To appear in the Proceedings of IEEE VR 2008.

Advanced Multi-Frame Rate Rendering Techniques

Jan P. Springer [1] Christopher Lux [1] Dirk Reiners [2] Bernd Froehlich [1]

[1] Bauhaus-Universität Weimar [2] University of Louisiana at Lafayette

ABSTRACT

Multi-frame rate rendering is a parallel rendering technique that renders interactive parts of the 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 and displayed.

This paper presents advanced multi-frame rate rendering techniques, which remove limitations of the original approach and reduce artifacts. The interactive manipulation of light sources and their parameters affects the entire scene. Our multi-GPU deferred shading splits the rendering task into a rasterization and lighting pass and distributes the passes to the appropriate graphics card to enable light manipulations at high frame rates independent of the geometry complexity of the scene. We also developed a parallel volume rendering technique, which allows the manipulation of objects inside a translucent volume at high frame rates. Due to the asynchronous nature of multi-frame rate rendering artifacts may occur during the migration of objects from the slow to the fast graphics card, and vice versa. We show how proper state management can be used to avoid these artifacts almost completely. These techniques were developed in the context of a single-system multi-GPU setup, which considerably simplifies the implementation and increases performance.

Keywords: Multi-Frame Rate Rendering, Multi-GPU Systems, 3D Interaction

Index Terms: F.1.2 [Computation by Abstract Devices]: Modes of Computation—Interactive and Reactive Computation; I.3.2 [Computer Graphics]: Graphics Systems—Distributed/Network Graphics; I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

1 INTRODUCTION

Parallel rendering is commonly used to improve graphics performance for very large models. These approaches aim 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 [1] is a parallel rendering technique that purposely splits the workload unevenly. Interactive parts of the scene are updated at the highest possible frame rates on one graphics sub-system (fast client), while the rest of the scene is rendered at lower frame rates on a second system (slow client). The results of these asynchronous image generation processes are digitally or optically merged and displayed.

Multi-frame rate rendering has been shown to be quite effective for virtual environment applications, where only a small subset of the objects in the scene are actively manipulated. However, due to the parallel and asynchronous nature of the image generation process, some limitations and artifacts are inherent to the approach. While the original implementation by Springer et al. [1] was based on a cluster of graphics workstations, we explore a single-system implementation using a dual-graphics card setup. Such an implementation does not suffer from limited network throughput and increased latency. Additionally, advanced techniques for distributing the work load between graphics cards become feasible, which further increases the flexibility and perceived interactivity of the system while reducing the overall complexity of the implementation.

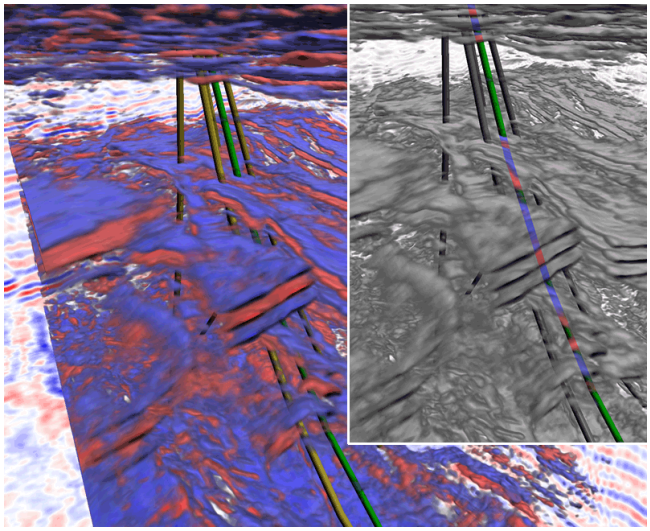


Figure 1: Volume rendering application using multi-frame rate rendering on a single multi-GPU system. The image is a snapshot of the fast client showing the digital composition result at 1600×1200 pixels. Volume ray casting was used on the $1024 \times 439 \times 734$ volume data set with 2048 samples per ray. The fast client renders only the screen area in front of the manipulated well (green) at 30 Hz. The slow client renders the whole volume at $\approx$1 Hz. Top right shows the contributions of the slow client (gray scale) and the fast client (colored). The clients were each running on an NVIDIA Quadro FX 5600.

This work is motivated by requirements for rendering large data sets from the automotive and the oil & gas industry, where multi-frame rate rendering can be most useful. Manipulation of lights is important for the visualization of car models. However, lighting changes, which affect the whole scene, can only be updated at low frame rates in the original implementation. Volume rendering is extremely important in the oil & gas industry. A central task is the precise placement of virtual wells inside a seismic volume. Since the semi-transparent volume has to be rendered after the manipulated well for correct occlusion, the volume would have to be handled on the fast graphics system as well, making multi-frame rate rendering useless for this situation. In addition, intermittent rendering artifacts resulted in limited user acceptance even though they might not affect interaction performance.

