The Asynchronous Time Warp for Virtual Reality on Consumer Hardware

J.M.P. van Waveren *
Engineering Manager, Oculus VR, LLC.

Abstract

To help create a true sense of presence in a virtual reality experience, a so called "time warp" may be used. This time warp does not only correct for the optical aberration of the lenses used in a virtual reality headset, it also transforms the stereoscopic images based on the very latest head tracking information to significantly reduce the motion-to-photon delay (or end-to-end latency). The time warp operates as close as possible to the display refresh, retrieves updated head tracking information and transforms a stereoscopic pair of images from representing a view at the time it was rendered, to representing the correct view at the time it is displayed. When run asynchronously to the stereoscopic rendering, the time warp can be used to increase the perceived frame rate and to smooth out inconsistent frame rates. Asynchronous operation can also improve the overall graphics hardware utilization by not requiring the stereoscopic rendering to be synchronized with the display refresh cycle. However, on today's consumer hardware it is challenging to implement a high quality time warp that is fast, has predictable latency and throughput, and runs asynchronously. This paper discusses the various challenges and the different trade-offs that need to be considered when implementing an asynchronous time warp on consumer hardware.

Keywords: virtual reality, latency, image warping

Concepts: • Computing methodologies → Virtual reality • Computing methodologies → Computer graphics • Computing methodologies → Concurrent computing methodologies

1 Introduction

A typical virtual reality headset uses lenses that are positioned over the eyes to bring a display in focus with a large field-of-view (FOV) stereoscopic perspective. Due to various trade-offs between size, weight, material, and quality, the lenses typically introduce optical distortion, usually in the form of a pincushion effect. A barrel distortion is applied when mapping the rendered images to the display to correct for the pincushion distortion of the lenses.

High frame rates and low latency are required to create a true sense of presence in an interactive virtual reality experience. To achieve this, a so called "time warp" may be used. This time warp does not only warp the rendered images for the optical aberration of the lenses in the virtual reality headset, it also transforms the images based on the very latest head tracking information to significantly reduce the motion-to-photon delay (or end-to-end latency).

*e-mail: janpaul.vanwaveren@oculus.com

Copyright is held by the owner/author(s).
VRST '16, November 02-04, 2016, Garching bei München, Germany
ISBN: 978-1-4503-4491-3/16/11.
DOI: http://dx.doi.org/10.1145/2993369.2993375

The time warp operates as close as possible to the display refresh, retrieves updated head tracking information, and transforms a stereoscopic pair of images from representing a view at the time it was rendered, to representing the correct view at the time it is displayed.

If the time warp is a fast operation with predictable latency and throughput, then the time warp can significantly reduce the motion-to-photon delay, even when rendering of the stereoscopic images takes a considerable amount of time or has significant latency. If the time warp runs asynchronously to the stereoscopic rendering, then the time warp can be used to increase the perceived frame rate and to smooth out inconsistent frame rates. Asynchronous operation can also improve the overall graphics hardware utilization by not requiring the stereoscopic rendering to be synchronized with the display refresh cycle.

While the time warp has the potential to significantly improve a virtual reality experience, implementing a high quality time warp that is fast, has a predictable latency and throughput, and runs asynchronously, comes with its own set of challenges. One of these challenges is transforming a stereoscopic pair of images from representing a view at the time it was rendered, to representing the correct view at the time it is displayed. This is particularly challenging considering the time warp needs to use very little time to be effective. On some systems it is challenging to achieve the necessary performance to warp the stereoscopic images onto a high resolution display at high frame rates. On other systems it is challenging to run the time warp asynchronously.

2 Previous Work

A typical head mounted display (HMD) employs an optical system using lenses to bring the display in focus with a wide field of view. Due to various trade-offs between size, weight, material and quality, the lenses typically introduce some form of optical distortion. Warping the stereoscopic images onto the display has turned out to be an effective technique to correct for optical aberration [Warren and Rolland 1991, Watson and Hodges 1995, Bauer et al. 2012, Pohl et al. 2013].

While it is desirable to reduce latency as much as possible, this has proven to be difficult. Therefore, a variety of research has focused on compensating for the latency that is inherent to a system. A common approach is to predict head movements so the stereoscopic images can be rendered from the position the head is likely going to be in the near future [List 1984, Azuma and Bishop 1995, Himberg et al. 2012]. However, even with sophisticated prediction algorithms, artifacts occur as head movements are initiated and changed.

Latency can also be reduced by using updated head tracking information to transform a stereoscopic pair of images from representing a view at the time it was rendered, to representing the correct view at the time it is displayed. Some research in this area uses simple hardware techniques to reduce latency for changes in orientation [Griffin and So 1991, Griffin and So 1992, Mazuryk and Gervautz 1995, So 1997, Yasuyuki et al. 1998, Kijima et al. 2001, Kijima and Yamada 2002, Jerald 2004, Jerald et al. 2007].

Other research uses environment mapping with compositing to fully reduce orientation latency, and partially reduce positional latency [Regan and Pose 1992, Regan and Pose 1993, Regan and Pose 1994, Regan and Pose 1998, Pose 1996, Pose 2000]. More recent research uses transformations in image space to reduce latency in changes to both orientation and position [Castella 1993, Mine and Bishop 1993, McMillan and Bishop 1995, Olano et al. 1995, McMillan 1995, Mark et al. 1995, Mark et al. 1996, Mark et al. 1997, Mark and Bishop 1998, Mark 1999, Peek et al. 2013, Peek et al. 2014]. Other research also uses motion fields to reduce the latency of object motion [Smith et al. 2007, Smith et al. 2008, Smith et al. 2009, Smith 2009, Smith et al. 2010]. Most of these implementations don't cover the complexities of running the algorithms asynchronously on consumer available hardware.

3 Time Warp

The time warp fundamentally does two things:

- Corrects for optical distortion
- Reduces the motion-to-photon delay

The time warp lends itself to a variety of implementation options that can be categorized as follows:

- Asynchronous Operation
- Degrees of Freedom
- Quality
- Performance

3.1 Correcting for optical distortion

The lenses in a virtual reality headset typically introduce some form of optical distortion, which usually results in a pincushion effect. To cancel this pincushion effect, the stereoscopic images need to be warped onto the display with the corresponding barrel distortion. The optical aberration of a round lens and its correction can be described using a polynomial with a small number of terms. The following is an example of such a polynomial:

$$d_i = d_d \cdot (k_0 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_3 \cdot r^6)$$

This polynomial describes the barrel distortion as a displacement from the optical axis of symmetry, where d_i is the displacement on the rendered image and d_d is the displacement on the display with a distance r from the optical axis of symmetry and lens specific constants $k_{0\text{-}3}$ that define the optical aberration.

The lenses in a virtual reality headset typically also introduce some form of chromatic aberration. Chromatic aberration is not discrete and occurs across the full light spectrum where every wavelength of light is refracted differently. However, for displays that use the RGB color space, the chromatic aberration can be significantly reduced by separately warping the red, green, and blue color components of the stereoscopic images.

$$d_{i\text{-}rgb} = [\ d_i \cdot (c_0 + c_1 \cdot r^2),\ d_i,\ d_i \cdot (c_2 + c_3 \cdot r^2)\ ]$$

This results in three separate displacements, one for each color component, where $c_{0\text{-}3}$ are lens specific constants.

