *IEEE VR 2005 Workshop: Emerging Display Technologies*, pages 15–17. IEEE Computer Society, 2005.

Boost.Interprocess Library. URL http://igaztanaga.drivehq.com/interprocess/.

D. A. Bowman, E. Kruijff, J. J. Laviola, and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Professional. Addison-Wesley, Boston, MA, USA, 2004.

S. Bryson. Implementing Virtual Reality. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 1.1.1–1.6.6, 16.1–16.12. ACM, 1993. Course #43.

S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, an Information Workspace. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 181–186. ACM, 1991. DOI 10.1145/108844.108874.

E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM*, 19(10):547–554, 1976. DOI 10.1145/360349.360354.

S. Comes and B. M. Macq. Human Visual Quality Criterion. *Proceedings of SPIE*, 1360:2–13, 1990. DOI 10.1117/12.24100.

W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96. Eurographics, 2002. DOI 10.1145/800064.801272.

T. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Computer Science Department, 1993.

A. Dayal, C. Woolley, B. Watson, and D. Luebke. Adaptive Frameless Rendering. In *Proceedings of the 2005 Eurographics Symposium on Rendering*, Rendering Techniques 2005, pages 265–275. Eurographics, 2005. DOI 10.1145/1198555.1198763.

M. Deering and D. Naegle. The SAGE Graphics Architecture. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 683–692. ACM, 2002. DOI 10.1145/566654.566638.

M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, volume 22(4), pages 21–30. ACM, 1988. DOI 10.1145/378456.378468.

D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, 2003. DOI 10.1109/PVGS.2003.1249046.

R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, volume 22(4), pages 65–74. ACM, 1988. DOI 10.1145/378456.378484.

R. Dumont, F. Pellacini, and J. A. Ferwerda. A Perceptually-Based Texture Caching Algorithm for Hardware-Based Rendering. In *Proceedings of Eurographics Workshop on Rendering*, pages 249–256. Eurographics, 2001.

R. Dumont, F. Pellacini, and J. A. Ferwerda. Perceptually-Driven Decision Theory for Interactive Realistic Rendering. *Trans. on Graphics*, 22(2):152–181, 2003. DOI 10.1145/636886.636888.

J. Durbin, R. Gossweiler, and R. Pausch. Amortizing 3D Graphics Optimization Across Multiple Frames. In *UIST '95: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, pages 13–19. ACM, 1995. DOI 10.1145/215585.215633.

J. El-Sana and A. Varshney. View-Dependent Topology Simplification. *Virtual Environments*, pages 11–22, 1999.

M. Eldridge. *Designing Graphics Architectures Around Scalability and Communication*. PhD thesis, Stanford University, 2001.

K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

Ensemble. URL http://dsl.cs.technion.ac.il/projects/Ensemble/.

J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The Realization. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 57–68. ACM, 1997. DOI 10.1145/258694.258714.

J. A. Ferwerda. Three Levels of Realism in Computer Graphics. In *SPIE Human Vision and Electronic Imaging '03*, pages 290–297. SPIE, 2003. DOI 10.1117/12.473899.

Folding@home. URL http://folding.stanford.edu/.

B. Fröhlich, J. Hochstrate, J. Hoffmann, K. Klüger, R. Blach, M. Bues, and O. Stefani. Implementing Multi-Viewer Stereo Displays. In *WSCG 2005*, pages 139–146, 2005.

T. A. Funkhouser and C. H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. ACM, 1993. DOI 10.1145/166117.166149.

M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. In *Proceedings of ACM SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 209–216. ACM, 1997. DOI 10.1145/258734.258849.

G. Godin, J.-F. Lalonde, and L. Borgeat. Projector-Based Dual-Resolution Stereoscopic Display. In *Proceedings IEEE Virtual Reality 2004 Conference*, pages 223–224. IEEE Computer Society, 2004a. DOI 10.1109/VR.2004.1310080.

G. Godin, P. Massicotte, and L. Borgeat. Foveated Stereoscopic Display for the Visualization of Detailed Virtual Environments. In *EGVE04: 10th Eurographics Symposium on Virtual Environments*, pages 7–16. Eurographics, 2004b.

H. Gouraud. Continuous Shading of Curved Surfaces. *IEEE Trans. Comput.*, C-20 (6):623–629, 1971.

S. Hargreaves and M. Harris. Deferred Shading. 2004. URL http://developer.nvidia.com/object/6800_leagues_deferred_shading.html.