The main contributions of this paper are advanced multi-frame rate rendering techniques, which are applicable to cluster-based as well as single-system multi-frame rate rendering setups. However, a single-system environment simplifies the implementation and also significantly increases performance. Our multi-GPU deferred shading approach allows the interactive manipulation of lights, which affects the whole scene. It also allows for balancing the handling of

shading computations between the fast and slow client. Our parallel volume rendering method has been shown to be very effective for supporting highly interactive manipulation of objects inside large volumes rendered with extreme quality. In addition, artifacts occurring during migration of objects from the slow to the fast graphics card, and vice versa, can be almost completely hidden by prediction and intelligent bookkeeping. These techniques considerably extend the potential application domains for multi-frame rate rendering and enable users to interactively manipulate objects within complex virtual environments.

2 Related Work

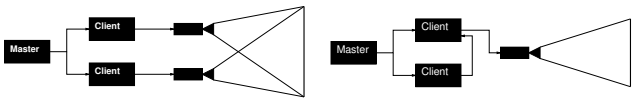
Support for multiple graphics cards in a single computer system once was the domain of specialized high-end graphics systems (e. g. SGI's RealityEngine [2] or InfiniteReality [3] platforms). With the introduction of the PCIe bus system this capability became recently available for single PCs. Besides using multiple graphics boards in such a PC system for dedicated tasks, vendor-specific solutions exist which allow the combined usage of these GPUs for the generation of a single image (e. g. NVIDIA SLI [4] or ATI CrossFire [5]). These solutions can be classified as Sort-First approaches [6]. This is in contrast to multi-frame rate rendering where a Sort-Last composition is needed. Also, with these vendor solutions the image generators are tightly coupled for load balancing and they are presented as a single graphics device to an application. Unfortunately, there is no support for application-specific Sort-Last composition using the vendor-provided high-bandwidth low-latency interconnect between graphics cards. This can only be achieved by transferring data from one card to the other via host system memory (cf. sections 3.1 and 3.2).

In this paper we present an approach to adapt direct volume rendering for our multi-frame rendering framework. 3D texture-based direct volume visualization methods sample the volume data by using a stack of typically viewport-aligned slice planes as proxy geometry [7]. 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 number of slice planes that must be generated and rendered directly depends on the data complexity as well as the desired output quality. In contrast volume ray casting [8] samples the volume data at discrete locations along rays originating at 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 [9, 10] were dependent on multiple render passes and temporary image buffers for storing intermediate results. Later methods [11] 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). However, 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 recently demonstrated [12, 13]. The main advantage of GPU-based volume ray casting techniques is their independent ray traversal, allowing the implementation of advanced acceleration techniques like 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 decent output quality using current graphics hardware, i. e. NVIDIA G80 or newer. However, larger data sets, even if fitting into available graphics memory, still suffer from low frame rates, impeding interaction with objects that need to be placed precisely inside volumes.

For the interactive manipulation of lights we adapted the deferred shading approach [14, 15], which is a method that separates the computation of relevant per-fragment shading information from the actual shading. This is achieved by storing the necessary information in intermediate buffers for later use in the composition stage (e. g. using G-buffers [16]). While hardware implementations do exist for some time (e. g. PixelFlow [17, 18]), consumer-product GPUs allowing for implementations that exhibit interactive frame rates became available only recently (e. g. NVIDIA NV40) [19]. Deferred shading is a multi-pass rendering method. The first pass collects relevant per-fragment shading information and writes it to preconfigured buffers on the graphics card. The second pass performs the shading only for actually visible fragments. The separation of generating the per-fragment shading information and the actual shading itself allows for efficient implementations of sophisticated lighting models.

3 Multi-GPU System Support

Multi-frame rate rendering and display is a parallel rendering technique improving interaction fidelity in complex virtual environments [1]. The results of asynchronously running image generators is displayed using either optical superposition or digital composition (cf. figure 2). The setup consists of a master node, a slow client (SC), and a fast client (FC). SC is designated for rendering all non-interactive parts of the scene. FC is responsible for rendering only those parts of the scene that are currently relevant to the user interaction (e. g. selected object(s), menus, cursors). The scene as a whole is distributed by the master node to the render clients. Objects migrate from SC to FC upon request (e. g. user selection) and vice versa (e. g. deselection). The number of objects that must be rendered by FC is (usually) much smaller than on SC, which leads to higher frame rates on FC and therefore to better interaction response while SC may render at low or even non-interactive frame rates.



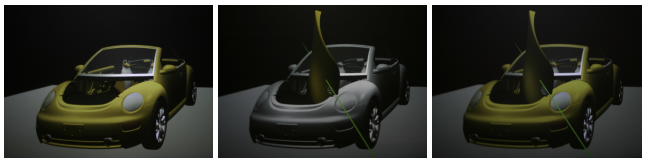
(a) Optical superposition. **(b)** Digital composition.