3.2 Reducing the motion-to-photon delay

The motion-to-photon delay is the total time between movement of the user's head and the display emitting photons for the updated stereoscopic images that reflect that movement. This delay, also known as end-to-end latency [Mine and Bishop 1993], usually includes sampling of the head tracking sensors, combining the sensor information (sensor fusion), rendering of the stereoscopic images, the display reading from the frame buffer, up to the display emitting photons. Ideally, the total delay is no more than a couple of milliseconds, but it should be 20 milliseconds or less for compelling virtual reality experiences [Zhang, Luo 2012].

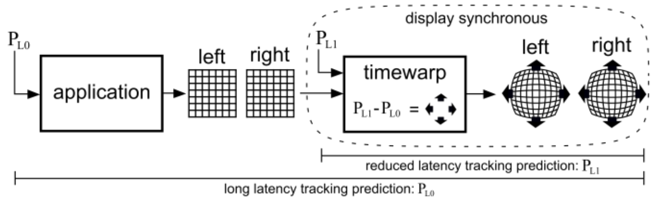
display synchronous
P_L0
application
left
right
P_L1
timewarp
P_L1-P_L0 =
left
right
reduced latency tracking prediction: P_L1
long latency tracking prediction: P_L0

Figure 1: *Rendering pipeline with time warp.*

The time warp significantly reduces the motion-to-photon delay by operating as close as possible to the display refresh, retrieving updated head tracking information, and transforming a stereoscopic pair of images from representing a view at the time it was rendered, to representing the correct view at the time it is displayed (Figure 1). This does, however, not necessarily remove all latency from the system.

Most modern displays introduce a significant latency between reading from the frame buffer and emitting photons. Even if this latency is minimal, most displays do not update or refresh the whole screen instantaneously, but instead, present an image incrementally as it is scanned out from memory. For instance, a 60 Hz display may change the bottom of the screen more than 16 milliseconds later than the top of the screen. This is rarely a problem on a static display, but on a head-mounted display it can cause the whole world to appear to stretch, compress, or shear as the head rotates because the stereoscopic images are generated for an instant in time, while different parts are presented at different times [Mine and Bishop 1993]. Compensating for the incremental display refresh is equivalent to preventing a variable motion-to-photon delay across the display. To achieve a significantly reduced and constant motion-to-photon delay across the display, the time warp needs to continuously retrieve updated head tracking information to transform the stereoscopic images just ahead of the display refresh. For instance, the time warp can retrieve updated head tracking information and select the to-be-displayed elements of the stereoscopic images for every pixel or scanline, right before the display emits their corresponding photons. This is referred to as "beam racing" [Mine and Bishop 1993].

On many systems, however, it may not be feasible to run the time warp just ahead of the display refresh, or it is not feasible to continuously retrieve updated head tracking information. In this case, prediction can be used to determine which elements of the stereoscopic images need to be displayed at every pixel. Based on the most recent head tracking information, the position and orientation can be predicted for the time the display refresh starts, and the time the display refresh ends. Interpolation can then be used to derive the position and orientation of the head for each pixel.

It is interesting to note that compensating for the incremental display refresh using the latest head tracking information does not account for eye movements. Just like head movement, eye movement can cause the world to appear to stretch, compress or shear. Accurate and low latency eye tracking is necessary to fully compensate for the incremental display refresh. However, in practice, the artifacts from eye movements are not noticeable, in particular while the practical field of view of most virtual reality headsets is limited. The user becomes unaware of the visual information when the eyes are in flight during saccades, and to see artifacts when the eyes are in a smooth pursuit, an object would have to move at supersonic speeds at most distances.

3.3 Asynchronous Operation

The time warp could simply run in a serial fashion after rendering of the stereoscopic images has completed, however, it is desirable to run the time warp asynchronously. If the time warp runs asynchronously to the stereoscopic rendering, then it can be used to increase the perceived frame rate and to smooth out inconsistent frame rates. As long as the time warp stays synchronized with the display refresh, it can maintain a smooth view into the virtual world by appropriately warping the last completed stereoscopic images using the latest head tracking information.

Asynchronous operation can also improve the overall graphics hardware utilization by not requiring the stereoscopic rendering to be synchronized with the display refresh. To account for variable frame rendering times, most rendering engines leave a significant amount of graphics hardware execution time unused, just to make sure the rendering completes in time for the next display refresh. When rendering of the stereoscopic images is decoupled from the display refresh cycle, the rendering engine can attempt to fully utilize the graphics hardware because the asynchronous time warp will continue to present a smooth view into the virtual world even when the rendering engine is unable to deliver new stereoscopic images in time for the next display refresh.

3.4 Degrees of Freedom

In its simplest form, the time warp only rotates the rendered images based on an updated head orientation. This does not change the perspective within the rendered images based on positional changes. Note that even if the position of the head does not change significantly, the orientation of the head changes the position of the eyes. The lack of adjustment for positional changes may result in positional judder, which is typically perceived as jerking or doubling of image elements. This is most noticeable for nearby objects and surfaces with a large parallax. Next to changes in view point, objects within the scene may also move or animate. The lack of adjustment for object motion may result in object judder.

If depth information is available, then the perspective within the rendered images can be changed as well. However, to correctly change the perspective in scenes with transparency, these scenes need to be rendered as a set of separate layers, each with its own depth information. Significant positional changes may also introduce artifacts at geometric silhouette edges where internal parallax disoccludes surfaces that are not visible in the original rendering. The situation is even more complicated in the presence of view dependent lighting effects such as specular reflections. Even if additional lighting information is available, it is non-trivial to recalculate view dependent lighting effects within a scene. The lack of adjustment of view dependent lighting effects in a scene may present itself as judder of specular highlights.

Fortunately, changing the perspective within the rendered images and recalculating view dependent lighting effects, is not absolutely necessary to implement compelling virtual reality experiences. Changes in orientation result in the largest registration error [Jerald 2004]. Changes in position produce much smaller registration error, in particular for large environments where objects are distant. Even for close objects, registration error due to motion parallax is relatively small for typical user movements [Jerald 2004]. In other words, simply applying small rotations to the stereoscopic images based on changes in orientation works quite well in practice.

The images rendered for each eye, however, may not provide enough data for new head orientations. To provide additional image data, some research renders the six sides of a cube map to provide images that completely encapsulate the user's head [Regan and Pose 1992, Regan and Pose 1993, Regan and Pose 1994, Regan and Pose 1998, Pose 1996, Pose 2000]. However, this makes the rendering process take up to six times longer compared to rendering images for a 90 degree FOV. Alternatively, the stereoscopic images can be rendered with a slightly larger FOV [Jerald et al. 2007]. The FOV may also be dynamically adjusted based on angular velocity, and a predicted head orientation can be used to generate a more useful range of image data. However, if there is a sudden, significant change in orientation, there may still not be enough image data for the time warp. Fortunately, it is minimally distracting to temporarily pull in black on the sides without enough image data [Griffin and So 1991, Griffin and So 1992].

