# An Image-Warping Architecture for VR: Low Latency versus Image Quality

F.A. Smit[*] CWI, Amsterdam
R. van Liere[†] CWI, Amsterdam
S. Beck[‡] Bauhaus-Universität Weimar
B. 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. 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 [9]. 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.

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.

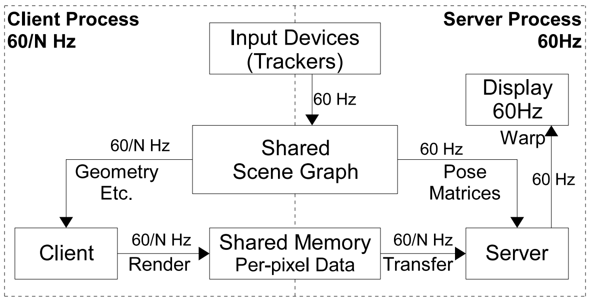

Figure 1: Overview of the proposed real-time image-warping architecture. A client and server process run in parallel, where the client generates new application frames at a fraction of 60Hz (60/N) and the server produces new display frames at 60Hz. The scene graph is shared to allow the server access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

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 holes 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. Furthermore, transferring image buffers from the client to the server can be time consuming, thereby limiting the amount of data that can be used.

This paper has a number of important contributions. 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. Our single-GPU implementation runs client and server on the same GPU and relies on the scheduling of the OpenGL server implementation. By partitioning the geometry data in small chunks, we achieved that this implementation also runs at a nearly constant 60Hz for monoscopic images. An important advantage is that per-pixel data is now shared on the single GPU and need not be transferred.

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. Naviga-

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

tion and object manipulation particularly benefit from our architecture. In addition, object selection and most system control tasks are executed at the display refresh rate due to the direct influence of the latest scene graph state on the warped images.

## 2 RELATED WORK

Image-based rendering by 3D warping was introduced by McMillan and Bishop [8]. Layered depth images (LDI) [11] 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 [3]. A good overview of many different warping algorithms is given by Mark [6]. Other approaches have been explored in order to reduce latency for interactive VR-systems. A very common method is to use predictive Kalman filtering [4] on the received tracker reports prior to the rendering of a new frame. Further examples are the PixelView architecture proposed by Stewart et al. [15], the Reflex HMD by Kijima et al. [5], the SLATS system by Olano et al. [10] and the Talisman architecture by Microsoft [16]. 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 resolutions 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 in [13].

In the past years we have implemented an alternative multi-GPU image-warping architecture [12, 13]. In addition to per-pixel color and depth information, the client generates a per-pixel 3D motion field. Using this information, the server performs depth-image warping to produce updated views of the scene. The motion field is used to linearly extrapolate the motion of the warped pixels. Image-warping is performed using simple point-cloud rendering. This architecture suffered from a number of drawbacks. A performance bottleneck was the amount of data (12 bytes per pixel) that needed to be transferred from client to server. Furthermore, due to the simplified point-cloud rendering, image quality was negatively affected by warping artifacts. Finally, since only per-pixel color, depth and motion information was available on the server, the latest input device poses for objects could not be used, which significantly increased interaction latency. The architecture proposed in this paper differs in a number of important ways:

- As was shown in Figure 1, the client and server share the scene graph. Instead of generating and transmitting a motion field, the client assigns object IDs to all pixels. The server uses these object IDs to fetch the corresponding object matrix transforms in the scene graph and compute object motion. An important benefit of this method is that it allows for object matrices to be re-sampled from the tracking devices at the server side, allowing for latency reduction at the object level. An additional advantage is that motion extrapolation for non-tracked objects can now be performed using general 4x4 matrices instead of 3D vectors, allowing for orientation as well as position extrapolation. This results in more accurate motion predictions for these objects. Finally, scene-graph updates that do not require geometry information directly, such as object selection, can be performed directly at 60Hz by the server, without waiting for the next client frame.

  A practical benefit of sharing the scene graph is that it reduces the amount of data required to be transmitted to the server: by only transmitting per-pixel color, depth and object IDs, the per-pixel data size is reduced from 12 to 7 bytes. This results in a 70% performance gain on data transfers.