T. He, L. Hong, A. Varshney, and S. Wang. Controlled Topology Simplification. *IEEE Trans. Vis. and Comput. Graph.*, 2(2):171–184, 1996. DOI 10.1109/2945.506228.

H. Hoppe. Progressive Meshes. In *Proceedings of ACM SIGGRAPH 1996*, Computer Graphics Proceedings, Annual Conference Series, pages 99–108. ACM, 1996. DOI 10.1145/237170.237216.

H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. In *Proceedings IEEE Visualization*, pages 35–42. IEEE Computer Society, 1998. DOI 10.1109/VISUAL.1998.745282.

H. Hoppe. View-Dependent Refinement of Progressive Meshes. In *Proceedings of ACM SIGGRAPH 1997*, Computer Graphics Proceedings, Annual Conference Series, pages 189–198. ACM, 1997. DOI 10.1145/258734.258843.

G. Humphreys, M. Eldridge, I. Buck, G. Stoll, P. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 129–140. ACM, 2001. DOI 10.1145/383259.383272.

G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *Proceedings of ACM SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 693–702. ACM, 2002. DOI 10.1145/566654.566639.

M. Isenburg and S. Gumhold. Out-of-Core Compression for Gigantic Polygon Meshes. *Trans. on Graphics*, 22(3):935–942, 2003. DOI 10.1145/1201775.882366.

J. T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, volume 20(4), pages 143–150. ACM, 1986. DOI 10.1145/15922.15902.

J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL® Shading Language, (Language Version 1.20, Rev. 8, 07-Sept-2006)*. OpenGL ARB, 2006.

J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, pages 38–43. IEEE Computer Society, 2003. DOI 10.1109/VIS.2003.10001.

D. Kurz, C. Lux, J. P. Springer, and B. Fröhlich. Improved Interaction Performance for Ray Tracing. In *Proceedings of Eurographics 2008*, pages 283–286. Eurographics, 2008. Annex to the Conference Proceedings, Short Papers.

A. Lastra, S. Molnar, M. Olano, and Y. Wang. Real-Time Programmable Shading. In *SI3D '95: Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 59–66. ACM, 1995. DOI 0.1145/199404.199414.

M. Levoy. Efficient Ray Tracing of Volume Data. *Trans. on Graphics*, 9(3):245–261, 1990. DOI 10.1145/78964.78965.

K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Building and Using a Scalable Display Wall System. *IEEE Comput. Graph. Appl.*, 20(4):29–37, 2000. DOI 10.1109/38.851747.

P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 259–262. ACM, 2000. DOI 10.1145/344779.344912.

P. Lindstrom. Out-of-Core Construction and Visualization of Multiresolution Surfaces. In *SI3D '03: Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 93–102. ACM, 2003. DOI 10.1145/641480.641500.

P. Lindstrom and C. T. Silva. A Memory Insensitive Technique for Large Model Simplification. In *Proceedings IEEE Visualization*, pages 121–126. IEEE Computer Society, 2001. DOI 10.1109/VISUAL.2001.964502.

S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heinrich. Scalable Interactive Volume Rendering Using Off-the-Shelf Components. In *IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pages 115–121. IEEE Computer Society, 2001. DOI 10.1109/PVGS.2001.964412.

M. Lounsbery, T. D. DeRose, and J. Warren. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. *Trans. on Graphics*, 16(1):34–73, 1997. DOI 10.1145/237748.237750.

D. Luebke and C. Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. In *Proceedings of ACM SIGGRAPH 1997*, Computer Graphics Proceedings, Annual Conference Series, pages 199–208. ACM, 1997. DOI 10.1145/258734.258847.

D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Computer Graphics Series. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

B. MacIntyre and S. Feiner. A Distributed 3D Graphics Library. In *Proceedings of ACM SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 361–370. ACM, 1998. DOI 10.1145/280814.280935.

A. Majumder. *A Practical Framework to Achieve Perceptually Seamless Multi-Projector Displays*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 2003.

A. Majumder and G. Welch. Computer Graphics Optique: Optical Superposition of Projected Computer Graphics. In *EGVE/IPT 2001: 7th EG Workshop on Virtual Environments & 5th Immersive Projection Technology Workshop*, pages 209–218. Center of the Fraunhofer Society Stuttgart IZS, 2001.

K. Mania, D. Wooldridge, M. Coxon, and A. Robinson. The Effect of Visual and Interaction Fidelity on Spatial Cognition in Immersive Virtual Environments. *IEEE Trans. Vis. and Comput. Graph.*, 12(3):396–404, 2006. DOI 10.1109/TVCG.2006.55.

