A Programmable Display Layer for Virtual Reality System Architectures

Ferdi Alexander Smit, Robert van Liere, and Bernd Froehlich

Abstract—Display systems typically operate at a minimum rate of 60 Hz. However, existing VR-architectures generally produce application updates at a lower rate. Consequently, the display is not updated by the application every display frame. This causes a number of undesirable perceptual artifacts. We describe an architecture that provides a programmable display layer (PDL) in order to generate updated display frames. This replaces the default display behavior of repeating application frames until an update is available. We will show three benefits of the architecture typical to VR. First, smooth motion is provided by generating intermediate display frames by per-pixel depth-image warping using 3D motion fields. Smooth motion eliminates various perceptual artifacts due to judder. Second, we implement fine-grained latency reduction at the display frame level using a synchronized prediction of simulation objects and the viewpoint. This improves the average quality and consistency of latency reduction. Third, a crosstalk reduction algorithm for consecutive display frames is implemented, which improves the quality of stereoscopic images. To evaluate the architecture, we compare image quality and latency to that of a classic level-of-detail approach.

Index Terms—Display algorithms, virtual reality, image-based rendering.

1 Introduction

Modern VR-applications typically consist of a head-tracked user, a simulation process, (photo)realistic rendering, and a stereoscopic display system. To generate images on the display, the application first calculates one time step in the simulation. A scene graph is constructed from the simulation data, which is then rendered by the graphics hardware into frame buffers for the left and right eyes. Head tracking information is used to determine a correct viewpoint for the rendered scene graph. Finally, the left and right buffers are scanned out to the stereoscopic display by a buffer swap. We call such an update of the display by the application an *application frame*. This is shown in Fig. 1a, corresponding to the VR-latency model proposed by Mine [1].

Display systems typically operate at a minimum of 60 Hz for monoscopic viewing, or 120 Hz in the case of stereoscopic viewing (60 Hz per eye). This implies that consecutive images on the display are only visible for approximately 16.7 ms. We call these consecutive images on the display *display frames*. The process of running a simulation and rendering typically takes much longer, either due to a time-consuming simulation process or due to the rendering of complex geometry and lighting. Consequently, the display is not updated by the application every display frame. The default behavior of a display system is to repeat a display frame when no updates are provided by the application. We call this repetition *repeated application frames*, since an application frame is repeated for several display frames until an update is provided by the application. This is shown in Fig. 1b. When a 60 Hz frame rate cannot be realized otherwise, classic VR-architectures often use level-of-detail approaches to reduce the number of polygons rendered by such an amount that new application frames can be generated at 60 Hz. However, this comes at the cost of reduced image quality.

Current graphics APIs and display hardware are highly rigid with respect to display updates. The default behavior of repeating the same display frame over and over in the absence of application updates can virtually never be changed. The only real control the application has over the display is in filling a frame buffer and requesting a buffer swap. The frame buffer is then read from memory and scanned out to the display whenever the hardware sees fit, usually at the next display refresh signal. The slow update rate and repetition of application frames on the display cause a number of problems and limitations for VR-applications. Examples of such limitations are inefficient latency reduction [2], inefficient crosstalk reduction [3], and perceptual artifacts in the form of judder [4].

Our motivation for this paper is that we desire an architecture where the display can be controlled more specifically. In particular, we want to replace the default behavior of repeating display frames by a programmable layer that can execute a custom program for every individual display frame. This layer should run independent of the application in such a way that the generation of new application frames is not postponed. We call this layer the *programmable display layer (PDL)*, which is illustrated in Fig. 2. The application is responsible for generating new application frames. In addition to color and depth, it also generates a

- *F.A. Smit and R. van Liere are with Centrum Wiskunde and Informatica (CWI), PO Box 94079, NL-1090 GB Amsterdam, The Netherlands. E-mail: {ferdi.smit, robert.van.liere}@cwi.nl.*
- *B. Froehlich is with Bauhaus-Universität Weimar, Bauhausstr. 11, Fakultät Medien, 99423 Weimar, Germany. E-mail: bernd.froehlich@medien.uni-weimar.de.*

Manuscript received 15 Feb. 2009; revised 29 Apr. 2009; accepted 7 May 2009; published online 29 June 2009.
Recommended for acceptance by E. Kruijff.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2009-02-0034.
Digital Object Identifier no. 10.1109/TVCG.2009.75.

1077-2626/10/$26.00 © 2010 IEEE Published by the IEEE Computer Society

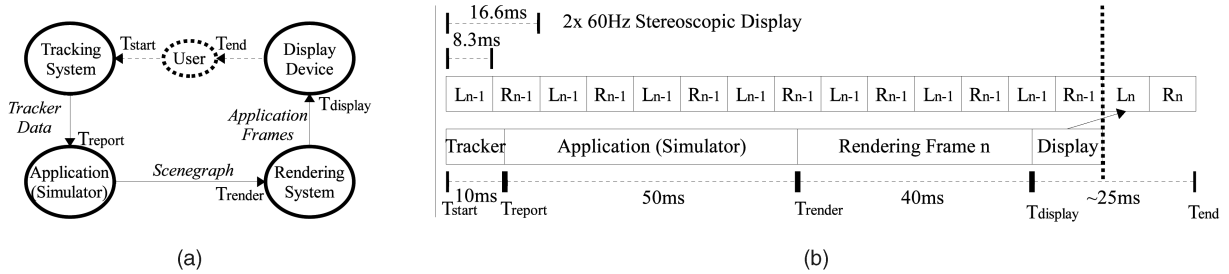


Fig. 1. (a) Logical overview of a classic VR-architecture. The user initiates an action at T_{start}, the tracking system reports a pose at T_{report}, the application completes the generation of a scene graph at T_{render}, the rendering system renders the scene graph, and the resulting application frame is sent to the display at $T_{display}$. The new application frame will be visible at T_{end}. (b) Timeline overview of the classic VR-architecture. Two processes are running in parallel: the application and the display. During the time required to process tracking data, generate a scene graph, perform rendering and sending the resulting application frame to the display, and several consecutive display frames are presented on the display. Application frames are repeated several times on the display, until a new application frame has been generated.

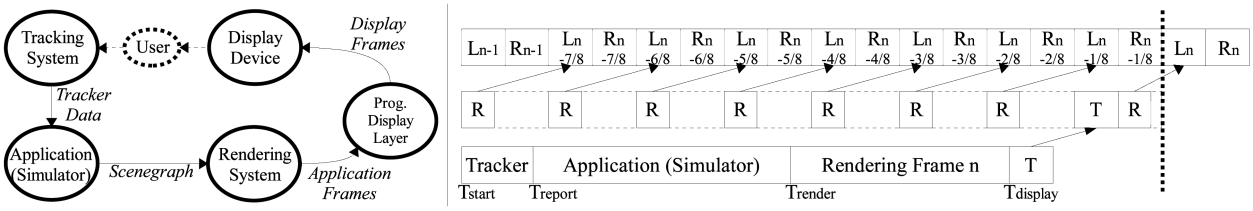


Fig. 2. (a) Logical overview of the image warping architecture. To change the default display behavior of repeating frames until an update is available, we have added an extra PDL. The architecture operates similar to a classic architecture, except that application frames are sent to the PDL instead of the display. The PDL performs image warping and sends intermediate frames to the display every time a new display frame is required. (b) Timeline overview of the new architecture. Three processes are running in parallel: the application, the PDL, and the display. New applications frames are sent to the PDL at $T_{display}$ (T). The PDL uses these application frames to continuously generate new, intermediate display frames and sends them to the display when required (R).

per-pixel 3D motion field. The PDL receives application frames in the form of per-pixel color, depth, and motion data. In order to generate consecutive display frames, the PDL performs per-pixel depth-image warping by utilizing the application data and 3D motion field. In this way, image warping is performed in real time and parallel with the application that is busy generating new application frames.

The PDL architecture can be used for a wide range of algorithms. We will describe three algorithms that have immediate benefit for VR. The first algorithm provides smooth motion, or judder reduction, by generating intermediate display frames by depth-image warping. This is especially beneficial for walk-throughs of large scenes. Second, we describe an algorithm for fine-grained latency reduction at display frame level. Using the PDL, we are able to apply viewpoint prediction for every display frame, instead of only once every application frame. In this way, our architecture provides an environment where perceived overall latency is reduced. This will reduce motion sickness and user fatigue [5] and increase task performance [6]. Third, a crosstalk reduction algorithm for consecutive display frames is implemented. This simplifies the implementation and increases the quality of the crosstalk reduction algorithm, resulting in improved stereoscopic images.

Furthermore, the implementation of the PDL architecture has a number of practical advantages. First, no special hardware is required; only commodity components are used to implement the architecture. Second, existing applications require only minimal modifications to their rendering pipeline. As long as an application can provide a motion field, the architecture can be used. The generation of a motion field can be realized effortlessly when geometry is specified in a scene graph. In this case, the application requires no changes at all, as the motion field can be generated during common scene graph processing. Third, there are no limitations to the number or types of primitives rendered. The architecture is pixel-based and shows constant runtime performance, regardless of the complexity of application processing. Therefore, for many common applications, the use of the architecture is virtually free of cost.

2 Related Work

Shaw et al. [7] proposed the Decoupled Simulation Model (DSM). The DSM utilizes four decoupled components for computation, geometry, interaction, and presentation. Each of these components can independently generate events. Most other VR architectures operate using a similar model, although the exact components may be different. All of these architectures share the fact that application frames are repeated because the display frame rate is not the driving rate of presentation. Olano et al. [2] noted the need to separate the image generation from the display update rate in order to combat rendering latency. They propose the SLATS system, which guarantees only one display frame (16.7 ms) of latency. This is achieved by insisting that all work for one display frame is finished during the frame immediately before it. The architecture consists of a number of graphics processors, a ring network, and an additional number of rendering processors. The graphics processors

generate rendering primitives in batches, which are then sent over the ring network to the rendering processors. In turn, the rendering processors are responsible for rendering the primitives and scanning out the resulting images to the display. In this way, the rendering processors operate independently of the graphics processors in updating the display, and a single frame of latency is guaranteed. This method is limited by the constraint on rendering time available to the rendering processors; only a small number of primitives can be rendered, and shading must not be complex and time-consuming. The architecture proposed in this paper provides the same guarantee with respect to updates; however, it does not place any constraints on the number of primitives or complexity of rendering.

