University of California
Santa Barbara

A Positional Timewarp Accelerator for Mobile Virtual Reality Devices

A thesis submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Electrical & Computer Engineering

by

Russell M. Barnes

Committee in charge:

Professor Yuan Xie, Chair
Professor Tobias Höllerer
Professor Li-C. Wang

June 2017

The Thesis of Russell M. Barnes is approved.

Professor Tobias Höllerer

Professor Li-C. Wang

Professor Yuan Xie, Committee Chair

June 2017

A Positional Timewarp Accelerator for Mobile Virtual Reality Devices

Russell M. Barnes

To my past educators Andy Long, Robert Remler, Cindy Bradley-Cross, Kate Molony, Bill Ekroth, Roger Traylor and Prof. Yuan Xie. Thank you for encouraging, captivating & inspiring me.

Acknowledgements

I would like to thank Professor Xie for supporting my graduate school experience and allowing me to teach and conduct research in SEAL; Arthur Marmin and Industrial Technology Research Institue of Taiwan for collaborating on the project formation, design exploration and software prototype; Brandon Pon for his help with data collection and design exploration; Prof. Yiyu Shi and the National Science Foundation IRES program for sending me to Taiwan and Prof. TingTing Hwang for hosting me at NTHU; Prof. Hung-Kuo Chu for his advocation; Prof. Tobias Höllerer and Ehsan Sayyad of UCSB Four Eyes Lab for their support; and finally, my parents and grandparents for supporting me through college.

Abstract

A Positional Timewarp Accelerator for Mobile Virtual Reality Devices

by

Russell M. Barnes

Mobile virtual reality devices are becoming more common, and yet their performance is still too low to be considered ideal. Frame rate and latency are two of the most important areas that should improve in order to provide a high-quality virtual reality experience. Meanwhile, positional tracking is improving the immersive experience of new mobile virtual reality devices by allowing users to physically move about in space and see their corresponding view matched in the virtual world. Timewarping is a technique that can improve the perceived latency and frame rate of virtual reality systems, but the positional variant of timewarping has proven to be difficult to implement on mobile devices due to the performance demands. A depth-informed positional time warp cannot be fully parallelized due to the depth test required for each pixel or group of pixels.

This thesis proposes a *positional timewarp hardware accelerator* for mobile devices. The accelerator accepts a rendered frame and depth image and produces an updated frame corresponding to the user's head position and orientation. The accelerator is compatible with existing deferred rendering engines for minimal modification of the software structure. Its execution time is directly proportional to the image resolution and is agnostic of the scene complexity. The accelerator's size can be adjusted to meet the latency requirement for a given image resolution. It can be integrated into a system-on-chip or fabricated as a separate chip.

Three examples are designed and simulated to show the performance potential of this accelerator architecture. The designs provide latencies of 15.43 ms, 11.58 ms and

9.27 ms for frame rates of 64.7, 86.4 and 107.9 frames per second, respectively. Although the visual side-effects may be insufficiently few to completely disregard the GPU's frame rate, the accelerator can still improve the end-to-end positional latency and is also capable of substituting the GPU in the case of dropped frames.

Contents

Chapter 1

Introduction

Virtual reality is a technology that, when executed successfully, provides a new medium for understanding information, watching stories, visiting virtual environments and interacting with people the world over. As a medium in its infancy, however, the promise of virtual reality has yet to meet the increasingly high expectations from users regarding both the content and the platform itself. Although it can be argued that a "killer app" is needed for virtual reality to make the leap to mainstream adoption, the technology plays an important role, too, as it delineates the experiences of watching something on a screen and becoming immersed in another environment.

This latest wave of virtual reality activity was brought on by the ingenuity of Oculus founder Palmer Luckey and the fostering of computer graphics pioneer John Carmack [1]. Since the release of the first development kit from Oculus, several major head-mounted displays (HMDs) for desktop computers and consoles have entered large-scale manufacturing, notably the Oculus Rift, HTC Vive and PlayStation VR [2, 3, 4]. These devices use the full capabilities of powerful graphics processing units (GPUs) to deliver the best experiences that are currently possible. Their major downsides include demanding hardware performance requirements, leading to a high total cost, and the use of a cable and

Figure 1.1: Immersion in virtual reality [8]

external tracking devices to tether users to one particular indoor space. As with many electronic devices prior, virtual reality is heading toward a mobile form factor. Samsung and Oculus partnered to release the Gear VR, an add-on device for smartphones that pairs an inertial measurement unit and lenses with a smartphone's high-resolution display and processing capabilities [5]. Gear VR's lower barrier to entry led to more units sold in 2016 than the top three high-end devices combined [6]. The Google Daydream platform is leading to new smartphone-based and standalone mobile virtual reality devices [7]. Mobile solutions benefit from the absence of any cables and space limitations but suffer from limited performance and power budgets.

Two important metrics for measuring the quality of a virtual reality experience are frame rate and latency. The frame rate is the frequency with which new frames are presented on the display. The latency is the time it takes from when the user's head position is sampled to when the user sees the corresponding view appear on the display. A key component for achieving high performance in both of these metrics is the GPU. A fast GPU can process more details and effects for a given span of time, leading to higher visual quality, a higher frame rate, or both. Mobile devices are at a disadvantage

with producing quality virtual reality experiences because of their GPUs. Mobile GPUs typically cannot contain as many transistors as desktop or console GPUs because of the physical space limitation of the smartphone form factor. Their heat dissipation is also constrained due to the lower thermal design power of smartphones [9], specifically causing an issue with using positional tracking for virtual reality [10]. Because of the characteristics of smartphones, it is difficult for mobile GPUs to come close to matching desktop and console GPUs, the high end of which are used to power virtual reality devices.

Timewarping, also known as reprojection and image warping, is a technique first developed in the 1990s to increase the frame rate and reduce the latency in 3D graphics systems [11]. The timewarping algorithm in its most basic form, which will be referred to as rotational timewarp, accepts an image, the camera orientation that the image is rendered for and the desired camera orientation, and it produces an image that resembles one rendered for the desired camera orientation. A more thorough form that will be referred to as positional timewarp accepts a depth image consisting of a colormap and depthmap, as well as the original and desired camera positions, and produces an image that resembles a rendered image from the desired position. When done efficiently, this technique can be used to update a rendered frame with the latest position data before displaying it, reducing the perceived latency of the graphics system, and in some cases increasing the frame rate.

