Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments

Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek

Saarland University, Computer Graphics Group
Im Stadtwald - Building 36.1, 66123 Saarbrücken, Germany
Email: {apomi|marmitt|wald|slusallek}@cs.uni-sb.de

Figure 1: Some example mixed reality applications using our framework: a) A virtual room illuminated by live video on the TV set, b) Two video billboards integrated into a scene with correct shadows and reflections, and c) A virtual car in a real environment, also with ray traced shadows and reflections.

Abstract

The realm of mixed reality applications lies in blending rendered images with images of the real world. This requires highly realistic rendered images in order to seamlessly blend between those two worlds. However, current rasterization technology severely limits the achievable realism and imposes strict limits on the scene complexity and the optical effects that can be simulated efficiently. Real-time ray tracing can overcome many of these constraints and enables completely new approaches for mixed reality applications.

This paper explores this design space based on a framework for live streaming of video textures in a real-time ray tracing engine. We also suggest a novel approach to video-based AR by integrating image compositing with shading computations. We demonstrate the approach with a number of VR/AR applications including video inserts, video billboards, and dynamic lighting from video and HDR video streams. Being seamlessly integrated into the ray tracing framework, all our applications feature ray traced effects, like shadows, reflections and refraction.

1 Introduction

Mixed reality applications range from augmenting videos with photo-realistic synthetic objects (e.g. in design review applications or assembly guides) up to inserting real persons into virtual worlds (e.g. virtual TV studios [10] or immersive tele-conferencing). For combining the real and the synthetic world the renderer needs as much information about the real world as possible, including accurate geometry, lighting conditions, and material properties. Due to its ability to correctly simulate light transport, the ray tracing algorithm is perfectly suited for this task. However, it has been much too slow in the past.

Recent advances in real-time ray tracing systems [25] provide new opportunities for high-quality rendering including physically correct lighting, reflection, and refraction effects, which paves the way to novel interactive applications. This suggest that it is well worth exploring this design space for interactive mixed reality applications.

Since the real-time reconstruction of accurate 3D models of non-trivial real environments is still very limited, interactive applications often use rather simple models, such as live video captured from the surrounding environment. The integration of live video into the ray tracing process is therefore a nec-

essary first step for realizing AR/VR applications.

In the following we provide a framework for streaming and synchronizing live video in a distributed ray tracing system using it as *video textures* in the shading process. We also suggest a method for compositing video streams and normal shading data for video-based augmented reality (AR) applications [1]. Note that the term *video textures* is also used for pseudo-randomly looped, animated textures introduced by Schödel et al. [21]. In the remainder of this paper we use the term to refer to textures from live video streams.

1.1 Outline

The rest of this paper is structured as follows: In the next section we briefly review real-time ray tracing and discuss the problems arising in the context of integrating video textures into a distributed ray tracing engine. We suggest a framework for integrating streaming video textures in the rendering system using multicast networking (Section 3) and streaming AR view information (Section 4.1). We then describe our implementation based on the OpenRT system (Section 4) and discuss a number of sample applications that benefit from our modular approach (Section 5). Finally we summarize and discuss our results and suggest future applications and improvements.

2 Video Textures for Distributed Ray Tracing

Recently real-time ray tracing has become feasible due to algorithmic improvements, optimized implementations, and distributed computing on a cluster of commodity PCs [25]. Already today, real-time ray tracing enables novel applications such as interactive global illumination [3] or the visualization of complex refraction and reflection scenarios [2]. A new API for ray tracing (OpenRT [8]) that is similar to OpenGL simplifies the design and implementation of new applications profiting from the advanced optical effects and the high modularity of ray tracing.

While hardware support for ray tracing is being developed [20, 19, 17], these technologies are not yet available. High rendering quality at realistic resolutions and frame rates still requires to use parallel and distributed ray tracing, for example by using a cluster of PCs.

A typical distributed ray tracing architecture consists of a rendering server and a number of rendering clients connected by commodity networks (see Figure 2a). The server runs the ray tracing application which communicates with the rendering engine through the OpenRT API [8]. The ray tracing library transparently performs the distribution and parallelization of rendering jobs across the clients by splitting the frame buffer into rectangular *tiles* (e.g. 16×16 pixels) and distributing them to clients on request. As long as enough jobs are available this allows the rendering performance to scale efficiently by simply adding more rendering clients.