Several other architectures follow the approach of Olano et al. by attempting to update the display at every display frame; however, different means of achieving this are often used. Kijima and Ojika [8] proposed an architecture to reduce latency effects on HMDs. Scenes are first rendered with a greater field-of-view than the HMD provides. Next, the user's head motion is extrapolated at every new display frame and the image is shifted accordingly on special HMD hardware. Stewart et al. [9] proposed the PixelView architecture. Instead of rendering a single 2D image for a specific viewpoint, they construct a 4D viewpoint-independent buffer. Then, for every display frame, a specific view is extracted from the 4D buffer according to a predicted viewpoint. The architecture requires the entire scene to be subdivided into points, or alternatively into specific types of primitives the system can handle. An implementation is provided using custom-built hardware. Finally, Regan and Pose proposed a virtual address recalculation pipeline [10], [11], where for each application fame, individual objects are rendered into distinct buffers. All these buffers are then transformed and combined to produce display frames at a fast rate. This approach shows similarities with the Talisman architecture by Torborg and Kajiya [12] and to some extent, layered depth images [13].

Depth-image warping is a technique that generates novel views from a given reference image considering per-pixel color and depth and performing a reprojection step. Image-based rendering by 3D warping was introduced by McMillan and Bishop [14]. Postrendering 3D warping is a particular technique that attempts to increase the overall frame rate of an interactive system by generating new views between the current viewpoint and a predicted one [15]. Layered depth images (LDIs) [13] combine several depth images from nearby views into a layered representation to address occlusion artifacts. A common problem with all of these approaches is that they often require custom hardware, or that the existing applications' rendering loops need to be modified significantly. Also, most approaches take into account viewpoint changes in static scenes only. Our approach is primarily based on a combination of these earlier proposed image warping techniques. In particular, the core image warping equations consist of a modified version of the image warping equations proposed by McMillan and coworkers [14], [15] adapted for modern graphics hardware and dynamic scenes. There are a number of advantages to our approach. First, the implementation operates in real time at a 60 Hz stereoscopic display using commodity hardware only. Second, since our approach records optic flow in the form of per-pixel motion, we support dynamic scenes by providing prediction and extrapolation for both viewpoint changes and moving simulation objects (see Section 4.2). Third, multiple client-side viewpoints are supported, where the placement of these viewpoints has been based on the optic flow information of the scene. We show that for dynamic scenes, this results in higher quality images than approaches based on camera prediction alone.

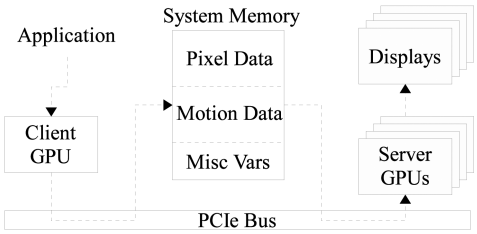


Fig. 3. Logical hardware overview of the image warping architecture. The application runs on a client GPU and sends pixel and motion data to shared system memory over the PCIe bus. Server GPUs implement the PDL and read the pixel and motion information to generate display frames. Multiple server GPUs are supported in order to drive several displays, as required for tiled displays or CAVE-settings. Our implementation currently uses a single client and server GPU to drive a single desktop-VR display.

We proposed a different variant of the PDL architecture in the past [16]. While the governing ideas and design of the two architectures are the same, some of the implementation details vary; most notably with respect to image warping. The earlier work is focused at improving performance and image warping quality and introduces a number of extensions to achieve this. Instead of using a 3D per-pixel motion field, the client assigns a unique ID to each geometric object in the scene and transmits per-pixel object IDs and the corresponding object transformation matrices. In this way, per-pixel motion can be estimated according to matrix transforms. The benefit of this approach is that data size is reduced, thereby increasing runtime performance. Additional extensions are the improvement of image quality due to more advanced image warping algorithms using dynamic splat sizes, and the ability to use two client-side viewpoints in real time due to increased performance. In this paper, we are primarily concerned about the high-level architecture design, its use for VR, and the comparison with classic VR-architectures. We examine the effect of client-side camera placements and compare the architecture to a classic level-of-detail method. A motion field is used, instead of object IDs, to perform image warping and client-side camera placement. The architecture is not restricted to this approach, and the previously mentioned extensions can be implemented as well. In fact, some of the improved image warping algorithms have been used in Section 5.

3 Architecture

An overview of the architecture's hardware implementation is given in Fig. 3. The architecture is implemented using a parallel multi-GPU system. The GPUs are connected over the PCIe bus and communicate using a large segment of

shared system memory. The first GPU, which we call the *client*, is responsible for rendering application frames. These frames contain a fixed number of rendered scenes from various viewpoints. The viewpoints used for rendering are not necessarily equal to the user's viewpoint. In addition to pixel color information, the client also generates per-pixel motion and depth information. Once generated, all application frame data are transferred over the PCIe bus into a synchronized circular producer/consumer buffer in shared system memory. The second GPU, which we call the *server*, is responsible for generating intermediate display frames and updating the display device. Whenever the display needs to be refreshed, the server polls the shared system memory to determine if a new application frame is available. If a new frame is available, it is copied from shared memory onto the GPU. Otherwise, the previously received application frame is reused. Image warping is then used to generate an intermediate display frame from the latest pixel and motion data, which are subsequently sent to the display device.

3.1 Client Implementation

The client starts by generating a left and right-eye stereoscopic scene for rendering as normal. It then renders the scenes into GPU memory using multiple render target (MRT) and frame-buffer object (FBO) functionality. For every geometric object that is to be rendered, the client stores its transformation matrix for the previous application frame M_{prev} and its current transformation matrix M_{cur} in a hardware vertex program. Then, the object is sent to the GPU for rendering using a vertex and fragment program. For each vertex V_i, the vertex program calculates the previous position $P_{prev} = M_{prev} \cdot V_i$ and the current position $P_{cur} = M_{cur} \cdot V_i$. Both vectors are then passed to the fragment program, where they are automatically interpolated per pixel by the hardware. For every pixel, the fragment program calculates the value $\Delta P = P_{cur} - P_{prev}$, resulting in a 3D motion vector per pixel. Since vertices are transformed into camera space, the motion vector ΔP also resides in camera space. Finally, the fragment program outputs two data vectors to the hardware FBO. The first vector consists of four 8-bit integer color values $P_{color} = (B, G, R, A)$ representing the rendered pixel's color. The alpha channel is used as a flag to indicate the motion data is valid for this pixel. The second vector consists of four 16-bit floating point values $P_{motion} = (\Delta P^x, \Delta P^y, \Delta P^z, V^z_{cur})$ representing the pixel's motion data in the first three components and its depth in the last component. The required space per pixel is, therefore, 12 bytes.

Once the pixel and motion data have been generated, a free slot in the circular producer/consumer buffer is acquired and the FBO is downloaded from the GPU hardware into shared system memory using pixel-buffer objects (PBOs). Additionally, the current and previous camera matrices are stored in shared memory to allow the server to make an estimate of the camera motion. This is described in more detail in Section 3.2. The client then loops and proceeds to generate the next application frame. Note that the client generates a left- and a right-eye application frame, each with their own pixel and motion data. Both frames are stored in shared memory in a single iteration; therefore, we effectively treat a 120 Hz stereoscopic display as a single 60 Hz display with twice the resolution.

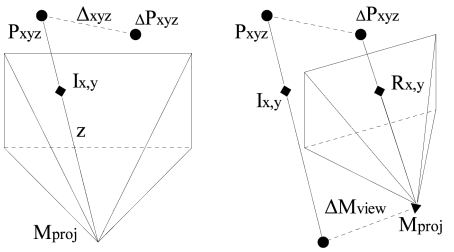


Fig. 4. Server image warping implementation. A 2D pixel $I_{x,y}$ is converted back to a 3D pixel P_{xyz} using its depth z and the camera projection matrix M_{proj}. Next, P_{xyz} is additively warped for object motion by Δ_{xyz}, resulting in ΔP_{xyz}. Camera viewpoint warping is performed by transforming ΔP_{xyz} by the predicted camera difference ΔM_{view} and projecting it back to 2D using M_{proj}. Finally, the pixel is rendered at location $R_{x,y}$.

3.2 Server Implementation

The server first polls the producer/consumer buffer in a nonblocking fashion to see whether a new application frame is available or not. If a frame is available, an OpenGL hardware buffer is mapped and the data are uploaded directly to the GPU from shared memory. Since data transfers are relatively slow, we chose to calculate all the required data from a minimum of transmitted data. Once the new application frame is uploaded to the GPU, server processing continues in the same fashion as when no new frame was available.

When no new frame is available, the server simply binds the previous frame to OpenGL. This requires no data transfer, as the frame still remains in GPU hardware memory. A Δt value is calculated to determine how much motion extrapolation is necessary, which is then passed to a vertex program. In order to render the extrapolated display frame, we bind the pixel motion data P_{motion} to an OpenGL vertex array and the pixel color data P_{color} to a color array. Both arrays have a size of $w * h$, where w and h are the display width and height in pixels, respectively. All the $w * h$ pixels are drawn through a single call to *glDrawArrays(GL_POINTS)*, which renders every pixel as a separate point/vertex.

