A Shared-Scene-Graph Image-Warping Architecture for VR: Low Latency versus Image Quality

Ferdi Smit* CWI, Amsterdam
Robert van Liere† CWI, Amsterdam
Stephan Beck‡ Bauhaus-Universität Weimar
Bernd Froehlich§ Bauhaus-Universität Weimar

ABSTRACT

Designing low end-to-end latency system architectures for virtual reality is still an open and challenging problem. We describe the design, implementation and evaluation of a client-server depth-image warping architecture that updates and displays the scene graph at the refresh rate of the display. Our approach works for scenes consisting of dynamic and interactive objects. The end-to-end latency is minimized as well as smooth object motion generated. However, this comes at the expense of image quality inherent to warping techniques. To improve image quality, we present a novel way of detecting and resolving occlusion errors due to warping. Furthermore, we investigate the use of asynchronous data transfers to increase the architecture's performance in a multi-GPU setting. Besides polygonal rendering, we also apply image-warping techniques to iso-surface rendering. Finally, we evaluate the architecture and its design trade-offs by comparing latency and image quality to a conventional rendering system. Our experience with the system confirms that the approach facilitates common interaction tasks such as navigation and object manipulation.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

1 INTRODUCTION

Most current rendering systems used in virtual environments are scene-graph based. The scene graph consists of several entities required for rendering, such as cameras, lights and geometric objects, along with their respective pose matrices and dependencies. The usual method of operation is to first update the state of the scene graph based on the most recent input sensor data and then render the scene out to the display for each application frame. Tracker sensors usually generate pose updates at 60Hz or more. The display system operates at a fixed frequency, typically 60Hz, which defines the optimal frame rate for an application. Rendering, however, can often not be performed in less than 16.6ms required for a 60Hz update rate. Thus, the scene graph update is constrained by the render process, and input data arriving at a higher rate than the application frame rate is ignored. Rendering at lower frame rates than the display refresh results in two problems. The end-to-end latency directly depends on the application frame rate and is therefore high for low frame rates [12]. Second, because the display is refreshed at 60Hz, the previous application frame is repeated by the display until a new one becomes available. This repetition of application frames results in visually displeasing artifacts for moving objects, such as judder [1] and jerky motion. Both problems significantly affect interaction tasks such as navigation and object manipulation.

*e-mail: Ferdi.Smit@cwi.nl
†e-mail: Robert.van.Liere@cwi.nl
‡e-mail: Stephan.Beck@medien.uni-weimar.de
§e-mail: Bernd.Froehlich@medien.uni-weimar.de

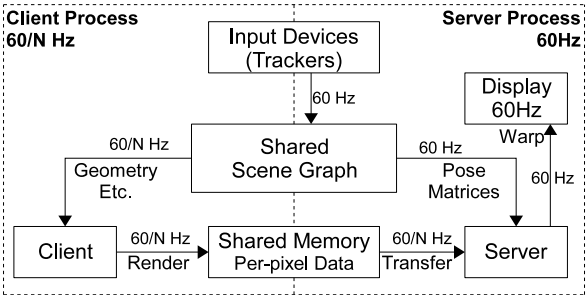

Figure 1: Overview of the proposed real-time image-warping architecture. A client and server process run in parallel, where the client generates new application frames at a fraction of 60Hz (60/N) and the server produces new display frames at 60Hz. The scene graph is shared to allow the server access to geometry and the latest pose information.

In this paper, we present an architecture that updates and renders the scene graph at the display refresh rate. This is achieved by an image-warping architecture using a parallel client and server process, which access a shared scene graph (Figure 1). The client is responsible for generating new application frames at its own frame rate depending on the scene complexity. The server performs constant-frame-rate image warping of the most recent application frames, based on the latest state of the scene graph. The central idea for enabling image warping based on the state of the scene graph is to tag each pixel in the client rendering process with an object identification (object ID). This enables the server process to warp each individual pixel based on the latest pose information from the corresponding object in the scene graph.

An important aspect is the manner in which data is shared, in particular when the client and server run on different GPUs. In this case, per-pixel data needs to be transferred from the client to the server GPU. Since the server has a strict performance requirement of less than 16.6ms per frame, it is a bottleneck to wait for a time-consuming synchronous data transfer to complete at each frame. To overcome this problem, we implement the architecture using asynchronous data transfers. In this way, data is transferred to the server GPU by a background thread while the server itself is busy processing. This reduces the impact of data transfers and allows for more per-pixel data to be shared. A single-GPU implementation runs client and server on the same GPU, with the advantage that per-pixel data is now shared on this GPU and need not be transferred.

Image warping is an image-based technique that generates novel views from reference images by performing a per-pixel reprojection. In this way, the application frames generated by the client can be transformed and re-projected for new object poses and camera viewpoints at the server. Since the warping operates on fixed image sizes (e.g. 1024x768), it results in constant image update rates and constant latency independent of the scene complexity. However, image warping comes at the expense of some trade-offs; it can have a negative effect on image quality, such as

sampling artifacts and occlusion errors due to missing image information. In addition, the quality of warped images depends on the distance to reference images and thus the frame rate of the client render process. We quantify the number and types of errors introduced by image-warping, and present a novel method for detecting and resolving occlusion artifacts. These methods can be used to increase the produced image quality, at the expense of longer processing times.

A limitation of the proposed architecture is that image-warping methods have difficulties handling scene transparency and dynamic object deformations. For certain simple cases of transparency this can be resolved by generating one or more extra depth-layers of per-pixel information on the client, much like LDIs [14]; however, this is highly inefficient for applications such as volume rendering using many transparent slices. Object deformations are especially difficult to handle due to changing surface topology. We will investigate how image-warping can be used efficiently for volume visualization through iso-surface rendering.

The presented architecture incorporates input sensor updates into the scene graph and the displayed image at the display refresh rate, which results in the minimal latency achievable through non-predictive methods. Our investigation of different image-warping methods show that point-splat warping is currently the fastest method providing a reasonable image-quality trade-off. It achieves constant 60Hz warping for 1024x768 stereoscopic images when running client and server on different GPUs. From our experience with the implementation, we conclude that our low latency system running at 60Hz significantly improves the interactivity, besides eliminating judder and jerky motion. Navigation and object manipulation can particularly benefit from our architecture due to the reduced latency [3]. In addition, object selection and most system control tasks are executed at the display refresh rate because of the direct influence of the latest scene graph state on the warped images.

The main focus in this paper will be on world-in-hand scenes consisting of very large models. We believe this scenario to be typical for many scientific visualization applications, especially in combination with large models obtained by scanning real-life objects. Furthermore, we aim at providing good solutions for desktop-VR systems where head motions are typically small and individual object motion, often realized using 6-DOF input devices, is extensive. Although all of the presented techniques also work in different settings, such as for walk-through scenes and HMDs, these may require different, fine-tuned configurations to avoid introducing additional errors due to rapid changes in viewing direction and increased occlusions.

2 Related Work

Image-based rendering by 3D warping was introduced by McMillan and Bishop [11]. Layered depth images (LDI) [14] is a technique that combines several depth images from nearby views into a layered representation in order to reduce occlusion artifacts and holes. In the context of auto-stereoscopic displays, image-warping using splatting was used to generate the multiple required shifted view-points from a single rendered view [4]. A good overview of many different warping algorithms is given by Mark [9]. Other approaches have been explored in order to reduce latency for interactive VR-systems. A very common method is to use predictive Kalman filtering [5] on the received tracker reports prior to the rendering of a new frame. Further examples are the PixelView architecture proposed by Stewart et al. [19], the Reflex HMD by Kijima et al. [6], the SLATS system by Olano et al. [13] and the Talisman architecture by Microsoft [20]. All of these systems either require special hardware to be used in real-time, impose constraints on the scenes used for rendering, or were test-bed systems that did not operate in real-time for realistic resolution and scenes. Furthermore, the focus of these systems mostly lies on static scenes and viewpoint changes, with no support for moving objects. Our approach supports dynamic scenes and operates in real-time at 60Hz for stereoscopic displays using commodity hardware. Because of this, we support realistic latency reduction for head-tracking as well as general 6 DOF input devices in common, practical VR-environments. A longer overview of related work is given by Smit et al. [16].

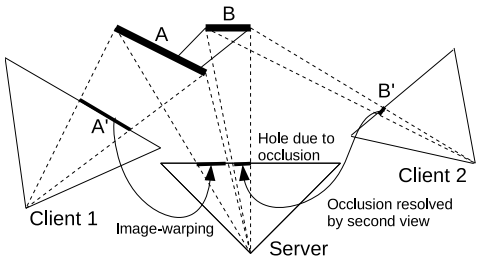

Figure 2: Using a second client view in order to reduce occlusion artifacts in the form of holes. Object A and B are visible in the server's projection; however, object B is occluded by object A in the first client view's projection, resulting in a hole. By adding a second client view the occlusion issue can be resolved.

We have previously evaluated the difference in image quality between using image-warping and a classic level-of-detail (LOD) method to achieve a 60Hz frame rate for large models [16]. A standard edge-collapse LOD method was used decimate the set of polygons to a pre-specified number that the hardware can render at 60Hz. It was shown that in this case image-warping produces better quality images than those produced by the LOD method. This result increases our believe that image-warping methods can provide a good trade-off between image quality and low latency systems.