3.5 Quality

For the best quality, the stereoscopic images are rendered at a high resolution with high quality anti-aliasing. The time warp then samples these images in linear space using a high quality filter such as an anisotropic or an elliptical weighted average (EWA) filter. However, while high quality linear space texture sampling improves the quality, the time warp is still effective when using bilinear texture sampling of gamma corrected images.

3.5 Performance

For the time warp to be effective, it needs to complete very quickly. The faster the time warp, the better it can reduce the motion-to-photon delay. Ideally, the time warp takes no more than a couple of milliseconds to support very high framerates. To be able to schedule the time warp so it completes just in time for the display refresh, it needs a predictable latency and throughput which may be hard to achieve on today’s power and thermal constrained processors.

4 Time Warp Transform

For the best performance, the time warp implementation used here only rotates the rendered images based on the latest head orientation. The 2D input coordinates to the time warp transform are the tangent-angles of the rays from the eye through the pixels on the display, typically in the range [-1, 1] for a 90 degree FOV. To correct for optical aberration, a barrel distortion is applied to these coordinates which may push the coordinates into the [-2, 2] range. The time warp transformation is then applied to these distorted coordinates, and transforms them into [0, 1] image space such that they can be used directly to sample the stereoscopic images.

The time warp transform is represented as a 4x4 homogeneous transformation matrix. To transform the 2D tangent-angle input coordinates, they are interpreted as 3D homogeneous coordinates with Z = -1 and W = 1. To compensate for the incremental display refresh, two transformation matrices are used, one for when the display refresh starts, and one for when the display refresh ends. Interpolation is used to get the correct transformation at each pixel along the display refresh. If the latest head tracking information is retrieved right before the display refresh starts, then the head orientation needs to be predicted only one display refresh ahead.

5 CPU Time Warp

To allow fully asynchronous operation to all graphics rendering on a Graphics Processing Unit (GPU), the time warp can be implemented to run on one or more cores of a Central Processing Unit (CPU). Unlike a GPU, a CPU does not have specialized hardware for texture mapping with texture sampling.

If the display is 1920x1080 pixels, then the time warp needs to texture map over 2 million pixels. When using a single core of a typical CPU that runs at 2 GHz with one clock-cycle-per-instruction (CPI), every instruction executed per pixel adds about one millisecond of execution time to the time warp. Considering the time warp cannot use much more than a dozen milliseconds per frame to update a 60 Hz display, no more than about a dozen instructions can be executed per pixel. At higher refresh rates or higher resolutions, the time warp needs to use even fewer instructions per pixel.

Things are further complicated on mobile CPUs which, at run-time, may be forced to run at lower frequencies for thermal reasons. This can make it hard to achieve a predictable throughput which is needed to appropriately schedule the time warp as close as possible to the display refresh. The problem can be mitigated by using multiple cores that are clocked at a lower frequency to reduce the power draw, but aggressive optimization is still necessary to stay within thermal boundaries.

5.1 CPU Memory Access

The stereoscopic images are typically rendered on the GPU. The CPU will need access to these images in order to warp them onto the display. The images could be copied to CPU accessible memory, but ideally the CPU has direct access to these images and the front buffer without the need for a copy. If direct memory access is available, then both the stereoscopic images and the front buffer may need to be configured to use a linear layout instead of a tiled format to avoid time consuming and hardware dependent tiling operations on the CPU.

If the stereoscopic images live on cacheable memory then there is typically less than 10% overhead from cache misses. Memory prefetching can be used to reduce the cache misses, but even though the memory access patterns are predictable, the prefetch distance changes with the barrel distortion and it is not trivial to get the prefetch timing right for every CPU.

The front buffer should live on write-combined memory to avoid large scale cache pollution. If the front buffer lives on write-combined memory, then the pixels have to be written in order to maintain good performance. It is, however, not necessary to write to the front buffer in a completely sequential fashion because the write-combining buffers are relatively small. Alternatively, the front buffer can live on write-through cacheable memory without the risk of displaying stale data, but at the cost of unnecessary cache pollution. If the front buffer lives on write-allocate cacheable memory, then the caches need to be explicitly flushed to avoid displaying stale data.

5.2 CPU Texture Mapping

In its simplest form, the time warp walks the screen in raster scan order and for each pixel samples the stereoscopic images using warped texture coordinates. If the optical aberration of the lenses is described by one or more polynomials, then these polynomials need to be evaluated at every pixel to calculate the warped texture coordinates. Warped texture coordinates could be pre-calculated and stored for every pixel but this would require significant storage and bandwidth.

The time warp can be optimized by using a piecewise linear approximation which maps the stereoscopic images onto screen aligned quads with texture coordinates only calculated or stored at the quad corners. Linear interpolation can then be used to calculate the texture coordinates for each pixel inside a quad. Multiple texture coordinates may have to be interpolated to be able to correct for chromatic aberration. If the display is 1920x1080, then using quads that cover 32x32 destination pixels requires a modest amount of compute or storage space for the texture coordinates, while linear interpolation over 32 pixels introduces minimal distortion to the time warp transformation. Using 32x32 pixel quads reduces the compute and/or storage space requirements by over a factor 1000.

5.3 CPU Texture Filtering

The texture filter can be optimized in various ways that trade quality for performance. While using nearest sampling results in the best performance, there are significant aliasing artifacts and the quality is generally not acceptable [Regan and Pose 1994]. The quality can be improved by using linear texture sampling which interpolates between adjacent texels. While this reduces the aliasing artifacts, they are still very noticeable. To produce acceptable quality, the time warp needs to use bilinear texture sampling which interpolates between the four texels surrounding the sample location. For chromatic aberration correction the time warp needs to sample from three different locations with bilinear filtering. Storing the source data in separate color planes helps to reduce the overhead of sampling from different source locations.

Without specialized hardware, anisotropic and EWA filters are very inefficient on the CPU. Instead of using a weighted combination of multiple texels, the stereoscopic images can also be rendered at, or upsampled to a much higher resolution after which these higher resolution images are simply point sampled. For decent quality, the stereoscopic images need to be upsampled to at least 4x the resolution along each axis. While a GPU can efficiently upsample the images, this requires 16x the memory to store these images. Instead, the CPU can locally upsample without ever committing the fully upsampled data to memory, however, upsampling the images on the CPU takes more time than is gained from point sampling the data because the upsampling tends to add at least 4 clock cycles per source texel.

5.4 CPU Instruction Level Optimization

The texture filter can be optimized by taking advantage of the available CPU instruction sets. Some instructions simply take more time than others, or instructions need to be scheduled and paired appropriately to efficiently use all the execution units available on a CPU. While compilers have improved significantly over the years, code that needs to be highly efficient may still need to be optimized manually using either intrinsics or the assembler language. Most of today's CPUs also implement a Single Instruction Multiple Data (SIMD) instruction set that allows multiple calculations to be performed in parallel. While in limited cases a vectorizing compiler can take advantage of a SIMD instruction set, most cases require significant programmer effort to optimally use SIMD instructions.