The core of the motion extrapolation algorithm is contained in the server's vertex program, which is executed for every pixel sent by the client. The steps of the algorithm are outlined in Fig. 4. The first step is to reconstruct the pixel's 3D coordinates in camera space from its 2D pixel coordinate and the depth V^z_{cur} stored in P_{motion}. The 2D pixel coordinates $I_{x,y}$ are not explicitly sent to the GPU, but calculated to save bandwidth. The Nvidia G80 series of GPUs is capable of performing integer logic in the vertex program; additionally, every vertex that is drawn in a call to glDrawArrays() receives a unique linear integer ID describing its position in the array. This ID is available in the vertex program as gl_VertexID. Since every vertex maps exactly to 1 pixel for our point-cloud rendering, we can calculate the pixel's coordinates $I_{x,y} = (gl_VertexID\%w, gl_VertexID/w)$, where w is the display width in pixels. The 2D integer coordinates are then converted to a $[-0.5, 0.5]$ range on the camera near-

plane, after which the 3D coordinates can be calculated using the camera's projection matrix: $P_x = (-V^z_{cur} \cdot P_{hskew}) - 2 \cdot P_{hfov} \cdot V^z_{cur} \cdot I_x$ and $P_y = 2 \cdot P_{vfov} \cdot V^z_{cur} \cdot I_y$, where P_{hskew} is the horizontal off-axis projection parameter gl_ProjectionMatrixInverse[3][0], and P_{hfov}, P_{vfov} are the projection field-of-view parameters gl_ProjectionMatrixInverse[0][0]/[1][1].

After the pixel's 3D coordinates have been recovered, the corresponding 3D motion vector ΔP is retrieved, scaled by Δt and subsequently added to the pixel's coordinates. This results in an extrapolated scene. For the extrapolated camera transformation, we either make a prediction based on the previous camera transforms provided by the client, or we sample the head tracker device for an updated pose if one is available. In this way, we can provide greater accuracy for the head tracking prediction. This will be discussed in more detail in Section 4.2. Let the original camera transform be given by M_{view} and the updated sample or prediction by M_{pred}, we can then transform the pixel by $M_{pred} \cdot M^{-1}_{view}$ in order to warp the image for camera motion. Finally, the 3D pixel is projected back to 2D, making use of the original projection matrix. The entire scene is rendered to a frame-buffer object in GPU memory.

Once an intermediate, warped display frame has been generated, the server performs a crosstalk reduction algorithm on it. Both hardware buffers containing the left- and right-eye images are used as input to the nonuniform crosstalk reduction algorithm proposed by Smit et al. [3]. This is described in more detail in Section 4.3.

3.3 Image Warping Errors

Image warping is a technique that generates pixel errors in the output proportional to the magnitude of warping required. We distinguish between three types of image warping errors: prediction errors, occlusion errors, and warping artifacts. Prediction errors are due to the fact that per-pixel motion prediction is imperfect; therefore, warped pixels are potentially rendered at a different location than they should be. Occlusion errors occur when parts of the scene in the original application frame are occluded by other geometry. Since image warping is only performed for the depth layer closest to the virtual camera, these occluded pixels are not warped. This results in a small gap of missing geometry in the warped frame. Finally, warping artifacts are due to the n-1-mapping of warped pixels; if the original surface of pixels is smaller than the required warped surface, there will always be some screen pixels left blank. This effect results in small holes in the warped mesh.

Occlusion errors and warping artifacts can, in part, be reduced by implementing more advanced and complicated image warping techniques; however, doing so may be prohibitively time-consuming, as the server component has very limited available processing time. A quantitative evaluation of the magnitude of the introduced warping errors, compared to a classic level-of-detail method, is given in Section 5.

3.4 Camera Placement Strategies

The PDL architecture supports the use of multiple client-side viewpoints. The client generates application frames from multiple viewpoints, which are transmitted and subsequently warped to a single viewpoint on the server side.

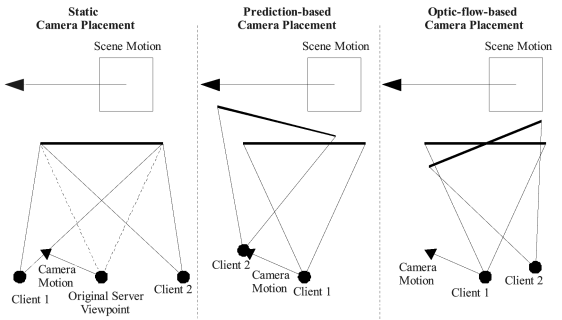


Fig. 5. Schematic overview of the three different client-side camera placement strategies. The static strategy always places the client cameras in fixed positions relative to the server viewpoint. The prediction-based strategy places the second camera according to the predicted pose of the camera in the next frame, ignoring scene motion. The optic-flow-based strategy implicitly takes the combined scene and camera motion into account and places the second camera according to averaged per-pixel optic flow. For the latter two strategies, the first camera is always placed at the server's viewpoint.

The challenge is to determine how many client-side cameras to use and how to choose the pose of these cameras in order to achieve the maximum warped image quality at the server side. The most notable and disturbing type of error in image warping is due to occlusion. Therefore, client viewpoints should be chosen in such a way as to minimize potential occlusions during warping. Another aspect is that of viewport clipping; geometry outside the client's view frustum cannot be warped into the server's frustum because it is never rendered. For this reason, the field-of-view (FOV) of the client cameras should, in general, be larger than the server's FOV to avoid viewport clipping. However, these two approaches may result in loss of image quality. Since image warping errors are dependent on the distance between the source and target images, the further client viewpoints deviate from the target viewpoint, the larger the errors become. Also, increasing FOV while keeping the number of pixels in the viewport equal effectively decreases resolution, and thus, image quality. Hence, the camera placement strategy should also maximize image quality.

We distinguish between three types of client-side camera placement strategies: static, predictive, and optic flow based. These strategies are schematically depicted in Fig. 5. A static strategy simply places the cameras in fixed positions relative to the latest known sensor data. Other strategies have been developed to efficiently place client cameras based on sensor prediction [17]. It was shown that reasonable results can be obtained by using two client-side views: one rendered from the camera viewpoint in the current frame and other from the predicted future camera viewpoint in the next frame. These strategies assume static scenes where the camera is the sole moving entity. For static scenes, once an image has been rendered from a specific viewpoint, it will remain valid throughout the lifetime of the application. Therefore, this strategy can reuse old imagery and the client only renders and transmits a single, predicted viewpoint for each frame. However, for dynamic scenes, good client-side camera placement is more challenging. Since objects are moving, client viewpoints should not be determined based solely on camera movement. What is needed is an intelligent camera placement algorithm that is based on the motion of

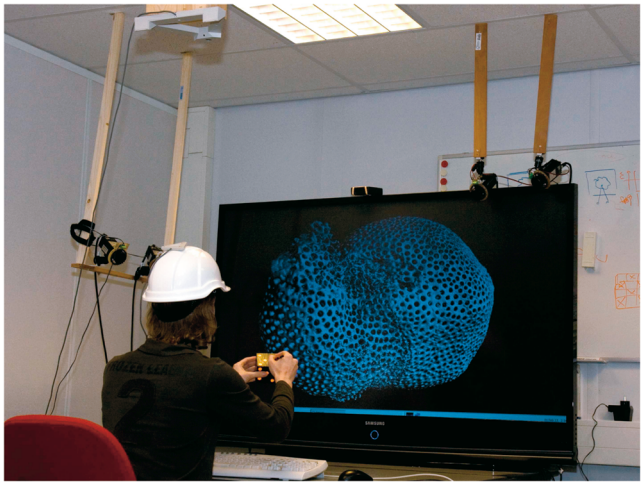


Fig. 6. Interactive visualization of a large model. A user is examining the 17M polygon coral model on a Samsung HL67A750 60 Hz stereoscopic DLP TV using LCS glasses (for image capturing purposes, a monoscopic image is displayed at screen-filling HD resolution). Head tracking is performed by a Logitech head tracker running at 50 Hz. The model can be interactively rotated using 6 DOF optical tracking running at 60 Hz. The rendering of the model is done at 6 Hz only. Therefore, on a classic architecture, new application frames are generated at a maximum rate of 6 Hz. Using our PDL architecture, we can generate intermediate display frames at 60 Hz and sample the input devices accordingly.

individual objects in the scene. We believe that this is possible by making use of per-pixel optic flow. Instead of placing client cameras according to the predicted motion of the server camera, one could place client cameras according to the optic flow of the scene at that time. In this way, object and camera movement both implicitly contribute to camera placements. In Section 5, we will evaluate the effect on image quality for these three types of camera placement strategies in a practical setting.

4 PDL Benefits for VR

4.1 Smooth Motion

The experimental psychology literature describes an effect where motion causes a single object to be perceived as multiple objects [4], [18]. In the video-processing community, this effect is also called judder and is caused by repeated application frames, as shown in Fig. 7. If application frames are generated at a lower rate than the display frequency, multiple display frames will be displaying a moving object in the same position. Next, when a positional update is generated, the object will suddenly jump to this new position. The human visual system has difficulties interpreting these sudden large changes, and the result is that multiple objects are perceived. The presence of judder degrades perceived image quality, and may lead to increased user fatigue. Using our proposed architecture, we can eliminate repeated application frames by generating extrapolated display frames. Observers reported that the use of our architecture completely eliminated the judder effect. Hence, one of the benefits of using our architecture is the appearance of smoothly moving objects due to the absence of judder.

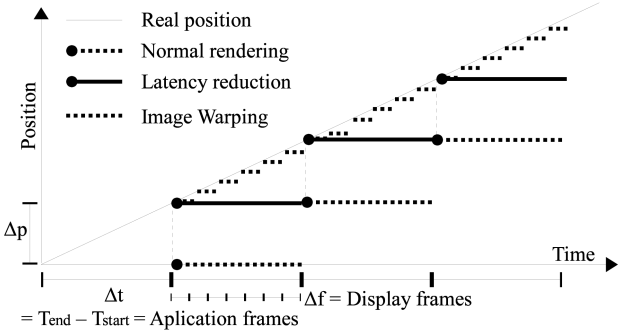


Fig. 7. Timeline for latency reduction. Real object position is depicted by the solid line. Rendering an application frames takes an amount of time equal to Δt, during which the object moves by an amount of Δp. Normal rendering samples the object's position at the beginning of rendering and renders it in that location; however, in reality, the object has already moved by Δp when the frame is finally displayed. Classic latency reduction predicts the object's position at the time of display instead of the time of rendering. Since new display updates only occur once every application frame, the object remains in the same position for the entire duration Δt. With our image warping architecture, the object's position is predicted every display frame Δf, resulting in more accurate prediction on average.