Figure 2: Multi-frame rate display methods [1].

We have implemented multi-frame rate rendering for a single computer system running with two graphics cards configured as independent devices using optical superposition and digital composition. Optical superposition does not provide correct occlusion of objects rendered by SC and FC. It is useful as a simple mechanism for overlaying interactive elements that do not need to be handled in a depth correct way (e. g. system controls). However, for the handling of interactive elements within the scene, where correct occlusion is necessary, digital composition provides a more viable approach. Therefore we will not further pursue optical superposition in the remaining discussion. Figure 3 shows digital photographs of an application prototype using multi-frame rate rendering by digital composition on a single-computer system with two GPUs.

3.1 Buffer Transfer Methods

There are several ways for transferring frame buffer content from SC to FC in a single-system multi-GPU setup using digital composition. First, existing infrastructure for multi-frame rate rendering in a graphics cluster can be reused. Instead of assigning SC and FC to different cluster machines both client applications are executed on the same machine but assigned to different GPUs. Frame buffer transfer uses the local loopback device. Second, by using process-shared memory instead of using the local loopback device. Third, depending on the scene graph API involved it is also possible to use process-local memory together with a multi-threaded application architecture.



(a) Slow part on GPU 1. **(b)** Fast part on GPU 2. **(c)** Digital composition.

Figure 3: Multi-frame rate rendering by digital composition on a single computer system using multiple image generators. (a) Scene part rendered by SC on GPU 1. (b) Scene part rendered by FC on GPU 2; frame-buffer image from SC in gray scale. (c) Final image on GPU 2 as perceived by the user.

We implemented multi-frame rate rendering prototypes with different scene graph APIs. We were using Avango [20], which is based on OpenGL|Performer [21], OpenSG [22], and an in-house 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 approach the actual scene graph traversal can not be separated in an asynchronous way; there is only one application traversal per application available which internally synchronizes the available cull and draw processes. In this case separate application instances need to be used 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 turned out to be non-trivial since this is an uncommon use case for a scene graph API. Thus, we decided to develop a minimal scene graph prototype allowing for asynchronously running graphics contexts. In both cases, i. e. OpenSG and our in-house development, process-local memory is available to share resources and communicate results between the graphics contexts.

Sending frame buffer content from SC to FC via local loopback device involves the use of a networking API usually based on sockets. A raw byte buffer representation of the frame buffer is created that also contains management information such as overall buffer size, frame counter, and the logical graphics buffer (left or right buffer for stereo applications). This raw byte buffer will be handed to the network API for sending. The network API however will copy that buffer internally at least once before actually sending it over the physical network. The buffer may also be split into several packets depending on the network protocol used. Since the actual send operation takes some time, it is important to decouple reading from the graphics device and sending with the network API because otherwise the send and flush operations of frame buffer sized buffers would stall the draw process, further delaying SC. Similar reasoning holds for receiving buffers on FC. Because the local loopback device does not actually read or write to an external device a speedup compared to actually using a physical Gigabit Ethernet device was expected. However, the network API's internal buffer copy for full screen sized frame buffers is still a limiting factor. In our experience the local loopback device will only provide a bandwidth of 2 Gbit/s on current Linux systems (based on kernels from the 2.6.x series).

To reduce the latency between reading the frame buffer on SC and uploading it on FC we implemented the buffer send/receive mechanism using process-shared memory between multiple application instances on the same system using a copy mechanism. This was achieved by reusing the infrastructure from the networking API and adding custom copy functionality. Unfortunately the bandwidth we achieved was just slightly better than the bandwidth achieved via local loopback device. This is propably due to the fact that reading from and writing to memory not only consumes bandwidth but also needs processing power. Memory bandwidth in modern PC systems is at least a magnitude higher than Gigabit Ethernet transfer performance. Concurrently reading and writing memory will degrade this bandwidth advantage.

Thus, copying frame buffers is not an option even on systems providing process-shared memory. Because of this we resorted to a zero-copy mechanism—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 for direct buffer transfers from one GPU to another without intermediate buffering in host memory. Process-shared memory must be used instead; a unique location in machine-local memory that is mapped to a virtual address within the processes attached to the shared memory segment. Because the attached processes all see the same memory only pointer values to the current buffer(s) in the shared memory segment must be adjusted to realize buffer updates. Our implementation uses a triple buffer approach so as to minimize the impact on the involved render processes. One buffer is assigned to SC for writing its frame buffer into it. Another buffer is used by FC for uploading the buffer's content into its frame buffer. The third buffer is a transfer buffer, which is either outdated after swapping with FC or it contains the most recently read frame buffer from SC. Once SC has read its frame buffer content, a lock is entered which is located in shared memory. While holding the lock SC swaps pointers to its buffer and the currently unused transfer buffer as well as setting a flag in shared memory. FC, as part of its frame update cycle, will also enter the shared lock once per frame and, upon finding the flag set, swap the pointer to the transfer buffer with its own buffer and reset the flag. We found that the overhead for this scheme is nearly unnoticeable, lock retention usually ≤ 0.1 ms. For scene graph APIs capable of asynchronous render traversals, such as OpenSG and our own 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 process-local memory (or heap memory) which avoids the use of a separate allocation API. However, the latency is similar to the case which uses shared memory.