An important aspect of image warping is the link between image quality and the number of client cameras and their placements. Most previous camera placement strategies (e.g. [9]) were concerned with static scenes only. For dynamic scenes in particular, good client-side camera placements can reduce the amount of errors resulting from image warping. We have shown that by using two client-side cameras along with prediction based on the optic flow of the scene, errors caused by occlusion can be reduced significantly [16]. In this paper, however, we use a simple client-side camera placement where two client cameras are placed at fixed positions relative to the server camera.

Previous image-warping architectures did not share the scene graph; although the use of object IDs has been proposed [9], this was not widely adopted. There are several important benefits to assigning object-IDs to pixels and sharing the scene graph completely between client and server. First, the latest pose for objects as well as cameras can be re-sampled from the input devices or animation files. Consequently, late input device re-sampling is no longer restricted to camera pose updates alone, and allows for latency reduction at the object level and for dynamic scenes. Furthermore, certain scene graph updates, such as object selection, can be performed directly at 60Hz by the server, without waiting for the next client frame. Finally, since the server has access to all geometry, errors in the image-warping output can be detected and the corresponding geometry can be re-rendered at these locations by the server. This increases image quality.

3 Architecture

An overview of the presented architecture was previously shown in Figure 1. The server performs image warping to generate a transformed and re-projected server view using images generated by the

client. The implementation is free to choose the number of images generated by the client for each frame, as well as the client viewpoints used for this purpose. However, as can be seen in Figure 2, a minimum of two client views are generally required to produce a single server view without serious occlusion artifacts [9]. In this case, both client views are warped to the same server view where the depth buffer handles overlapping pixels. Consequently, for stereoscopic rendering, the server needs to perform image-warping four times: twice for each of the left- and right-eye server views.

An important aspect of this architecture design is the shared scene graph. Both the client and the server have full access to the complete scene graph state and geometry. This enables the server to use the latest available object pose information for warping, and also the possibility to render parts of the geometry.

3.1 Client

The purpose of the client is to render images from different viewpoints, in such a way that the server can use this data to perform effective image-warping. We call the data generated by the client for this purpose a *client frame*. To generate the required data, the client renders the scene from multiple viewpoints. Our implementation uses two viewpoints per client frame, but depending on application requirements more can be added. At the start of a client frame, a buffer slot is acquired for data storage (see Section 3.3). Next, the head-tracker device is sampled to determine the latest camera pose. Since we assume stereoscopic displays, we can render a left- and a right-eye view for this camera pose. In order to avoid some clipping at the borders of the screen after warping, we increase the field-of-view of the cameras. The eye-separation is also increased in an attempt to avoid some warping occlusion artifacts. For each viewpoint, the inverse of the corresponding camera's projection and modelview matrices are stored in the buffer slot as $C_{img\rightarrow cam}$ and $C_{cam\rightarrow wld}$, respectively.

Next, the scene graph is rendered for each of the viewpoints. For every geometric object that is to be rendered, a static 16-bit object ID i is assigned to that object and the inverse of the corresponding object matrix is stored in an array in the buffer slot as $C^i_{wld\rightarrow obj}$. The scene is rendered using a custom shader program that outputs per-pixel color, normal, depth and object IDs. The pixel's color is stored in a 4-component 8-bit BGRA format, where the alpha component is used for the low eight bits of the object ID. The normal is converted to the $[0,1]$ interval for each of three components and stored as 16-bit fixed point. No shading is performed, since deferred shading will be done on the server using the stored normal. Post-projection depth information is also converted to $(0,1)$ and stored in a 16-bit fixed point integer format as $2^{16}\cdot z/w$. Finally, the high eight bits of the object ID are stored in a single 8-bit component. This results in a total required storage space of 13 bytes per pixel.

3.2 Server

The server starts by polling the client in a non-blocking fashion to determine if a new client frame is available. Details of different approaches to achieve this are given in Section 3.3. If a new frame is available, it is used as the source for warping; if not, the server continues using the previously received frame. A ratio is kept that indicates the number of frames the server renders until a new client frame becomes available; for example, if the server renders three frames for every client frame, this ratio equals 3:1. Next, a Δt value can be computed according to the number of times a new client frame was not available since the last received frame and the kept ratio. This value is useful for motion extrapolation, which will be described below. Finally, the server generates a left- and right-eye stereoscopic view from a newly sampled head-tracker pose $S_{wld\rightarrow cam}$ by warping the two client viewpoints for each view individually.

Suppose that we know the 3D homogeneous coordinates of a pixel P_{xyzw} in the client's post-projection space and the pixel's corresponding object ID i; in that case, it is possible to warp the pixel to the server's new viewpoint by first unprojecting the pixel, transforming it back to world-space, applying object transforms and finally reprojecting it using the server's camera: $P'_{xyzw} = M^i \cdot P_{xyzw}$, where M^i equals:

$$S_{cam\rightarrow img}\cdot S_{wld\rightarrow cam}\cdot S^i_{obj\rightarrow wld}\cdot C^i_{wld\rightarrow obj}\cdot C_{cam\rightarrow wld}\cdot C_{img\rightarrow cam}$$

The matrices denoted by S are the server's projection, camera and object matrices, and the ones denoted by C are the client's corresponding matrices. The procedure is depicted in Figure 3. The per-pixel normals are stored in the client's camera space. For the purpose of deferred shading, the normals need to be transformed to the server's updated camera space. This is achieved by computing the following normal-matrix:

$$N^i = ((S_{wld\rightarrow cam}\cdot S^i_{obj\rightarrow wld}\cdot C^i_{wld\rightarrow obj}\cdot C_{cam\rightarrow wld})^{-1})^T$$

The matrices M^i and N^i are calculated for each object once per server view, and are then uploaded to the GPU. In this way, a GPU image warping algorithm can warp a pixel or a normal by a single 4x4 matrix multiply, where the required matrix is pre-computed per-object instead of being computed for each pixel. Furthermore, all the warping equations are expressed in terms of general 4x4 homogeneous matrix computations, allowing for arbitrary transforms to be easily inserted in the warping pipeline and effortless integration with the standard OpenGL rendering pipeline. We found that these computation do not generally cause a large performance overhead for several hundreds of objects. For a very large amount of objects, the computations may impact performance, and the implementation should be extended to perform them on multiple cores and in parallel with the GPU operations. Some additional work that can be pre-computed, such as extrapolated matrices for fixed time steps, could be performed by the client.

So far we have not yet mentioned the significance of the server's object matrices $S^i_{obj\rightarrow wld}$, nor how they are calculated. If the server's object matrix for any particular object is equal to the client's object matrix for that object, the object will only appear to change pose whenever a new client frame becomes available. This results in non-smooth motion, or judder, and high latency. However, using image warping, we are free to change the server's object matrices as well as it's camera matrix. This can be done either by sampling the newest pose from a 6 DOF interaction device, or by extrapolating object motion when it is not linked to an interaction device. In order to extrapolate object motion, the server keeps the client's object matrices for the previous client frame as well as for the current client frame. Using these two object poses and the previously calculated Δt value, we can extrapolate the pose by performing a quaternion spherical linear extrapolation on the rotational part, and a regular linear extrapolation for the translational part. In effect, this is a linear prediction. Note, no such prediction is performed for object poses that can be updated in the scene graph at 60Hz.

3.3 Data transfers and synchronization

A mechanism is required to move data from the client to the server and to synchronize these transfers. The most efficient way to do this depends heavily on whether the architecture is implemented on a multi-GPU or on a single-GPU system. In the former case, the data needs to be transferred from the client to the server GPU over the PCIe bus, while in the latter case buffer data can be left in the video RAM of the single GPU and need not be explicitly transferred. For either case, buffer synchronization is required. Due to this dependence on the underlying system, we distinguish between

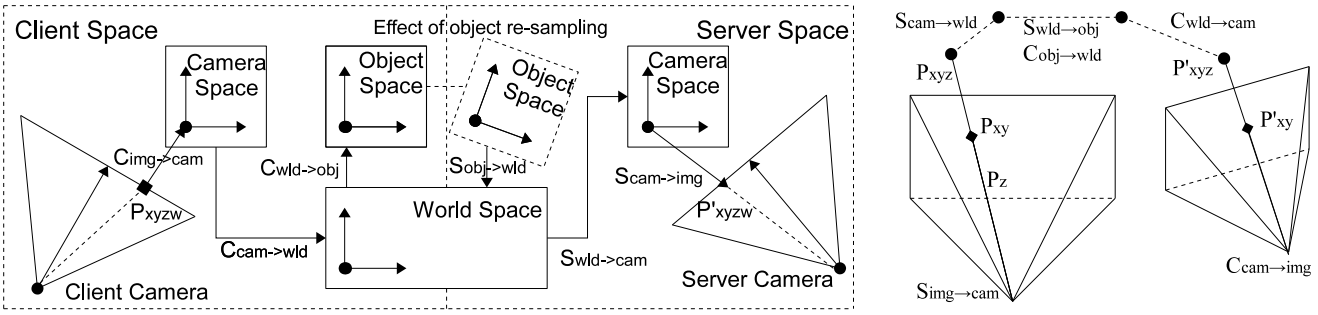

Figure 3: (left) Schematic overview of warping a client pixel P_{xyzw} to the server pixel P'_{xyzw}. Re-sampling of object poses is essentially achieved through the server's object-to-world transform. All six required matrices are concatenated into a single warping matrix for each different object. (right) Simplified 3D depiction of the same process.

data transfer and synchronization implementations for a multi- and single-GPU system.