W. R. Mark, L. McMillan, and G. Bishop. Post-Rendering 3D Warping. In *SI3D '97: Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM, 1997. DOI 10.1145/253284.253292.

M. D. McKenna and D. Zeltzer. Three Dimensional Visual Display Systems for Virtual Environments. *Presence: Teleoperators & Virtual Environments*, 1(4): 421–458, 1992.

L. Moll, A. Heirich, and M. Shand. Sepia: Scalable 3D Compositing Using PCI Pamette. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 146–155. IEEE Computer Society, 1999. DOI 10.1109/FPGA.1999.803676.

S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, volume 26(2), pages 231–240. ACM, 1992. DOI 10.1145/142920.134067.

S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, 1994. DOI 10.1109/38.291528.

J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. Infinitereality: A Real-Time Graphics System. In *Proceedings of ACM SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 293–303. ACM, 1997. DOI 10.1145/258734.258871.

M. Newell. *The Utilization of Procedure Models in Digital Image Synthesis*. PhD thesis, University of Utah, 1975.

NVIDIA GPU Programming Guide. revision 2.5.0, 2006. URL http://developer.nvidia.com/object/gpu_programming_guide.html.

NVIDIA SDK. URL http://developer.nvidia.com/object/sdk_home.html.

NVIDIA SLI. URL http://www.slizone.com/page/home.html.

M. Ogata, S. Muraki, X. Liu, and K.-L. Ma. The Design and Evaluation of a Pipelined Image Compositing Device for Massively Parallel Volume Rendering. In *VG '03: Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume Graphics*, pages 61–68. ACM, 2003. DOI 10.1145/827051.827060.

OpenSceneGraph. URL http://www.openscenegraph.org/.

OpenSG. URL http://www.opensg.org/.

J. D. Owens. Towards Multi-GPU Support for Visualization. *J. Phys.: Conf. Ser.*, 78(1):012055 (5 pp), 2007. DOI 10.1088/1742-6596/78/1/012055.

R. Pausch. Virtual Reality on Five Dollars a Day. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 265–270. ACM, 1991. DOI 10.1145/108844.108913.

PCI Express. URL http://www.pcisig.com/specifications/pciexpress/.

E. Persson. Programming for Crossfire™. 2005. URL http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/Programming_for_CrossFire.pdf.

B. T. Phong. Illumination for Computer Generated Pictures. *Commun. ACM*, 18 (6):311–317, 1975. DOI 10.1145/360825.360839.

C. Prince. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington, Department of Computer Science and Engineering, 2000.

P. M. Rademacher. *Measuring the Perceived Visual Realism of Images*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 2002.

R. Raskar. *Projector-Based Three Dimensional Graphics*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 2002.

M. Reddy. The Effects of Low Frame Rate on a Measure for User Performance in Virtual Environments. Technical Report ECS-CSG-36-97, University of Edinburgh, Department of Computer Science, 1997.

D. Reiners. *OpenSG: A Scene Graph System for Flexible and Efficient Realtime Rendering for Virtual and Augmented Reality Applications*. PhD thesis, Technische Universität Darmstadt, Fachbereich Informatik, 2002.

D. Reiners, G. Voss, and J. Behr. OpenSG: Basic Concepts. In *1. OpenSG Symposium OpenSG 2002*, 2002.

S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, pages 231–238. IEEE Computer Society, 2003.

J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for 3D Graphics. In *Proceedings of ACM SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 381–394. ACM, 1994. DOI 10.1145/192161.192262.

S. Rose Clay. Optimization for Real-Time Graphics Applications. http://www.sgi.com/products/software/performer/presentations/tune_wp.pdf, 1996.

J. Rossignac and P. Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. In *Modeling in Computer Graphics: Methods and Applications*, pages 455–465. Springer-Verlag, 1993.

R. J. Rost. *OpenGL® Shading Language*. Professional. Addison-Wesley, Boston, MA, USA, 2004.

M. Roth. *Parallele Bildberechnung in einem Netzwerk von Workstations*. PhD thesis, Technische Universität Darmstadt, Fachbereich Informatik, 2005.

M. Roth and D. Reiners. Sorted Pipeline Image Composition. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 119–126. Eurographics, 2006.

T. Saito and T. Takahashi. Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, volume 24(4), pages 197–205. ACM, 1990. DOI 10.1145/97879.97901.