This technique has been popularized in recent years as a tool for improving the quality of virtual reality systems. Notably, Oculus and Samsung implemented rotational timewarp in the Gear VR to compensate for smartphone GPU limitations [12]. However, Gear VR is a rotation-only virtual reality device that does not consider the translation of the user's head in three-dimensional space. Thus, it does not require anything beyond rotational timewarping to improve the latency. Newer mobile virtual reality devices

utilize more advanced inside-out tracking methods to provide six degrees of freedom for both rotation and position changes of the user's view in the virtual world [7]. For these devices, positional timewarp capability would be beneficial to the end user experience.

The rotational timewarp used in mobile devices today is executed in the GPU. It will be shown in Chapter 4 that a positional timewarp requires higher memory bandwidth and is more difficult to parallelize than its rotational counterpart. According to Oculus regarding the Gear VR, they "do not have the performance headroom on mobile to have TimeWarp do a depth buffer informed reprojection" [13]. While the processing power of mobile devices will indeed continue to improve over time, the resolutions and frame rates of virtual reality devices will also increase in order to improve immersion and bring the experience closer to resembling reality, potentially leaving little allowance for a positional timewarp solution using only the GPU.

The idea that this thesis puts forth is a positional timewarping accelerator for mobile virtual reality devices. As the GPU produces frames, the accelerator reads the most recent frame and its head position along with the newest head position data from shared memory. Then, it generates an updated frame using a positional timewarp. A tile-based design allows for the accelerator to process more pixels per unit time by adding more hardware and memory interfaces, allowing for the accelerator to generate timewarped frames faster than the GPU is capable of producing them. This accelerator could be integrated into a system-on-chip or produced on its own.

Background material and previous works are reviewed in Chapter 2. Chapter 3 introduces the proposed accelerator and system architectures, and implementation details are discussed in Chapter 4. Three register-transfer level (RTL) prototypes of this accelerator were designed and simulated, and the results are presented and discussed in Chapter 5. A summary and ideas for future work based on this accelerator are found in Chapter 6.

Chapter 2

Background and Related Work

Before describing the accelerator architecture in detail, it is necessary to review some basic concepts of three-dimensional computer graphics. Then, specifics related to rendering 3D graphics for virtual reality are discussed. The timewarping algorithm is described in both rotational and positional forms, along with their respective trade-offs. Related works in the areas of timewarp algorithms and acceleration are also reviewed.

2.1 Computer Graphics & Graphics Processing Units

The goal of computer graphics is to turn data stored in a digital medium into an image to be displayed. These images can show content that is two-dimensional, such as an operating system desktop, or three-dimensional, in the case of 3D animated films, video games and virtual reality. For the remainder of this paper, the focus will be on three-dimensional computer graphics.

At the core, these computer graphics are made up of vertices connected by lines. When three points are connected, they form a triangular polygon. Using polygons, many

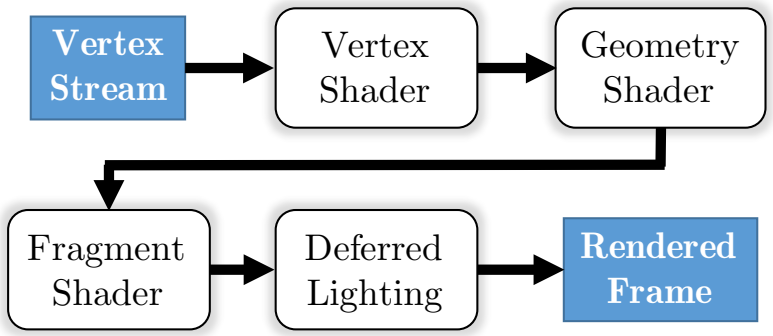


Figure 2.1: Graphics pipeline with deferred rendering

three-dimensional volumes can be constructed. For example, two equilateral polygons can make a square, and six of those squares can make an enclosed cube. With a sufficiently large number of very small polygons, they can be arranged to create complex virtual shapes with smooth or textured surfaces that are nearly indistinguishable from real ones.

Triangular polygons are common in computer graphics because a triangle will always form a plane, so certain assumptions can be made with regards to its properties when working in three dimensions. However, the memory-intensive calculations involved to display these shapes are inefficient when executed on a central processing unit (CPU). CPUs are capable of processing computer graphics, but because they are made to run any kind of computation, they are not optimized for this particular task. When graphical user interfaces and three-dimensional graphics applications took off in the 1990s, graphics processing units (GPUs) were invented to perform these calculations more efficiently and quickly.

GPUs were originally designed to perform a predetermined fixed-function pipeline of tasks, but since their inception, they have become more generalized and programmable. The basic graphics work flow in OpenGL [14], a prevalent graphics programming interface, begins with a list of vertices. These are passed through fragment shaders before being connected to form polygons and processed in geometry shaders. After color and

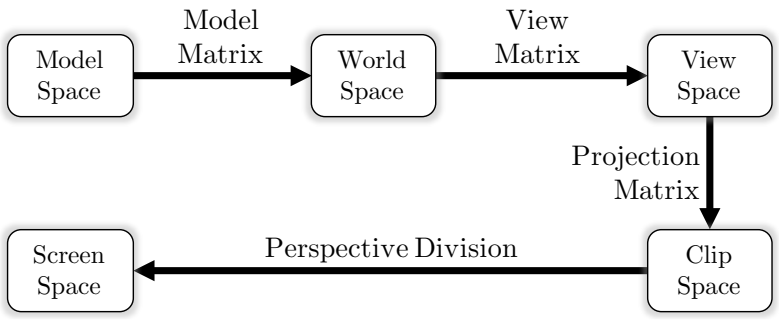
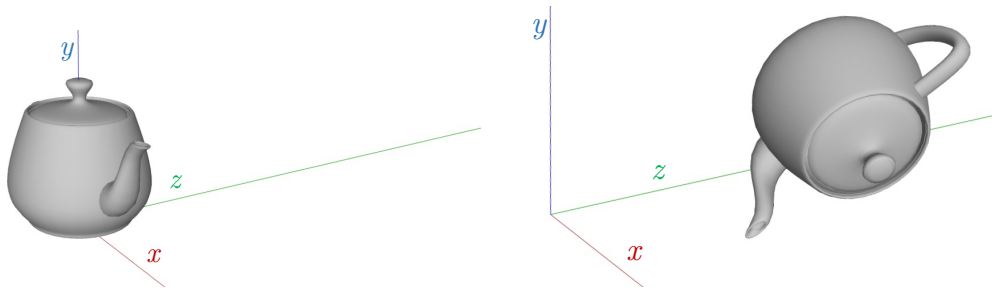