A GPU uses no more than 8 bits of sub-texel precision for texture filtering and there is no need to use more bits on the CPU. To be able to address sufficiently large source images, the source texture coordinates can be stored in 32-bit registers using a 24.8 fixed-point representation. The barrel distortion magnifies the source image in the center and minifies the source image towards the edges. Because the barrel distortion results in a limited minification towards the edges, the area of a source image that is sampled to produce 32x32 destination pixels is limited. If the source texture coordinates are localized within this confined space, then they can be reduced to 8.8 fixed-point. This still allows an 8x minification going from 256x256 source texels to 32x32 destination pixels. By reducing the coordinates to 8.8 fixed-point, both the X and Y coordinate can be stored and maintained in a single 32-bit register.

Bilinear weighting uses four fractions: (1.0 - fracX, fracX, 1.0 - fracY, fracY). When using a SIMD instruction set, all four fractions can be stored in a single SIMD register and they can be updated with a single SIMD add instruction to move from one destination pixel to the next. The fracX and fracY can simply be updated by adding the fractional deltas using 8-bit addition with binary wrapping. The fraction (0.255 - x) can be derived from the regular fractions by exclusive-OR-ing with 0xFF because (0xFF - x) == (x ^ 0xFF). The identity ((x + y) & 0xFF) ^ 0xFF) == (((x ^ 0xFF) - y) & 0xFF) allows a single SIMD addition to be used by adding (srcX ^ 0xFF, srcX, srcY ^ 0xFF, srcY) + (- deltaX, deltaX, - deltaY, deltaY). This produces fractions (0.255 - x) instead of (1.0 - x). This can be trivially fixed by adding one to these fractions. In practice, however, the addition of 1 can be omitted because the slight loss in brightness is not noticeable.

One way to take advantage of a SIMD instruction set is through a compressed computation where independent but equivalent operations in the scalar code are performed in parallel with SIMD instructions. In the case of bilinear filtering of RGB data, the same weighting is applied to all three components and these calculations can be performed in parallel using SIMD instructions. Most SIMD instruction sets operate on an even number of elements and there are three RGB components that need to be filtered. This approach to exploiting parallelism typically results in redundantly filtering the alpha component when the texture data is stored as packed RGBA in memory.

Better performance can typically be achieved by increasing the throughput and processing 4 or 8 pixels in parallel. For this approach, the data needs to be collected in a structure of arrays (SoA) format while the RGB source data is typically stored in memory in an array of structures (AoS) format. Most SIMD instruction sets do not implement efficient gather instructions to collect the same component of all texels into a single register. Individual components can be inserted into SIMD registers but that requires a significant number of instructions. Instead, all four components of a texel can be loaded into a single SIMD register after which the elements of multiple SIMD registers are transposed to collect the same component of each texel in a single register. This will still require a significant number of instructions to transpose the data.

Some instruction sets, like ARM® NEON, provide structure load and store instructions. While a gather instruction loads elements from multiple memory locations into a single register, a structure load instruction loads elements from a single memory location into multiple registers. This can significantly reduce the number of instructions needed to exploit parallelism.

On many CPUs all memory addressing is done with general purpose registers (GPRs). A significant number of instructions (up to half a dozen) may be needed to calculate the memory addresses for a single texture sample. Instead, 4 or 8 memory offsets could be calculated in parallel with SIMD instructions after which individual offsets are extracted to a GPR for addressing. While that significantly cuts down the number of instructions, on some platforms it is very time consuming to move data from a SIMD register to a GPR. On some processors, there is no direct path from a SIMD register to a GPR and the data has to be moved through memory possibly causing a load-hit-store. On other processors, the SIMD pipeline is staggered behind the scalar pipeline and there is a high latency for moving data from a SIMD register to a GPR. For instance, on an ARM® Cortex® A8 processor, the NEON pipe is staggered after the main pipe which causes a significant latency of up to 20 clock cycles when moving data from a NEON register to a GPR. The pixel processing itself can be staggered to hide these latencies but this typically results in high register pressure and intermediate values may spill to memory.

6 DSP Time Warp

Many of today's platforms use a System on a Chip (SoC) architecture. Such a SoC integrates a variety of components including a CPU and a GPU. Many of today's SoCs also integrate one or more Digital Signal Processors (DSPs). While on some platforms these DSPs are inaccessible, on other platforms these DSPs are user programmable and available to offload work. An example is the AMD Kaveri APU which includes multiple audio optimized programmable DSP cores. Another example is the Qualcomm© Snapdragon™ SoC which implements the Hexagon™ programming environment for mobile multimedia optimization on the DSP.

These DSPs are designed to deliver a consistent throughput with very good power efficiency while still being able to crunch through significant computational workloads. However, compared to most CPUs, these DSPs run at lower frequencies to reduce the power draw. For instance, the Hexagon™ DSPs run at speeds ranging from 200 to 800 MHz. Even though the Hexagon™ DSP can execute up to four powerful SIMD instructions per clock cycle by exploiting a Very Long Instruction Word (VLIW) architecture, the reduced clock frequency limits the overall throughput.

6.1 DSP Memory Access

Not all DSPs have direct access to High-Level Operating System (HLOS) memory. While the AMD Kaveri APU implements a unified memory architecture, the Qualcomm© Snapdragon™ 800 series SoC does not. The Qualcomm© Hexagon™ QDSP6 V5 can only access contiguous pinned memory. In other words, the stereoscopic images and the front buffer will need to live on contiguous pinned memory to avoid additional data copies.

While many modern CPUs have advanced logic for automatic prefetching, most DSPs do not, and manual prefetching is required to achieve good performance. To make manual prefetching easier, some DSPs implement very powerful prefetch instructions. The Hexagon™ QDSP6 V5 has a "box prefetch" which allows prefetching a rectangular area of memory using a single instruction. However, the Hexagon™ QDSP6 V5 can only have a single outstanding "box prefetch" at a time. While this is fine for prefetching packed color data, this does not allow simultaneous prefetching of source data stored in separate color planes.

The Hexagon™ QDSP6 V5 also has other powerful cache control instructions, such as, an instruction to flush the cache, and an instruction to allocate a zero filled cache line which avoids the time it takes to fill the cache line from memory. With these instructions, the front buffer does not necessarily have to live on write-combined memory. Ideally the front buffer lives on write-combined memory to avoid large scale cache pollution.

6.2 DSP Instruction Level Optimization

Many of the same optimizations that apply to the CPU also apply to the DSP. Most DSPs implement not only a SIMD instruction set but also exploit a VLIW architecture to take advantage of instruction level parallelism (ILP) with minimal hardware complexity. In other words, significant programming effort is required to not only execute multiple calculations with SIMD instructions, but also pack multiple instructions into VLIWs.

The Hexagon™ QDSP6 V5 uses the same register set for both SIMD operations and memory addressing. This allows multiple memory addresses to be calculated in parallel using SIMD instructions, without the need to move the calculated addresses to a different register set.