- The client now generates two views to be warped instead of only one, resulting in significantly less occlusion artifacts during image warping (see Figure 2). In addition, various camera placement strategies can be used on the client side.

- The image-warping methods have been improved with higher quality algorithms, using dynamic splat sizes and hole-filling methods. This reduces the amount of visual artifacts and improves image quality. Although these algorithms could also have been implemented on the previous architecture, the server would not achieve the required frame rate of 60Hz. However, due to the performance gain on data transfers, the server can now use higher quality algorithms.

- In addition to a multi-GPU implementation, we show a multi-threaded single-GPU implementation of the architecture. The single-GPU implementation has the benefit that no data transfers are required between server and client.

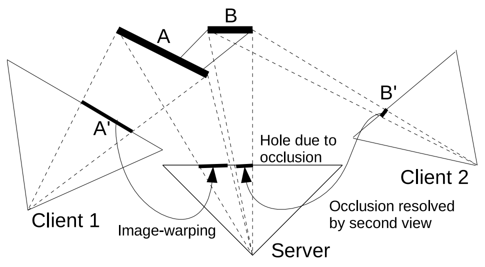

Figure 2: Using a second client view in order to reduce occlusion artifacts in the form of holes. Object A and B are visible in the server's projection; however, object B is occluded by object A in the first client view's projection, resulting in a hole. By adding a second client view the occlusion issue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

In summary, the proposed architecture provides more accurate motion extrapolation and update flexibility due to a shared scene graph, lower latency due to late input device sampling, and significantly better image quality due to multiple views and more advanced image-warping techniques.

## 3 ARCHITECTURE

The server performs image-warping to generate a transformed and re-projected server view using images generated by the client. Such a reprojection has to be generated for every different server view. For stereoscopic rendering, two warped server views need to be generated for each client image. The implementation is free to choose the number of images generated by the client, 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 [6]. In the case of two client views, both are warped to the same server view where the depth buffer handles overlapping pixels. This means that for stereoscopic rendering, the server needs to perform image-warping four times: twice for each of the left- and right-eye views.

### 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

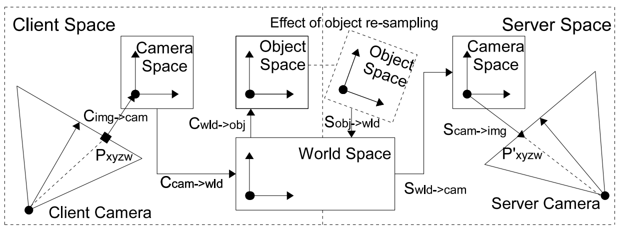

Figure 3: Schematic overview of warping a client pixel $P_{xyzw}$ to the server pixel $P'_{xyzw}$. Re-sampling of object poses is essentially achieved through the server's object-to-world transform. All six required matrices are concatenated into a single warping matrix for each different object.

this purpose a *client frame*. A mechanism is required to synchronize the creation of client frames and the reading of them by the server. For this purpose we use a synchronized producer-consumer buffer. The client starts by requesting a free write slot in the buffer and blocks waiting until it acquired one.

To generate the required data, the client renders the scene from multiple viewpoints. Our implementation currently uses two viewpoints per client frame. At the start of a client frame, 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 $C_{img\to cam}$ and $C_{cam\to wld}$, 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 $C^{i}_{wld\to obj}$. The scene is rendered using a custom shader program that outputs per-pixel color, 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. Post-projection depth information is 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 required storage space of seven bytes per pixel.

## 3.2 Server

The server starts by polling the producer-consumer buffer in a non-blocking fashion to determine if a new client frame is available. If so, the corresponding data is transferred from the client; if not, the server simply 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 $\Delta t$ 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 $S_{wld\to cam}$ 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:

$$S_{cam\to img}\cdot S_{wld\to cam}\cdot S^{i}_{obj\to wld}\cdot C^{i}_{wld\to obj}\cdot C_{cam\to wld}\cdot C_{img\to cam}$$

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 matrices $M^i$ 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 by a single 4x4 matrix multiply, where the required matrix is pre-computed per-object instead of being computed for each pixel. Furthermore, 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.