Figure 2.2: Coordinate systems

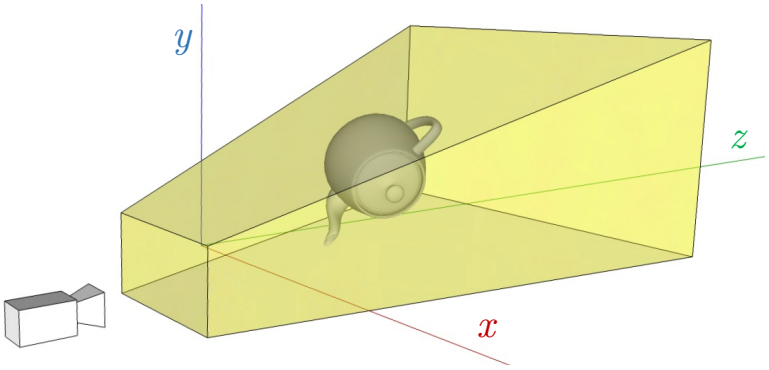
effects have been applied to the polygons, the program may perform view culling, which eliminates the polygons that are not directly visible to the user, and then rasterization is performed, which converts the graphics in 3D space into a grid of pixel-like fragments that is almost ready to be displayed on the screen. The fragments are passed through fragment shaders, and in programs that use a method called deferred rendering, certain lighting effects are calculated. Lighting effects are done at this stage instead of earlier in the pipeline in order to constrain the computational complexity of the calculations in accordance with the number of pixels on the screen as opposed to the number of polygons present in the scene. Finally, the frame is completed and submitted to the display, and the process starts over to generate the next frame.

Three-dimensional graphics programs usually generate images of the scene that resemble what a human eye might see. To translate an abstract set of polygons onto an image with a realistic perspective, a camera model is used. The camera model describes parameters including the field of view and aspect ratio of the desired perspective, and the camera can be positioned and oriented anywhere in the three-dimensional scene. The pinhole camera model is favored for its simplicity.

To translate an abstract idea of three-dimensional objects into an image on a screen, a system of coordinate transformations is required to map data between coordinate spaces



Model space, left, to world space



View space to clip space

Figure 2.3: Model, world and view spaces

(Figure **??**), eventually terminating in screen space, the two-dimensional image that is displayed on the screen. A popular system for expressing these transformations is with model-view-projection matrices. The model-view-projection system is comprised of linear algebra transformations that are chained together to produce an image of the desired scene. Initially, polygons are grouped together to form models, or discrete objects in the scene. Each model has a local origin, and the polygons within the model are located with respect to that origin. Their locations are described with three values corresponding to x, y and z coordinates. To represent objects at an infinite distance, such as a sky or faraway mountains, homogeneous coordinates are used; this system add a fourth value, ω, to the coordinates of each point, creating a 4×1 vector.

Figure 2.2 shows the transformations that occur between coordinate spaces. First,

4×4 model matrices are applied to each object to position and orient it within the three-dimensional scene. After the models are properly positioned, a 4×4 view matrix is applied to translate and orient the camera at a desired position within the scene. Then, a 4×4 projection matrix applies a linear transformation that forms the frustum of a camera, transforming the data into clip space. Finally, a perspective division of the points within the view frustum is performed by dividing x_h, y_h and z_h by ω_h, resulting in a flat image on the camera face of the frustum in screen space, where x and y are used for screen coordinates $(u\ v)^T$.

Algebraically, the model-view-projection transformations are applied from right to left, as shown in Formula 2.1, where P is the projection matrix, V is the view matrix, M is the model matrix and x_h, y_h, z_h and ω_h are the homogeneous coordinates of the vertex.

$$\begin{pmatrix} x_h \\ y_h \\ z_h \\ \omega_h \end{pmatrix} = PVM \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{2.1}$$

2.2 Rendering Environments for Virtual Reality

Virtual reality systems differ from typical interactive systems in their data flow and performance requirements. Typical graphics data flows use a keyboard and computer mouse or joystick as inputs, which are then transferred to the GPU via the CPU. The GPU then renders a frame and submits it to the display. In virtual reality systems, the data flow involves additional steps. In addition to peripheral control inputs that may include a keyboard, game controller or tracked hand controllers, the head-mounted display (HMD) itself provides an input to the rendering process. In most virtual reality systems, the HMD contains an inertial measurement unit to sense head motion, and it

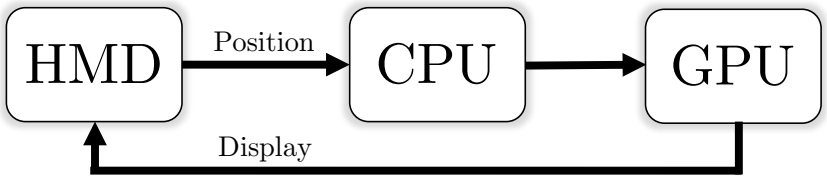


Figure 2.4: The flow of data in a virtual reality system

may utilize external or internal vision sensors for more accurate tracking. After this data is gathered and processed by the CPU, it is transferred to the GPU, where it acts as an input to the camera model.

Instead of rendering one view, virtual reality systems must render two independent viewpoints to provide stereo vision. The GPU can be optimized to reuse data for both viewpoints, but regardless, it leads to more effort than rendering only one. After the stereoscopic frame is rendered, a barrel distortion and chromatic aberration correction is applied to each eye. The barrel distortion is necessary to compensate for the pincushion distortion caused by the lenses used to focus and fill the user's field of vision with the light from the HMD's display, and chromatic aberration correction redistributes colors at the fringes of the view. Finally, the completed frame, exemplified in Figure 2.5, is transferred to the display via a high-speed video link.