For video textures we need a way to stream the video information to all rendering clients. The simplest way – an explicit network connection (TCP/IP) to each client – does not scale with the number of clients as the network bandwidth of the server increases linearly with each new client.

As an alternative we could use a demand driven approach, having clients requesting texture data from a video server on demand. Because not all clients need all texels, this would reduce the network bandwidth. However, the high latency due to the commodity network is not tolerable for real-time applications. Another alternative would be to provide the video information via a separate wiring to the clients, and installing a video frame grabber in each client. However, this approach is technically not very practical and would require even more sophisticated synchronization methods.

Because all clients might require some part of the video data, a 1-to-N communication mechanism such as *IP multicast* seems the best solution. This is particularly true for local networks that minimize the routing issues of multicast traffic. However, it requires that the system can cope with synchronization issues and packet losses typically appearing in such networks.

3 Video Streaming with IP Multicast

IP multicast transport [23] has been designed for scalable streaming of audio and video across the Internet. It provides a simple solution for distributing data to a number of hosts in parallel. The multicast IP address represents not a single host but a group of hosts and multiple hosts can send packets on the

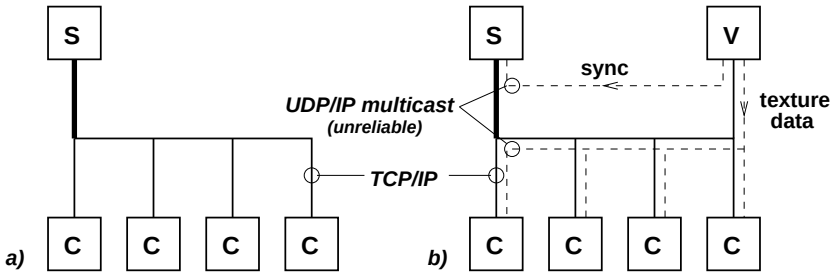

Figure 2: a) OpenRT distributed ray tracing system. Each client (C) has a (reliable) TCP/IP connection to the rendering server (S). The server has a higher bandwidth network connection (e.g. Gigabit). b) System extended with streaming video textures. The video texture server (V) sends sync information and texture data on separate (unreliable) multicast groups.

same group. Hosts can dynamically join and leave multicast groups. The network routes the packets to hosts that have subscribed to a stream and copies the packets at routers and switches as required.

Due to the 1-to-N relation between sender and the receivers IP multicast only works for UDP datagram packets. The lack of an acknowledging mechanism causes reliability issues with traditional multicast, i.e. there is no way to guarantee that packets have been properly delivered.

Loss of packets in LAN environments is mostly caused by packet queue overflows within switches and the operating system. This requires careful control of the maximum network bandwidth. Several protocols (e.g. Reliable Multicast Protocol RMP [27]) have been designed to increase the reliability of IP multicast. However, they are designed for WAN use and increase latency, which makes them unsuitable for interactive applications.

3.1 System Architecture

Our framework uses a video server for streaming the video textures to the ray tracing clients (Figure 2b). The video servers run on separate hosts to minimize the bandwidth impact on the rendering server. A typical server uses a frame grabber board for live video input where resolution and frame rate can be adjusted dynamically. For testing purposes it can also use pre-recorded video from files.

UDP packets cannot exceed a size of 64Kb on most systems and since media like Fast Ethernet uses packet sizes of 1500 byte, video frames need to be split into packets before streaming. A packet consists of a header and part of the texture color data (e.g. one image row). The header contains information about the texture format (width, height, color coding), the position of the texture color data in the frame, as well as a synchronization timestamp (video frame number).

3.2 Synchronization

Distributed ray tracing systems render tiles of the final image frame asynchronously, i.e. different hosts might start computation of a frame at different times. In order to avoid tiling artifacts, we need a synchronization mechanism to assure that all clients rendering the tiles of one *rendering frame* use the corresponding *video texture frame*. While the frame rate of the video texture is fairly constant, the rendering frame rate might vary depending on the cost of the current view.

To avoid synchronization overhead, the clients are usually unaware of the processing of all other clients. To synchronize the clients' video textures, the server therefore has to perform synchronization [25]. If the server also listens to the streamed video it knows which texture frames have just been transmitted to the clients and can distribute this information with each new frame.

Because the rendering server is already a network bottleneck we should not increase its load with the video data. As it only needs timestamps and not the video content, we transmit small packets containing the texture ID and the timestamp of the last frame sent on a separate multicast group. Thus the server must only receive this sync information for making its global decision. Once the server has decided on the video texture frame to be used for the given ren-

dering frame, it can safely sent this single integer value to all clients via reliable unicast without having to consider bandwidth limitations.