Even though the Hexagon™ QDSP6 V5 vectors are only 64 bits wide, this DSP implements very powerful instructions that allow significant parallelism to be exploited. One such instruction individually multiplies the 8 bytes from one register with the 8 bytes from another register, after which the lower and upper four results are accumulated. Optionally the two 32-bit results of this operation are accumulated with two 32-bit values in the destination register. In other words, 8 multiplications plus 6 to 8 additions are performed with a single instruction. Two of these instructions can be packed into a single VLIW, allowing up to 16 multiplications plus 16 additions per VLIW.

Unfortunately, the Hexagon™ QDSP6 V5 instructions that convert from a larger to a smaller integer data type, with either rounding, truncation, and/or saturation, take a 64-bit input register and output to a 32-bit register. In other words, SIMD parallelism is lost if these instructions are used in the middle of a SIMD code sequence.

7 GPU Time Warp

A Graphics Processing Unit (GPU) has specialized hardware for mapping images onto surfaces which makes it a natural fit for the time warp. However, the GPU is already tasked with rendering the stereoscopic images. For asynchronous operation, the GPU either needs to be able to run the time warp and stereoscopic rendering concurrently, or the GPU needs to be able to preempt normal rendering to run the time warp. Unfortunately, not all of today's GPUs and/or graphics drivers allow GPU tasks to run concurrently, or task switching comes with significant inefficiencies or delays.

The time warp can be implemented on the GPU by rendering a full screen quad and, in the fragment program, performing a warped sampling of the stereoscopic images. However, calculating the warped texture coordinates in the fragment program takes time. Instead, the time warp can also be implemented by rendering a uniformly tesselated grid of triangles over the whole screen, where the texture coordinates are setup for a warped sampling of the stereoscopic images. Rendering a grid of triangles with warped texture coordinates basically results in a piecewise linear approximation of the time warp. Multiple texture coordinates will have to be provided per vertex to be able to correct for chromatic aberration. If the destination is 1920x1080, then tesselating the triangle mesh down to quads that cover 32x32 pixels results in a high enough triangle density to accurately represent the transformation. It is interesting to note that a triangle mesh based piecewise linear approximation uses barycentric interpolation of triangles which produces slightly different results than the quad based interpolation used on the CPU and DSP.

Ideally the GPU performs texture filtering in linear space by sampling stereoscopic images that are either stored in linear space or sampled as sRGB textures. While bilinear filtering works well in practice, high end GPUs allow higher quality anisotropic and EWA filters to be used.

7.1 Immediate-Mode GPUs

An immediate-mode GPU rasterizes all scene geometry into shaded fragments that are applied directly to the full-screen frame buffer. A depth buffer (or Z-buffer) is typically used to discard fragments that are farther away than the opaque fragments that already applied to the frame buffer. Examples of immediate-mode GPUs are:

- AMD Radeon™ series
- Intel® HD Graphics series
- NVIDIA® GeForce® series

While an immediate-mode GPU implements a very straight forward graphics pipeline, there may be significant overhead from shading fragments that are ultimately invisible. This overhead does not only include the computations performed per fragment, but also the memory bandwidth used to fetch texture data and to write to, and/or read from the frame buffer. Therefore, most modern immediate-mode GPUs implement various techniques to reduce fragment shading and memory bandwidth. Examples of such techniques are:

- Z-Cull or Hierarchical-Z
- Early-Z
- Vertex, texture and pixel caches
- Tiled texture and framebuffer layouts
- Texture and framebuffer compression

Immediate-mode GPUs are typically used for high performance desktop graphics where power draw is less of a concern and active cooling can be used to dissipate heat.

7.1.1 Immediate-Mode Context Priorities

OpenGL, at least up to and including version 4.5, does not have a concept of prioritizing work from one context over another. DirectX does have a concept of context priorities but it is not fully exposed at the application level. Using DirectX, a process has a priority class and a rendering context has a priority within this class. There is no DirectX API to change the priority class, but it can be changed through the Microsoft Direct3D Kernel Mode interface (D3DKMT). The process does need admin privileges. In the case of the time warp, it is desirable to run the process in the D3DKMT real-time priority class. Once the priority class has been raised to real-time, the priority of a context can be changed within this class.

7.1.2 Immediate-Mode Task Switching Latency

Modern immediate mode GPUs support multiple hardware queues for graphics, compute, DMA, display, video decode, etc. Even though GPUs are massively parallel machines, most graphics hardware cannot run multiple graphics tasks concurrently. On newer GPUs (like the AMD Kaveri APU) compute can be executed concurrently with graphics, however, preemption of queues is necessary to implement multi-tasking of graphics. Preemption of a queue requires stopping the current work in a queue at a point where the current state can be saved so it can be restored later. In most graphics hardware, this happens at the "method boundary" such as individual draw calls. The problem is that some draw calls may take a long time to complete which results in a poor task switching granularity. Draw calls can be broken up to always stay below a certain number of primitives, but even with preemption at the "primitive boundary", a single triangle can still cover the whole screen with a very time consuming fragment program.

Task switching can happen at the method boundary on operating systems like Mac OS and Linux, but is more complicated on versions of Microsoft Windows before 10. The first version of the Windows Display Driver Model (WDDM) is designed for graphics hardware that does not support switching between graphics tasks. The WDDM exposes the different hardware queues (compute, graphics, DMA etc.) as "engines" and the GPU kernel drivers tells the OS which engines are available. However, the WDDM only allows the kernel driver to specify one graphics engine per GPU. The WDDM does support task switching but this is done completely in software, and preemption can only occur at the "command buffer segment boundary". The WDDMv2 that comes with Windows 10 lifts these restrictions.

7.2 Tile-Based GPUs

A tile-based GPU first assigns the scene geometry to screen-space tiles or bins, after which each tile is rendered individually to a small on-chip buffer of memory. Rendering individual tiles allows local spatial coherence to be exploited, and facilitates the use of limited hardware rendering resources. Tile-based GPUs are typically used in power and thermal constrained environments. Rendering tiles to a fast on-chip memory can significantly reduce the power draw and heat by minimizing the bandwidth to main memory. Examples of tile-based GPUs are:

- ARM® Mali™ series
- Imagination® PowerVR® series
- Qualcomm© Adreno® series

The tiles themselves can be rendered in several different ways. Some GPUs use Tile-Based Deferred Rendering (TBDR) where the GPU defers all fragment shading until the GPU has determined exactly which fragments are actually visible. Other GPUs use Tile-Based Immediate-Mode Rendering (TBIMR) where the GPU directly rasterizes the triangles in a tile.

7.2.1 Tile-Based Context Priorities

OpenGL ES is the dominant graphics API for tile-based GPUs. The OpenGL ES extension EGL_IMG_context_priority adds support for context prioritization, and this extension is available on most current tile-based GPUs.

7.2.2 Tile-Based Task Switching Latency

Tile-based GPUs present a unique opportunity to switch tasks at every tile or every batch of tiles. Different tiles may take a different amount of time to complete and a high triangle density or very long running fragment program can still cause a single tile to take a long time to complete, but in practice, preemption at the "tile boundary" results in reasonable task switching granularity. The task switching granularity does depend on the batching of tiles, where large batches of tiles introduce task switching latency. However, very small batches may introduce kernel driver overhead. On current tile-based graphics hardware, the context priorities are handled at the kernel driver level and multiple parameters may control the effective task switching granularity.