As has been shown there are several ways of achieving multi-frame rate rendering using digital composition on a single-system dual-GPU setup. The methods vary in their degree of efficiency and depend mainly on the bandwidth available by the underlying transport mechanism. However, the zero-copy approach is clearly the method of choice, i. e. swapping pointers to buffers provides the lowest latency. Our implementations also show that multi-frame-rate rendering not only works on a dual-GPU setup but that infrastructure can be build allowing for selective use of the right communication and transfer path for specific application scenarios as well as a variety of hardware configurations.

3.2 End-to-End Latency Analysis

Latency decreases in a multi-GPU based multi-frame rate rendering system because the frame buffer image is transferred to and from host memory only within a single computer compared to the original cluster solution [1]. We will analyze the end-to-end latency for conventional rendering and multi-frame rate rendering using digital composition at a resolution of 1280×1024 pixels with a frame

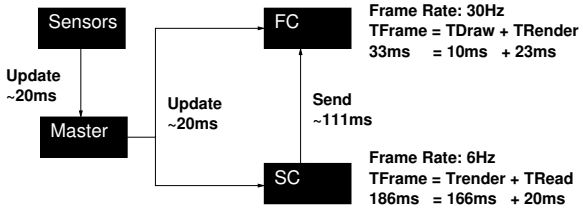


Figure 4: End-to-end latency for multi-frame rate rendering using digital composition in a cluster setup.

		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 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.

rate of 6 Hz for SC and 30 Hz for FC (cf. figure 4). In all cases sensor data and event updates are received with a latency of 40 ms by participating clients assuming an externally running tracking system sending new values at a rate of 50 Hz, i. e. every 20 ms. The end-to-end latency in a conventional rendering setup consists then of this sensor-update latency and the frame time of the render client, which is roughly the same as the render time of SC (TRender in figure 4). In the example this is 206 ms if event updates arrive at frame start which we assume for the following discussion. In the worst case the time for one tracking frame (20 ms) needs to be added.

In a cluster-based multi-frame rate rendering setup using digital composition sensor data is also received with a latency of 40 ms. On FC the frame time is split into the time needed for uploading the frame buffer from SC (TDraw in figure 4) and the time for rendering the relevant scene parts for the current interaction (TRender in figure 4). TDraw is the accumulated time for uploading color and depth values at a certain resolution. Table 1 indicates TDraw to be ≤ 10 ms for an image size of 1280×1024 pixels. This amounts to an end-to-end latency of 73 ms for FC. On SC the frame time consists of the time needed for rendering the relevant scene parts (TRender in figure 4) and the time for downloading the frame-buffer image (TRead in figure 4). 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 1. We assume TRead to be ≤ 20 ms. Additionally, the just read frame-buffer image must be transferred over the network to FC. For a resolution of 1280×1024 pixels this takes ≈111 ms as can be seen in table 2. The final end-to-end latency for an event to be incorporated into the frame-buffer image of SC, transferring the frame buffer to FC, and uploading it to FC's graphics card is then 337 ms, or 390 ms at most considering an additonal lag of 20 ms for sensor data as well as an additional render frame for FC of 33 ms. However, the end-to-end latency relevant for interaction is the end-to-end latency on FC. This is only 73 ms compared to 206 ms for the conventional rendering case.

End-to-end latency for event updates on FC 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, since we are still running a master application on a separate machine. Only FC and SC are running on a single machine. Because buffer transfer from SC to FC is realized by a zero-copy approach network-transfer latency is avoided. For our example the event-update latency on SC decreases in a multi-GPU setup to 226/279 ms.

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 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).

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 ≈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 SC and FC reduces the end-to-end latency of frame-buffer image updates from SC to FC 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 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 exhibited a significantly better balanced system load.