3.3 Data Formats

Video texture data comprises not only the color information for the texels but also the resolution of a texture frame (width and height) and a *format* identifier. The latter is used for decoding and correct texture interpolation. A texture interpolator is a function to map texture coordinates to memory addresses with additional texture filtering (e.g. bilinear interpolation).

By providing a table of those interpolators we can use different texture coding formats, from packed 16bit RGB (RGB565) up to formats containing an alpha channel or even special formats like RGBE [26] for videos with high dynamic range data. One could also use data compression to decrease the bandwidth or exploit scalable video coding with multi-resolution formats for hiding packet losses (e.g. MPEG).

In addition, we sometimes require streaming of arbitrary data structures (e.g. a table of light samples, see Section 5.4). A *raw format* without interpolation can provide such a mechanism. It takes special care of packet losses (e.g. by using fragmenting writes [23]).

Since multicast is based on UDP packets there is no guarantee that video packets arrive on every client or even leave the sending host. Optimizing buffer sizes and a carefully timed sender decrease the amount of lost packets but still provide no reliability. Instead we favor a rather simple approach: Lost parts of a texture frame are tolerated by reusing the information stored in previous frames. In practice this seems to work well if single rows get lost. 'Burst losses' can easily be hidden by sending the image rows in random order.

4 Implementation Within OpenRT

A major goal of our implementation in the OpenRT system [8] was to leave the rendering kernel unchanged. OpenRT is expandable by a number of run-time loadable plug-ins like shaders, camera models, light source types etc. This makes a plug-in implementation of streaming video textures simple and straightforward.

The plug-in on the server receives the information from the sync multicast group, updates a table listing the latest timestamp for each video texture, and forwards it to clients for new frames. A *Video Texture Manager* on each client provides access to the textures for any shader. It creates a separate thread for receiving the data and manages the synchronization. A separate texture object for each video stream allows multiple shaders to access the same video. Via unique IDs assigned by the video server the application can dynamically switch between multiple video textures used by a specific shader.

A video texture server is a stand-alone application running on an arbitrary host. It uses a custom server library that provides a simple API for streaming textures and data structures. This library hides the synchronization mechanism to the user. We implemented a simple server using the Video4Linux [16] API for access to frame grabber boards. Another implementation offers playback of previously recorded video files.

By combining the video texture server library with the Network Multimedia System NMM [9] more complex scenarios could be realized. Examples could be the playback of DVDs, streaming of digital television programs, and video conference codecs.

4.1 AR View Compositing

AR applications augment a view of the real world with computer generated images. One option for compositing are special AR hardware devices like semi-transparent video glasses that physically performs the composition. As an alternative, we can use a camera to capture the user's view and perform compositing in the computer (*video-based augmented reality* [1]). In this case, the renderer provides not only an image of the synthetic part of the scene but also a *matte* in form of an alpha channel that masks the foreground objects. The matte is used for compositing the two video images on the rendering host [4] or with external *keyer* hardware [12].

Obviously, one could use the same mechanisms with the OpenRT architecture. However, we prefer to use the built-in shaders to do the compositing operations. We provide the video background color as an input for the shading of a pixel. Hence we can overwrite the color for opaque objects or just mod-

Figure 3: Room with TV set. a) Video texture on the TV screen with ambient lighting. b) The same scene with the video texture acting as a light source. The TV screen is subdivided into 5x5 light samples that illuminate the scene. The ambient term adapts to the sample average. Note the reflections on the table and the soft shadows cast by the table's legs.

ify it for transparent objects. This also allows us to perform differential rendering [6, 18] for creating lighting effects like shadows or caustics on the real background.

As an example we can add a shadow to a synthetic object by approximating the geometry and reflectance of the real scene with simplified synthetic stand-in objects (like a 'shadow catching' plane on the floor [13]). We use the shader of this plane to modulate the background video input color according to the ratio of incident light with and without the synthetic objects (see application example in Section 5.4) [18].

We could use the above multicast mechanism to also stream the background video to all clients. However, in order to minimize latency and improve reliability the rendering server distributes this video to the clients together with the tiles using the reliable TCP/IP connection. In contrast to the multicast method a client only receives the background for its part of the final rendered image and not the whole background video image. The use of TCP/IP eliminates disturbing artifacts caused by packet losses.