The Qualcomm© Adreno™ 420 GPU uses a variable tile size based on the amount of memory needed per tile, which is based on the format of the frame buffer (depth, stencil, color, MSAA, etc.). Batches of tiles are inserted into a prioritized kernel driver queue, where the batch size is parameter controlled. Individual tiles can be inserted into the prioritized kernel driver queue but this may result in significant CPU overhead. The kernel driver submits batches of tiles into the hardware queue, where the batch size is controlled by a different parameter. In other words, the task switching latency depends not only on the fragment shading time of individual tiles, but also the format of the frame buffer and multiple parameters that control the different queues.

The ARM® Mali™ T760 GPU uses a fixed tile size of 16x16 pixels. The Mali™ T760 hardware implements a job manager that will schedule fragment work in batches of 2x2 tiles (32x32 pixels) to each shader core. The kernel driver is responsible for coarse-grained scheduling based on context priority. When new work from a higher priority context is submitted, the kernel driver will stop the work for the current context, while allowing currently scheduled work to run to completion. Once the scheduled work has completed, work from the higher priority context will be started. In other words, the task switching latency depends on the rendering time of 4 tiles of 16x16 pixels, plus the overhead of stopping one batch of work and starting the next.

The binning pass, where geometry is assigned to tiles, can also introduce task switching latency. On today's hardware, the binning pass cannot be preempted and has to be completed as a whole before another graphics task can start. Fortunately, the binning pass tends to be fast and does not introduce significant task switching latency. However, very computationally expensive vertex programs can cause the binning pass to take more time, because at least the vertex positions have to be calculated to be able to assign geometry to tiles.

7.3 Compute

On hardware where compute can run concurrently with graphics, the time warp can run fully asynchronous as a compute task, while the stereoscopic images are rendered through the regular graphics pipeline. For instance, the AMD Kaveri APU implements a GPU that is capable of executing compute concurrently with graphics. The AMD Kaveri APU has a basic hardware scheduler that can interleave work at the dispatch level. It will slot pending compute work into the compute units based on a priority system and the usage level from the other queues. If the graphics work is not bottlenecked by the compute units, then the compute can be opportunistically scheduled with little to no impact on other tasks.

The current AMD DirectX 11 and OpenGL 4.5 drivers submit all compute work on the graphics queue which prevents this work from being executed concurrently. However, OpenCL uses a separate queue and can be executed concurrently with graphics.

7.3.1 Compute Context Priorities

On hardware that cannot run graphics and compute concurrently, using compute may still help to improve the task switching granularity. Either way, the GPU compute units are shared between all graphics and compute tasks. It is therefore important to be able to prioritize the time warp compute task so it will be scheduled onto the compute units in favor of other compute or graphics tasks. Unfortunately, there is currently no way to prioritize a OpenCL task over other tasks.

7.3.2 Compute Data Sharing

If the time warp is implemented as a compute task, then it will need access to the stereoscopic images and the front buffer that are typically managed by the graphics driver. OpenGL and DirectX textures can be shared with OpenCL as memory objects. Unfortunately, there is no direct access to the back/front buffer. OpenGL and DirectX render buffers can be shared with OpenCL as memory objects but the contents will need to be copied to the actual back/front buffer. This requires synchronization with the graphics driver which defeats the purpose of running OpenCL concurrently.

7.3.2 Compute Kernels

In the conventional graphics pipeline, the piecewise linear approximated time warp uses the vertex program to warp the texture coordinates. Interpolation of these warped texture coordinates happens in the attribute interpolation stage right before running the fragment program. Similar to having a separate vertex and fragment program, the compute implementation can be broken up into two kernels. One kernel transforms the texture coordinates, and a second kernel warps the stereoscopic images using the interpolated texture coordinates. Compute kernels, however, are not preceded by an attribute interpolation stage which means that interpolation has to be done inside the second kernel. Performing barycentric or bilinear interpolation inside the compute kernel may take considerable time. Instead, the first kernel can store the warped texture coordinates in a texture, allowing the second kernel to sample this texture with bilinear filtering. While this results in a dependent texture lookup, it makes bilinear interpolation of the warped texture coordinates very fast by taking advantage of the dedicated texture hardware.

8 Results

For purposes of this paper, several experimental versions of the time warp have been implemented to run on different consumer available processors. All implementations use a piecewise linear approximation by texture mapping quads of 32x32 pixels. Each listed execution time represents the total time it takes to warp a stereoscopic pair of images (1024 x 1024 texels each) onto a display of 1920 x 1080 pixels. The stereoscopic pair of images is stored either in packed form with 4 bytes per texel, or in planar form with 1 byte per color plane. In all cases, the stereoscopic pair of images lives on cacheable memory, and for the CPU implementations the destination lives on write-combined memory.

There is relatively little performance benefit from using a wider SIMD instruction set like AVX2 because the execution time is dominated by loading of the source data. Sampling exhibits good spatial locality, but the source texels for adjacent destination pixels are not laid out contiguously in memory. Even though AVX2 includes instructions to gather 32-bit data elements, they sometimes take more time than separately inserting individual elements. The NEON structure loads and stores significantly reduce the number of instructions. However, the number of clock-cycles-per-instruction (CPI) is too high on the Krait CPU to achieve the desirable throughput using a single core. Multiple cores need to be used to be able to warp onto the display at high frame rates. The Hexagon™ DSP load instructions can only read data types that are naturally aligned. For chromatic aberration correction every byte of source data has to be read and inserted into a register separately, resulting in 3x the number of load instructions compared to the packed RGBA case. The GPU implementations achieve the best performance. The compute implementation takes longer because it uses two kernels due to the lack of an attribute interpolation stage.

Table 1 shows the performance of the time warp on a single core of a 2.6 GHz Intel® Core™ i7-4960HQ CPU.

Table 1: Single core of a 2.6 GHz Intel® Core™ i7-4960HQ

sampling	source data format	source code	instruction set(s)	milliseconds	megapixels per second
nearest	packed RGBA	C	x64	2.8	724
nearest	packed RGBA	intrinsics	x64 + SSE2	1.9	1062
nearest	packed RGBA	intrinsics	x64 + SSE4	1.9	1056
nearest	packed RGBA	intrinsics	x64 + AVX2	1.5	1311
linear	packed RGBA	C	x64	9.6	211
linear	packed RGBA	intrinsics	x64 + SSE2	4.2	475
linear	packed RGBA	intrinsics	x64 + SSE4	2.9	699
linear	packed RGBA	intrinsics	x64 + AVX2	2.6	766
bilinear	packed RGBA	C	x64	18.0	112
bilinear	packed RGBA	intrinsics	x64 + SSE2	5.9	341
bilinear	packed RGBA	intrinsics	x64 + SSE4	4.9	413
bilinear	packed RGBA	intrinsics	x64 + AVX2	4.2	482
bilinear chromatic	planar RGB	C	x64	23.9	84
bilinear chromatic	planar RGB	intrinsics	x64 + SSE2	10.4	194
bilinear chromatic	planar RGB	intrinsics	x64 + SSE4	8.9	226
bilinear chromatic	planar RGB	intrinsics	x64 + AVX2	7.8	259

Table 2 shows the performance of the time warp on a single core of a 2.6 GHz Qualcomm© Krait™ 450 quad-core CPU.