E. J. Scher Zagier. Perceptually-Driven Graphics. Technical Report TR97-017, University of North Carolina, Computer Science Department, 1997.

W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, volume 26(2), pages 65–70. ACM, 1992. DOI 10.1145/142920.134010.

M. Segal and K. Akeley. *The OpenGL® Graphics System: A Specification, (Version 2.1 – December 1, 2006)*. OpenGL ARB, 2006.

SETI@home. URL http://setiathome.berkeley.edu/.

D. A. Silverstein and J. E. Farrell. The Relationship between Image Fidelity and Image Quality. In *International Conference on Image Processing*, volume 1, pages 881–884, 1996. DOI 10.1109/ICIP.1996.559640.

M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1994.

J. P. Springer, C. Sladeczek, M. Scheffler, J. Hochstrate, F. Melchior, and B. Fröhlich. Combining Wave Field Synthesis and Multi-Viewer Stereo Displays. In *Proceedings IEEE Virtual Reality 2006 Conference*, pages 237–240. IEEE Computer Society, 2006. DOI 10.1109/VR.2006.33.

S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics 2005*, pages 187–195. Sony Book, New York, 2005.

W. R. Stevens. *UNIX Network Programming*, volume 1 – Networking APIs: Sockets and XTI. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.

G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 141–148. ACM, 2001. DOI 10.1145/383259.383273.

P. S. Strauss. IRIS Inventor: A 3D Graphics Toolkit. In *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 192–200. ACM, 1993. DOI 10.1145/165854.165889.

P. S. Strauss and R. Carey. An Object-Oriented 3D Graphics Toolkit. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, volume 26(2), pages 341–349. ACM, 1992. DOI 10.1145/133994.134089.

I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *Computing Surveys*, 6(1):1–55, 1974. DOI 10.1145/356625.356626.

G. Tharp, A. Liu, L. French, and L. Stark. Timing Considerations of Helmet Mounted Display Performance. *Proceedings of SPIE*, 1666:570–576, 1992. DOI 10.1117/12.136003.

H. Tramberend. Avocado: A Distributed Virtual Reality Framework. In *Proceedings IEEE Virtual Reality '99 Conference*, pages 14–21. IEEE Computer Society, 1999. DOI 10.1109/VR.1999.756918.

H. Tramberend. *Avocado: A Distributed Virtual Reality Framework*. PhD thesis, Universität Bielefeld, Technische Fakultät, 2003.

G. Voß, J. Behr, D. Reiners, and M. Roth. A Multi-Thread Safe Foundation for Scene Graphs and its Extension to Clusters. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 33–37. Eurographics, 2002.

I. Wald, P. Slusallek, and C. Benthin. Interactive Distributed Ray Tracing of Highly Complex Models. In *Rendering Techniques 2001: Proceedings of the Eurographics Workshop*, pages 277–288, 2001.

I. Wald, C. Benthin, and P. Slusallek. Distributed Interactive Ray Tracing of Dynamic Scenes. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 77–85, 2003. DOI 10.1109/PVGS.2003.1249045.

C. Ware and R. Balakrishnan. Reaching for Objects in VR Displays: Lag and Frame Rate. *Trans. on Computer-Human Interaction*, 1(4):331–356, 1994. DOI 10.1145/198425.198426.

B. Watson, N. Walker, W. Ribarsky, and V. Spaulding. Effects of Variation in System Responsiveness on User Performance in Virtual Environments. *Human Factors*, 40(3):403–414, 1998.

B. Watson, A. Friedman, and A. McGaffey. Measuring and Predicting Visual Fidelity. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 213–220. ACM, 2001. DOI 10.1145/383259.383283.

B. A. Watson, D. Luebke, and A. Dayal. Breaking the Frame: A New Approach to Temporal Sampling. Technical Sketch, ACM SIGGRAPH, 2002.

T. Whitted and D. M. Weimer. A Software Test-Bed for the Development of 3-D Raster Graphics Systems. In *Computer Graphics (Proceedings of ACM SIGGRAPH 81)*, volume 15(3), pages 271–277. ACM, 1981. DOI 10.1145/357290.357295.

C. D. Wickens. *Engineering Psychology and Human Performance*. HarperCollins, New York, NY, USA, 2nd edition, 1992.

L. Williams. Pyramidal Parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, volume 17(3), pages 1–11. ACM, 1983. DOI 10.1145/964967.801126.