Note that this method does not significantly increase the network load compared to using the multicast approach as we are just using the unloaded direction of the full-duplex Ethernet connection.

By using a simple shader that just returns the background color for stand-in objects we can selectively hide parts of the synthetic scene and simulate view dependent occlusion. However, as we only have access to the background color directly behind a shaded object this approach does not allow for computing refraction effects.

5 Applications

The above framework now allows for novel approaches by using the seamless integration of video textures into a real-time ray tracing system. In the following we will briefly discuss some example applications.

5.1 Live Video Inserts

The most obvious application of streaming video textures is live video inserting, i.e. using the video stream as a dynamically changing texture for any type of shader.

The scene shown in Figure 3a consists of a simple living room scenario with a television set. The shader used for the screen material simply returns the color from the appropriate video texel. The integration into a ray tracing system automatically provides the reflection on the table and other reflective objects.

5.2 Dynamic Lighting from Video Textures

Figure 3b shows the same scene but this time the room is illuminated by the TV set. To achieve this effect we simply placed an array of point lights in front of the TV screen. The light source shaders have access to the video texture, and compute their color and intensity based on the texture data. We also use the video texture to control the level of ambient light in the room. Since lighting is calculated from scratch for each frame the direct illumination in the scene resembles the light emitted by the TV screen.

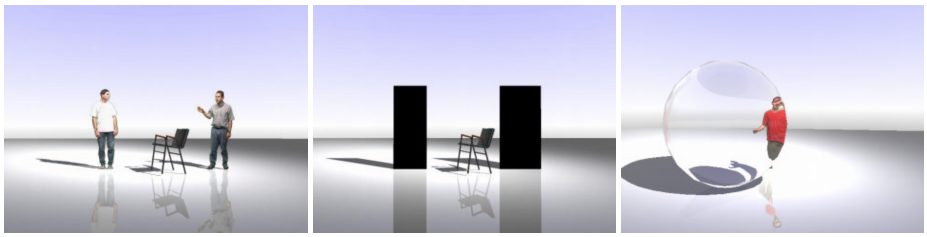

Figure 4: Video billboards: Chroma keying is done in the shader, which automatically makes the keying information available for other shaders computing shadows, reflections and refractions. a) Two persons in a virtual environment. The persons are captured separately in front of a green-screen. b) The billboard objects without video textures. c) Refraction of a person standing behind a glass sphere.

This scenario can easily be extended for indirect lighting by using the instant global illumination algorithm of the OpenRT system [24, 3].

5.3 Video Billboards

Another interesting application of video textures is augmenting a rendered scene with real persons e.g. for tele-presence applications or virtual TV sets. The input video from a camera is segmented into foreground and background using background subtraction or chroma keying methods [22]. However, instead of segmenting the signal on the video server and streaming the signal together with the matte information (alpha channel) we can take advantage of distributed ray tracing and perform the segmentation also in the shader. This reduces the bandwidth and performs segmentation only for texels actually visible in the current frame.

The areas of a video billboard segmented as foreground simply show the video texture, possibly modified by the shader, while the background area is computed by tracing a transparency ray. Since this works not only for primary but for arbitrary rays, shadow and reflection effects even on curved surfaces are simulated without any additional effort (Figures 4 and 7).

Due to their 2D nature billboards have some drawbacks: They can look distorted for shallow viewing angles. Furthermore, if the person moves back and forth relative to the video camera, the silhouette may move up and down in the video image causing the person to 'hover' in the composite image. A 3D reconstruction of the person using a *visual hull* [5] and a volume shader combined with video textures for the view cameras would provide a more realistic solution. However, this has not yet been implemented.

5.4 Dynamic Lighting With Real World Illumination

Video textures and real-time ray tracing also allow for interactive lighting of virtual objects with real-time captured lighting. Together with a live background video stream (see Section 4.1) this significantly improves the realism of AR scene (Figures 5 and 8).

We use a video camera with remotely controllable shutter time together with a 180 degree fisheye lens to capture the incident light in the upper hemisphere (Figure 5a). By quickly changing the shutter times and capturing frames at different exposure levels we can reconstruct a high dynamic range (HDR) video stream [7, 14]. HDR calculation is done on a special video texture server that then streams a high dynamic range video in RGBE format [26]. This video texture server also samples the HDR lightprobe image and streams a table of directional light samples to all clients.