So far we have not yet mentioned the significance of the server's object matrices $S^{i}_{obj\to wld}$, 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 $\Delta t$ 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 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 in the client's texture data. Other per-vertex information that is available in 2D client textures is the post-projection depth $P_z\in(0,1)$, the object ID $i$ 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 expansion factor of this transform 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. This final form is equivalent to the warping equations used by McMillan [8]; 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.

We have implemented three different warping algorithms that all 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. The point-splat method computes the Jacobian matrix as described earlier 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 [7] 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.

Finally, a post-processing hole-filling step is applied to the resulting warped image. We set a flag for every destination pixel that is reached by a warped fragment; every pixel missing this flag is considered a hole and should be filled. Several hole-filling techniques are available, but many do not scale well to parallel GPU execution. Therefore, we used a simple scan along the epipolar camera direction as described by Mark [6]; however, we scan in both opposite directions and choose the pixel that is furthest away. This strategy is needed for dynamic scenes where objects as well as the camera can move, and is somewhat similar to the mesh-based hole filling strategy proposed earlier by Mark et al. [7, 6]. Hole-filling issues will be discussed further in Section 6.

## 3.4 Implementation

We will now describe implementation of the architecture on a a multi-processing dual-GPU system and a multi-threaded single-GPU system. For the multi-GPU implementation, the client and server each run in a separate process using an independent GPU and each acquires a separate OpenGL context. The client and server are executed in parallel on a multi-core CPU. The producer-consumer buffer is implemented using interprocess shared memory. Texture data is copied from the client GPU into the provided slots in shared memory. The server polls the producer-consumer buffer for a new slot and subsequently uploads the client data to the server GPU when available.

The single-GPU implementation approach runs both client and server in different threads where they share the same GPU. The two OpenGL contexts for the different threads are also shared, such that OpenGL resources can be accessed from either thread. The CPU code for client and server is again executed in parallel. In this case, the slots of the producer-consumer buffer consist of several shared texture IDs instead of shared memory regions. The client reads a texture ID from an acquired slot and directly renders into that texture. The server polls the buffer as before, and directly maps the read texture ID to the GPU for reading. In this way, the implementation simply cycles between a number of synchronized texture IDs controlled by the producer-consumer buffer and never copies texture data from the GPU into system shared memory. Special care has to be taken to allow proper scheduling between threads. Since a single GPU is shared between two threads, the OpenGL driver and the internal GPU scheduling is responsible for allotting time slices on the GPU to the client and server threads. In practice, once a rendering command has been issued to the GPU, such as a large geometry display list, this command will be completed before any other command is processed. This means that, in order to allow the server rendering time for warping, the client's rendering must be split into smaller chunks. This is illustrated in Table 1.

| **Chunk Size** | **0.5K** | **1K** | **5K** | **25K** | **30K** | **50K** |
|---|---|---|---|---|---|---|
| **Average (ms)** | 15.7 | 15.8 | 15.8 | 15.6 | 16.1 | 24.7 |
| **St.Dev.** | 1.2 | 0.9 | 1.1 | 1.6 | 2.3 | 7.4 |
| **Max (ms)** | 26 | 20 | 21 | 22 | 32 | 36 |

Table 1: The effect of different chunk-sizes in triangles on the client for a single-GPU warping implementation. The client renders geometry in small chunks to allow the server in-between GPU time. A chunk size of 1K seems to be the best choice, while for a size of 30K or higher the server does not get sufficient processing time according to the standard deviation. It can also be seen from the maximum time that the server can not always guarantee 60Hz, so for a small number of frames it drops back to 30Hz even for good chunk sizes.

.

## 4 Results

This section gives an overview of the acquired results using our architecture for the frame rate, image quality, observed latency and single-GPU performance. The multi-GPU 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. The stereoscopic display consists of an iiyama Vision Master Pro 512 22 inch CRT monitor operating at 120Hz in order to achieve 60Hz per eye. The single-GPU version was realized on the same system using only the Nvidia GeForce 8800 GTX. A number of geometric models are used in this section: a 17M polygon model of a CT-scanned coral structure, the 13M polygon UNC powerplant model and the 10M polygon Thai Statue model.