In a previous multi-GPU implementation, we have used direct, synchronous data transfers [15]. The time required for synchronous transfers turned out to be a bottleneck; therefore, we have implemented indirect, asynchronous data transfers. To achieve this, the architecture runs four threads: a client rendering thread, a server warping thread, a client frame downloading thread and a server frame uploading thread. Each of these threads is executed in parallel on a quad-core CPU. The client rendering thread and the server warping thread each open a separate OpenGL context on a different GPU and share this OpenGL context with either their corresponding down- or uploading thread, respectively. Client frames are written to and read from a circular producer-consumer buffer containing a number of slots where client frames can be placed. With such thread-synchronized buffers, we can either request a new slot to write to, or poll (optionally blocking) for a new slot to read from. Since separate GPUs can not share memory directly, we need three instances of such circular buffers: one on the video RAM of the client GPU, one in system shared memory and one in the video RAM of the server GPU. The slots in shared system memory can be accessed by data pointers, while the slots in video RAM are accessed by OpenGL buffer object IDs. These producer-consumer buffers are shared between the threads to allow communication; the client's GPU buffer is shared by the client's render and download threads, the shared system memory buffer by the client's download and the server's upload threads, and the server's GPU buffer by the server's upload and the server's warping threads. This is shown schematically in Figure 4.

The transfer of client frames to the server GPU now proceeds as follows. First, the client render thread acquires a free slot to write to from its producer-consumer buffer in client GPU space, after which it renders the frame to that GPU buffer. The client's download thread polls for newly rendered buffers and downloads them to system shared memory. Next, the server's upload thread will detect a new buffer in shared memory and proceeds to upload it to the server GPU. Finally, the server warping thread detects that a new frame is available in its GPU memory, and switches to that GPU buffer as the source for image-warping. All these threads operate asynchronously and data transfers are performed using the OpenGL PBO extension's DMA transfers. The effect of transferring data in this fashion is that the image-warping server does not need to spend time on uploading data to the GPU; however, new client frames will arrive a short time later due to background transfers. An important observation is that updates received by the server of newly rendered client frames were already infrequent (say every 200ms), so an extra delay of a few milliseconds is hardly significant with respect to the total update time between new client frames; on the

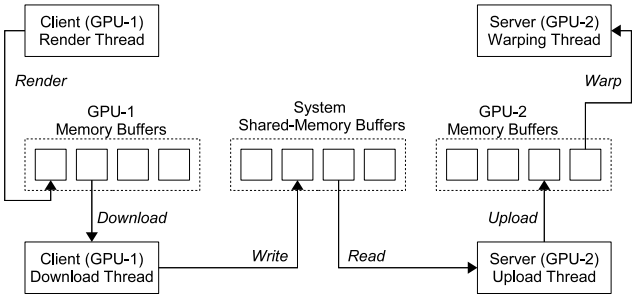

Figure 4: Schematic overview of the implementation of asynchronous data transfers for image warping. A situation is shown where every thread is working on a different buffer slot. In practice, other situation may occur; for example, since data transfers are relatively fast and client rendering relatively slow, the server will often be using the previous slot used for client rendering.

other hand, since only 16ms are available for image-warping, had this delay of a few milliseconds occurred then, even such a small delay would have a significant impact on image-warping performance, taking up a large percentage of the available 16ms. With asynchronous transfers, the server does receive new client updates with some delay, but it can still continue to perform image warping in the meanwhile; therefore, the delay slightly reduces image quality due to the larger warping distance between slower client updates, but it has minimal impact on warping performance. The resulting latency is unaffected because the server still manages to perform image warping for every display frame.

For the single-GPU implementation, no data transfers are required; therefore, it suffices to synchronize multiple buffers that are kept in the GPU's video RAM. Two threads are now used: one for the client's rendering and one for the server's warping. Both threads open an OpenGL context on the same GPU, which is shared to allow accesses to all resources from either thread without copying data. We use the same circular producer-consumer buffer as described previously; however, only a single instance is now required that manages the frames in the single GPU's video RAM. Note, in earlier work [15] we introduced the notion of client chunk-size on a single GPU, where the client split the geometry into small chunks for rendering to ease the scheduling and time-slicing of the GPU. With the next generation of graphics drivers and vertex buffers this appears to be no longer necessary. We ran some tests that confirmed that chunk size has minimal influence on the achieved frame rates; therefore, all client-side geometry is now rendered efficiently in a single batch.

3.4 Image warping

Image-warping is performed entirely on the GPU by custom vertex, geometry and fragment shaders. Rendering begins by drawing a static, screen-aligned grid consisting of a number of evenly spaced vertices equal to the client's resolution. There is a one-to-one correspondence between vertices in the grid and pixel-centers in the client texture. Therefore, per-pixel image warping can be achieved by transforming the individual vertices in the grid. Each vertex contains the post-projection 2D coordinates $P_{xy} \in [-1,1] \times [-1,1]$ of the corresponding pixel's center. Other per-vertex information that is available from buffer generated by the client is the post-projection depth $P_z \in (0,1)$, the object ID i, the normal n and the pixel's color. Furthermore, it is assumed that the homogeneous coordinate $P_w = 1$. Each vertex is warped to its new location in a vertex shader. First, we fetch the pre-constructed warping matrix M^i corresponding to the pixel's object ID from GPU memory with four texture reads. Since the combination of the pixel's 2D coordinates and depth results in a valid vertex in post-projection space, it can be warped easily by pre-multiplying it by the warping matrix: $P'_{xyzw} = M^i \cdot P_{xyzw}$. Homogeneous or perspective division will occur later in the OpenGL pipeline, so this P'_{xyzw} can be send directly to the fragment shader.

Next, we need to determine the projected size of the warped pixel. Suppose that the depth for a given pixel with coordinates (x,y) is given by a two-dimensional function $z = f(x,y)$. In that case, the warp can be seen as a general 2D coordinate transform: $P'_{xyzw} = M^i_4 + M^i_1 x + M^i_2 y + M^i_3 f(x,y)$, where M^i_n is the n-th column of M^i. This follows directly from $M^i \cdot P_{xyzw}$. The corresponding Jacobian matrix of this transform has 2x2 dimensions because the function can be interpreted as to map 2D screen pixel coordinates to warped 2D pixel coordinates. The expansion factor of this transform, representing the change in size of a warped pixel, is given by the Jacobian, which is the determinant of the Jacobian matrix of partial derivatives, after homogeneous division:

$$J = \begin{pmatrix} \partial(P'_x/P'_w)/\partial x & \partial(P'_y/P'_w)/\partial x \\ \partial(P'_x/P'_w)/\partial y & \partial(P'_y/P'_w)/\partial y \end{pmatrix}$$

After simplification of the derivatives and substitution by the elements of P'_{xyzw}, we find that:

$$J_{11} = \frac{P'_w M^i_{11} - P'_x M^i_{41} + (P'_w M^i_{13} - P'_x M^i_{43}) \cdot \partial f(x,y)/\partial x}{(P'_w)^2}$$

The other three elements are found through very similar equations. The partial depth derivatives for $f(x,y)$ can be approximated by using the depth buffer gradients: $\partial f(x,y)/\partial x = (f(x+1,y) - f(x-1,y))/2$, and so on. However, a more robust depth derivative can be computed from the pixel's original normal n as $(Sn_{xy})/n_z$ where S is the size of a pixel in post-projective space, which is $(2/w, 2/h)$ for a resolution of wxh. This final form is equivalent to the warping equations used by McMillan [11]; however, the ones here are written in general 4x4 homogeneous matrix form for easy concatenation of transforms. In this way, the Jacobian matrix can be computed quite efficiently in the vertex shader, given that P'_{xyzw} is already computed. Finally, the pixel's normal must be warped to camera space for the purpose of deferred shading. This is achieved by pre-multiplying the normal by the pre-computed normal matrix $n' = N^i \cdot n$.

4 Image-warping Errors

Image-warping techniques generally introduce a number of errors in the output images. The magnitude of errors depends on the distance between the warped source view and the resulting target view and, consequently, the client's frame rate. We distinguish between two types of errors: sampling errors and occlusion errors. Sampling errors are due to errors in pixel-surface reconstruction and shading, while occlusion errors result in missing geometry in the output images.

Figure 5: A quality comparison of a 60x60 pixel close-up from a 10M polygon statue model. From left to right: point-splat, quad-splat and mesh-based image warping methods and the directly rendered reference. Some noise appears for the splat-based methods due to overlapping splats; however, this is generally not very noticeable when viewed from a distance. The mesh-based result shows slightly too much blurring.

4.1 Sampling errors

Sampling errors are caused by the use of image-warping to reconstruct continuous surfaces using only discrete pixel information. Generally, these sampling errors manifest as small errors in shading and slightly thicker edges at sharp depth boundaries. When the distance between the source and target images is large, a loss of resolution can also be distinguished. However, in practice these degradations of image quality are not very noticeable, especially not for animated scenes.