4 Migration Artifacts

Multi-frame rate rendering migrates objects from the slow client to the fast client if they are involved in an interaction. These objects have to be migrated back to the slow client once they are no longer manipulated. Since our implementations work with a replicated scene graph copy in each rendering process, migration requires only toggling a node mask on the concerned object. However, because parallel as well as asynchronous rendering processes are involved on the fast as well as the slow client, the following 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 typically takes 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. This will become only apparent if the user starts manipulating the object right away before the updated frame buffer from the slow client arrives. In this case both the moved object and the object in its original location will be shown. Otherwise the depth buffer takes care of the doubly rendered object and this artifact will not be visible to the user.
- **Release Artifact:** Once an object is no longer needed on the fast client it is deactivated on the fast client and 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 incorporating this change. During the intermediate period the object is not displayed at all and users perceive it as a popping artifact.

Our idea is to ameliorate the situation by using prediction and appropriate bookkeeping. For predicting the selection of an object we simply use the "over" status similar to 2D interfaces. Assuming ray-based selection the object is activated on the fast client as soon as the ray intersects the object. At the same time it is deactivated on the slow client. In our experience this heuristic works quite well since in most cases users need some time from entering the over status until selection. Fixing the release artifact seems simple at first glance: just keep the object active on the fast client until it is incorporated into the frame buffer of the slow client. However, it is slightly more complicated since 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 complicates state handling further.

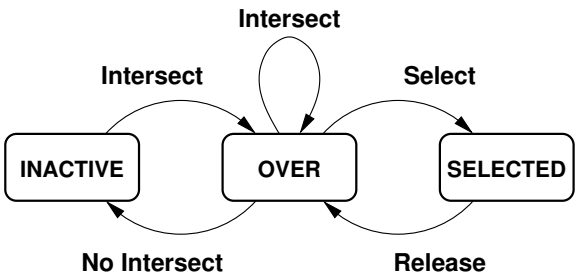


Figure 5: The state transition diagram for possible states of an object during an interaction.

To deal with this problem we need to know which objects were actually rendered into the frame buffer of the slow client that is currently in use on the fast client. Thus, the slow client not only needs to pass the frame buffer to the fast client, it also needs to provide the list of rendered objects (or the list of missing objects). On the fast client the selection and release process for an object can be described by the state diagram shown in figure 5. The fast client can be in one of three states with respect to an object in the scene graph: INACTIVE, OVER, or SELECTED. In addition there is the information if the object is contained in the currently used frame buffer from the slow client. Hence, there is only a total of six state combinations, which may occur.

Table 3 considers all possible state combinations and determines if the object needs to be rendered on the fast client or not. State combination 1 shows an INACTIVE object, which is not yet contained in the frame buffer of the slow client. Thus, it needs to be rendered on the fast client. State combinations 2 and 3 represent an object that is in state OVER or SELECTED but no longer contained in the frame buffer of the slow client. Thus, it needs to be rendered on the fast client as well. State combination 4 represents an INACTIVE object that is contained in the frame buffer of the slow client. State combinations 5 and 6 are the most interesting ones. State combination 5 represents a selected object, which is still contained in the frame buffer of the slow client, but the user may have started to move 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 6 may have several reasons. The user may have just moved 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 the 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. Then the object needs to be rendered on the fast client—possibly using a shadow object as well.

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

Table 3: Possible state combinations of an object on the fast client. The state variable SC denotes that the considered object is contained in the 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. States 5 and 6 indicate that a shadow object might be drawn on the fast client instead of the actual object.

Shadow objects are also used in distributed applications (e. g. by Benford and Mariani [23]) to indicate 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 as a regular object 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 real object. If the lock cannot be acquired the shadow object vanishes. We have experimented with the use of shadow objects, but since our current implementation of the above described state handling rarely generates these problematic cases, it is surprising to users if shadow objects actually appear for a split second while doubly drawn objects remain mostly unnoticed. For this reason we do not use shadow objects at the moment.

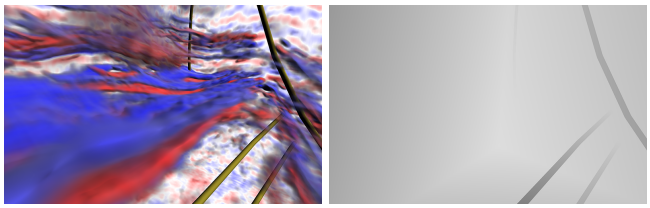
5 Volume Rendering Support

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 inside the semi-transparent seismic volume is necessary (e. g. for specifying a well path). Even though head tracking is available they often do not move much while they are placing and manipulating geometry representations such as well paths inside a translucently rendered volume. This observation was a driving motivation for developing multi-frame rate rendering in the first place [1].

In the context of multi-frame rate rendering such an interaction 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 SC carrying the volume rendering load and FC providing high frame rates for the interaction the final image would exhibit incorrect occlusions. The digital composition of SC's frame buffer and FC's render process cannot incorporate transparency information correctly. This is due to the fact that the color buffer from SC is an already finished image. This means that interactive geometries can appear correctly only in front of but never inside or behind the translucent volume. To ensure correct occlusion the volume has to be drawn after all opaque geometries, in particular after the interactively manipulated well geometry and thus shifting the computational load of the volume rendering to FC and diminishing all performance advantages.