Frame Rate  As described in Section 3.3, 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 is illustrated in Figure 4. The server

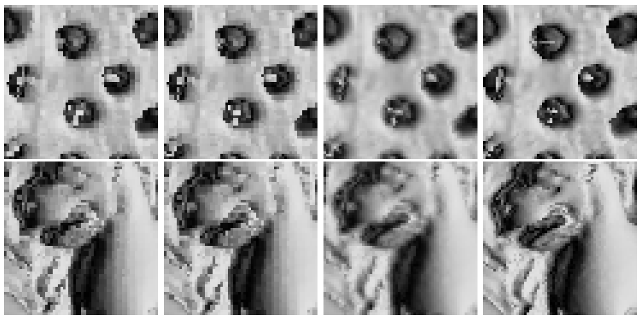

Figure 4: A quality comparison of a 60x60 pixel close-up from the 17M polygon coral model (top) and the 10M polygon Thai statue model (bottom). From left to right: point-splat, quad-splat and mesh-based image warping methods and the direct rendered reference. Some noise appears for the splat-based methods 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.

| | 640x480 | 800x600 | 1024x768 | 1280x960 |
|---|---|---|---|---|
| **Point Splat** | 217 / 162 | 142 / 105 | 87 / 65 | 55 / 41 |
| **Quad Splat** | 72 / 64 | 49 / 44 | 30 / 27 | 19 / 17 |
| **Mesh-based** | 51 / 48 | 33 / 31 | 20 / 19 | 13 / 12 |

Table 2: Multi-GPU server frame rate at various resolutions for three warping algorithms. The server performs image warping four times, due to the use of two client views and stereoscopic rendering. The first number refers to frames without data transfers, while the second is for frames with data transfers.

frame rate of the three algorithms for a stereoscopic multi-GPU implementation is given in Table 2 for various resolutions. Two types of frames should be distinguished: those where no new client frame is available and the server can directly warp the previous frame, and those where a new client frame is available and data needs to be transferred. The performance for both types of frames is given.

Since our architecture is strictly constrained by a run-time performance of 60Hz, we are primarily interested in the slowest frames that include data transfers. For a resolution of 1024x768 and two client views, the amount of pixel data per client frame equals 10.5MB. The current implementation, using the aforementioned hardware setup, is capable of uploading data from shared memory to the server GPU at approximately 2.5 GB/s. Therefore, the server will spend 4.2ms on uploading pixel data to the GPU. Additionally, for each object in the scene, an object specific 4x4 warping matrix needs to be uploaded; however, this cost is usually negligible. For example, the space requirement for 8192 floating point 4x4 matrices is only 0.5MB, corresponding to an additional 0.2ms of transfer time. Some additional time is spent on constructing the warping matrices on the CPU, which requires three matrix multiplies and a matrix extrapolation per object (see Section 3.2). However, the extrapolation step is not required during frames that require data transfers because the matrix can be used directly, and additional time is available in frames without data transfers. Two additional optimizations are possible but were not implemented. First, the matrices could possibly be stored using 16-bit floating point, effectively reducing the data size by 50%. Second, since image warping is performed on the GPU, and the CPU is mostly idle, extrapolation computations can be performed in parallel on the CPU during the warping of the previous frame.

For stereoscopic displays and two client viewpoints, the server needs to perform four image-warpings at 60Hz. This means that the warping of a single view must be done in less than approximately 2.5ms (400 FPS), considering the amount of time required for data transfers and matrix computations. Table 2 gives an overview of the final frame rates achieved for various resolutions and warping algorithms, including data transfers. From this we can deduce that mesh-based reconstruction is too slow to be practically used currently, as is quad-based splatting for higher resolutions. Therefore, for the remainder of this section, we use point splatting only; 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.