Table 2: Single core of a 2.6 GHz Qualcomm© Krait™ 450 CPU

sampling	source data format	source code	instruction set(s)	milliseconds	megapixels per second
nearest	packed RGBA	C	ARMv7-A	11.3	179
nearest	packed RGBA	assembler	ARMv7-A	7.4	273
linear	packed RGBA	C	ARMv7-A	35.3	57
linear	packed RGBA	assembler	ARMv7-A + NEON	11.7	173
bilinear	packed RGBA	C	ARMv7-A	49.5	40
bilinear	packed RGBA	assembler	ARMv7-A + NEON	16.5	122
bilinear chromatic	planar RGB	C	ARMv7-A	99.1	20
bilinear chromatic	planar RGB	assembler	ARMv7-A + NEON	30.7	66

Table 3 shows the performance of the time warp on a single 800 MHz Qualcomm© Hexagon™ QDSP6.

Table 3: Single 800 MHz Qualcomm© Hexagon™ QDSP6 v50

sampling	source data format	source code	instruction set(s)	milliseconds	megapixels per second
nearest	packed RGBA	C	QDSP6 v50	14.7	138
nearest	packed RGBA	intrinsics	QDSP6 v50	9.0	225
linear	packed RGBA	C	QDSP6 v50	33.4	61
linear	packed RGBA	intrinsics	QDSP6 v50	16.4	123
bilinear	packed RGBA	C	QDSP6 v50	61.1	33
bilinear	packed RGBA	intrinsics	QDSP6 v50	28.8	70
bilinear chromatic	planar RGB	C	QDSP6 v50	90.2	22
bilinear chromatic	planar RGB	intrinsics	QDSP6 v50	71.6	28

Table 4 shows the performance of the time warp as an OpenGL graphics task on a NVIDIA® Geforce® GT 750M GPU.

Table 4: NVIDIA® Geforce® GT 750M Graphics

sampling	source data format	source code	milliseconds	megapixels per second
nearest	packed RGBA	OpenGL + GLSL	0.4	5184
bilinear	packed RGBA	OpenGL + GLSL	0.4	5184
bilinear chromatic	packed RGBA	OpenGL + GLSL	0.6	3456

Table 5 shows the performance of the time warp as an OpenCL compute task on a NVIDIA® Geforce® GT 750M GPU.

Table 5: NVIDIA® Geforce® GT 750M Compute

sampling	source data format	source code	milliseconds	megapixels per second
nearest	packed RGBA	OpenCL + C kernel	1.7	1192
bilinear	packed RGBA	OpenCL + C kernel	1.7	1192
bilinear chromatic	packed RGBA	OpenCL + C kernel	1.8	1126

Table 6 shows the performance of the time warp as an OpenGL graphics task on a Qualcomm© Adreno™ 420 GPU at 500 MHz.

Table 6: Qualcomm© Adreno™ 420 Graphics

sampling	source data format	source code	milliseconds	megapixels per second
nearest	packed RGBA	OpenGL + GLSL	1.9	1091
bilinear	packed RGBA	OpenGL + GLSL	2.0	1037
bilinear chromatic	packed RGBA	OpenGL + GLSL	2.7	768

Table 7 shows the performance of the time warp as an OpenCL compute task on a Qualcomm© Adreno™ 420 GPU at 500 MHz.

Table 7: Qualcomm© Adreno™ 420 Compute

sampling	source data format	source code	milliseconds	megapixels per second
nearest	packed RGBA	OpenCL + C Kernel	2.2	943
bilinear	packed RGBA	OpenCL + C Kernel	2.2	943
bilinear chromatic	packed RGBA	OpenCL + C Kernel	3.2	648

9 Conclusion

On today's multi-core CPUs, one or more cores can be dedicated to run the time warp. Unfortunately, it is difficult to implement high quality texture sampling on a CPU while delivering the necessary throughput to warp stereoscopic images onto a high resolution display at high frame rates. The time warp needs to use at least bilinear sampling to deliver acceptable quality. However, even bilinear sampling without correcting for chromatic aberration is time consuming on a CPU. On a high performance CPU with relaxed power constraints it is possible to implement a high performance time warp. However, on a power and thermal constrained CPU it is difficult to achieve the necessary throughput.

While a dedicated DSP could provide a power efficient alternative, current DSPs do not deliver the necessary throughput to warp the stereoscopic images onto a high resolution display at high frame rates. Current DSPs can execute multiple powerful instructions per clock cycle, but they tend to run at lower frequencies to increase the power efficiency resulting in insufficient throughput. Note that new DSPs with much wider SIMD vectors are now available which may be able to provide the necessary throughput.

A GPU is highly optimized for mapping images onto surfaces making it a natural fit for the time warp, but is already tasked with rendering the stereoscopic images. To implement an asynchronous time warp, the GPU either needs to be able to run multiple rendering and/or compute tasks concurrently, or the GPU needs to be able to preempt normal rendering to run the time warp. Unfortunately, most GPUs cannot execute multiple tasks concurrently and not all of today's GPUs and/or graphics drivers allow GPU rendering to be interrupted at any time without inefficiencies or delays. Advances in GPU multi-tasking will allow the time warp to run asynchronously on the processor that is most suited for the task.

References

AZUMA, R., BISHOP, G., 1995, A Frequency Domain Analysis of Head Motion Prediction, *Proceedings of the 22nd SIGGRAPH*, Pages 401-408

BAUER, A., VO, S., PARKINS, K., RODRIGUEZ, F., CAKMAKCI, O., ROLLAND, J.P, 2012, Computational Optical Distortion Correction using a Radial Basis Function-Based Mapping Method, *Optics Express*, Volume 20, Issue 14, Pages 14906-14920

COSTELLA, J. P., 1993, Motion Extrapolation at the Pixel Level, *School of Physics, The University of Melbourne*, Australia

GRIFFIN, M.J., SO, R.H.Y., 1991, Effects of Time Delays on Head Tracking Performance and the Benefits of Lag Compensation by Image Deflection, *AIAA Flight Simulation Technologies Conference*, New Orleans, Louisiana

GRIFFIN, M.J., SO, R.H.Y., 1992, Compensating Lags in Head-Coupled Displays Using Head Position Prediction and Image Deflection, *AIAA Journal of Aircraft*, Volume 29, Number 6, Pages 1064-1068

HIMBERG, H., MOTAI, Y., BRADLEY, A., 2012, A Multiple Model Approach to Track Head Orientation with Delta Quaternions, *IEEE Transactions on Systems, Man, and Cybernetics*. Part B, Cybernetics

JERALD, J., 2004, Latency Compensation for Head-Mounted Virtual Reality, *UNC Computer Science Technical Report*

JERALD, J., FULLER, A., LASTRA, A., WHITTON, M., KOHLI, L., BROOKS, F., 2007, Latency Compensation by Horizontal Scanline Selection for Head-Mounted Displays, *Stereoscopic Displays and Virtual Reality Systems XIV*, 64901Q