In an effort to reduce sampling errors, we have implemented three different warping algorithms. All three algorithms use the same warping equation, but vary in the way the warped pixels are rendered: a screen aligned point splat, a general quadrilateral splat and a mesh-based reconstruction. Sample output of these three algorithms is given in Figure 5. The point-splat method computes the Jacobian matrix as described earlier (see Section 3.4) and then determines a per-pixel splat size $s = ceil(max(J_{11} + J_{12}, J_{21} + J_{22}))$. This size is set in the vertex shader on a per-pixel basis using the *gl_PointSize* variable; hence, this method can only render screen aligned squares of size *sxs* pixels. The splat size is ceiled to avoid hole artifacts resulting from discrete splat sizes that are too small to cover the surface entirely. A drawback is that this generates slightly thicker edges and overlaps individual splats. A more flexible approach is the use of a general quadrilateral splat. In this case, the geometry shader inputs a pixel to be warped and outputs a single quad. The four vertices of this quad are computed by post-multiplying the Jacobian matrix by half-pixel offsets $(x \pm 0.5, y \pm 0.5)$ of the warped pixel's position. This results in less overlap for individual splats. Note that the previously described discrete point splat size S is equal to the smallest square completely covering this quad. Finally, a mesh-based reconstruction similar to the one proposed by Mark and McMillan [10] treats the grid of pixels as a connected triangle mesh. Each vertex in this grid mesh is warped, and the fragment shader renders connected triangles accordingly. Because the grid is completely connected, there is no need to calculate a splat size and the Jacobian matrix need not be computed.

4.2 Occlusion errors

Image-warping potentially leaves holes in the produced images caused by occlusion in the warped source images. An example has been given in Figure 2, and Figure 7 depicts what these errors look like in a practical application. Errors caused by holes are perceptually very noticeable, especially in animated scenes, and should be minimized as much as possible. A post-processing step can be applied after warping to first find hole pixels and then resolve or fill them. In the past, a number of hole-filling strategies have been proposed [9]. The problem with these methods is that not all hole

pixels are found correctly, and the methods for resolving the holes that are found do not work well for dynamic scenes and a GPU implementation. Therefore, we now describe a novel way of both detecting and resolving holes in dynamic scenes.

4.2.1 Detecting holes

To detect hole pixels, we must first define what constitutes a hole pixel. We distinguish between three types of holes:

- Level-0 holes are those destination pixels where no pixels warp to, i.e. pixels that are left blank in the final image. These are the classic holes found by methods as described by Mark et al.[9].

- Level-1 holes are pixels where an object is visible, but not the correct object. This case occurs frequently in dynamic scenes, or when using multiple client views, when a part of geometry is missing in the warped image due to an occlusion error and part of some different geometry warps into this gap, behind the geometry that should have been there.

- Level-2 holes are pixels that do show the correct object, but not the correct part of it. For rigid object transforms this case occurs infrequently.

Level-0 holes can easily be detected by scanning the warped output image and marking every pixel that does not have some flag set to indicate it is the destination of a warped pixel. Detecting level-1 and level-2 holes requires more effort. The governing idea is to compare a warp map consisting of object IDs and depth of warped pixels to a reference map of which IDs and depth should be at those locations. If the object IDs in the two maps do not match, then we flag the pixel as a level-1 hole. If they do match, but the warped pixel is further away than the depth of the bounding box, up to a threshold, we have found a level-2 hole. In this way, level-0 holes will always be found because their object ID of zero does not occur in the scene graph.

Our implementation outputs per-pixel object IDs and depth for every warped pixel into a separate buffer making up the warp map. Per-pixel object IDs are directly available from the regular shared-scene-graph warping, so this step is trivial. Next, we construct the reference map. To obtain a perfect reference map, we would have to render the entire scene, so we use an approximation instead. To achieve this, we start by pre-computing a kd-tree for every object in the scene. Building this kd-tree currently takes a couple of seconds for models consisting of millions of polygons and is performed on-the-fly when a model file is loaded; however, the kd-tree could also be pre-computed once in advance and stored on disk. The kd-tree gives us a convex-hull bounding-box hierarchy of increasing resolution, with leaf nodes consisting of small subsets of the geometry. We now render a convex-hull approximation for each object by rendering all the bounding boxes at a specific resolution in the corresponding kd-tree. Rendering several thousand of these bounding boxes can be achieved in about one or two milliseconds using hardware instancing, so this quickly gives us an approximated reference map of object IDs and depth. Because of the nature of image-warping and occlusions, hole pixels always have a larger depth value (further away from the viewer) than the correct geometry would have. Therefore, we can detect holes by comparing the warp map to the approximated reference map made up of convex-hull bounding boxes.

As an optimization, we always re-draw the entire background quad after warping to solve many common holes. A hole-detection optimization is also possible here. If the approximated reference map tells us that a pixel is a background pixel, then this must be so due to the convex hull nature of the kd-tree bounding boxes. Therefore, if the warped pixel also turns out to be a hole, we can safely skip it, since it must be a background hole that will be filled anyway by redrawing the background quad. This reduces the number of candidate holes found significantly. The same approach is not possible when only scanning for level-0 holes, since there is no distinction between holes and background then.

A problem with the described hole detection implementation is that it will detect some amount of false-positives, i.e. pixels that are flagged to be holes, but in reality are not. This happens most frequently at the edges of objects for the level-1 holes, and at sharp depth edges for level-2 holes. The reason is that the bounding boxes cover a larger portion of screen-space than the actual geometry, and these pixels receive incorrect object IDs and depth from the approximated reference map. Therefore, for performance reasons, we restrict the algorithm to finding level-1 holes only, and comparing object IDs only, as this reduces the amount of false-positives. Note, however, that detecting false positives only costs us performance, and not image quality, after trying to resolve these holes.

4.2.2 Resolving holes

In the previous section it was described how to find a bitmap of candidate hole pixels. Now we will resolve the found holes by directly rendering in the missing geometry on the image-warping server using the shared-scene-graph geometry. The idea is to shoot rays through the hole pixels, trace those rays through the pre-computed kd-tree to find the intersected leaf nodes, and to re-render these complete leaf nodes that form a super-set of the intersected geometry. Rendering entire leaf nodes, which in our implementation consist of approximately 1K polygons, avoids the need to test every face in the node to find an exact intersection, which is too slow in practice. However, it does cause more polygons to be redrawn, but this can usually be done much quicker on a GPU.

To resolve holes we need to download the hole map to the CPU. To reduce the size of data, we test blocks of 4x4 pixels for holes on the GPU and combine the output of these 16 tests into a single 16bit unsigned integer texture. The hole map now becomes so small that it can be downloaded to the CPU in a fraction of a millisecond. Next, we iterate over the 16-bit integer hole map on the CPU, searching for integers that are not equal to zero, indicating at least one of the pixels in the corresponding 4x4 block is a hole pixel. This saves many comparisons for the generally sparse hole map. A straight-forward approach would be to test every pixel in the block and trace a ray through it if that pixel's bit is set, indicating it to be a hole; however, this requires many rays to be traced, degrading performance. Therefore, we only trace a single ray through the center of the 4x4 block of pixels. The corresponding rays for the other potential hole pixels in the block are so close to this center ray that they usually intersect the same leaf nodes. Since these leaf nodes are to be re-rendered anyway, it usually suffices to trace only the single center ray. This approach may cause us to miss the right geometry, leaving the hole unresolved, but it greatly improves performance. Next, we determine the leaf nodes in the kd-tree that the center ray intersects, and store the indices of these leaf nodes in a unique set, ensuring that we never re-render the same leaf node twice.

Due to the structure of the kd-tree, intersected leaf nodes are found in front-to-back order along the ray; however, this does not guarantee that the first-found intersected leaf node contains the correct geometry to resolve the hole. It is possible for the ray to hit the bounding box for the geometry contained in the leaf node, but not to hit the actual geometry itself. This is often the case at object boundaries and sharp depth edges. In that case, other geometry in leaf nodes further away is actually visible in the place of the hole. To be absolutely certain we re-render the right geometry, we would need to re-render all the intersected leaf nodes along the ray. Another optimization is to only re-render the first few hit leaf nodes. Again, this may cause the correct geometry to be missed, failing to

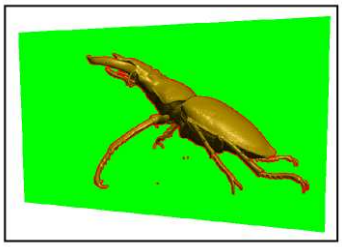

Figure 6: Visualization of the post-warping reconstruction pass on the server. Green area: no starting position available. Red area: starting position available but no iso-surface found during the binary search.

resolve the hole. We found that re-rendering the first three hit leaf nodes generally provides good quality.

5 ISO-SURFACE RENDERING

In the context of volume rendering, iso-surface rendering is a common task: to find and render the iso-surface within a volume data set for a given iso-value. Both steps can be done separately or in a direct manner. The Marching Cubes algorithm [7] is able to extract the iso-surface by traversing a volume voxel by voxel. The iso-surface can be stored as a polygonal mesh and then be rendered. Another technique is to use (direct) iso-surface ray-casting. By ray-casting a volume in screen space, the detected iso-surface along each ray can be rendered directly. Direct iso-surface ray-casting can be implemented on programmable GPUs [18]. The computational cost for ray-casting depends on screen size, volume size, the searched iso-value and the actual voxel data. For small and medium sized volumes up to 256^3 voxels, iso-surface ray-casting can be done in less than 16ms (60Hz) on current graphics hardware. Rendering larger volume data sets is still too expensive to achieve interactive frame rates.

We investigated ways to integrate iso-surface ray-casting within our image-warping architecture in a natural way. Our first approach is to extended the image-warping architecture for low-latency iso-surface rendering, analogous to the low-latency polygonal rendering. As a second approach, we propose a modified warping architecture that aims for better iso-surface reconstruction on the server side. We will now describe these two approaches in turn.