The RGBE video texture is then applied as an environment map [11] for illumination [15] and for specular reflections. A plane under the object acts as stand-in for the real floor and is used for shadow generation [6, 18].

6 Results

Most of the applications shown here could also be implemented using rasterization hardware. However, they are much easier and straightforward to

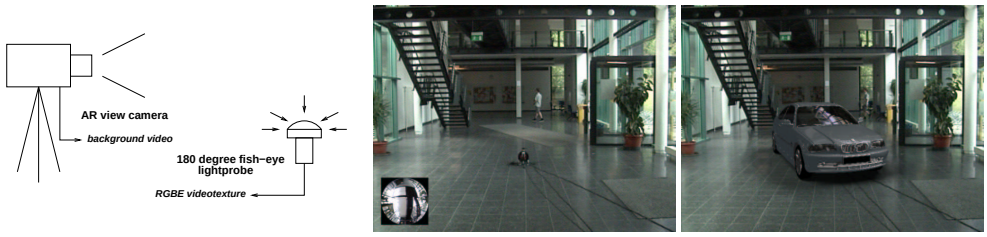

Figure 5: Dynamic lighting of a virtual car. a) Setup: A video camera captures the scene view while a second camera with a 180 degree fish-eye lens is used to capture the incident light in the upper hemisphere. b) The background with the lightprobe. c) The final composited image. Note the soft shadow on the floor.

scene	figure	triangles	#CPUs	resolution	video texture	fps
TV w/o lighting	3a + 6	7237	24	640x480	130x100 @ 25fps	19
TV w. lighting	3b + 6	7237	24	640x480	130x100 @ 25fps	5.5
BBD w. lighting	4a + 7	424	24	640x480	220x420 @ 25fps	10
BBD w. glass sphere	4c + 7	824	24	640x480	220x420 @ 25fps	6.4
AR car	5 + 8	208259	24	640x480	350x350 @ 1fps	2.6

Table 1: Frame rates achieved on our example applications. Note that due to limitations in camera speed the environment map in the HDR lighting example (car) is updated at only 1 fps.

implement in a ray tracing environment. The modularity of an OpenRT based rendering system assures high re-usability of code like shaders and easy combination of rendering effects like reflections, refraction, and shadows. All shading effects combine automatically and correctly with all other objects in the scene.

The frame rates in the example applications are given in Table 1. The rendering clients are dual Athlon MP 1800+ PCs. They are connected via FastEthernet (100Mbit/s) to a multicast capable switch. The rendering server is connected via a Gigabit (1000Mbit/s) uplink. No compression is used for video textures.

The AR view composition method presented in Section 4.1 runs at 19 fps at video resolution (640x480). For lower resolutions the full video frame rate of 25 fps is being achieved. Note that our current OpenRT implementation is limited to 19 fps for resolutions higher than 640x480 due to load balancing issues. Video delay is about four frames. No change in the rendering frame rate is noticeable when switching video streaming on and off in the AR view compositing.

7 Conclusion

In this paper we presented a framework that extends the real-time OpenRT system by using streaming video textures and multicast networking. We have demonstrated its use with several AR and mixed reality applications.

Compared to sophisticated rasterization hardware with video stream support used for hard real-time applications (i.e. virtual TV studios), interactive ray tracing on commodity hardware cannot provide the low latency in terms of video delay in combination with fixed frame rates (e.g. PAL 50 fields rendering in TV applications) yet. This is mainly caused by distribution aspects like low network bandwidth and load balancing issues. For applications with weaker requirements where image quality and modularity in terms of programming is important, interactive ray tracing provides a cheaper and more flexible alternative compared to specialized hardware.

7.1 Future Work

The streaming mechanism described here could be improved by frame rate feedback to the video servers. The existing synchronization multicast

group could be used for this purpose. The video server could then adapt sending rate of the video to the rendering frame rate.

Furthermore, the video texture bandwidth could easily be reduced by introducing image compression. Compression and error-correction schemes should be used that can correct or hide lost packets.

In the future we will concentrate our work on the 3D reconstruction briefly mentioned in Section 5.3. This would allow for real 3D compositing and better integration of the real and virtual worlds. We believe that 3D compositing is a major key for realistic mixed reality rendering.

Acknowledgements

The authors wish to thank Jörg Schmittler for the TV room scene and also Andreas Dietrich, Carsten Benthin, Kim Herzig, Simon Hoffmann, and Christian Linz for their help.

This work has been supported by Intel Corp.