An example is shown in Fig. 6. A 17 million polygon isosurface model is made from a CT scan of a coral. The user is viewing the model on a stereoscopic display operating at 60 Hz using active stereoshutter glasses. Head tracking is provided by an acoustic head tracker that operates at 50 Hz. Finally, the model can be interactively rotated by means of 6 DOF optical tracking that generates reports at 60 Hz. The rendering of such a large model, however, is relatively slow and is done at a rate of only 6 Hz. Therefore, application frames are generated at 6 Hz. This situation is similar to a walk-through of a large, static scene. In these situations, when the rendering of VR-applications becomes very slow, users will observe latency effects due to the slow update rates of the display (once per application frame). Using our architecture, we can update the display with extrapolated display frames at a higher rate. A combination of camera prediction and sampling is used, as described in Section 4.2, resulting in a feeling of a more responsive virtual world.

4.2 Latency Reduction

End-to-end latency in VR is defined as the time delay between an action and its observed effect. Generally, this is the delay between moving an interaction device and seeing the rendered result on the display. A typical scenario is depicted in Fig. 1. First, a user initiates an action at time T_{start} and the tracking device reports a pose at T_{report}. The application then generates an updated scene graph and starts rendering this at time T_{render}. Once rendering is complete, the newly generated application frame is scanned out to the display at time $T_{display}$. The first updated display frame is completely visible at time T_{end}. For a stereoscopic display, it is somewhat difficult to determine exactly when T_{end} occurs, so a reasonable estimate is at the display's second vertical blank signal. End-to-end latency is now defined as $\Delta t = T_{end} - T_{start}$.

Fig. 7 shows the effect of end-to-end latency on rendering in VR. Suppose we run an application where there is an

interaction device (usually, a head tracker) or object moving at a constant velocity over a 1D path. The position of the object set out against the application time is then a straight line. The application starts by sampling the object's position, and then, renders it. Generating and rendering an updated application frames takes an amount of time shown as Δt. However, during this time Δt, the object moves by an amount of Δp; therefore, when the updated application frame is visible on the display, the object is already at a different location using normal rendering. An often-used solution is to predict the position of the object at a time Δt in the future using Kalman filtering. In this way, the object is rendered at the position where it should be when the display is actually updated. This approach is called latency reduction or dead reckoning. Note that the object remains visible at the same location on the display until a new application frame is generated and displayed, regardless of the display frame rate.

Since our architecture renders predicted, extrapolated display frames, it is well suited for performing latency reduction. For every new display frame, we extrapolate all the pixels by image warping according to a prediction of Δt and the scene's motion information. In this way, the scene is warped to where it should be at the time of display, precisely corresponding to normal latency reduction. Contrary to normal latency reduction, we do not only predict for a time step Δt in the future, but for every display frame, we also predict an extra time step Δf. This scenario is also shown in Fig. 7.

Instead of predicting all motion, it is also possible to obtain additional samples of the interaction device. For every display frame, we poll the interaction device to see if a new pose report is available. If this is the case, the latest known pose is used instead of a prediction, after which the scene is warped accordingly. This allows for a higher sampling rate of the interaction device, resulting in fewer prediction errors than with regular latency reduction, where the device is sampled only once per application frame. Furthermore, as can be seen from Fig. 7, the rate of visual feedback is higher using our architecture for latency reduction. Even with regular latency reduction, the object's position on the display is only updated once every application frame. Using our architecture, the position is predicted and possibly sampled every display frame, resulting in a faster observed response from the interaction device. This is especially beneficial for applications using head tracking, as end-to-end latency in head tracking is considered to be the primary source of motion sickness [5].

A problem occurs when only the user's viewpoint is extrapolated, but not the simulation or scene itself. This is an effect that may occur when scene graphs are rendered repeatedly from different viewpoints before they are updated by the simulation, which is a typical approach for existing architectures providing viewpoint extrapolation (e.g., [9], [8], [7]). An example is given in Fig. 8. Imagine a head-tracked observer is following a moving simulation object. Simulation updates are generated in the form of new application frames at time t_0 and t_1; therefore, the object jumps from one location to the next at t_1. When predicting head tracking motion, intermediate display frames will be generated between these two application frames. A sample

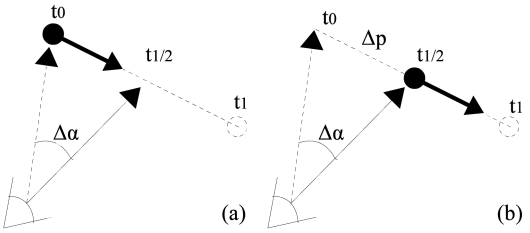


Fig. 8. Synchronization issues between head tracker predictions and simulation. (a) A user is following a moving simulation object from t_0 to t_1. Viewpoint extrapolation predicts the user's viewpoint Δa at $t_{1/2}$; however, this does not take into account object position, causing the user to miss the object. (b) Our architecture also takes object motion into account and predicts the object's position Δp at $t_{1/2}$, in addition to the viewpoint Δa, allowing for the object to be followed correctly.

intermediate frame is shown for time $t_{1/2}$ in Fig. 8a. The user's viewpoint change is predicted by Δa and the scene is temporarily updated for that specific viewpoint. However, when the simulation is not updated as well, the object will remain where it was at time t_0 until a new application frame is available. This causes a visual artifact when the user is trying to follow the moving object. Using our architecture, we perform motion extrapolation for the entire scene, including the simulation objects and their predicted motion. Therefore, in the case of our architecture, the object's position at time $t_{1/2}$ is also predicted by Δp and the user can follow the object correctly. This is shown in Fig. 8b. In effect, our architecture synchronizes the prediction of both simulation rendering and head tracking.

4.3 Crosstalk Reduction

Active stereo displays operate by showing sequential display frames for the left and right-eye views, in combination with active liquid crystal shutter (LCS) glasses, which block the view for the eye currently not displayed. Stereoscopic displays suffer from crosstalk or ghosting, an effect that reduces or even inhibits the user's ability to perceive depth. Crosstalk is caused by the slow decay of CRT display phosphors, LCS glasses that do not go completely opaque, and inexact timing of the LCS glasses [19]. The causes of crosstalk are all inherently related to the sequential display of left and right-eye display frames, as light leaks from the previous to the current display frame.

Methods exist to reduce or eliminate the effect of crosstalk in software. One such method was proposed by Smit et al. [3], [20]. They reduced visible crosstalk by estimating the amount of light leakage between the previous and the current application frame, and then, performed a correction step to compensate for this added intensity. However, this correction has to be performed between display frames, not application frames.

Since our architecture guarantees updates of individual display frames, crosstalk reduction can be performed by the programmable display layer for display frames instead of application frames. Crosstalk is now guaranteed to be reduced for consecutive display frames, as it should be, thereby eliminating reduction errors introduced due to repeated application frames. The benefit of our architecture in this case is better quality crosstalk reduction and a more straightforward implementation.

TABLE 1
Architecture Performance for Various Resolutions

Resolution	640x480	800x600	1024x768	1280x960
Number of Pixels	307200	480000	786432	1228800
Frame Size (MB)	7.0	11.0	18.0	28.1
Data Transfer (ms)	3.1	4.7	7.4	11.6
Rendering (ms)	3.8	5.7	8.8	13.6
Total Time (ms)	6.9	10.3	16.2	25.1
Performance (Gpix/s)	1.03	1.06	1.11	1.12
Throughput (GB/s)	2.28	2.36	2.42	2.43

Data transfers and rendering take almost the same amount of time. Per-pixel performance is nearly constant, showing that performance scales linearly with the amount of pixels.

5 Results

Our implementation of the PDL architecture is realized using an Nvidia GeForce 8800 GTX for the client GPU and a stereoenabled Nvidia Quadro FX5600 for the server GPU. The system consists of an Intel Q6600 2.4 Ghz quadcore processor; therefore, the client and server processes can each utilize a separate core, as well as a GPU. Two separate stereoscopic displays have been used: an iiyama Vision Master Pro 512 22″ CRT monitor operating at 120 Hz in order to achieve 60 Hz per eye, and a Samsung HL67A750 60 Hz stereoscopic DLP TV. For head tracking, a Logitech Acoustic head tracker running at 50 Hz is used.

5.1 Architecture Performance

In order to examine the architecture's runtime performance, we run a sample application program on the client and measure the time required by the server for data transfers and rendering. We only examine the server process as it is the limiting factor of the architecture. Client processing can take an arbitrary amount of time; however, the server processing must be performed at the display refresh rate to guarantee updates every display frame. There are two different types of display frames the server is required to render: newly received application frames that require data transfers, and intermediate extrapolated display frames that do not require any data transfer. The server guarantees to update the display at every refresh, regardless of the type of frame currently being rendered. Therefore, we only measure the performance for the slowest type of frame, which are frames requiring data transfers.

Table 1 provides an overview of the server's performance for a number of different resolutions. Since the stereoscopic display operates at 2×60 Hz, there are approximately 16.6 ms available for the server to produce a new display frame. Table 1 shows us, therefore, that the maximum achievable resolution for this hardware setup is $1{,}024 \times 768$ pixels. At $1{,}280 \times 960$, the server can no longer guarantee updates of the display every refresh. The overall performance expressed in giga pixels per second is almost constant. Larger resolutions are relatively faster due to the reduced impact of initialization overhead, but the effect is almost negligible. This means performance scales linearly with the number of pixels. The same effect shows for the throughput in gigabyte per second for transferring data.

It can also be seen from Table 1 that data uploads from system-shared memory to the GPU and the rendering of pixels are virtually equal contributors to the total processing time per display frame. This shows the validity of our approach to calculate the 3D positions of all pixels from their depth, and so minimize data transfers. In case, we do not calculate these positions in the vertex program, an additional 4 bytes need to be transferred per pixel in case of 16-bit floating point values. The per-pixel transfer size is then increased from 12 to 16 bytes, increasing transfer times by 33 percent. Performance evaluation of the vertex program shows that only a small portion of time is spent on these calculations; definitely not more than one-third of the total rendering time. A large amount of rendering time is actually spent on crosstalk reduction.