The first approach is a client-side extension, where the client renders and stores iso-surface information instead of polygonal surface information. This is done by ray-casting the volume in a fragment program. For each fragment, a ray traverses the volume at a fixed sampling distance in front-to-back direction and searches for a given iso-value. If two consecutive samples along the ray indicate that the iso-surface lies in-between them, the ray segment is refined using binary search to find the sampling position that matches the iso-value best. Whenever an iso-surface is found, the gradient is computed and interpreted as surface normal. The surface normal is then stored along with the object-ID and depth in a client buffer. The surface information stored in each client view is similar to the surface information which would be stored in the polygonal warping-architecture; therefore, we have the possibility to warp and reconstruct the extracted iso-surfaces in the same way for both polygonal data and volume data. This means that the server does not need to be modified at all in order to do the warping, deferred shading and reconstruction work.

The second approach aims for a better reconstruction of the iso-surface on the server side. It is currently only implemented for a single volume in a world-in-hand scenario, without the benefit of a full scene graph. As there is only one object, the volume, no object-IDs are needed. We modified the previously described volume ray-casting on the client and the warping mechanism on the server in the following way. The client's ray traversal stops at the point when the binary search would begin to refine the sampling position. At this point, the client stores the depth at this sampling position. The server now warps the depth information from the client view into the depth buffer of the actual view, without performing any shading. After this warping step is done, the newly filled depth buffer contains either a valid volume position, or no information at each pixel. In a second pass, the bounding box of the volume is rendered, and the depth buffer is bound to a texture unit and used as a look-up for a valid starting position for the binary search, which is now done on the server side (see Figure 6). This post-warping reconstruction pass enables us to search in the volume for the iso-surface in a short range along the ray, similar to the method proposed by Magnus et al.[8]

There are three benefits to this second approach. First, we are able to reconstruct the iso-surface without traversing the volume from the volume entry point. Second, we are able to fill small holes that tend to open up during warping at the silhouettes by slightly enlarging the point-size during the warping of the starting positions on the server. Third, warping the starting positions for the deferred binary search into the actual view saves traversal steps on the server.

Multi-GPU								
	640x480		800x600		1024x768		1280x960	
	Warp	Upd.	Warp	Upd.	Warp	Upd.	Warp	Upd.
Point	5.8	212	9.0	214	14.7	218	23.1	222
Quad	10.2	212	15.9	214	25.9	218	41.0	222
Mesh	15.7	212	24.5	214	40.3	218	63.2	222
Single-GPU								
Point	9.2	900	11.8	1150	16.3	1586	25.9	2325
Quad	14.5	1426	20.1	1883	30.3	2857	44.5	4319
Mesh	48.7	654	76.0	655	123.5	659	193	672
Reg.	207.0		207.1		207.4		207.9	

Table 1: Frame rates for single- and multi-GPU implementations that warp two views into a stereoscopic image; hence, image-warping needs to performed four times. Occlusion artifacts are neither detect nor resolved. All data are in milliseconds (60Hz corresponds to 16.6ms available time). The bottom row shows the time required for a regular stereoscopic renderer to produce the same images. For each resolution, the first column shows the time required for warping on the server, while the second column represents the time between fresh buffer updates arriving at the server from the client. The pipeline of a single GPU is quickly congested under heavy warping loads; in combination with the scheduling overhead, this leaves little time for new client updates to be rendered. (Intel Core2 Quad Q9950 2.83Ghz + 2x NVidia GeForce GTX 260)

6 RESULTS

6.1 Frame rate

We have tested the performance of the multi- and single-GPU implementation of the warping architecture on a system consisting of of an Intel Core2 Quad Q9950 2.83Ghz and two NVidia GeForce GTX 260 video cards connected over a PCIe 2.0 16x bus. For the single-GPU implementation one of these video cards was disabled. The test scene consisted of the 28M polygon Stanford Lucy angel model orbited by twelve torus knots. Table 1 shows that a regular stereoscopic renderer would takes 207ms to render this scene in stereo. We will use the output of this regular renderer as error-free reference images, to compare the quality of the output produced by image-warping.

We implemented three different image-warping algorithms for our architecture: point splat, quad splat and mesh-based reconstruction. In general, we found the perceived quality of point splats and quad splats to be roughly equal, with quad splats showing slightly better quality in smooth shaded regions. The quality of the mesh-

based approach appeared to be better than either point or quad splats, although it resulted in slightly too much image blurring. The additional blur may have given a false sense of better image quality, which is not equal to the reference. This was illustrated in Figure 5. For each of these reconstruction methods, the server's image-warping frame rates are given in milliseconds per frame in Table 1 for various resolutions, both for single- and multi-GPU implementations. In all cases stereoscopic image-warping using two client views is performed. Note, to achieve a 60Hz frame rate, image-warping must be performed in less than 16.6ms. From these data we can deduce that mesh-based reconstruction is too slow to be practically used currently, as is quad-based splatting for higher resolutions. The performance of point splatting is higher than the other reconstruction methods, while resulting in almost the same image quality. Therefore, we use point splatting exclusively throughout this paper; although, in the future, the other techniques may become feasible. Using point splatting we can guarantee a 60Hz frame rate for stereoscopic resolutions up to 1024x768, both for single- and multi-GPU implementations. Performance was found to be linear in the number of warped pixels.

Every second column in Table 1 gives the time between consecutive client frame updates that arrive at the server. While there is no strict performance threshold here (such as 16.6ms for the server), the longer it takes for new client frames to arrive as a source for image warping, the larger the image warping errors become. For the multi-GPU implementation these updates times are stable and are equal to the rendering time (207ms) plus the time taken to perform asynchronous, background data transfers. However, for the single-GPU implementation the update times vary greatly. This is due to the sharing of a single GPU for both warping and client rendering and the implied scheduling and context switching. Since only a short time is available for client rendering, the relative impact of the overhead of the context switch becomes larger. It can be seen that the larger the server's warping load becomes, and the more closely it fills the allotted 16.6ms per frame, the less time available to render new client frames. Therefore, the ratio between server and client frames is increased, resulting in an increase of warping errors.

6.2 Image quality

For the evaluation of the image quality produced by our architecture, we use a sample scene consisting of the 28M polygon Stanford Lucy angel model, orbited by twelve torus knots. All objects rotate about their own Y-axis at 60 deg/s, where each torus knot rotates in opposite direction. The torus knots rotate about the angel at 20 deg/s. The camera is tilted upwards by ten degrees. A sample frame of this animation is shown in Figure 7. We artificially fixed the client's frame rate to 6Hz, resulting in a 10:1 server:client frame ratio, i.e. ten frames are warped for every client frame update. This is possible because we use off-line rendering and can run the client and server in a special synchronized mode where both wait for each other according to a given fixed ratio. In all cases point splatting was used as the image-warping algorithm, and two client views were warped to the left-eye image of a stereoscopic pair.

We distinguish between two types of errors: occlusion errors and sampling errors. The former are caused by occluded geometry and are the most perceptually disturbing kinds of errors, while the latter are caused by small errors in shading, splat-size calculations and overlapping splats. In practice, sampling errors are not very noticeable, unless the scene is still and observed very closely. To evaluate image quality, we compare warped images to directly rendered reference images. Furthermore, we compare the warped depth to a reference depth map to detect the difference between occlusion and sampling errors. When the depth of a warped pixel lies further away than a set threshold from the depth of the corresponding reference pixel, it is marked as a hole pixel. Note, when a warped

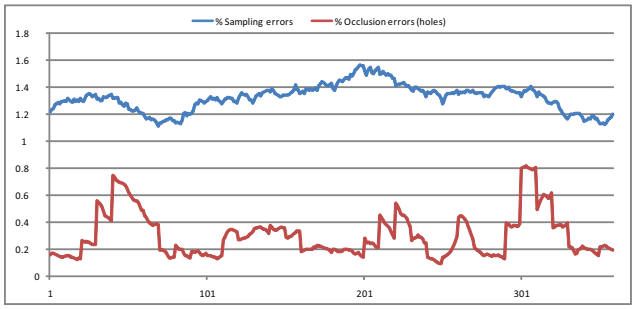

Figure 8: The percentage of pixels that constitute occlusion or sampling errors for each frame of a 360 frame animation of the 28M polygon scene. Figure 7 depicts frame number 42, which exhibits a near-maximum amount of occlusion errors in the sequence. This frame will be used as a sample frame throughout this paper.

pixel is closer to the viewer than the depth of the reference, it can not be a hole pixel and so it is marked as a sampling error. This kind of sampling errors occurs frequently at depth edges, where the splat-size is too large and covers a few pixels behind it. To detect sampling errors, we choose to convert the both images to the Lab perceptual color space and compare them per-pixel. If the distance in Lab space between two corresponding pixels is larger than a threshold value of 20 units, the pixel is marked as sampling error. This threshold is used to avoid marking small differences that are hard to perceive as errors.

An example frame where the detected occlusion errors are shown in red and the sampling errors in blue is shown in Figure 7. Quantitative data is plotted in Figure 8. The chosen sample frame #42, and this complete 360 frame scene animation, will be used for the rest of this results section. Note, this is a particularly bad frame that shows near-maximum occlusion errors.