References

[1] R. Azuma. A survey of augmented reality. *Computer Graphics (SIGGRAPH'95 Proceedings, Course Notes #9: Developing Advanced Virtual Reality Applications)*, pages pp. 1–38., 1995.

[2] Carsten Benthin, Ingo Wald, Tim Dahmen, and Philipp Slusallek. Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of Eurographics Workshop on Parallel Graphics and Visualization (PGV)*, pages 81–88, 2002.

[3] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A Scalable Approach to Interactive Global Illumination. to be published at Eurographics 2003, 2003.

[4] Ron Brinkmann. *The Art and Science of Digital Compositing*. Morgan Kaufmann Publishers, Inc., first edition, 1999.

[5] Chris Buehler, Wojciech Matusik, Leonard McMillan, and Steven Gortler. Creating and rendering image-based visual hulls. Technical Report MIT/LCS/TR-780, Massachusetts Institute of Technology, 1999.

[6] Paul Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. *Computer Graphics*, 32(Annual Conference Series):189–198, 1998.

[7] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. *Computer Graphics*, 31(Annual Conference Series):369–378, 1997.

[8] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The OpenRT Application Programming Interface – Towards A Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium*, Darmstadt, Germany, 2003. Available at http://www.openrt.de.

[9] NMM Network-Integrated Multimedia Middleware for Linux. http://www.networkmultimedia.org/.

[10] Oliver Grau and Graham Thomas. Use of image-based 3D modelling techniques in broadcast applications. *2002 Tyrrhenian International Workshop on Digital Communications*, 2002.

[11] Ned Greene. Environment mapping and other applications of world projections. *IEEE CG&A*, 6(11):21–29, November 1986.

[12] Jack Keith. *Video Demystified. A Handbook for the Digital Engineer*. Harris Semiconductor, second edition, 1996.

[13] Doug Kelly. *Digital Compositing In Depth*. Coriolis, 2000.

[14] Steve Mann and Rosalind W. Picard. Being 'undigital' with digital cameras: Extending dynamic range by combining differently exposed pictures, 1995.

[15] Gene S. Miller and C. Robert Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environments. *SIGGRAPH 84 Advanced Computer Graphics Animation seminar notes*, 1984.

[16] Video4Linux Project. http://www.exploits.org/v4l/.

[17] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002. (Proceedings of SIGGRAPH 2002).

[18] Imari Sato, Yoichi Sato, and Katsushi Ikeuchi. Acquiring a radiance distribution to superimpose virtual objects onto a real scene. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):1–12, 1999.

[19] Jörg Schmittler, Alexander Leidinger, and Philipp Slusallek. A Virtual Memory Architecture for Real-Time Ray Tracing Hardware. *to appear in Computer and Graphics, Volume 27, No. 5*, 2003.

[20] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR – A Hardware Architecture for Ray Tracing. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 27–36, 2002.

[21] Arno Schödl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video textures. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 489–498. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[22] Alvy Ray Smith and James F. Blinn. Blue screen matting. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 259–268. ACM Press, 1996.

[23] W. Richard Stevens. *UNIX Network Programming. Networking APIs: Sockets and XTI.* Prentice-Hall, second edition, 1998.

[24] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques 2002*, pages 15–24, 2002. (Proceedings of the 13th Eurographics Workshop on Rendering).

[25] Ingo Wald and Philipp Slusallek. State-of-the-Art in Interactive Ray-Tracing. In *State of the Art Reports, Eurographics 2001*, pages 21–42, 2001.

[26] Greg Ward. Real pixels. In James Arvo, editor, *Graphics Gems II*. Academic Press, 1992.

[27] Brian Whetten. A reliable multicast protocol. *Theory and Practice in Distributed Systems. Birman, Mattern and Schiper, eds., Lecture Notes on Computer Science, 938*, 1995.

Figure 6: Room with TV set. The video texture is used for both: providing the video image on the TV screen and controlling the direct lighting of the room. Note the shadows on the floor and the reflections on the table. *Left top, left bottom and center top*: Different lighting situations. *Center bottom*: Without lighting by the TV set.

Figure 7: Video billboards. *Left top*: With and without shadow and reflection. The person seems to hover without shadow. *Left bottom*: Glass refraction and reflections. *Right*: Two persons rendered with separate video textures.

Figure 8: Dynamic AR lighting. *Left*: Background video for AR view compositing and fish-eye HDR lightprobe. *Right*: Car model composed over background. Note the soft shadow on the real floor.