Since performance scales linearly with respect to the number of pixels, faster hardware will have a direct impact on the maximum achievable resolution. Ideally, data transfers between the GPUs should be done directly, without the need to send data over the PCIe bus to shared system memory; however, this is not possible on current GPUs—Nvidia's SLI uses a video signal and cannot be used to transfer digital information. Perhaps, future generation GPUs will provide such capabilities, making the shared system memory obsolete. Bandwidth over the PCIe 2.0 bus has been doubled compared to the currently used version 1.1 and will be doubled again by version 3.0; however, GPUs cannot as of yet take full advantage of this increased bandwidth. Future GPU versions are likely to perform data transfers much faster. Therefore, we believe that even higher resolutions, such as 1080p HDTV, are within reach.

5.2 Level-of-Detail Quality Comparison

When a frame rate of 60 Hz is to be guaranteed in a classic VR-architecture, the only method to achieve this is to reduce the computational load. To this extent, static level-of-detail (LOD) methods are often used to reduce the number of polygons rendered. The geometric models are decimated by successively removing all those polygons that are considered to have the least visual significance, until a target number of polygons are reached for which a 60 Hz frame rate can be guaranteed. In this section, we make a comparison between the image quality of the proposed image warping architecture and that of a static level-of-detail method.

Two models are used for this comparison: the 10M polygon Thai Statue model from XYZ RGB, Inc., and a 17M polygon coral model. The former is a model of a scanned statue with a relatively smooth, low-frequency surface, while the latter is a model of a CT scan of a coral consisting of high-frequency data and many holes in the surface. Both models are good examples of large real-life polygonal data sets. To estimate image quality, we recorded the 60 Hz image-sequence output of the PDL image warping server for approximately 1,600 animation frames. A prerecorded animation sequence is used where the models rotate about their Y-axis and the camera hoovers around the models according to recorded user inputs. We call this the dynamic animation sequence, since the objects as well as the camera move. In Section 5.3, we will also use a static animation sequence where the camera moves in similar fashion but the objects remain still. In addition, we also record similar output of a stand-alone reference implementation that renders every animation frame as it should appear without error. Finally,

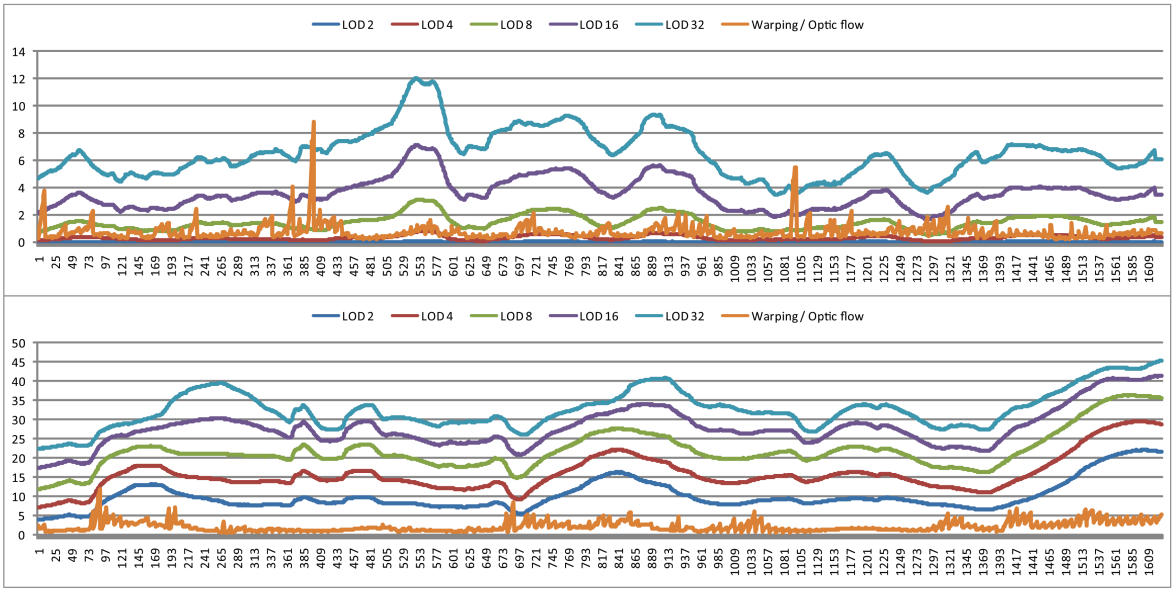


Fig. 9. Percentages of error pixels for each frame of a dynamic animation sequence. The top plot shows the errors for the 10M polygon statue scene; the bottom plot the 17M polygon coral scene. The camera placement strategy used for image warping is the optic-flow-based approach. This strategy occasionally results in a poor camera setup, causing the error spikes near frames 400 and 1,100. An overview of the average errors is given in Table 2 under the dynamic scene heading.

we render the same reference scene using corresponding LOD models that are decimated to various decreasing amounts of polygons. For the PDL server, rendering and recording are performed offline by running the client and server in a special synchronized mode. In this way, a client frame rate of 6 Hz is simulated in combination with a 60 Hz server rate, i.e., for every application frame, we need to generate 10 warped display frames. To improve the image quality of warping, we used some extensions proposed by Smit et al. [16], most notably a dynamic splat size to avoid undersampling errors and the ability to use two client viewpoints (see Section 3.4). In all cases, a resolution of $1{,}024 \times 768$ pixels is used. The rate of object rotation about the Y-axis for the dynamic scenes is 45 degrees/s. The extent of the axis-aligned bounding boxes in rendering units of the coral and the statue object are (130, 94, 83) and (235, 396, 203) units. The translational and angular velocities for the camera are different for each scene. For the coral scenes, the average translational velocity of the camera is 27.2 and 37.2 units/s for the dynamic and static scenes, while the average angular velocities are 43.6 and 49.3 degrees/s, respectively. For the statue scenes, the average dynamic and static animations' translational velocities are 48.7 and 66.2 units/s, with average angular velocities of 34.2 and 49.2 degrees/s.

For the level-of-detail models, we reduced the number of polygons of each of the two models by fractions 2, 4, 8, 16, and 32 of the total amount of polygons. For example, LOD-4 of the coral model consists of 4.25M polygons, while the same level for the statue model consists of 2.5M polygons. We were unable to find free, out-of-the-box software for mesh decimation that could efficiently handle data sets of 10M polygons or more; most of the tools we found simply hanged or quickly ran out of memory. Therefore, we split the used models in separate chunks of 6M polygons each, generated LODs for each chunk and then recombined the meshes into a single whole. In order to achieve this, we made use of the MeshLab software (version 1.1.1) [21]. Mesh decimation was performed using the quadric edge collapse algorithm with a default quality threshold of 0.3 to reduce the mesh to the various target numbers of polygons.

From the recorded images of the output of the various algorithms and a reference implementation, we can compare each image to the reference image in order to determine the amount of error in the output images. For this comparison, we use a simple and straightforward technique. First, we filter both images using a small three-pixel wide Gaussian kernel. Next, we convert both images to the Lab perceptual color space. Finally, we compare the two images on a pixel-by-pixel basis and mark a pixel as being an error pixel if the distance between them is larger than a threshold value of 10 units in the Lab space. The Gaussian filter and the threshold value are used to avoid marking very small individual pixel differences, which are generally unperceivable, as errors. The final error value that is reported for the image frame is the percentage of pixels marked as error pixels.

Fig. 9 gives an overview of the errors for the dynamic statue and coral scenes for each animation frame. Average errors and their standard deviation are summarized in Table 2. It is immediately obvious that the image warping quality for the coral scene is superior to that of the LOD approach for all levels. For the statue scene, the average warping quality lies somewhere between LOD-4 and LOD-8. There are two reasons for these large differences. First, the original statue model appears to be severely oversampled, since reducing the amount of polygons by half from 10M to 5M (LOD-2) hardly introduces any error at all. This is an effect of the inherent smoothness of the statue model's surface. The second reason is that the coral model consists of very high-frequency data and almost twice the number of

TABLE 2
Overview of the Average Errors and Standard Deviations for Various LOD Methods and Different Camera Placement Strategies

	Statue		Coral	
Dynamic Scene	**Error**	**Stdev**	**Error**	**Stdev**
LOD 2	0.03	0.01	10.01	3.84
LOD 4	0.33	0.16	15.85	4.55
LOD 8	1.45	0.54	21.88	4.92
LOD 16	3.58	1.09	27.76	4.93
LOD 32	6.51	1.73	32.61	4.94
Warping / Static	1.48	1.39	2.16	2.08
Warping / Prediction	1.15	1.10	2.01	1.18
Warping / Optic flow	0.72	0.61	2.10	1.31
	Statue		**Coral**	
Static Scene	**Error**	**Stdev**	**Error**	**Stdev**
Warping / Static	0.95	1.15	2.01	2.11
Warping / Prediction	0.63	0.35	1.90	1.03
Warping / Optic flow	0.58	0.41	1.80	1.08

Error comparisons can only be made vertically between equal scene/model combinations because different animations were used for each. The PDL architecture was used to perform image warping using a static, camera-prediction-based, and optic-flow-based client-side camera placement strategy.

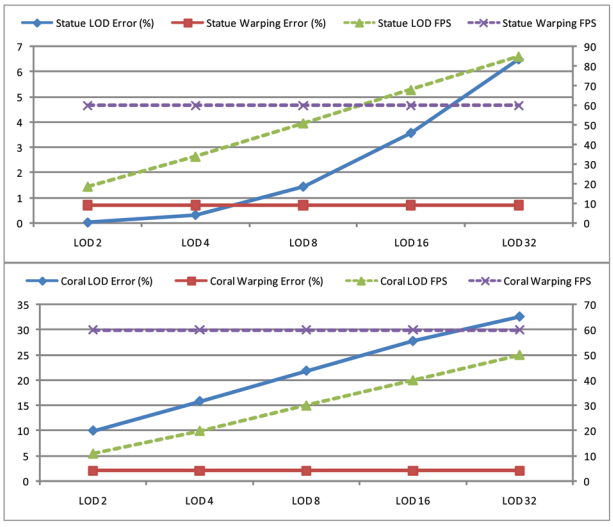


Fig. 10. Average errors and frame rates for level-of-detail methods and image warping, set out against the various detail levels. The left vertical axis indicates an average error percentage, while the right axis is used to indicate frames per second. The points of interest are where the error and FPS lines cross for image warping and the LOD approach. This shows that when both approaches run at 60 Hz, image warping results in the best image quality.

polygons. Since LOD-2 already introduces a significant error for the coral, this model does not share the smooth, oversampled nature of the statue and is less-suited to LOD methods. Another observation that can be made from Table 2 is that the standard deviation of the error for the statue scene is generally lower in the LOD case. The standard deviation gives a reasonable estimate of the amount of occlusion errors that appear and disappear from frame to frame. For warping, the error is lowest when a new frame is received from the client, and usually, highest when extrapolation is at its maximum just before the receipt of a new frame. This fluctuation in error leads to higher standard deviations.