6.2.1 Hole detection

To evaluate the hole-finding methods described in Section 4.2.1, we compare the pixels that were identified by these methods as holes to the map of reference holes. This reference map is constructed in the way described in Section 6.2, using a reference and warped depth map. Figure 9 shows the hole maps found for frame #42. The left image depicts the reference holes in a warped image where no hole filling is applied. To better illustrate the results, we did not redraw the background quad here, so holes in the background are also visible. The middle image depicts the holes found by a level-0 hole finding method (see Section 4.2.1), which has been a popular method of detecting holes in the past [9]. This method only finds holes where no pixels have been warped to. Since any pixel that is not reached by a fragment is necessarily a hole, the method never finds false-positives. However, the method fails to find any level-1 type of hole, which occur frequently in the animation. The right image depicts the holes identified by the level-1 hole finding method. Correctly identified holes are shown in green, where dark green indicates the subset of pixels that is classified as a background hole. Since we re-draw the background quad anyway, it is not required to ray trace these hole pixels. A few background holes remain that are not classified as such. False-positives, i.e. non-hole pixels identified as holes, are depicted in yellow and occur most frequently at the edges of objects, as expected.

The percentage of holes correctly identified by the two methods is plotted in Figure 10 for the 360 animation frames. Here we compare the percentages of correctly identified holes that are not background holes. The level-1 method is capable of finding most of the holes in the animation, except for a small number of level-2

Figure 7: (left) A reference image produced in approximately 207ms by a regular stereoscopic render for a 28M polygon scene. Only the left-eye view is shown, as will be the case throughout this paper. (center) The same scene produced by image-warping in only 14ms. (right) Errors in the warped image. Occlusion errors, or holes, are depicted in red and sampling errors in blue.

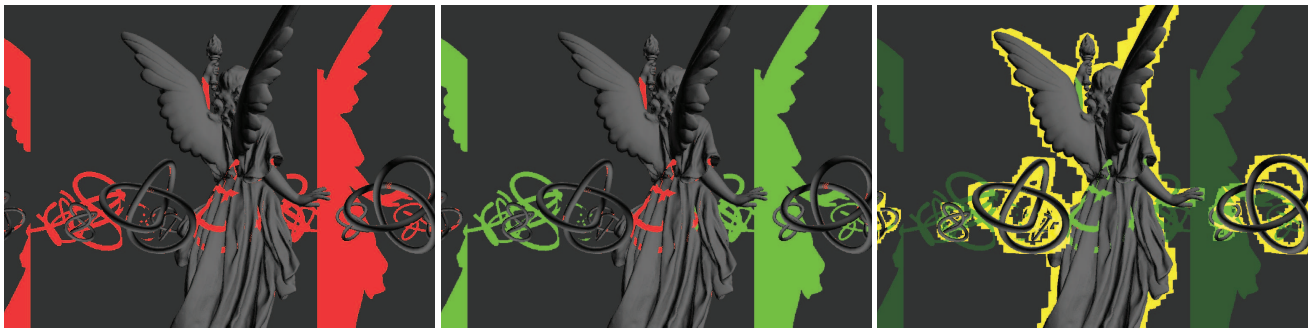

Figure 9: (left) Reference holes in red for a warped image where no hole filling is applied, including background holes. (center) Holes correctly detected by a level-0 hole finding algorithm are shown in bright green, while holes that were not found are shown in red. (right) Holes found by our level-1 hole finding algorithm. Green indicates a hole is correctly found, where the dark green sub set represents pixels correctly identified as background holes. False-positives are depicted in yellow. Holes that were not found are depicted in red; however, this occurred for only a few individual pixels in the shown frame.

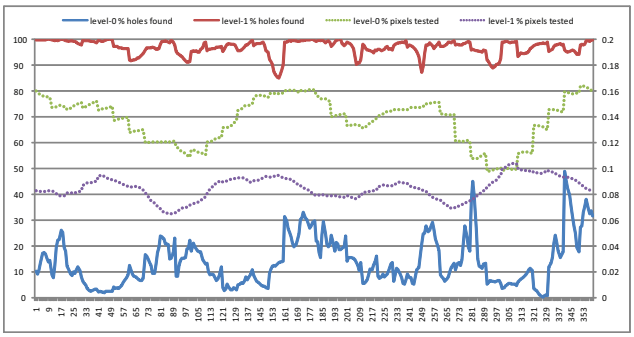

Figure 10: (left axis) Percentage of holes found correctly for the level-0 (blue) and level-1 (red) hole finding methods, excluding background holes. The level-0 method fails to find many holes, while the level-1 method finds most of the holes. (right axis) The percentage of pixels identified as candidate non-background holes. While the level-1 method (purple dotted) finds more holes, it actually scans fewer pixels than the level-0 method (green dotted).

holes. The level-0 method performs poorly in identifying holes. Although the level-1 method finds more holes and a number of false-positives, it actually scans fewer pixels than the level-0 method due to the background-hole optimization. This is because the level-0 method has no way to distinguish between background holes and other types of holes, so it has to scan all of them. We conclude that for our purposes the level-1 method shows superior performance in identifying holes, and we use it as the algorithm of choice.

6.2.2 Hole filling

Given a map of candidate hole pixels, we can re-render the first number of leaf nodes that are hit in back-to-front order along the ray, as was described in Section 4.2.2. We have evaluated the number of holes that can be resolved correctly in this way and the number of polygons that need to be re-rendered. For the hole map we used the output of our level-1 hole finding method, which was described in Section 4.2.1. Figure 11 shows the sample frame #42 with hole filling applied for the first, the first three and all intersected leaf nodes. The larger the number of intersected leaf nodes, the larger the number of polygons that need to be re-rendered. This is plotted in Figure 12. The percentages of hole pixels that were correctly resolved are plotted in Figure 13. It can be seen that re-rendering the first three leaf node intersection along the ray results in a reasonable trade-off between the number of resolved holes and the number of re-rendered polygons; most of the holes are resolved, while re-rendering only a small percentage of the original 28M polygons. For illustration, we also plotted the number of resolved holes using a level-0 hole finding method in Figure 13. This shows the importance of good hole finding, as it forms a baseline for

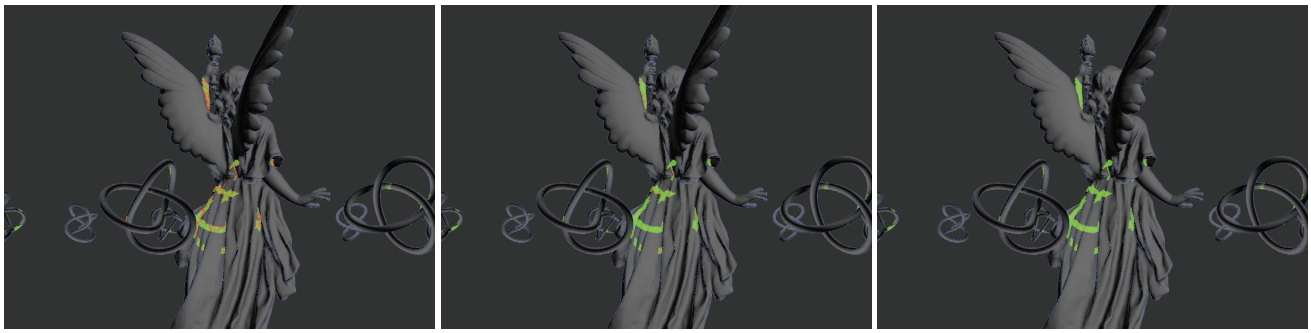

Figure 11: Example of hole-filling. Green pixels depict correctly filled holes, while red pixels depict holes that were not filled. Blue represents remaining sampling errors. (left) Resolving holes by re-rendering the first leaf node intersection; some holes remain. (center) Re-rendering the first three intersection resolves most holes (right) Re-rendering all intersection resolves all hole pixels that were correctly identified.

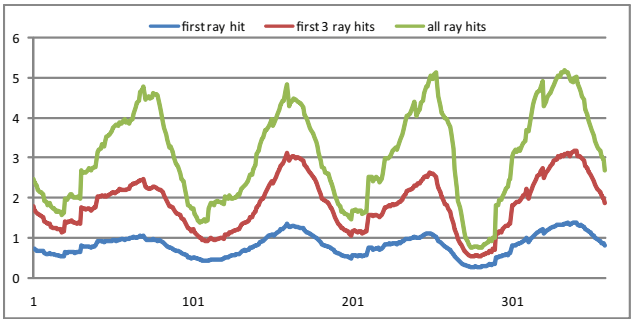

Figure 12: Millions of polygons re-rendered by the three hole-filling strategies for each frame of the animation. the original scene consists of 28M polygons. We found that the first three leaf node intersections provide a good trade-off between achieved quality and the number of polygons re-rendered.

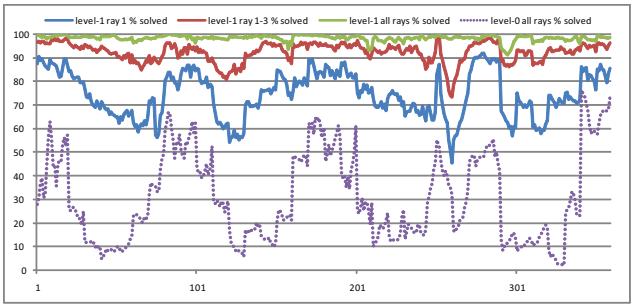

Figure 13: Percentage of holes resolved for the three hole-filling strategies for each frame of the animation. Many pixels identified as holes are not correctly filled when only re-rendering the first leaf node intersection found. However, re-rendering the first three intersections resolves most of the holes. When re-rendering all the intersections, every identified hole is resolved and only the holes that were not found are left. In comparison, the dotted line shows resolved holes when using a level-0 hole finding method that re-renders all intersections.

the number of reference holes than can be resolved: holes that are not found can not be resolved, no matter how many leaf node intersections we re-render. Finally, these results show that it is possible to create very high quality images using image-warping techniques and good hole filling, depending on the available time.