When rendering environments for virtual reality systems, the performance requirements to deliver a sufficient experience are increased significantly over traditional real-time graphics applications. For one, the display resolution factors into how much detail can be perceived in the virtual world. The latest virtual reality desktop systems contain either one or two displays for a resolution of about $2,160\,\text{px} \times 1,200\,\text{px}$ per eye [2, 3]. While this display density leads to an impressive pixels per centimeter figure for a smartphone screen held at arm's length, viewing these displays from several centimeters away through a lens can lead to the discernment of individual pixels and, depending on the

Figure 2.5: A frame in a virtual reality system

underlying display technology, even the empty gaps in-between them (the screen-door effect). Hence, resolutions in virtual reality devices must undergo a great increase, with an ideal target of around $32,000\,\text{px} \times 24,000\,\text{px}$ for producing images that approach real-world clarity [15]. As display resolution increases, so does the number of computations required to render a frame. This can be a challenge for mobile virtual reality devices in particular, as the GPU is expected to render for a display resolution that already exceeds many standard desktop displays. Some applications need to lower their resolution in order to keep up with the required frame rate [16]. An accelerator can potentially assist the GPU by generating new frames at a higher frequency while the GPU renders them at a slower rate.

Latency is another crucial component of virtual reality systems. As defined in Chapter 1, latency in virtual reality systems is the time it takes from when the user's head position is sampled to when the user sees the corresponding view appear on the display. Note that the frame rate provides a lower bound for latency. If a system displays new frames at 60 frames per second, the latency cannot be consistently faster than 16.7 ms. If a program

samples a new head position 5 ms before displaying a new frame but only produces 30 frames per second, the per-frame latency is low, but the overall latency is high because of the 33 ms between frames.

The latency requirement in virtual reality systems is much more stringent than in traditional scenarios. When interacting with video games on a television, the input latency is rather high—roughly in the 80 ms to 100 ms range [17]. In virtual reality systems, the screen does not act as an object that is observed from a distance, but rather a tool that effectively replaces the eyes of the user. Studies have shown that a higher latency in HMDs leads to an increase in simulator sickness [18] and a loss of presence [19]. A NASA study showed that the latency of a virtual reality system should be no more than 17 ms [20], while a latency of as low as 3.2 ms [21] is ideal to eliminate the perception of any latency in a HMD.

The rotational asynchronous timewarping present in the Gear VR allows for a rotational input latency of less than 20 ms [22]. When positional tracking is added to mobile virtual reality devices, the performance limitations will be an issue in keeping up with a low latency in positional movement as well as rotation [13]. An accelerator can help in this area by reducing the time from when the user's head position is sampled to when a corresponding view is displayed and performing a full positional and rotational view adjustment.

Based on these parameters, it can be argued whether or not the most advanced virtual reality systems at present can deliver a sufficiently immersive experience, or whether there is progress to be made before this can be claimed. The resolution of both the display and the rendered image should increase, and the time it takes to respond to the actions of the user should decrease. There is still much progress to be made before virtual reality systems can deliver something that is capable of mimicking reality, and an accelerator can help to bridge this gap sooner on mobile platforms.

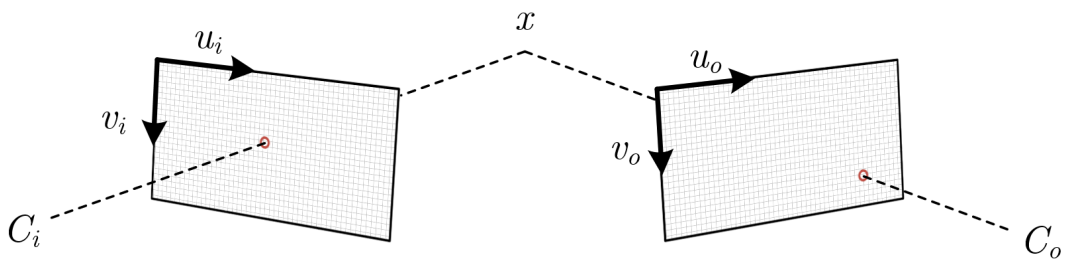


Figure 2.6: Reprojection from the reference perspective C_i to a desired perspective C_o

2.3 Timewarping

Timewarping, also known as reprojection or, more broadly, image-based rendering, is a technique used in computer graphics to generate new views from one source image. It was introduced by William Mark, Leonard McMillan and Gary Bishop in 1997 [11]. One or more depth images are used as reference frames, along with the known viewpoints at which those images were taken. A desired viewpoint at an arbitrary location is specified, and a new image is generated that corresponds to the perspective of the desired viewpoint. A depth image is an image with an extra data channel for the depth of each pixel, analogous to the Z axis in Cartesian coordinates.

This technique can be applied to generate new frames for an interactive graphics system that cannot meet the quality-of-service level for its use case. Using the timewarping algorithm to generate a new image is much less computationally expensive than rendering one from scratch, and its cost is irrespective of the content or complexity of the underlying scene, instead dependent only upon the image resolution.

There are two types of timewarp algorithms. The full timewarping algorithm presented in the original paper is referred to as positional timewarping because it corrects for any translation and rotation of the camera. Formula 2.2 shows this transformation with reference input transformation matrix T_i, desired output transformation matrix T_o

and input vector v_i.

$$v_o = T_o T_i^{-1} v_i \tag{2.2}$$

The second, simpler type of this algorithm is rotational timewarping, and it only corrects for the rotation of the camera. Unlike its positional counterpart, rotational timewarping does not require depth information and is injective. Its solution is simplified in that the translational component of $T_o T_i^{-1}$ is set to 0 as in Formula 2.3.

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \alpha_{1,3} & 0 \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & 0 \\ \alpha_{3,1} & \alpha_{3,2} & \alpha_{3,3} & 0 \\ \alpha_{4,1} & \alpha_{4,2} & \alpha_{4,3} & 1 \end{pmatrix} = T_o T_i^{-1} \tag{2.3}$$

When timewarping is used to generate a new frame in the place of a GPU, such as a system that alternates between GPU-generated images and timewarp-generated images every other frame, it is said to be synchronous. A timewarp procedure that is scheduled to operate on a frame just before a new image should be sent to the display, potentially interrupting the tasks on the GPU, is said to be asynchronous.