Our solution, which allows correct blending of geometry with the translucent volume, is to detect screen-space portions where the volume potentially overlays the interactive geometry and to redraw only these portions on FC. Usually only small parts of the volume actually overlay manipulated objects (e. g. geometry in a well planning task consists only of thin tubes as shown in figures 1 and 7d). For this reason the volume rendering load on FC will be significantly lower compared to rendering the full volume on SC. Our approach guarantees that exactly the screen-space fragments covered by the interactive geometry trigger the volume rendering algorithm on FC, thus preventing unnecessary work on FC.

To prevent screen-space fragments from being processed we use stencil testing. OpenGL specifies stencil tests to be performed after the fragment processing stage. However, current graphics hardware does perform the stencil test prior to this stage, rejecting fragments before entering the fragment processing stage and thus circumventing potentially expensive calculations. We generate the required stencil mask during the rendering of the interactive geometry on FC. Since this rendering pass is executed against the depth image received from SC the mask spans precisely the visible fragments of the interactive geometry. While the frame-buffer image from SC contains the base color image of the non-interactive geometry combined with the fully rendered volume (cf. figure 6a), the depth image must contain only values generated by the non-interactive geometry (cf. figure 6b) to allow correct clipping with geometry rendered on FC. Subsequently executing the volume rendering algorithm on



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

Figure 6: Volume rendering in a multi-frame rate rendering setup, SC part. (a) shows the color and (b) the depth buffer content.

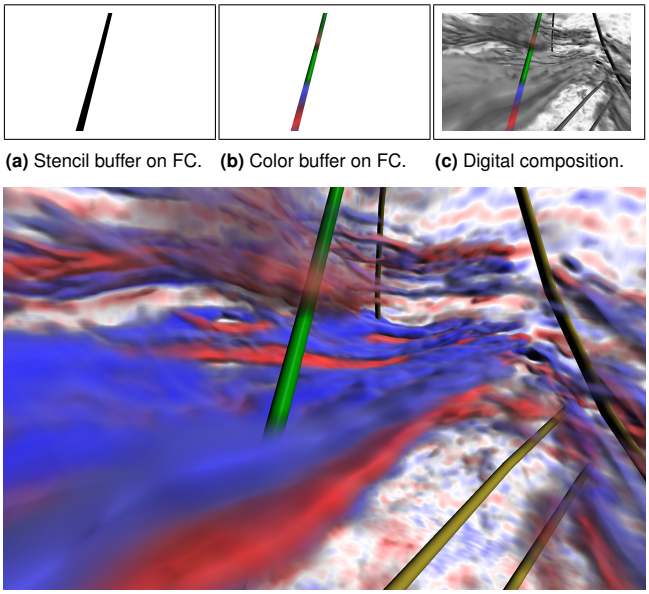
FC with stencil testing enabled will perform image updates for exactly those screen regions where the volume is actually covering interactive geometry.

Our volume visualization is based on ray casting similar to Stegmaier et al. [11]. By rendering only the bounding box of the volume data set a single fragment shader program is needed which implements the complete volume ray casting algorithm. Thus, the stencil tests discards complete rays instead of just fragments of slices as it would be the case for slice-based volume rendering, which renders a large stack of viewport-aligned polygons and 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 so prevents correct interaction of the volume with already rendered geometry. To achieve correct clipping behavior the fragment program needs to access depth information of previously rendered geometry. Therefore, we employ a two-pass approach. The first pass renders the entire scene geometry into off-screen buffers holding depth and color information. On FC 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 SC and FC is quite similar. Special care must to be taken on SC when to read the frame-buffer image components to be send to FC. The depth values must only contain the depth information of the geometry rendering pass. Therefore, the depth buffer content is downloaded directly after processing all scene geometry, preventing the volume rendering from modifying these depth values. After receiving the frame-buffer image on FC the geometry pass on FC is used to generate the stencil mask and update the volume rendering for a subset of screen-space fragments. Figure 7a shows a stencil mask generated on FC for the interactive geometry of the selected well object located inside the semi-transparent volume. As illustrated in figure 7b the actual scene contribution of FC is generated based on this mask. Figure 7c shows the digital composition of SC's and FC's contributions, highlighting the scene part generated on SC using gray scale. The final image composite as perceived by a user is shown in figure 7d.

Combining GPU-based volume ray casting with stencil testing on FC 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 that must be generated and traversed through the volume. The penetration depth determines the number of samples taken along the ray. The computational cost for traversing a ray through the volume is at most linear. Optimizations like early ray termination may reduce the number of samples that must be processed.