S. Winkler. Visual Fidelity and Perceived Quality: Towards Comprehensive Metrics. *Proceedings of SPIE*, 4299:114–125, 2001. DOI 10.1117/12.429540.

M. M. Wloka, R. C. Zeleznik, and T. Miller. Practically Frameless Rendering. Technical Report CS-95-06, Brown University, Computer Science Department, 1995.

C. Woolley, D. Luebke, B. Watson, and A. Dayal. Interruptible Rendering. In *SI3D '03: Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pages 143–151. ACM, 2003. DOI 10.1145/641480.641509.

J. Yang, J. Shi, Z. Jin, and H. Zhang. Design and Implementation of a Large-Scale Hybrid Distributed Graphics System. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 39–49. Eurographics, 2002.

Curriculum Vitae

Personal

Name : Jan Peter Springer
Birth Date : July 14, 1969
Birth Place : Gotha/Thuringia, Germany

Education

1976 – 1986 : POS „Andreas Reyher“ I, Gotha
1986 – 1988 : EOS „Ernst Wilhelm Arnoldi“, Gotha
1991 – 1999 : Diploma study in Computer Science at Bauhaus-Universität Weimar, Fakultät Informatik

Professional

1990 – 1991 : Power plant machine operator, ENAG
1999 – 2001 : Research assistant with the Virtual Environments Group, Institute for Media Communication, GMD — German National Research Center for Information Technology
2001 – 2008 : Research assistant with the Virtual Reality Systems Group, Faculty of Media, Bauhaus-Universität Weimar

Publications

J. P. Springer, H. Tramberend, and B. Fröhlich. On Scripting in Distributed Virtual Environments. In *Proceedings of the 4th Immersive Projection Technology Workshop*. Virtual Reality Applications Center, Iowa State University, 2000.

J. P. Springer, C. Sladeczek, M. Scheffler, J. Hochstrate, F. Melchior, and B. Fröhlich. Combining Wave Field Synthesis and Multi-Viewer Stereo Displays. In *Proceedings IEEE Virtual Reality 2006 Conference*, pages 237–240. IEEE Computer Society, 2006.

H.-F. Pabst, J. P. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray Casting of Trimmed NURBS Surfaces on the GPU. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*, pages 151–160. IEEE Computer Society, 2006.

J. P. Springer, S. Beck, F. Weiszig, D. Reiners, and B. Froehlich. Multi-Frame Rate Rendering and Display. In *Proceedings IEEE Virtual Reality 2007 Conference*, pages 195–202. IEEE Computer Society, 2007.

J. P. Springer, S. Beck, F. Weiszig, and B. Fröhlich. Multi-Frame Rate Rendering for Standalone Graphics Systems. In *Proceedings 4. Workshop „Virtuelle und Erweiterte Realität“ der GI-Fachgruppe VR/AR*, pages 55–64. Gesellschaft für Informatik, 2007.

M. Scheffler, J. P. Springer, and B. Froehlich. Object-Capability Security in Virtual Environments. In *Proceedings IEEE Virtual Reality 2008 Conference*, pages 51–58. IEEE Computer Society, 2008.

J. P. Springer, C. Lux, D. Reiners, and B. Froehlich. Advanced Multi-Frame Rate Rendering Techniques. In *Proceedings IEEE Virtual Reality 2008 Conference*, pages 177–184. IEEE Computer Society, 2008.

J. P. Springer and B. Froehlich. From Single-Frame Rate to Multi-Frame Rate Systems. In *IEEE VR 2008 Workshop: Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 45–48. Shaker Verlag, 2008.

D. Kurz, C. Lux, J. P. Springer, and B. Fröhlich. Improved Interaction Performance for Ray Tracing. In *Proceedings of Eurographics 2008*, pages 283–286. Eurographics, 2008.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlung- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich versichere, dass ich nach bestem Wissen die reine Wahrheit gesagt und nichts verschwiegen habe.

Weimar, September 2008

— Jan P. Springer —

Colophon

This document was created with the LaTeX typesetting system. Production tools were LaTeX2e/PDFTeX 1.40.3 using the `buw_thesis` class based on KOMA-Script/scrbook. Diagrams were created with and converted by the XFig suite of graphic tools. Digital photographs and bitmap images were processed with the GNU Image Manipulation Program (GIMP). The body text is set 12/14 pt on a 34 pc measure with the Latin Modern font family.

This is the online version intended for reading on a computer screen. The layout nominally uses a one-sided A4 paper sheet, if printing is desired.