In Fig. 10, both the frame rates of a stereoscopic reference implementation and the average errors for warping and the various levels-of-detail are given. For image warping, the number of polygons on the client side has no effect on either the frame rate or the quality, and therefore, plotted as straight lines. The points of interest in this figure are the crossing points between the quality and frame rate series. For the statue scene, the crossing point for frame rate lies somewhere half way between LOD-8 and LOD-16. This means that in order to achieve 60 Hz frame rates with LOD methods, we need to reduce the amount of polygons roughly by a factor of 10 to 1M polygons. However, the crossing point for the amount of error lies to the left of the FPS crossing, between LOD-4 and LOD-8, where the errors in both methods are considered equal. This means that if both image warping and LOD run at 60 Hz for the statue scene, the image quality for warping is better than that of the LOD approach. For the coral scene, the FPS crossing is not contained within the graph, meaning that even at LOD-32, a 60 Hz frame rate is not achieved. There is also no error crossing, which implies that the warping quality is always better in this case.

5.3 Camera Placement Strategies

In order to investigate the effect of client-side camera placements, we have implemented and evaluated three different camera placement strategies. In all three cases, we make use of two client viewpoints. The first is a static placement strategy, where the two cameras are always positioned at a fixed offset relative to the latest known head tracker pose. This is implemented by using a stereoscopic camera setup with larger-than-normal eye separation and increased FOV that is centered at the head tracker pose. The second approach is a prediction-based strategy. The first camera is positioned at the latest head tracker pose, while the second camera is placed according to the predicted future head tracker pose. Our implementation predicts three application frames ahead instead of one, since this results in better overall image quality. Furthermore, the FOV of the first camera is equal to that of the server camera, while the second camera is set to increased FOV. The third strategy is based on optic flow. It is similar to the prediction-based method, except that the position of the second camera is determined according to the per-pixel optic flow on the client side. First, the 3D motion field that is generated by the client for the first camera is projected onto the image plane. Next, the projected motion vectors are averaged into a single vector. The second camera is now rotated about the focal point of the first camera, where the opposite averaged motion vector is used to determine the magnitude and angle of rotation. The camera is placed according to the opposite direction of the optic flow because this strategy minimizes the introduction of occlusion artifacts. As before, only the second camera is set to increased FOV.

An important problem that arises when warping two client-side viewpoints to the same server viewpoint is that pixels from both viewpoints may be warped to the same

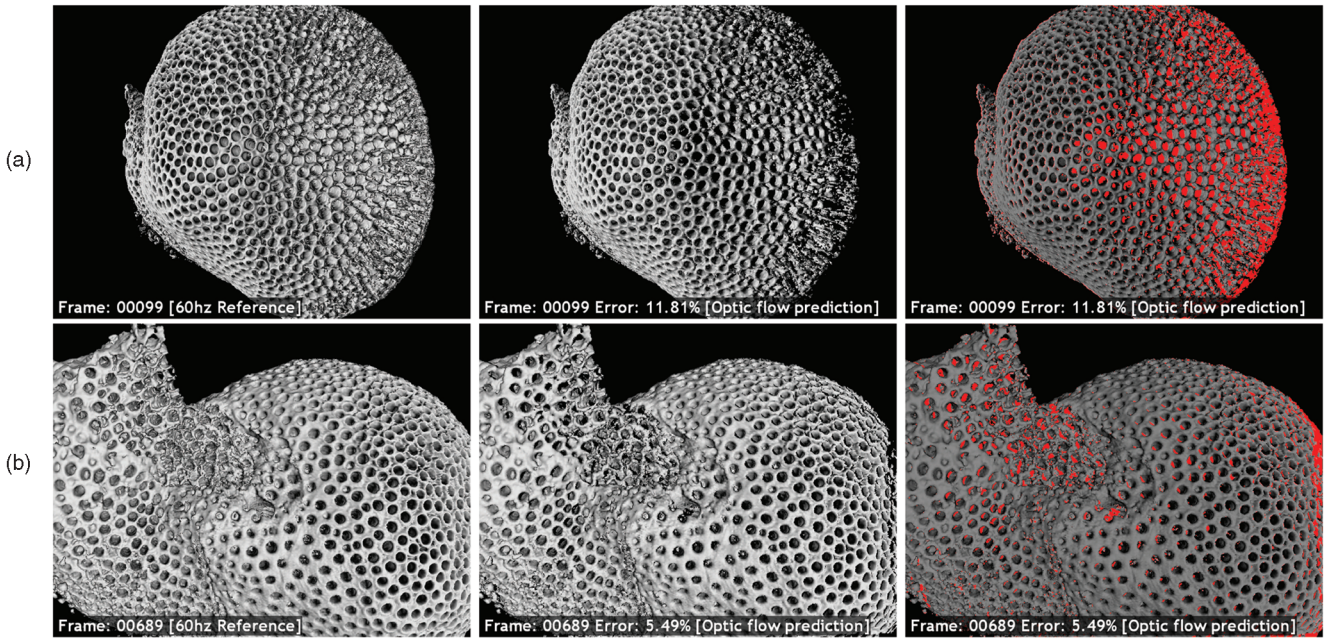


Fig. 11. From left to right are shown a reference image of how a rendered scene should look without error, the same scene as rendered by the image warping process, and the errors made depicted by red pixels. (a) The worst frame, with the largest error, for the dynamic coral scene animation sequence. (b) Another frame with a high amount of error. These large amounts of errors are almost always due to poor camera placements for the two warped views. Note that these frames show a particularly high error because of erroneously predicted camera placements. In practice, the amount of error is much lower for the vast majority of frames (see Fig. 9).

target pixel. The depth buffer correctly handles the situation when these pixels belong to different parts of the geometry. However, if the depth values are nearly the same, this indicates that we are dealing with the same geometry and one of the pixels needs to be discarded in favor of the other. Our implementation tries to discard the lowest quality pixel. Pixel quality is estimated according to the pixel's splat size, which is calculated using the expansion factor of the projective image warping transformation [16]. Generally, a pixel with a smaller splat size, or smaller warping distance, is a better quality pixel.

To compare image quality for the three-camera placement strategies, we recorded the same type of animation sequences as described in Section 5.2 for a dynamic and a static scene using the coral and statue models. In the dynamic scene, the models rotate about their Y-axis, while for the static scene, they remain still. Both scenes contain user-controlled camera movements. The results are tabulated in Table 2.

First, we examine the statue scene. In the dynamic case, we see that the average error is lowest for the optic-flow-based strategy, followed by that for the prediction-based and static strategies. The same is true for the standard deviations of the errors, which give an indication to the number of occlusion artifacts. The static strategy apparently has difficulties with the motion in the scene and results in many occlusion artifacts. The prediction strategy works better because it manages to reduce occlusion artifacts by predicting camera motion. However, object motion in the dynamic scene is not detected by the prediction strategy. On the other hand, the optic flow strategy is able to combine camera and dynamic object motion and results in even less error. In the static case, the results for the prediction and optic flow strategies are nearly equal. This is due to the fact that camera prediction alone almost perfectly predicts the motion in a static scene. Again, both methods result in less error than the static strategy.

The results for the coral scene are somewhat different. First, the average errors for all three methods are very close to each other. This can be attributed to the high-frequency nature of the model; many image warping errors are made at edges, hiding the impact of occlusion errors. Second, while the standard deviations are lower for the prediction and optic flow methods, there seems to be little difference between a static and a dynamic scene. This can be explained by the fact that the coral model is nearly convex with the exception of many deep holes in the surface at arbitrary orientations. The prediction methods succeed in dealing with large self-occlusions of the model, but almost always fail in handling occlusion for the surface holes due to their arbitrary directions. The motion of the surface of the coral is not a good indicator to handle all types of occlusion artifacts in this case. An interesting property of both the prediction-based and optic-flow-based camera placement strategies is that for scenes with little motion, the image quality converges to that of a directly rendered reference image. This is due to the fact that the setup of the first camera is equivalent to that of the server camera. This allows the user to inspect a nearly still scene at close-to-maximum quality.

5.4 Latency

In order to measure and compare the latency of our image warping architecture, we make use of a method for latency measurement originally proposed by Steed [22]. A tracked 6 DOF input device as well as a bright LED are attached to a swinging pendulum, resulting in sine-wave-shaped motion. The tracked spatial position of the input device is depicted

TABLE 3
Measured Latency in Milliseconds for a Stand-Alone Stereoscopic Reference Application and the PDL Image Warping Server

	Level-of-detail		PDL Image-warping	
LOD	Latency (ms)	Stdev	Latency (ms)	Stdev
1	374.4	15.7	57.4	3.0
2	190.4	9.8	58.4	5.4
4	101.0	7.9	54.3	2.6
8	73.1	5.2	56.2	3.6
16	45.3	1.6	53.6	2.8
32	46.1	1.1	50.8	2.6

The rows show various levels-of-detail. The measurements were repeated several time to produce an average and standard deviation. It can be seen that the PDL is able to guarantee low and relatively constant latency regardless the number of polygons rendered, which is something that cannot be achieved with classic LOD methods.

on the display as a small bright sphere. A video camera is used to capture the scene in such a way that both the rendered sphere as well as the LED are visible. Using image processing, the positions of the sphere and the LED relative to the extreme points of the pendulum trajectory can be determined. If the system is latency-free, these positions should match exactly; however, due to the latency, the sphere lags behind the LED. The amount of latency can be found quite accurately by determining the phase shift between the two sine-wave signals for the LED and rendered sphere using a Fourier transform.