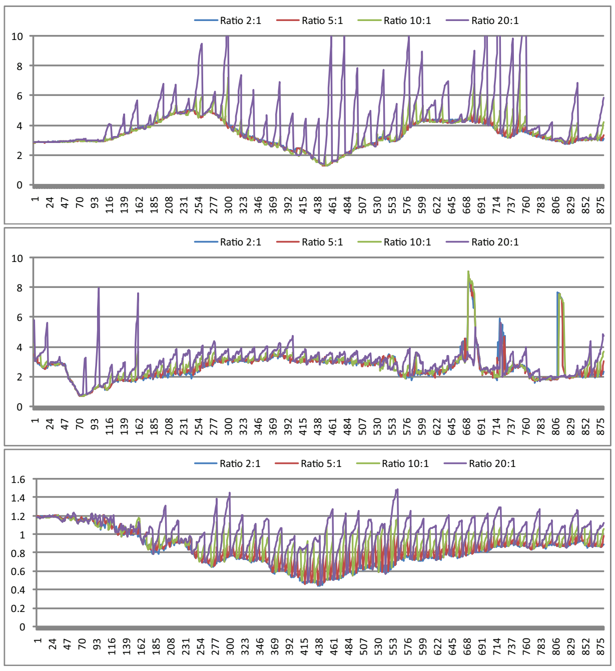

Figure 6: Percentage of error pixels between a directly rendered reference frame and a warped frame for various ratios between server and client frames. From top-to-bottom the results are shown for the three sample applications: the world-in-hand coral scene, the powerplant walk-through scene and the dynamic particle scene.

| | **Applications** | | | | | |
|---|---|---|---|---|---|---|
| | World-in-hand | | Walk-through | | Particle | |
| **Ratio** | Err.% | Rel.% | Err.% | Rel.% | Err.% | Rel.% |
| **2:1** | 3.37 | x | 2.60 | x | 0.81 | x |
| **5:1** | 3.39 | 0.63 | 2.62 | 0.49 | 0.86 | 5.42 |
| **10:1** | 3.54 | 5.00 | 2.73 | 4.91 | 0.93 | 13.9 |
| **20:1** | 4.41 | 30.8 | 2.91 | 11.6 | 1.02 | 25.3 |

Table 3: Average percentage of error pixels between reference and warped frames for the three sample applications for different server:client ratios, corresponding to Figure 6. The second column for each application shows the relative increase in error compared to the 2:1 ratio in percentages.

Image Quality For the evaluation of our architecture under various conditions, we use three sample applications for different usage scenarios and scenes: a world-in-hand scenario using the 17M polygon coral model, where the model is attached to a 6 DOF polhemus FastTrak pen-device; a camera walk-through of the powerplant model; and a particle simulation with many moving objects. In all cases the camera can either be controlled by the mouse, or by a Logitech acoustic head-tracker. Animations of 900 frames at 60Hz (15 seconds) were recorded for typical interaction sessions