Image-based rendering has found use in stereoscopic content delivery systems for 3D televisions where two views can be generated with small lateral perspective shifts [23]. However, there are a few reasons why this technique has not been popularized for use in typical interactive graphics applications such as video games. One of the primary issues with timewarping is the artifacts that occur when using only one source image. When rotational timewarping is used, empty areas appear where the reference frame ends. These empty areas grow in size the farther that the desired perspective rotates from the reference perspective. When positional timewarping is used, this remains true, and there is also the issue of disocclusion. If the camera undergoes a translation in the virtual environment, and there are objects of varying depths in view, the areas behind those objects can be exposed. Figure 2.7 shows an artifact from disocclusion due to a

Figure 2.7: Example of a disocclusion artifact in positional timewarp

perspective shift to the left. There is no information available in the reference frame about what lies behind the bird, so a black area appears on the bird's left side. Because only one reference frame from the most recent GPU render job is available in typical graphics systems, and this reference frame does not contain any data about what lies behind the visible objects in the scene, these disoccluded areas are empty.

Translucent surfaces and reflections present visual issues because timewarping moves objects with respect to their position and not their effects or properties. If the user looks through a pane of glass with a depth representative of the pane's location, the algorithm will move the image seen through the glass as though it were located within the pane itself instead of with a proper parallax motion. Also, any motion in the scene that is not due to a camera movement by the user will remain static until the GPU produces a new reference frame. This means that motion that does not come from the user's head movements, such as a flying bird or the view out of a moving car's window, will lag until the next GPU frame. In many video games, this would result in a noticeable stuttering effect if the base frame rate from the GPU is low.

Another reason as to why the timewarp algorithm has not been widely adopted is the high speed of camera rotations in traditional use cases. With video games and other applications that use a mouse or joystick for view rotation, users are able to achieve a high

angular velocity. Additionally, many applications involve a "third-person" camera view with fast positional displacement. This results in large view translations and rotations, which lead to large, noticeable artifacts. Finally, using asynchronous timewarping, which could lower the effective latency of the system by altering an image that was just rendered by the GPU, requires task preemption in order to execute. This is because time warping is better executed on the GPU than the CPU, and the GPU may be rendering a frame at the moment of the timewarp execution [24]. Task preemption is something that has only recently been added to personal computer GPUs [25, 26]. Also, latencies for traditional interactive graphics use cases have comfortably occupied a higher threshold [17].

With the latest wave of virtual reality devices, interest in timewarp in its various forms has reignited. Unlike in traditional graphics applications, virtual reality head rotation is limited to the speed of a human head, leading to smaller artifacts. Positional changes of the head are often smaller—fast virtual camera movements like running can create a disconnect from the experience and introduce simulator sickness in the user, so smaller view translations resulting from physical head movements are encouraged [18, 21]. Also, the inherent pursuit of high frame rates leads to better artifacts, as the artifacts improve when less time passes between reference image updates. However, care must be taken not to rely on timewarping too much as the visual side-effects mentioned above still occur.

2.4 Accelerators

The CPU is, as denominated, the center or "brain" of a computer. It is capable of performing many diverse tasks. Due to the general-purpose nature of the CPU, there are some workloads that it does not execute in an efficient manner. When a task is deemed to be worth the extra monetary expense to perform more quickly or efficiently, an accelerator is added to the system. An accelerator is hardware that is designed to only perform a

specific domain of tasks. Although the GPU is a programmable computing platform today, it used to be considered a dedicated accelerator for the graphics workload in a system [27]. A digital signal processor is another example of an accelerator; it computes signals of the analog domain, such as audio.

While performance is an important role of accelerators, energy efficiency is another important benefit. Mobile devices are constrained by a limited power source; in smartphones, accelerators are used to handle various features of the system, including image and audio processing, leading to lower power usage [28]. While desktop systems are not faced with similar constraints, economic and environmental factors encourage lower power usage in all electronics. Processors are also limited by the heat that they can produce, and mobile virtual reality devices are no exception [10].

2.5 Related Work

Three-dimensional image-based rendering was introduced by Bishop, Mark and McMillan in the 1990s [29, 30, 11, 31]. Their vision for the application of this technique was in a head-mounted display. They addressed image quality concerns by using multiple reference images to cover the holes in the warped image. New developments in GPU-based 3D image warping are being made by Schollmeyer, Schneegans, Beck, Steed, and Froehlich [32]. They address issues with translucency and disocclusion holes by choosing between two warping methods and designing a new hole filling technique.

Other approaches to image-based rendering involve rendering a field of view greater than what will be displayed in order to avoid gaps at the edges caused by view rotation [33, 34]. This approach has a higher visual quality but is more computationally expensive and does not address disocclusion of objects in the scene due to positional changes.

Some have taken a client-server approach to image warping [35, 36, 37]. This idea

involves rendering depth images remotely and transmitting it to a mobile device. More recently, Xiaochuan and Xiaohui designed an image-based rendering technique that uses motion prediction and multiple depth images for improved image quality [38].

In the realm of hardware acceleration of image-based rendering for virtual reality, Smit, van Liere and Froehlich designed a programmable display layer that performs high-quality image warping [39]. Their algorithm warps objects in the scene separately by using a motion field that is generated by the application. This allows for objects that are moving in the scene to continue moving in the new images that are generated. User motion prediction is used to continue producing frames with low latencies until a new reference frame is ready. To reduce artifacts, they use two reference depth images, increase the field-of-view of one of the reference perspectives to minimize edge holes and evaluate intelligent camera placement strategies. Their technique uses one GPU as a client and one GPU per display as servers. However, this work does not benefit mobile devices, which are limited to one performance- and energy-constrained GPU. An additional GPU in a mobile platform could be prohibitive in area and the thermal design power.

Research is also being done on using alternative display technologies to reduce latency. Lincoln et al. designed a system to deliver 80 µs of motion-to-photon latency [40]. They combine 3D image warping with a grayscale micromirror display for an extremely low-latency virtual reality display. However, digital micromirror displays are not currently in use as primary displays for desktop computers or mobile devices and require additional mechanical parts to project color images. As illustrated by their follow-up work for augmented reality [41], there is much work to be done in shrinking the size of the components before it can become an integrated display solution. Also, this latency figure does not include the delay incurred by typical sensors and pose calculation, which is orders of magnitude larger.