KIJIMA, R., YAMADA, E., OJIKA, T., 2001, A Development of Reflex HMD - HMD with Time Delay Compensation Capability, *International Symposium on Mixed Reality*, Japan

KIJIMA, R., OJIKA, T., 2002, Reflex HMD to Compensate Lag and Correction of Derivative Deformation, *IEEE Virtual Reality*, Pages 172-179

LIST, U.H., 1984, Nonlinear Prediction of Head Movements for Helmet-Mounted Displays, *Technical report AFHRL-TP-83-45*, Williams AFB, AZ: Operations Training Division

MARK, W. R., BISHOP, G., MCMILLAN, L., 1996, Post-Rendering Image Warping for Latency Compensation, *UNC Computer Science Technical Report TR96-020*

MARK, W. R., MCMILLAN, L., BISHOP, G., 1997, Post-Rendering 3D warping, *ACM Symposium on Interactive 3D Graphics*, Pages 7-16

MARK, W. R., BISHOP, G., 1998, Efficient Reconstruction Techniques for Post-Rendering 3D Image Warping, *UNC Computer Science Technical Report TR98-011*

MARK, W. R., 1999, Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping, *UNC Computer Science PhD Thesis*

MAZURYK, T., GERVAUTZ, M., 1995, Two-step Prediction and Image Deflection for Exact Head Tracking in Virtual Environments, *Computer Graphics Forum*, Volume 14, Issue 3

MCMILLAN, L., BISHOP, G., 1995, Head-Tracked Stereo Display Using Image Warping, *Symposium on Electronic Imaging Science and Technology*, Pages 21-30

MCMILLAN, L., 1995, A List-Priority Rendering Algorithm for Redisplaying Projected Surfaces, *UNC Computer Science Technical Report TR95-005*

MINE, M., BISHOP, G., 1993, Just-In-Time Pixels, *UNC Computer Science Technical Report TR93–005*

PEEK, E. M., LUTTEROTH, C., WUNSCHE, B. C., 2013, More for Less: Fast Image Warping for Improving the Appearance of Head Tracking on HMDs, *Image and Vision Computing New Zealand (IVCNZ)*, Pages 41-46

PEEK, E. M., WUNSCHE, B. C., LUTTEROTH, C., 2014, Image warping for Enhancing Consumer Applications of Head-Mounted Displays, *Proceedings of the Fifteenth Australasian User Interface Conference (AUIC)*, Volume 150, Pages 47-55

OLANO, M., COHEN, J., MINE, M., BISHOP, G., 1995, Combatting Rendering Latency, *Proceedings of the symposium on Interactive 3D Graphics*, Pages 19-24

POHL, D., JOHNSON, G., BOLKART, T., 2013, Improved Pre-Warping for Wide Angle Head Mounted Displays, *Proceedings of the 19th VRST*, Pages 259-262

POSE, R. D., 1996, ARP: A Specialized Computer Architecture Applicable to Virtual Reality, Telerobotics and Interactive Television, *Computer Architecture*, Pages 15-29

POSE, R. D., 2000, Parallelism in a Computer Architecture to Support Orientation Changes in Virtual Reality and other Immersive Interactive Graphics Systems, *Parallel and Real-Time Systems (PART)*, Pages 18-19

REGAN, M., POSE, R. D., 1992, A Low Latency Virtual Reality Display System, *Monash University Department of Computer Science*, Technical Report 166

REGAN, M., POSE, R. D., 1993, An Interactive Graphics Display Architecture, *IEEE Virtual Reality*, Pages 293-299

REGAN, M., POSE, R. D., 1994, Priority Rendering with a Virtual Reality Address Recalculation Pipeline, SIGGRAPH, Pages 155-162

REGAN, M., POSE, R. D., 1994, Display Memory Access Issues and Anti-Aliasing with a Virtual Reality Graphics Controller, Eurographics conference on Graphics Hardware (EGGH), Pages 23-27

REGAN, M., POSE, R. D., 1994, Techniques for Reducing Virtual Reality Latency with Architectural Support and Consideration of Human Factors, *Multimedia, Hypermedia, and Virtual Reality (MHVR)*, Pages 117-129

REGAN, M., POSE, R. D., 1994, Virtual Reality and Telerobotics Applications of an Address Recalculation Pipeline, *International Symposium on Measurement and Control in Robotics (ISMCR)*, Pages 31-36

REGAN, M., POSE, R. D., 1998, An Architecture for Orientation Mapping Post Rendering, *International Conference on Artificial Reality and Tele-Existence* (ICAT), Pages 121-130

SMIT, F. A., VAN LIERE, R., FROELICH, B., 2007, The Design and Implementation of a VR-Architecture for Smooth Motion, V*irtual Reality Software and Technology (VRST)*, Pages 153-156

SMIT, F. A., VAN LIERE, R., FROELICH, B., 2008, An Image-Warping VR-Architecture: Design, Implementation and Applications, *Virtual Reality Software and Technology (VRST)*, Pages 115-122

SMIT, F. A., VAN LIERE, R., BECK, S., FROELICH, B., 2009, An Image-Warping Architecture for VR: Low Latency versus Image Quality, *IEEE Virtual Reality Conference*, Pages 27-34

SMIT, F. A., 2009, A Programmable Display-Layer Architecture for Virtual-Reality Applications, *Technische Universiteit Eindhoven PhD thesis*

SMIT, F. A., VAN LIERE, R., FROELICH, B., 2010, A Programmable Display Layer for Virtual Reality System Architectures, *IEEE Transactions on Visualization and Computer Graphics*, Volume 16, Issue 1, Pages 28-42

SMIT, F. A., VAN LIERE, R., BECK, S., FROELICH, B., 2010, A Shared-Scene-Graph Image-Warping Architecture for VR: Low Latency Versus Image Quality, *Computers & Graphics 34*, Issue 1, Pages 3-16

SO, R.H.Y., 1997, Lag Compensation by Image Deflection and Prediction: a Review on the Potential Benefits to Virtual Training Applications for Manufacturing Industry, *Design of Computing Systems: Social and Ergonomic Considerations, 7th International Conference on Human-Computer Interaction*, Volume 2, San Fransisco, USA

WARREN, R., AND ROLLAND, J.P., 1991, A Computational Model for the Stereoscopic Optics of a Head Mounted Display, *Proceedings of SPIE 1457*, Stereoscopic Displays and Applications II, 140.

WATSON, B.A., HODGES, L.F., 1995, Using Texture Maps to Correct for Optical Distortion in Head Mounted Displays, *Proceedings of the IEEE Virtual Reality* Annual International Symposium, Pages 172-178

YASUYUKI, Y., INAMI, M., TACHI, S., 1998, Improvement of Temporal Quality of HMD for Rotational Motion, *IEEE International Workshop on Robot and Human Communication (RO-MAN)*, Japan, Pages 121-126

ZHANG, D., LUO, Y., 2012, Single-trial ERPs elicited by visual stimuli at two contrast levels: Analysis of ongoing EEG and latency/amplitude jitters, *IEEE Symposium on Robotics and Applications (ISRA),* Kuala Lumpur, Malaysia