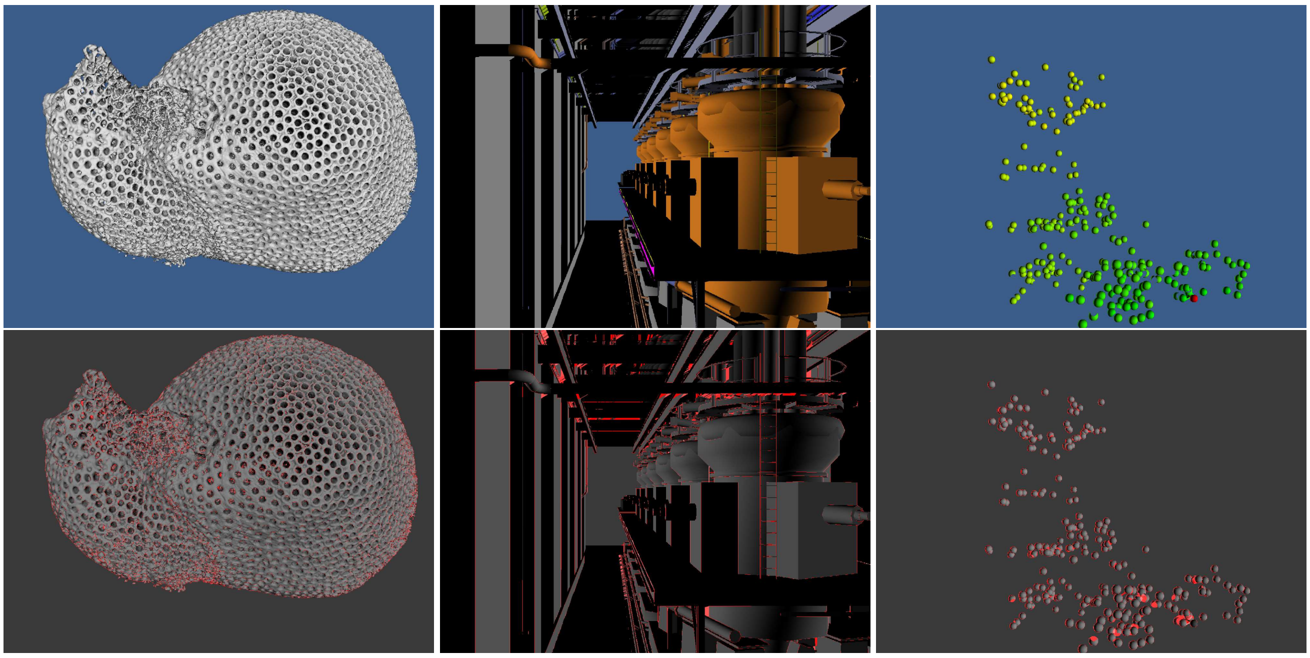

Figure 5: (top) From left to right warped server images are shown for three sample scenes: the world-in-hand coral scene, the walk-through powerplant scene, and the dynamic particle scene. The first scene is an example of a user inspecting a very large, complex model bound to an input device; the second, a head-tracked walk-through in a large environment with high-frequency geometry data; the third, a highly dynamic scene with many moving objects. (bottom) Corresponding error pixels in red compared to a directly rendered reference frame. Errors most notably appear at the edges of objects and for holes in the warping due to occlusion.

using these applications. For all three scenes a sample warped animation frame is shown in Figure 5 (top). All tracker reports for the input devices occurred at 60Hz and were recorded to an animation file in order to replay and capture off-line. For each of the 900 frames of the animation, we capture four images: a reference image obtained through regular stand-alone off-line rendering, a corresponding warped image obtained from the server using our architecture, and the last two client views received by the server. The two client view images show which images were warped to acquire the warped frame. This procedure was repeated four times using simulated server to client frame ratios of 2:1, 5:1, 10:1 and 20:1; that is, we artificially fixed the client's frame rate to 30Hz, 12Hz, 6Hz and 3Hz. 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.

Given a reference image and a warped image, we can compare them to evaluate warping quality. It is a non-trivial matter to compare the perceptual difference between two images (this is discussed further in Section 5). We choose to convert 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 an error pixel. This threshold is used to avoid marking small differences that are hard to perceive as errors. Figure 5 (bottom) shows a sample error comparison output for three different animation frames, where the found error pixels are marked in red. Plots of the percentage of error pixels per frame for the three animations are given in Figure 6. The average percentages are given in Table 3. Many small errors result at the edges of objects causing a constant base-line error. This is a result of the splatting algorithm that has difficulties in determining the depth buffer gradients at these points, resulting in erroneous splat sizes. Other errors are mostly caused by wrongly filled holes caused by occlusion. The plots also clearly show that after receiving a client frame the amount of error increases for a number of frames until a new client frame is received and the error is reduced again. This behavior is directly related to the server:client frame ratio and is especially noticeable for the world-in-hand and particle scenes because these consist largely of moving objects. Client viewpoints are currently set according to camera poses, so when objects change pose there is a greater possibility for occlusion artifacts during warping. The world-in-hand scene is mostly translated or still during the first 100 frames; therefore, most occlusion artifacts appear only when the object is rotated at a later stage. The same is true for the particle scene where object velocity increases over time.

| | **Reference Application** | | | **Image-warping Server** | | |
|---|---|---|---|---|---|---|
| **N** | Latency | SD | FPS | Latency | SD | FPS |
| **1** | 374.4 | 15.7 | 6 | 57.4 | 3.0 | 87/65 |
| **2** | 190.4 | 9.8 | 11 | 58.4 | 5.4 | 87/65 |
| **4** | 101.0 | 7.9 | 20 | 54.3 | 2.6 | 87/65 |
| **8** | 73.1 | 5.2 | 30 | 56.2 | 3.6 | 87/65 |
| **16** | 45.3 | 1.6 | 40 | 53.6 | 2.8 | 87/65 |
| **32** | 46.1 | 1.1 | 50 | 50.8 | 2.6 | 87/65 |

Table 4: Latency in ms, standard deviation of latency and frame rate for a stand-alone stereoscopic reference application and the server of our multi-GPU stereoscopic image-warping architecture. The rendered scene consists of the 17M polygon coral model, where the actual rendered number of polygons is divided by the value of N in the left-most column. The image-warping server is able to maintain low and constant latency. Server frame rates correspond to Table 2.