Other research into latency reduction involves changes in the graphics rendering process itself. One of the precursors to time warping is the address recalculation pipeline invented by Regan and Pose in 1994 [33], which separates the user's head position from the rendering process and is able to render different parts of the scene at different rates. Friston, Steed, Tilbury and Gaydadjiev recently designed a frameless renderer based on this work along with concepts of dataflow computing and ray casting [42]. Their system is able to render pixels in any order and time each one for completion just before the raster displays it. They implement the renderer in an FPGA hardware accelerator for testing with a desktop HMD. This work results in an impressive 1 ms of motion-to-photon latency for positional and rotational movements. The crux of the idea is a new rendering architecture, so all graphics applications would be required to shift away from GPU rendering to this new hardware, a change that may not work for area-limited mobile devices that need to retain their GPUs to run traditional graphics applications in addition to virtual reality. Also, it is uncertain if this technique could maintain the current standards of visual quality in graphics applications when coupled with another rendering technique to produce them. The test cases shown are very simple static scenes, and ray tracing is considered to be too demanding when rendering complex scenes, but progress is being made in this problem as well [43].

Numerous software solutions are currently available that use the existing hardware to lower the motion-to-photon latency. Both major GPU vendors provide software features that encourage low latency for virtual reality applications by enabling asynchronous task scheduling, namely found in NVIDIA's VRWorks [25] and AMD's LiquidVR [26]. Valve and HTC support asynchronous reprojection on the HTC Vive [44]. This reduces motion-to-photon delay by resampling the head input and interrupting the GPU when needed to generate a new frame. It currently does not correct for positional motion, but this is planned for a future update [45].

In late 2016, Oculus introduced Asynchronous Spacewarp for their Rift HMD [46]. This technique uses both the CPU and GPU to provide both positional and rotational timewarp capabilities and also update moving objects in the scene [47]. When the application experiences a drop in frame rate below the 90 frames per second target, the GPU only generates new reference frames at 45 frames per second, while it continues to deliver updated frames to the HMD at 90 frames per second using Asynchronous Spacewarp. The GPU produces motion vectors for each frame, and they are used to find the optical flow between frames and extrapolate animation and positional change [48]. The Spacewarp task can run concurrently with the rendering process. An asynchronous timewarp takes care of view rotation in a separate task [47]. A fairly new GPU is required to use these features, and it is not yet available on any mobile virtual reality device.

The state of latency reduction on mobile devices is more limited. Oculus and Samsung addressed the latency problem at the launch of their mobile product Gear VR with asynchronous timewarping [5, 12]. The version of asynchronous timewarp present on the Gear VR only corrects for view rotation—this is because of both the lack of positional tracking in the Gear VR and performance limitations [13]. To implement the rotational warp, they combined it with the lens distortion correction step so that each pixel need only be repositioned once [12].

Chapter 3

Proposed Architecture and Design

Mobile virtual reality devices are becoming more common, and yet their performance levels are still too low to be considered ideal, let alone those of their tethered counterparts [21, 15]. Positional tracking is improving the immersive experience of new mobile virtual reality devices by allowing users to move about in space and see their corresponding view matched in the virtual world [7]. Asynchronous timewarping can improve the perceived latency of virtual reality systems [12], but positional timewarping has proven to be difficult to implement on mobile devices due to the performance demands [13]. A basic positional time warp cannot be fully parallelized due to the depth test required for each pixel or group of pixels, which will be discussed in Chapter 4. Although the tile-based architecture of mobile GPUs allows for task preemption, special care must be taken to reliably execute an asynchronous time warp [24].

This thesis proposes a *positional timewarp hardware accelerator* for mobile devices. The accelerator is compatible with existing deferred rendering engines for minimal modification of the software structure. Its execution time is directly proportional to the image resolution and is agnostic of the scene complexity. The amount of hardware resources can be adjusted to meet a latency requirement for a given image resolution. This accel-

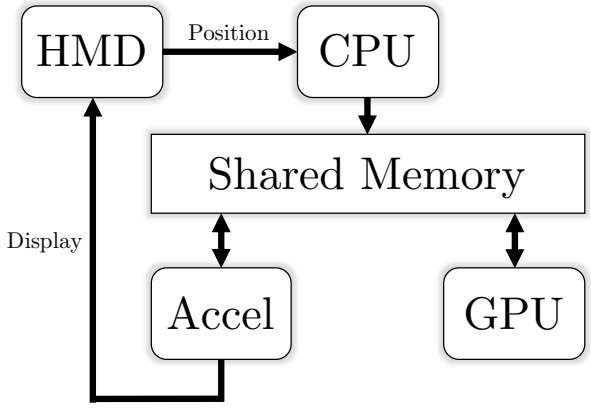


Figure 3.1: Proposed system architecture

erator can be integrated into a system-on-chip or, memory bus bandwidth permitting, fabricated as a separate chip.

As the accelerator is separate from the GPU, it is inherently asynchronous. The data flow begins with the sensor data being read from the HMD. The CPU reads and processes this data and passes the user's head position and orientation to the GPU as an input for the rendering process. When the GPU is finished rendering a new frame, this data is made available to the accelerator. Separately, the accelerator reads the most recent frame from the GPU and its corresponding head position data along with the newest head position from the CPU. Then, it generates an updated frame using a positional timewarp and issues it for display on the screen.

This accelerator takes advantage of the shared memory present in many mobile system-on-chip architectures [49] to read data from the GPU without a time-consuming memory copy. When the GPU begins rendering a new frame from scratch, which will be referred to hereafter as a reference frame, it records the user's view position. When a reference frame is completed, it is stored with its view position in shared memory. External timing logic keeps track of when a new frame is due to be displayed and triggers the accelerator with enough time to complete the timewarp task.