Our multi-frame rate volume rendering approach is based upon the same frame-buffer image as the original digital composition technique. However, depth and color values are downloaded at different points in time of the frame cycle on SC before being sent



(a) Stencil buffer on FC. **(b)** Color buffer on FC. **(c)** Digital composition.

(d) Final rendering on FC.

Figure 7: Volume rendering in a multi-frame rate rendering setup, FC part. (a) shows the stencil mask generated by the active geometry on FC while (b) shows the final image contribution from FC. (c) shows the composition of the contributions from SC (gray scale) and FC (colored). (d) shows the final image as perceived 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 NVIDIAQuadro FX 5600 at $\approx$1 Hz and 30 Hz, at a screen resolution of 1600×1200 pixels.

to FC. A potential pipeline stall is introduced on SC for reading the content of the depth buffer after the geometry rendering pass. Because the volume ray casting algorithm also requires the depth buffer content for correct interaction with already rendered geometry this stall is generated independently of the depth image read operation. The latency caused by the read operation can be hidden by using an asynchronous read-back mechanism (e. g. OpenGL's `ARB_pixel_buffer_object` extension), if no additional copying is necessary (e. g. when using a multi-threaded application scenario). This way the volume rendering is not delayed while the depth values of the geometry rendering pass are transferred to host memory.

Even though we used a volume ray casting approach the stencil-mask technique will also work for other volume rendering methods like traditional slice-based volume rendering. However, due to the overhead for the rasterization of the proxy geometry it will not be as efficient as using a ray casting approach.

6 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. Using multi-frame rate rendering this however will cause problems. While the geometry representing a light is rendered on FC, providing sufficient interaction speed, the underlying light abstraction from the graphics API is updated on SC only. This signals the user two disconnected visual cues: the light geometry or handle is updated appropriately and the light's influence on the scene is lagging behind. Using digital composition the final image contains the color and depth image generated by SC complemented by FC's additional rendering of the interactive scene parts. Because the base color image already contains lit pixel values the lighting stage cannot be moved to FC. However, if the frame-buffer image from SC would contain the information necessary for shading the pixels, the lighting can be shifted to FC. This can be

realized by using a multi-frame rate specific adaptation of deferred shading.

Deferred shading [14] separates rasterization and shading of geometry into two rendering passes. This is accomplished by first rendering the scene into multiple off-screen buffers storing per-fragment parameters needed for lighting calculations (e. g. position, normal as well as diffuse and specular reflectance information). 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 shading calculations. As a result the shading calculations are completely decoupled from the geometric complexity of the scene, thus allowing 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 target buffers instead of the usual color (and depth) buffer. For deferred shading four-component buffers are used. While 8 bits precision per component are sufficent for the specification of color parameters, 16 bits per component for the normal vectors yield better results [19].

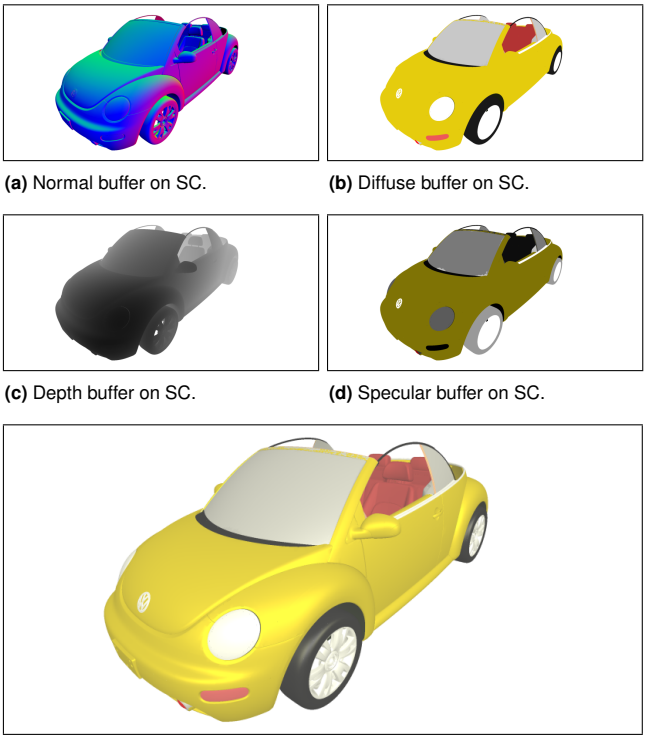
It is easy to see that the two-pass algorithm for deferred shading can be used within a multi-frame rate rendering setup. The geometry pass is run on SC while the shading pass is run on FC. Along with the depth value SC will also store the interpolated normal as well as diffuse and specular reflectance coefficients of the current material for each fragment. We use a 16 bits per-component float buffer for the normal data and 8 bits per-component float buffers for the material parameters. These buffers are transferred together with the depth buffer to FC (cf. figure 8a to 8d).