**Latency** Next, we evaluated the latency for both our image-warping architectures and a stand-alone regular renderer for reference using a method introduced by [14]. The scene that was used consisted of the 17M polygon coral model, attached to the 6 DOF

|  |  | Server |  | Reference |  |
|---|---|---|---|---|---|
| **Polygons** | **Ratio** | Avg (ms) | SD | Avg. (ms) | SD |
| **1.5M** | 1:1 | 15.7 | 2.3 | 16.0 | 0.3 |
| **3.0M** | 2:1 | 15.8 | 2.8 | 32.6 | 0.6 |
| **5.6M** | 3.8:1 | 15.9 | 1.9 | 49.0 | 0.3 |
| **8.9M** | 10:1 | 16.0 | 1.1 | 133 | 0.4 |
| **11.6M** | 19:1 | 17.1 | 10.2 | 226 | 8.3 |

Table 5: Comparison between a single-GPU warping architecture and a stand-alone reference implementation. Both render monoscopic images at 800x600 resolution. At a threshold of about 11M polygons, the server can no longer maintain 60Hz and a large standard deviation is introduced in the frame rate. However, the server can maintain a 60Hz frame rate for a larger amount of polygons than the reference renderer.

input device and rendered at 1024x768 in stereo. To measure the latency under different amounts of rendering load, we also rendered integers fractions 2, 4, 8, 16 and 32 of the total amount of polygons in the model. Both the image-warping server and the reference system rendered the scene as normal for the left and right eye and then cleared the 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 acquired results are listed in Table 4. This shows the average latency acquired over 15 runs of the experiment, as well as the standard deviation over those runs. It can be seen that the latency for the image-warping server is low and almost constant, regardless of the rendering load. The reference renderer's latency is much higher and depends on the frame rate. Also, the standard deviations indicate that the server's latency is of a more constant nature. Both low and constant latency are desirable properties for an interactive system and enable the use of further predictive filtering methods [10].

Multi-threaded single GPU The performance characteristics of the single-GPU implementation are somewhat different because the client and server share the same GPU; therefore, in this particular case, server performance is not independent of the rendered scene and depends on client load as well. Therefore, we have to describe the performance of a single-GPU implementation for specific numbers of polygons rendered. For a single-GPU we render at 800x600 monoscopic resolution due to the fact that only half the processing capability is available. A comparison between a reference renderer and the single-GPU warping architecture is shown in Table 5. It can be seen that in practice the system can maintain a frame rate of 60Hz for a load of approximately 11M polygons. Once this load is exceeded the server occasionally drops back to 30Hz, introducing an increased standard deviation in rendering time. Note, there is no such constraint on the number of polygons in the case of a multi-GPU implementation.

## 5 Trade-offs and Limitations

An important trade-off is that between image quality and camera placements. 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 aspect is that of occlusion; an object that is visible from the server's viewpoint but not from either client viewpoint will result in a hole in the rendering. Since hole-filling algorithms can only partially remedy this problem, the server should strive to generate as few holes as possible. 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, by doing this the client views will be far away from the server views, resulting in large required warping distances. In addition, if a client camera is placed inside an object, or very close to it, the image it renders is virtually useless. This aspect is very important in walk-through scenes, and collision detection should be used to avoid these degenerate camera placements. To overcome many of these issues, we placed the client cameras relatively close to the server camera, with slightly larger FOV and eye-separation. Finally, most previous camera placement strategies [6] were concerned with static scenes only. For dynamic scenes, where objects as well as the camera can move, there is not always a good client camera position available. This can be the case if multiple objects move in opposite directions: in that case there is no simple, universal camera placement strategy that can guarantee that all objects will be entirely visible for all frames.