The average frame times on a multi-GPU system for stereoscopic warping using these hole-fillings methods are approximately 65ms, 78ms and 92ms, for the re-rendering of the first hit leaf, the first three hits, and all hits, respectively. There are a number of reasons for this lower performance. The kd-tree is generated using a simple mid-point split, which is known to result in sub-optimal ray tracing performance. A tree build using a surface area heuristic would increase ray tracing performance significantly. Furthermore, ray tracing is done sequentially on just a single CPU core, and for each pixel independently. Using multiple cores or a GPU ray tracer, in combination with amortized or packet ray tracing, would further improve performance (a good overview of ray tracing techniques is given by Wald et al. [21]). One final optimization could be to start ray tracing at the depth found in the occlusion map. Due to the convex hull nature of the rendered bounding boxes, no intersections can possibly occur in front of this depth. One issue that remains is the amount of geometry that needs to be re-rendered. If this amount is too high, we won't be able to achieve a 60Hz frame rate without falling back to level-of-detail methods for the re-rendered geometry, which may be a viable alternative. Another option would be to implement a completely GPU-based ray tracer that can quickly re-render individual pixels, avoiding the need to re-render entire leaf nodes. However, while we can not currently maintain a 60Hz frame rate when using the described hole-filling methods, we still produce an almost-error-free stereoscopic image about three times faster than a regular stereoscopic renderer would (70ms versus 200ms, see Table 1). This result by itself may justify the use of image-warping techniques in some situations.

6.3 Latency

We evaluated the latency for both single- and multi-GPU image-warping implementations and a stand-alone regular renderer for reference using a method introduced by Steed [17]. The architecture was implemented on an Intel Q6600 2.4 Ghz quad-core processor system using an Nvidia GeForce 8800 GTX for the client GPU and a stereo-enabled Nvidia Quadro FX5600 for the server GPU. In the case of a single-GPU, both client and server used the NVidia Quadro FX5600. The stereoscopic display consisted of an iiyama Vision Master Pro 512 22 inch CRT monitor operating at 120Hz in order to achieve 60Hz per eye. For input we used a Polhemus Fastrak 6 DOF input device that generates tracking reports at 120Hz. Both the image-warping server and the reference system rendered the scene as normal for the left and right eye and then cleared the

	Reference			Multi-GPU			Single-GPU		
Faces	**lat.**	**σ**	**time**	**lat.**	**σ**	**time**	**lat.**	**σ**	**time**
64M	x	x	577	58.9	2.6	14.1	93.8	4.0	27.6
32M	780	14.2	289	61.1	2.5	14.3	95.6	4.8	27.6
16M	409	12.3	145	62.2	2.4	14.6	94.4	4.6	27.4
8M	220	8.4	72.9	58.6	2.8	15.4	96.2	2.3	27.4
4M	113	5.3	37.0	65.4	5.4	16.9	95.0	3.1	27.1
2M	68.5	0.5	18.8	74.0	3.5	19.8	89.9	3.5	27.3
1M	48.5	3.5	9.6	71.9	5.1	19.7	95.0	4.4	27.1

Table 2: Latency, standard deviation of latency and time required to warp, all in milliseconds, for a reference renderer and both our single- and multi-gpu image-warping implementations. In all cases a stereoscopic image is produced from two client views and no hole filling is performed. The reference latency for 64M faces could not be measured because it was larger than half the period of the pendulum used for measuring, introducing aliasing frequencies. (Intel Core2 Quad Q6600 2.4Ghz + Nvidia GeForce 8800GTX + Nvidia Quadro FX5600)

display in order to render a small sphere at the position of the input device for latency measurements. The reference system sampled the input device just before rendering the scene, while the server of our architecture re-sampled the input device prior to warping using the previously described algorithms. The scene consisted of various amounts of polygons.

The acquired results are listed in Table 2. This shows the average latency acquired over several sampling runs of the experiment, as well as the standard deviation over those samples. It can be seen that the latency for the image-warping server is low and almost constant, regardless of the rendering load. The single-GPU's latency is somewhat higher due to longer warping times, but still constant. Both low and constant latency are desirable properties for an interactive system and enable the use of further predictive filtering methods [13]. The reference renderer's latency is much higher and depends on number of polygons in the scene and the frame rate. Furthermore, the standard deviation of the reference is also much higher, indicating non-constant latency that would be hard to predict. When the number of polygons is reduced to about 4M and lower, the multi-GPU latency increases slightly. This is because client frames are updated almost every frame now, and the frequent asynchronous data transfers and context switches between the warp and upload thread start affecting warping performance. This has no effect on a single-GPU because no data is transferred. For small scenes it is probably better to use direct rendering; however, the data shows that when more than two million polygons are to be rendered, image-warping methods can reduce the latency.

6.4 Iso-surface rendering

We have implemented both iso-surface warping approaches that were described in Section 5 on a single GPU system and have measured the performance when rendering four different volumes: the 256x256x512 Carp, 492x492x442 Present, 512x499x512 Christmas tree and the 832x832x494 stag beetle volumes. All of these are 8-bit volumes. Figure 14 shows sample renderings of these test volumes at different iso-values.

For each volume and setup, we recorded drawing times for the image-warping client and server and a reference implementation that does a direct iso surface ray-casting for an animation of 20 seconds. The test system consisted of an Intel Core-i7 CPU 2.93GHz, 12GB RAM, and an NVidia Quadro FX 5800 with 4GB VRAM. Table 3 shows the performance results for our first approach compared to direct iso-surface rendering using the iso-values from Figure 14 for a monoscopic resolution of 1024x768. Table 4 shows the performance results for our second approach compared to direct iso-surface rendering using the iso-values from Figure 14 for a monoscopic resolution of 1024x768. Finally, Figure 15 gives a close-up of the better reconstruction during the post warping pass

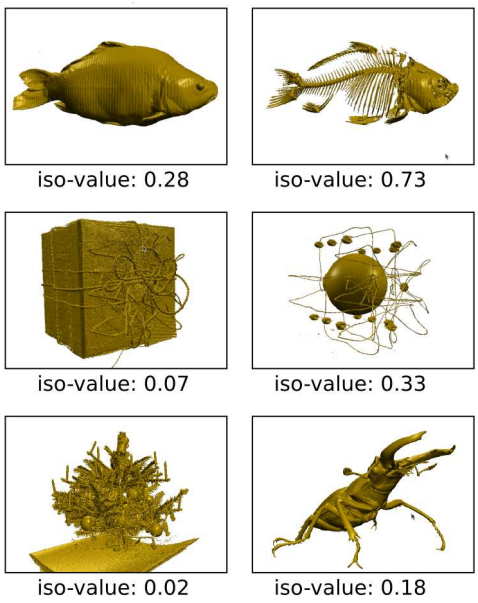

Figure 14: Direct renderings of the test volumes at different iso-values.

volume	**iso-surface warping**				**direct**	
	dt	σ	c. dt	σ	dt	σ
carp 0.28	**16.2**	0.9	51.3	9.0	**33.1**	1.2
carp 0.73	**16.2**	0.9	56.5	9.8	**35.3**	5.8
pres.0.07	**27.3**	7.7	181.0	47.8	**48.7**	15.7
pres.0.33	**32.8**	1.1	204.2	26.6	**76.5**	24.0
ctre.0.02	**16.3**	1.6	192.6	13.6	**74.7**	22.5
beet.0.18	**17.9**	4.7	170.5	45.3	**115.7**	38.5

Table 3: Drawtimes for iso-surface warping in a Single-GPU implementation compared to direct iso-surface ray-casting for the different test runs of our first approach at a resolution of 1024x768: server (dt) and client (c. dt) drawtimes along with standard deviations (σ), all timings are in milliseconds. (Intel Core-i7 2.93GHz + NVidia Quadro FX 5800)

compared to our first approach.

These results show that iso-surface rendering can benefit from our warping-architecture analogous to polygonal rendering. While we cannot guarantee a fixed frame rate of 60Hz, as the polygonal warping-architecture can, iso-surface warping can still provide us with lower latency. Compared to direct ray-casting, our first method can accelerate iso-surface rendering by a factor of two to six, depending on iso-value and volume size. Deferred binary search and gradient computation causes additional load on the server, but still gives better frame rates than our reference implementation. The benefit of this method is an improved image quality compared to direct iso-surface warping.