FC executes the shading pass using the buffers from SC as input textures for 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 the base image (cf. figure 8e). Alternatively, they may also be rendered into the multiple render target buffers from SC allowing for a unified lighting shader at the end of FC's render process.

Our multi-frame rate deferred shading method allows for user interaction with light representations as well as light parameters at interactive frame rates even though these manipulations affect the entire scene. However, there is also a cost involved because more buffers have to be transferred from SC to FC. For the original multi-frame rate rendering method using digital composition 64 bits per pixel are necessary; i. e. 32 bits color and 32 bits depth. The minimum bit size per pixel for deferred shading is rather 144 bits: 32 bits depth, 48 bits normal (i. e. 16 bits per component), 32 bits diffuse, and 32 bits specular color coeffecients. To increase the numeric precision the normal values might even be stored as 32 bit values per component instead of 16 bit values, which increases the 48 bits buffer size for the normals to 96 bits. This means that in any case more data must be transferred from SC to FC.

Beside the depth and normal buffer generated by SC the remaining buffers contain material descriptions. Instead of actually storing the material parameters, it is much more efficient to store only material indices on a per-fragment basis on SC. The material palette and the stored indices are then used on FC 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. On the other hand, a scene traversal is needed to associate all geometry with its material index as well as generating a material palette texture. This may be problematic in highly complex scenes or very dynamic environments.

Apart from the increased buffer sizes of this approach it also shifts computational load from SC to FC. In the original setup SC would not only have to transform and rasterize potentially large sets of geometry but also apply shading on the resulting fragments as well. With deferred shading these calculations are moved completely to FC. Even though the complexity is limited by the screen resolution, an actual shading program might implement an expensive lighting model, which may decrease the frame rate on FC more than is 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 currently not manipulated by the user. The shading for non-interactive lights can be moved back to SC leaving only the active lights for processing on FC. This approach bears the cost of requiring an additional render target buffer on SC, which must also be transferred to FC, but may outweigh the computational cost imposed by the choosen shading model.



(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 8: Deferred shading in a multi-frame rate rendering setup. Figure (a) to (d) show the content of the render targets from SC holding normals, diffuse reflection coefficients, depth, and specular reflection parameters, respectively. Figure (e) shows the final shading using the buffer contents from SC together with the currently active lights as computed on FC. The model shown consists of 3 M triangles. The clients were each running on an NVIDIA Quadro FX 5600 in active stereo mode using a screen resolution of 1600×1200 pixels. Frame times were 80 ms for SC and 6 ms for FC. Four point lights were enabled on FC.

7 Conclusions and Future Work

We have presented advanced multi-frame rate rendering techniques, which are generally applicable to cluster-based as well as single-system multi-frame rate rendering setups. The implementation of multi-frame rate rendering on a single multi-GPU system simplifies software development and significantly reduces buffer transfer limitations. Visual artifacts which may occur when objects need to migrate from the slow to the fast graphics card, and vice versa, can be mostly hidden by intelligent state management. Our volume rendering support makes use of the fact that object manipulation inside a volume often affects only a limited number of pixels—and only those pixels need to be updated on the fast client for every

frame. Our multi-GPU deferred shading approach computes per-pixel normals as well as material information for the whole scene on the slow client, while the actual shading is computed on the fast client enabling interactive light manipulation at high frame rates.

Multi-frame rendering techniques can also be applied to systems with only a single GPU by interleaving the rendering process of the fast and slow client. The entire scene needs to be partitioned such that each part requires approximately the same 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 our technique. In addition, since 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.

Depth-image warping is used in image-based rendering to generate novel views from given reference images considering per-pixel color and depth information, which is also available with our multi-frame rate rendering technique. Post-rendering 3D warping [24] 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 proven to be quite effective in exploiting view point coherence. We are planning to combine post-rendering 3D warping with multi-frame rate rendering to improve performance for navigation tasks.

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 virtual environments.

Acknowledgments

We thank the NVIDIA Professional Solutions Group Europe for continued support and hardware donations. This work was supported in part by the VRGeo Consortium. Seismic data from Wytch Farm oil field courtesy of BP, Premier Oil, Kerr-McGee, ONEPM, and Talisman. The VW New Beetle model courtesy of Volkswagen AG.

References

[1] 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, 2007.

[2] K. Akeley. Reality Engine Graphics. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 109–116. ACM, 1993.

[3] 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.

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

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

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

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

[8] M. Levoy. Efficient Ray Tracing of Volume Data. *Trans. on Graphics*, 9(3):245–261, 1990.

[9] 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, 2003.

[10] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, pages 38–43. IEEE, 2003.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] H. Tramberend. *Avocado: A Distributed Virtual Reality Framework*. PhD thesis, Universität Bielefeld, 2003.

[21] 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.

[22] 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.

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

[24] 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.