There are also trade-offs between image quality and performance of the algorithms described in section 3. Different image-warping methods may result in better perceived quality at the expense of longer rendering times. For example, 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. Also, rendering more than two client views may help to avoid holes. However, since we desire to guarantee a 60Hz stereoscopic server frame rate, many of these algorithms are too slow in practice. One could decide that a frame rate of half the display rate (30Hz) is also acceptable and consequently use better quality warping algorithms. We do not advise this approach, as it introduces dynamic artifacts in the form of judder.

There are a number of interesting trade-offs between a single- and multi-GPU implementation of the architecture. The benefit of the single-GPU approach is that client data is shared on the GPU and does not need to be transferred to the server. This allows us to almost freely store any amount of data required for warping. An immediate advantage is that higher precision data can be stored, for example 32-bit float depth values instead of 16-bit fixed point. Another example is that of deferred shading: the single-GPU client can additionally store per-pixel normals that can be used by the server to perform shading. In this way, we can not only warp viewpoints and object positions at the server at 60Hz, but also dynamic and view dependent lighting. For a multi-GPU approach this technique would cause a degradation of performance due to extra data transfers. Another example of extra data on a single-GPU is the use of completely different image warping techniques, such as layered depth-images [11]. This may allow for much better handling of occlusion artifacts and holes resulting from image-warping. A less obvious drawback for a single-GPU is that because we only have half the processing capability available, the client's rendering will be slower, even if the server maintains 60Hz. Therefore, the ratio between server and client frames is increased, resulting in an increase of warping errors.

A limitation of the proposed architecture is that, due to the nature of the currently used image-warping algorithms, scene transparency can not be handled correctly. 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 [11]; however, this is highly inefficient for applications such as volume rendering using many transparent slices. Another class of applications that our architecture can not 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 motion and structural changes of the deforming surface. Since image-warping is essentially a one-to-one mapping of pixels – even though larger splats may be comprised of several pixels – topologically changing surfaces require special attention. It may be possible to assign per-pixel motion vectors corresponding to the motion of the deformable surface; however this does not immediately solve the problem of changing topology and does not handle real-time interactions with the surface correctly. Another minor problem occurs for objects that newly appear in the scene: since no previous object matrices are available, the server is unable to perform extrapolation on the poses of these objects; however, the re-sampling of an attached input devices is still possible. As of yet, how to best handle volume rendering and deformable objects in an image-warping architecture is still an open problem.

## 6 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 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.

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. Methods such as VDP often excel for images that show an amount of random noise that is not perceived as disturbing: VDP will usually not report these pixel difference because they are hard to perceive. However, in our case we have digitally acquired images that are essentially noise-free. Furthermore, in this case we are dealing with animations and not still images; therefore, a different perceptual model, for example one including temporal frequency sensitivities, should be used. Since accurately predicting perceived differences is so hard, we chose to use a simple and straight-forward method that can be easily understood. One problem with the used comparison method is that because of the splatting most object edges are found to be errors, even though these errors are not very noticeable in practice. Finding a good comparison method for animation sequences that can pin-point disturbing errors would help a great deal in the further development of real-time warping systems.

The end-to-end latency of our system is around 50ms, 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 [4] 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.

## 7 Conclusion

We have shown a real-time image warping architecture 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. The architecture works for dynamic scenes and it was illustrated by three sample applications typical to VR. We presented a single- and a multi-GPU system implementation, and extensively discussed trade-offs and limitations that we became aware off during the development of the architecture. The most desirable hardware design for our approach would be a multi-GPU architecture with shared memory, which should not be difficult to build, since each GPU is already a multi-processor. Although we did not perform a formal study, users of the architecture were very 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 – at least in the near future.

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

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

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

[6] 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.

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

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

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

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

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

[12] F. A. Smit, R. van Liere, and B. Fröhlich. The design and implementation of a VR-architecture for smooth motion. In *Proc. ACM VRST*, pages 153–156, 2007.

[13] F. A. Smit, R. van Liere, and B. Fröhlich. An image warping VR-architecture: Design, implementation and applications. In *Proc. ACM VRST*, pages 115–122, 2008.

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

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

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