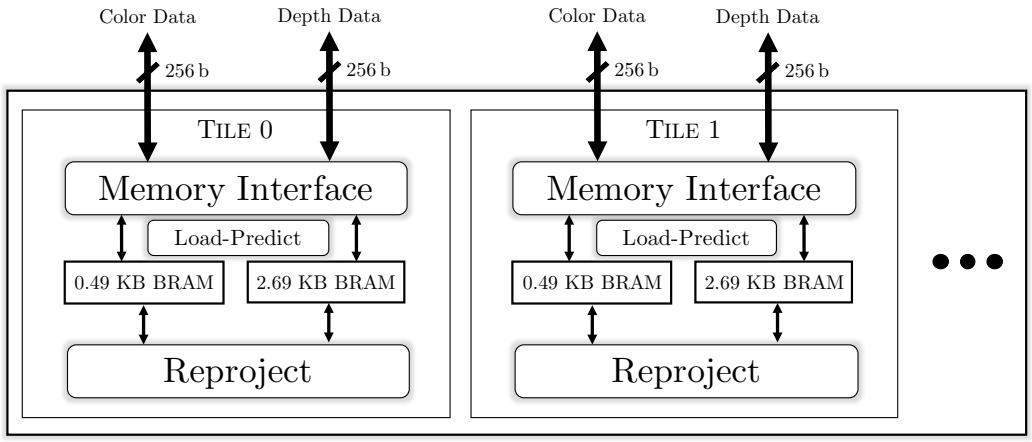


Figure 3.2: Proposed accelerator architecture

The accelerator as proposed does not attempt any form of hole filling, nor does it update any animation present in the scene. Users prefer a trade-off in visual quality for a lower system latency when performing tasks in virtual reality [50], and higher source application frame rates lead to smaller artifacts between frames. This is not to say that visual quality should be abandoned in the pursuit of low latency—according to Antonov, positional timewarping without hole filling and animation updates should not be relied on to continuously improve the frame rate of a slow application [12], and a reliance on an asynchronous accelerator without motion updates may need to trade lower positional latency for higher animation latency. However, this accelerator is a step toward a design that can provide a high visual quality, and it can also fill in frames that are dropped by the GPU, which is a net benefit.

Like rotational timewarping, positional timewarping creates empty areas at the edges of the image after view rotations. Average head rotation speeds when using HMDs are around $50\,^\circ\,\mathrm{s}^{-1}$, with $300\,^\circ\,\mathrm{s}^{-1}$ considered a rapid velocity [51]. Although eyes do undergo translation in space during a head rotation, the effect of motion smear [52] may help to hide small artifacts that occur due to this translation in static areas of the virtual

environment. Also, the edges of the image are obscured by the lens distortion, so it is more difficult to detect in an HMD than it would be on a normal display.

The input data to the accelerator is a reference depth image that includes three color channels and one depth channel. Each color channel is 8 bits, for a total of 24 bits per pixel, and each depth value is a 32-bit floating-point value. Space is required for both the reference image and final image. The reference and desired view positions are also recorded, with 16 floating-point elements in each matrix. The image data is too much to be stored on chip, so shared off-chip memory is used. At the hardware level, the accelerator uses multiple memory interfaces for parallel reads and writes to this shared memory.

To allow for the GPU render concurrently with the accelerator, two image buffers should be allocated in shared memory. If the accelerator is scaled to be a very low latency, it can run for every frame that the GPU submits. At higher latencies that approach the target frame rate, it may be a better idea to only display images from the accelerator during dropped frames. In this case, the CPU can arbitrate which source the image comes from.

$$\begin{cases} m = 3wh(3c + d) + 3(16f) \\ c = 8\,\mathrm{b} \\ d = 32\,\mathrm{b} \\ f = 32\,\mathrm{b} \end{cases} \tag{3.1}$$

The memory footprint is shown in Formula 3.1, where m is the memory footprint in bits, w is the total image width in pixels spanning both eyes, n is the number of image buffers that the GPU can write to, h is the image height in pixels, c is the bits per color channel, d is the bits per depth channel and f is the bits per position matrix element. One copy each of the depth image and position data are needed for the output, and two copies are needed for the input so that the GPU can modify new image data while leaving

the previous one intact.

Because image resolutions for mobile virtual reality systems are on the order of $1,920\,\text{px} \times 1,080\,\text{px}$, there is too much data for the accelerator to store and process at once. However, individual reads from off-chip memory for each pixel use too many clock cycles at the scale of this image resolution. Instead, the accelerator loads input data from the reference image in tiles of size $8\,\text{px} \times 8\,\text{px}$, grouping them together in burst reads for a more efficient overhead. It processes multiple of these tiles in parallel for increased throughput.

With positional timewarping, each pixel's destination is unknown until its depth value is read from memory, so an input tile of $8\,\text{px} \times 8\,\text{px}$ may not warp cleanly to a contiguous $8\,\text{px} \times 8\,\text{px}$ area in the output image. Nonetheless, individual writes to off-chip memory are also too expensive. Accordingly, an output cache of size $24\,\text{px} \times 16\,\text{px}$ is used to buffer writes and reads for the output image. The output cache is larger than the input tile to account for variation in the input tile's depth values and target locations.

When the accelerator loads an $8\,\text{px} \times 8\,\text{px}$ input tile from the reference frame, it performs a *motion-informed predictive cache load* from memory based on the predicted target locations of these pixels. If the predicted location is within the domain of the final image, then the data is passed to the reprojection and distortion correction logic for processing. Finally, the output cache is written back to memory.

Chapter 4

Implementation

This chapter discusses the implementation of proof-of-concept accelerator designs. First, a software prototype was used to evaluate the algorithm and visual quality. Then, three register-transfer level designs were created to evaluate the resource-latency trade-offs. The design was done in Vivado High-Level Synthesis in order to speed up development time.

4.1 Software Prototype

A software prototype was created using C++. This step served the purpose of implementing a version of the positional timewarp algorithm and testing the impact of design decisions on image quality. A lens distortion compensation step was also included.

Deferred rendering, mentioned in Section 2.1, is of particular interest to the positional timewarp algorithm. A requirement of deferred rendering is that a depth buffer is generated to indicate the three-dimensional depth of each fragment in the frame. The depth buffer stores a floating-point depth value for each pixel. As a natural byproduct of deferred rendering, this depth buffer can be fed into the positional timewarp.

Broadly, the positional timewarp transforms each pixel into homogeneous coordinates in three-dimensional space and then records a new view of these points with the desired camera perspective. First, the pixels of the reference depth image must be converted from screen coordinates plus depth into normalized device coordinates in three-dimensional space, as in Formula 4.1 where x_{sc} and y_{sc} are the screen coordinates of the pixel, d is the depth and w and h are the image width and height, respectively.