We measured the latency for a stereoscopic reference renderer and our image warping architecture. The results are shown in Table 3. The rendered scene consisted of the 17M polygon coral and the corresponding decimated versions thereof. The scenes were rendered or warped as normal, with the exception that after rendering the back buffer was cleared in order to render the bright sphere required for latency measurements. In each case, the input device was a 6 DOF Polhemus Fastrak pen device, which was either sampled just before rendering in the LOD case, or resampled just before warping in the image warping case. These measurements were repeated several times in order to produce an average latency and a standard deviation.

6 Discussion

We have described an architecture using a programmable display layer to generate individual display frames. Application frames are extrapolated using a per-pixel 3D motion field and image warping techniques. Applications gain the immediate benefit of judder, latency, and crosstalk reduction. Subjective user feedback indicates that motion appears more smooth and responsive; furthermore, crosstalk reduction causes images to appear sharper and depth perception is increased.

As was shown, the quality of the PDL architecture in combination with image warping is competitive, and in many cases, superior to classic LOD methods for large models, especially when a 60 Hz frame rate is required. In this case, the models need to be decimated to such extent that a large reduction in image quality occurs for LOD methods. image warping, on the other hand, can maintain high-quality images because the original, high-detail geometry is rendered, and subsequently, warped to produce 60 Hz display updates. One further advantage of image warping is that no preprocessing is required on the polygonal data sets. In contrast, generating the decimated data sets for the various levels-of-detail requires a considerable amount of preprocessing time and resources. Under certain circumstances, for example, when large, static data sets are regularly updated with newer versions, precomputing static

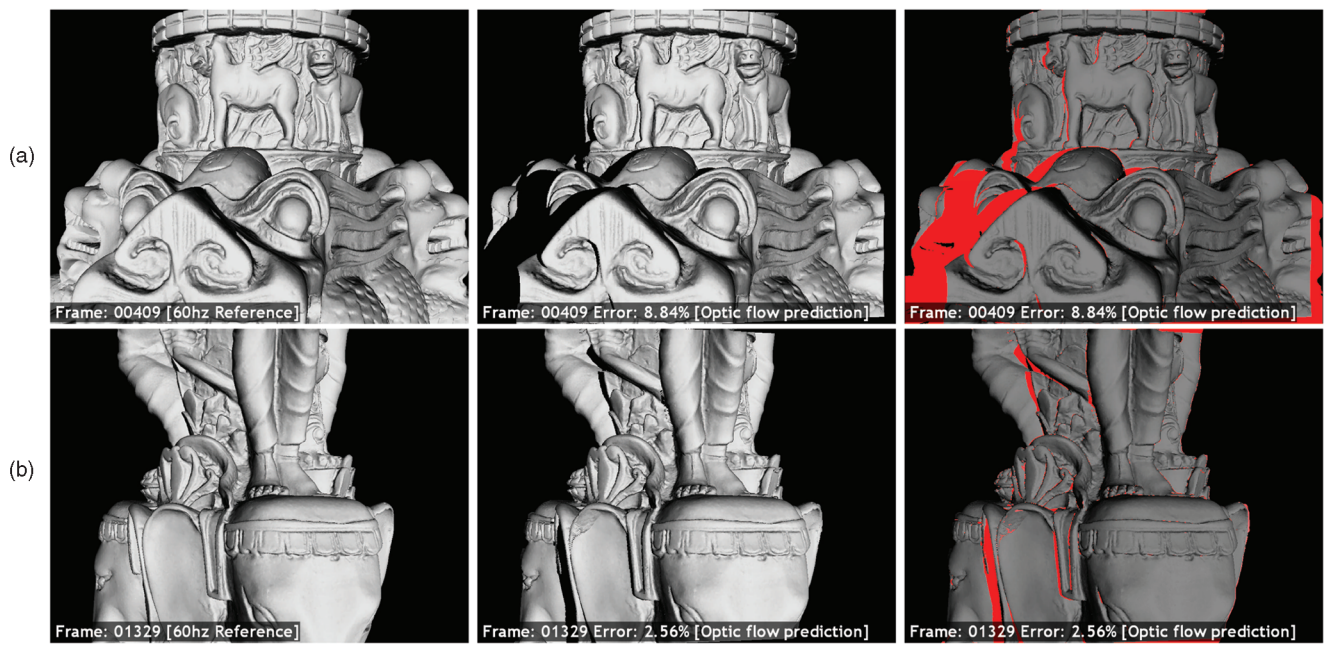


Fig. 12. Images similar to the ones shown in Fig. 11 for the dynamic statue scene. As before, from left to right are shown the reference and image-warped frame and the depiction of the error. (a) The worst error frame, which could be considered an outlier. (b) A typical frame with high error that occurs occasionally. Again, the error is much lower for the majority of frames.



Fig. 13. The importance of good client-side camera placements. From left to right, warped images of the dynamic statue scene are shown using no prediction, camera-only prediction, and prediction based on optic flow. (b) The amount of error. Using no prediction results in high error due to the relatively large scene velocity. Since the scene is dynamic, with a rotating object, the camera-only prediction method is unable to accurately predict the scene's motion and also results in high error. The optic-flow-based prediction method is able to correctly estimate the scene motion and results in an almost error-free image. This shows that with good camera placements, image warping can produce very high-quality images.

LODs may be infeasible. In this case, the PDL can be used to immediately inspect and interact with the updated data sets in real time with minimum latency.

An important challenge in VR is the production of low-latency systems. It can be seen from Table 3 that for classic LOD methods, the latency depends strongly on the number of rendered polygons and is generally high. Also, the fluctuation in frame rate, and thus, latency is high. On the other hand, for the PDL architecture, the latency remains relatively constant with low standard deviation. This result is due to the fact that the PDL operates at a constant 60 Hz display frame rate and the input device is resampled prior to warping. Low and constant latency are both important requirements for the use of subsequent predictive latency reduction methods [2]. While the PDL latency is already low, integrating further predictive latency reduction may result in a system that is almost latency-free, independent of the rendering load. This is an important result that is very hard to achieve using classic VR-architectures.

Due to the used image warping algorithms, the PDL architecture has a number of limitations. First, scene transparency cannot be handled correctly. While certain simple cases of transparency can be resolved by the generation of an extra depth layer, this is infeasible for applications such as volume rendering that require many transparent slices. A second class of geometry that our architecture cannot handle easily is that of deformable objects. While it is possible to warp the pixels belonging to deformable objects, it is difficult to predict the structural changes of the deforming surface. Per-pixel motion vectors corresponding to the motion of the deformable surface may help in dealing with this issue; however, this does not immediately solve the problem of changing surface topology. As of yet, how to best handle volume rendering and deformable objects in an image warping architecture is still an open problem.

We have used client-side optic flow in order to determine effective camera placements for warping. In this way, two full resolution client views were rendered and transmitted to the server, which warped all of the pixels in both views to the target viewpoint. When two pixels belonging to different views warped to the same location, the estimated best quality pixel was used. While this approach works, it is often quite wasteful in the amount of pixels that are transferred and warped only to be discarded in favor of a better quality pixel. The first view that is warped generally produces the best quality pixels, while the second view is mostly used to resolve occlusion artifacts; yet, all of its pixels are transferred and warped. A better approach would be to only render the subset of pixels of the second view that are useful to resolve occlusion artifacts. Furthermore, in the case of optic flow camera placement, the entire view was averaged to produce a single placement vector. In cases where the optic flow is very different across sections of the view, it would be beneficial to render these different subsections with different camera placements. Rendering only subsets of pixels allows the use of many different client viewpoints, depending on the optic flow of the subset, without significant loss of warping performance. However, one major obstacle is that standard rendering systems are ill-suited to render low-resolution images from many different viewpoints. This is due to the fact that a renderer basically has to render all the polygons for every such view, regardless the resolution—with the possible exception of some increased culling. We believe that real-time ray tracers are much better suited for such a task, since ray tracers traverse pixels and look up the corresponding

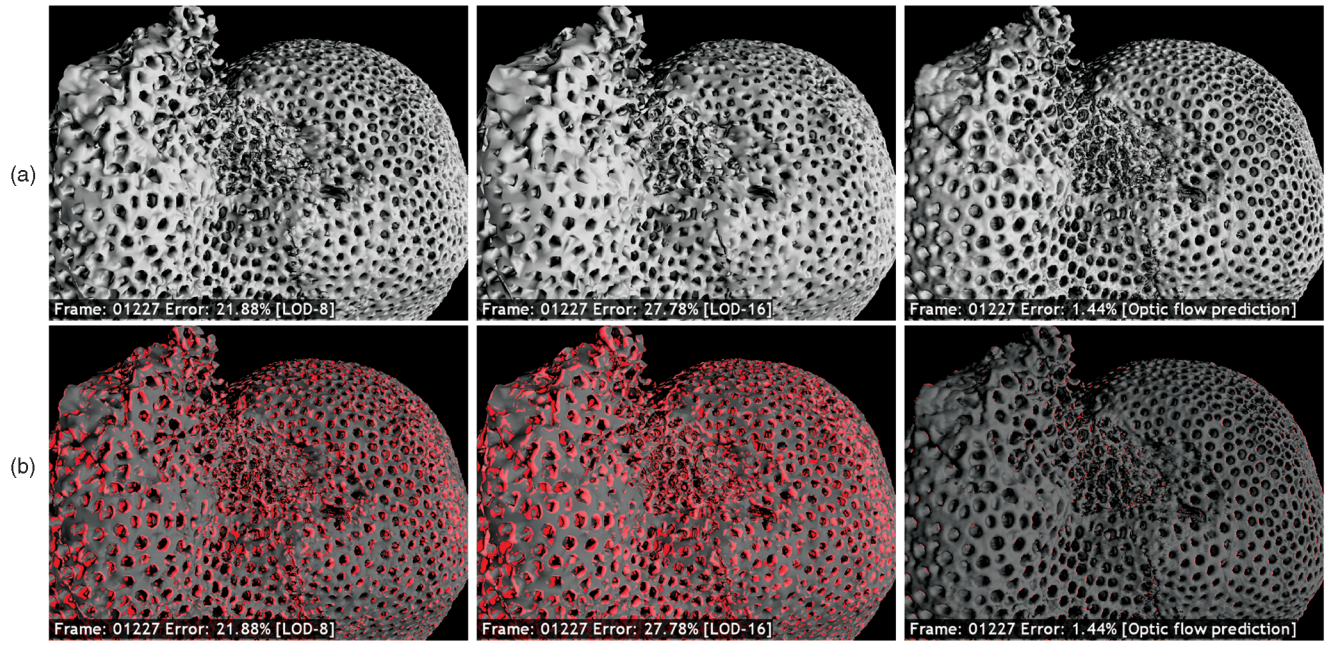


Fig. 14. Comparison between level-of-detail approaches and image warping. (a) The left-most image shows the 17M polygon coral model where the number of polygons is reduced by a factor of 8, while for the center image, the polygon reduction factor is 16. The right-most image shows the same frame produced by image warping. (b) An visual overview of the error locations. It can be seen that image warping results in much higher detail, at the expense of a different kind of errors in the form of holes. Note that the LOD-16 model is still not sufficiently decimated to allow for 60 Hz rendering.