7 Discussion

An interesting question is for which scenes our image-warping architecture is useful. For scenes containing many more than 10 million triangles and a display resolution of only one million pixels, we run into a sampling problem and thus aliasing occurs. Each pixel receives contributions from many triangles, which should be integrated above the pixel area. To a certain degree this can be achieved by over-sampling and anti-aliasing techniques, but for very large scenes this is not possible. What is needed are output-sensitive and display-resolution-sensitive techniques, such as occlusion culling

volume	iso-surface warping				direct	
	dt	σ	c. dt	σ	dt	σ
carp 0.28	**16.2**	0.7	88.7	18.5	**33.1**	1.2
carp 0.73	**16.2**	1.4	117.8	16.5	**35.3**	5.8
pres.0.07	**27.3**	7.7	181.0	47.8	**48.7**	15.7
pres.0.33	**32.8**	1.1	204.1	26.6	**76.5**	24.0
ctre.0.02	**33.0**	1.7	438.1	50.9	**74.7**	22.5
beet.0.18	**32.9**	4.2	197.8	49.2	**115.7**	38.5

Table 4: Drawtimes for iso-surface warping in a single-GPU implementation compared to direct iso-surface ray-casting for the different test runs of our second approach at a resolution of 1024x768: server (dt) and client (c. dt) drawtimes along with standard deviations (σ), All timings are in milliseconds. (Intel Core-i7 2.93GHz + NVidia Quadro FX 5800)

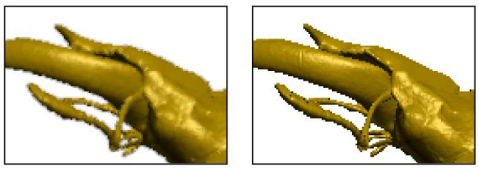

Figure 15: (left) Close-up of the stag beetle. Reconstruction using point-based warping in our first approach. (right) Deferred binary search and gradient computation.

and level-of-detail approaches, which generate triangle sets matching the display resolution. Future graphics hardware is likely to be able to render such intelligently decimated scenes at 60Hz. However, the ever increasing shading quality, as well as the computationally expensive dynamic level-of-detail and occlusion culling techniques, may limit the frame rate below 60Hz for some time. Thus, our architecture remains beneficial in all cases where a constant high frame rate and low latency cannot be guaranteed otherwise.

There are a number of trade-offs between image quality and the architecture's end-to-end latency. Different choices for the described algorithms may result in better perceived quality at the expense of longer rendering times and, consequently, higher latency. One example is the amount of sampling errors after warping. Three ways to render warped pixels were discussed: screen-aligned point splats, quadrilateral splats and and a mesh-based reconstruction method. Point splats have the highest performance but result in increased sampling artifacts, whereas mesh-based reconstruction generally resulted in the best image quality but was significantly slower. Which sampling algorithm to use depends on the exact application requirements; however, we believe that the increase in performance for point splats outweigh the loss of image quality compared to the other two methods.

Another trade-off exists for occlusion errors. Objects that are visible from the server's viewpoint but not from any client viewpoint will result in an occlusion error in the warped image. Since hole-filling algorithms can be expensive, the server should strive to generate as few holes as possible to begin with. Therefore, in order to avoid clipping objects that enter the camera space, the client view should have a larger field-of-view than the server camera. However, increasing the FOV results in smaller rendered object sizes and fewer warped pixels per object. This effect is almost equivalent to reducing the client resolution and reduces the overall quality of warping. Another good way to reduce occlusion errors is to warp multiple client views. This can be achieved by placing the client cameras in such a way that a large part of the scene is visible from different directions. However, this could result in the client views to be far away from the server view, resulting in large required warping distances and, consequently, larger errors. Using multiple client views also implies a direct performance reduction due to an increase of pixels to be warped, and an increase in shared pixel data. Furthermore, for multiple client views to be effective, intelligent camera placement strategies are required [16] that come with different performance characteristics.

Different algorithms require different amounts of pixel data to be shared between the client and server. For example, the layered depth images technique [14] provides a way to reduce occlusion errors, but requires much additional per-pixel data. Depending on the architecture's hardware, this may or may not be a bottleneck. A single-GPU implementation will not be affected much, since no data is transferred. For the multi-GPU case we have used asynchronous data transfers to reduce the impact of transfers; however, the transfers are still not entirely free as was seen in Table 2. Furthermore, accessing large amounts of data may introduce extra performance penalties in the form of cache misses, depending on the access pattern. Therefore, before increasing the per-pixel data requirements, these issues should be considered carefully.

Another trade-off occurs for the presented method of detecting and resolving occlusion artifacts. Resolving a larger number of occlusion errors correctly also required a larger number of polygons to be redrawn, in turn reducing performance. In this way, image quality can be increased depending on the available processing time. Another factor is the size of the leaf nodes in the kd-tree. Larger leaf nodes contain more geometry and require more polygons to be redrawn; however, smaller leaf nodes require more expensive ray tracing. A similar trade-off exists for hole detection; with bounding boxes at a higher resolution in the kd-tree, fewer false positives are detected, and fewer rays need to be traced, but more bounding boxes need to be rendered to construct the reference map. These problems may be partially remedied by using adaptive, dynamic resolutions for the leaf nodes and bounding boxes, but this has not been investigated.

One aspect that we have not discussed is the method used to compare image quality between the reference application and our image-warping architecture. We chose to do a simple frame-by-frame comparison using 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) [2]. 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. Furthermore, we are dealing with animations and not still images; therefore, a different perceptual model, for example one including temporal frequency sensitivities, should be used. Occlusion errors are perceived to be more disturbing than sampling errors because they cause rapid flicker in animations; a fact that can not be detected by classic image-comparison methods. Finding a good comparison method for animation sequences that can pin-point disturbing errors would help a great deal in increasing the image quality of real-time warping systems.

The end-to-end latency of our system is around 60ms, which is already quite low. We have not yet experimented with using prediction on the tracker data that is fed into the scene graph. Prediction based on a Kalman [5] filter could be very effective in creating a system with almost no perceivable latency, which might be a good trade-off for some image quality in many application domains.

8 Conclusion

We have shown a real-time image warping architecture for dynamic scenes that can be used to guarantee a 60Hz stereoscopic display rate up to a certain display resolution on current hardware. In this way, latency was significantly reduced at the expense of some reduction of image quality. We have quantified certain types of errors with respect to image quality, and discussed trade-offs be-

tween an increase of image quality and a reduction of latency. A novel method of detecting and resolving hole artifacts was also presented. Furthermore, the use of image-warping in combination with iso-surface rendering was investigated. An implementation of the architecture was described for both a single- and a multi-GPU system; the most desirable hardware design for our approach would be a multi-GPU architecture with shared memory, which currently does not exist. Although we did not perform a formal study, the preliminary impressions of users of the architecture — where various large, static models could be interactively examined in a world-in-hand fashion using 6-DOF input devices — were positive, especially with respect to the low latency and the improved interactivity. We believe that our real-time image-warping architecture will be an excellent choice for all kinds of interactive graphics applications dealing with large scenes.

REFERENCES

[1] P. J. Bex, G. K. Edgar, and A. T. Smith. Multiple images appear when motion energy detection fails. *Journal of Experimental Psychology: Human Perception and Performance*, 21:231–238, 1995.

[2] S. Daly. The visible differences predictor: an algorithm for the assessment of image fidelity. In *Digital images and human vision*, pages 179–206. MIT Press, 1993.

[3] S. R. Ellis, M. J. Young, B. D. Adelstein, and S. M. Ehrlich. Discrimination of changes in latency during voluntary hand movement of virtual objects. In *Human Factors and Ergonomics Society*, 1999.

[4] T. Hübner, Y. Zhang, and R. Pajarola. Multi-view point splatting. In *Proc. ACM GRAPHITE*, pages 285–294, 2006.

[5] R. E. Kalman. A new approach to linear filtering and prediction problems. *ASME Journal of Basic Engineering Vol. 82*, pages 35–45, 1960.

[6] R. Kijima and T. Ojika. Reflex hmd to compensate lag and correction of derivative deformation. In *Proc. IEEE VR*, page 172, 2002.

[7] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, 1987.

[8] T. K. Magnus, T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *In: Proceedings of IEEE Visualization 05*, pages 223–230. IEEE, 2005.

[9] W. R. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*. PhD thesis, University of North Carolina at Chapel Hill, 1999.

[10] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3d warping. In *Symposium on Interactive 3D Graphics*, pages 7–16, 180, 1997.

[11] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics*, 29:39–46, 1995.

[12] M. R. Mine. Characterization of end-to-end delays in head-mounted display systems. Technical report, 1993.

[13] M. Olano, J. Cohen, M. Mine, and G. Bishop. Combatting rendering latency. In *Proc. ACM SI3D*, pages 19–ff., 1995.

[14] J. W. Shade, S. J. Gortler, L.-W. He, and R. Szeliski. Layered depth images. *Computer Graphics*, 32:231–242, 1998.

[15] F. A. Smit, R. van Liere, S. Beck, and B. Froehlich. An image warping architecture for vr: Low latency versus image quality. In *Proc. IEEE Virtual Reality (VR)*, pages 27–34, 2009.

[16] F. A. Smit, R. van Liere, and B. Froehlich. A programmable display layer for virtual reality system architectures. In *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2009 [in press].

[17] A. Steed. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proc. ACM VRST*, pages 123–129, 2008.

[18] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. *International Workshop on Volume Graphics*, 0:187–241, 2005.

[19] J. Stewart, E. P. Bennett, and L. McMillan. Pixelview: a view-independent graphics rendering architecture. In *Proc. ACM HWWS*, pages 75–84, 2004.

[20] J. Torborg and J. T. Kajiya. Talisman: commodity realtime 3D graphics for the PC. In *Proc. ACM SIGGRAPH*, pages 353–363, 1996.

[21] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *STAR Proc. of Eurographics 2007*, pages 89–116, 2007.