polygons. This allows us to render individual pixels from widely varying viewpoints. An interesting approach would be to ray trace relatively small blocks of pixels on the client and determine the viewpoint used for ray tracing according to the optic flow of the block. Each block is then warped by the server as before. Such an approach may significantly reduce occlusion artifacts, and possibly even increase performance.

The method we used to compare image quality between reference animations and the output of the PDL is but one choice of many. We chose to do a simple frame-by-frame comparison using a small Gaussian filter kernel and a threshold in the Lab perceptual color space; however, many other ways are possible. Perceptual differences between images can be obtained using programs such as the visible differences predictor (VDP) [23]. We experimented with using VDP for our image comparisons, but found that the reported differences did not match well to the perceived errors when using our system. Since we are dealing with animations and not still images, a different perceptual model should be used. We are primarily interested in errors that draw the attention of the user in a disturbing way. Usually, errors at the far edges of the display, or far away from the point of attention, are not noticed at all by the user. Also, small errors in shading are usually not perceived as disturbing. The truly perceptually disturbing errors are often caused by rapid flicker in the animation caused by occlusion artifacts. Finding a good comparison method for animation sequences that can pinpoint disturbing errors like these is no trivial matter. We believe that such a method would help a great deal in the further development of real-time warping systems. Since accurately predicting perceived differences in animations is a difficult problem, we chose to use a simple and straightforward method that matched reasonably well with our experience.

7 Conclusions

We have described an architecture that provides a programmable display layer in order to generate display frames at the refresh rate of the display. Display frame motion extrapolation was performed on a dual-GPU architecture using 3D image warping. The architecture was shown to have a number of benefits, such as smooth motion of large models, latency reduction, and crosstalk reduction. Observers of our architecture reported that motion appeared more smooth and responsive and the judder effect disappeared; furthermore, crosstalk reduction caused images to appear sharper with increased depth perception. The architecture was compared to a classic static level-of-detail method. In this way, we showed that the PDL can produce images with competitive, if not superior, quality. Furthermore, the PDL architecture achieved low and constant latency regardless of the rendering load; something that could not be achieved with a classic VR-architecture. We conclude that the PDL provides a very good alternative to static LOD methods for some time to come.

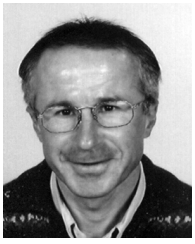
References

[1] M.R. Mine, "Characterization of End-to-End Delays in Head-Mounted Display Systems," technical report, 1993.
[2] M. Olano, J. Cohen, M. Mine, and G. Bishop, "Combatting Rendering Latency," *Proc. ACM Symp. Interactive 3D Graphics (SI3D)*, pp. 19-24, 1995.
[3] F. Smit, R. van Liere, and B. Froehlich, "Non-Uniform Crosstalk Reduction for Dynamic Scenes," *Proc. IEEE Virtual Reality (VR) Conf.*, pp. 139-146, 2007.

[4] P.J. Bex, G.K. Edgar, and A.T. Smith, "Multiple Images Appear When Motion Energy Detection Fails," *J. Experimental Psychology: Human Perception and Performance,* vol. 21, pp. 231-238, 1995.
[5] W. Bles and A. Wertheim, "Appropriate Use of Virtual Environments to Minimise Motion Sickness," RTO MP58, pp. 7.1-7.9, 2000.
[6] J.P. Springer, S. Beck, F. Weiszig, D. Reiners, and B. Froehlich, "Multi-Frame Rate Rendering and Display," *Proc. IEEE Virtual Reality (VR) Conf.,* pp. 195-202, 2007.
[7] C. Shaw, M. Green, J. Liang, and Y. Sun, "Decoupled Simulation in Virtual Reality with the MR Toolkit," *Information Systems,* vol. 11, no. 3, pp. 287-317, 1993.
[8] R. Kijima and T. Ojika, "Reflex HMD to Compensate Lag and Correction of Derivative Deformation," *Proc. IEEE Virtual Reality (VR) Conf.,* pp. 172-179, 2002.
[9] J. Stewart, E.P. Bennett, and L. McMillan, "Pixelview: A View Independent Graphics Rendering Architecture," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (HWWS),* pp. 75-84, 2004.
[10] M. Regan and R. Pose, "Priority Rendering with a Virtual Reality Address Recalculation Pipeline," *Proc. ACM SIGGRAPH,* pp. 155-162, 1994.
[11] R. Pose and M. Regan, "Techniques for Reducing Virtual Reality Latency with Architectural Support and Consideration on Human Factors," *Proc. Int'l Conf. Hypermedia, Multimedia, and Virtual Reality: Models, Systems, and Applications (MHVR),* pp. 117-129, 1994.
[12] J. Torborg and J.T. Kajiya, "Talisman: Commodity Realtime 3D Graphics for the PC," *Proc. ACM SIGGRAPH,* pp. 353-363, 1996.
[13] J.W. Shade, S.J. Gortler, L.-W. He, and R. Szeliski, "Layered Depth Images," *Proc. Ann. Conf. Computer Graphics,* pp. 231-242, 1998.
[14] L. McMillan and G. Bishop, "Plenoptic Modeling: An Image-Based Rendering System," *Proc. Ann. Conf. Computer Graphics,* vol. 29, pp. 39-46, 1995.
[15] W.R. Mark, L. McMillan, and G. Bishop, "Post-Rendering 3D Warping," *Proc. Symp. Int'l 3D Graphics,* vol. 180, pp. 7-16, 1997.
[16] F. Smit, R. van Liere, S. Beck, and B. Froehlich, "An Image-Warping Architecture for VR: Low Latency versus Image Quality," *Proc. IEEE Virtual Reality Conf.,* Mar. 2009.
[17] W.R. Mark, "Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth Image Warping," PhD dissertation, Univ. of North Carolina at Chapel Hill, 1999.
[18] J.E. Farrell, M. Pavel, and G. Sperling, "The Visible Persistence of Stimuli in Stroboscopic Motion," *Vision Research,* vol. 30, no. 6, pp. 921-936, 1990.
[19] A.J. Woods and S.S. Tan, "Characteristic Sources of Ghosting in Time-Sequential Stereoscopic Video Displays," *Proc. SPIE,* pp. 66-77, 2002.
[20] F. Smit, R. van Liere, and B. Froehlich, "Three Extensions to Subtractive Crosstalk Reduction," *Proc. Eurographics Symp. Virtual Environments (EGVE),* pp. 85-92, 2007.
[21] P. Cignoni, M. Corsini, and G. Ranzuglia, "Meshlab: An Open-Source 3D Mesh Processing System," *ERCIM News,* vol. 73, pp. 45-46, Apr. 2008.
[22] A. Steed, "A Simple Method for Estimating the Latency of Interactive, Real-Time Graphics Simulations," *Proc. ACM Symp. Virtual Reality Software and Technology (VRST),* pp. 123-129, 2008.
[23] S. Daly, "The Visible Differences Predictor: An Algorithm for the Assessment of Image Fidelity," *Digital Images and Human Vision,* pp. 179-206, MIT Press, 1993.



Ferdi Alexander Smit received the MSc degree in computer science from the Vrije Universiteit in Amsterdam in 2005. He is currently working toward the PhD degree in the Virtual Reality Group at the Centrum Wiskunde Informatica (CWI) in Amsterdam, The Netherlands. His PhD supervisor is Robert van Liere. He expects to receive his PhD degree in computer science in October 2009, after which he plans to continue work as a postdoctoral researcher. His research interests lie in virtual reality, real-time parallel computer graphics, and computer vision and tracking.



Robert van Liere received the master's degree in computer science from the University of Delft, and the PhD degree in computer science from the University of Amsterdam. He is a principle investigator at CWI in Amsterdam, where he heads the Visualization and Virtual Reality Research Group. He also holds a part time position as a full professor at the Technical University in Eindhoven. His research interests involve interactive visualization, virtual environments, and human-computer interaction.



Bernd Froehlich received the MS and PhD degrees in computer science from the Technical University of Braunschweig in 1988 and 1992, respectively. He is a full professor with the Media Faculty at Bauhaus-Universität Weimar, Germany. From 1997 to 2001, he held a position as a senior scientist at the German National Research Center for Information Technology (GMD), where he was involved in scientific visualization research. From 1995 to 1997, he worked as a research associate with the Computer Graphics Group at Stanford University. He served as a program cochair for the IEEE VR, 3DUI, IPT/EGVE, and VRST conferences. He is also a coinitiator of the 3DUI symposium series and won the 2008 Virtual Reality Technical Achievement Award. His research interests include real-time rendering, visualization, 2D and 3D input devices, 3D interaction techniques, multiviewer display technology, and support for collaboration in colocated and distributed virtual environments.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.