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} 2\dfrac{x_{sc}}{w} - 1 \\ 2\dfrac{y_{sc}}{h} - 1 \\ d \end{pmatrix} \tag{4.1}$$

Then, ω is set to 1 to create a homogeneous coordinate vector.

$$\begin{pmatrix} x_h \\ y_h \\ z_h \\ \omega \end{pmatrix} = \begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \\ 1 \end{pmatrix} \tag{4.2}$$

The positional timewarping transformation of the vector is taken with input head position V_i and output head position V_o (Formula 4.3).

$$\begin{aligned} \begin{pmatrix} x_f \\ y_f \\ z_f \\ \omega_f \end{pmatrix} &= PV_oM_oM_i^{-1}V_i^{-1}P^{-1} \begin{pmatrix} x_h \\ y_h \\ z_h \\ \omega \end{pmatrix} \\ &= PV_oV_i^{-1}P^{-1} \begin{pmatrix} x_h \\ y_h \\ z_h \\ \omega \end{pmatrix} \end{aligned} \tag{4.3}$$

The model matrices are not known, but they form an identity when multiplied and can be ignored. Finally, rasterization of the data is performed by dividing each three-dimensional coordinate by its homogeneous component, flattening it to the new image plane and discretizing the resulting value into a pixel grid, which comprises the final

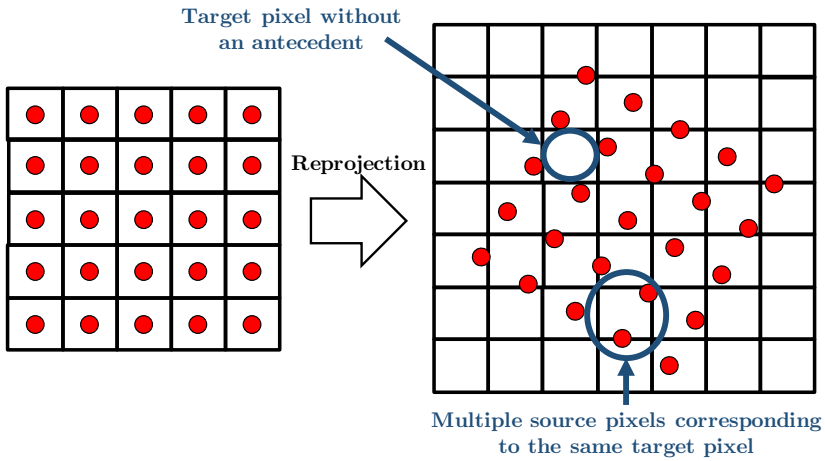


Figure 4.1: Post-reprojection reconstruction issues

image to be displayed (Formula 4.4).

$$\begin{pmatrix} x_{sc} \\ y_{sc} \end{pmatrix} = \begin{pmatrix} \dfrac{x_h}{w\dfrac{\omega_h}{2}} + 1 \\ \dfrac{y_h}{h\dfrac{\omega_h}{2}} + 1 \end{pmatrix} \tag{4.4}$$

One issue with positional timewarping is the potential for non-surjective and non-injective pixel mappings onto the source image. With the former case, normalized device coordinates may not align neatly with discretized screen space, leaving some target pixels empty while others with multiple mappings and leading to noticeable artifacts. Figure 4.1 illustrates these issues. Multiple solutions were considered for filling in the blank pixels, including mesh interpolation and splatting. In the end, the latter was chosen for its simple hardware implementation. Splatting resamples each mapped pixel's color and depth onto three adjacent pixels so that if the source pixel in question is hidden by another, it may still be displayed.

To solve the issue of multiple depth-informed source pixels mapping to the same target, depth tests are required. Before each source pixel writes its color and depth

data to a target pixel in the output image, it must check that either the target pixel is empty or the target pixel's depth is farther away from the new camera than its own; to overwrite a foreground object with something behind it would be incorrect. This depth test introduces a limiting factor in the performance of positional timewarp that leads to difficult implementation on mobile devices. Consider moving laterally in front of a chain-link fence with a distant background—pixels from the fence can traverse great portions of the view. The timewarp task cannot be fully parallelized because for an arbitrary scene, any pixel could theoretically overwrite another one.

HMDs use lenses to fill the user's field of vision with data from the display. To produce an image resembling one that the human eye perceives from reality, the images on the display must compensate for the pincushion distortion of the lenses with a barrel distortion. For the accelerator to be the last stage of the display chain, it must consider this distortion by either calculating the pixel reprojection in distortion space or applying the barrel distortion to an undistorted frame. This accelerator takes the latter approach. A simplified version (Formula 4.5) of the Brown–Condray model [53] was used to correct for radial distortion. For approximating a lens correction used in a real HMD, OpenCV was used to extract K_1 and K_2 from the HTC Vive.

$$\begin{cases} x_o = x_i(1 + K_1 r^2 + K_2 r^4) \\ y_o = y_i(1 + K_1 r^2 + K_2 r^4) \\ r = \sqrt{(x_i - \frac{w}{2})^2 + (y_i - \frac{h}{2})^2} \end{cases} \tag{4.5}$$

4.2 High-Level Synthesis

C++ is a popular programming language that can express an algorithm with a reasonable amount of code. It abstracts some memory management details away from the programmer and includes useful libraries to perform special functions. Hardware de-

scription languages (HDLs), meanwhile, are just that—a class of languages that describe digital hardware. The most common HDLs today are Verilog and VHDL. Although HDLs do not design the hardware directly, they define the behavior that the hardware should have with respect to the data, so there can be more than one hardware implementation of an HDL description. High-level synthesis (HLS) is a system for converting common computer programming code into HDL in order to enable rapid prototyping, faster development time and a lower barrier to entry for engineers to design digital hardware. Xilinx offers a product called Vivado High-Level Synthesis for its FPGA platforms [54] that originated from work by Cong et al. [55].

For this work, Vivado HLS was chosen as a tool to achieve a faster design cycle given a limited time and resource budget. The positional timewarping software prototype was used as a basis for a hardware design in the Vivado HLS toolchain. Limitations of Vivado HLS were discovered along the way, and the adapted software prototype required significant modification and revisions to produce a functional design with a reasonable performance.

One of the key differences between designing software and static hardware functions is the inability for hardware to dynamically allocate data storage for itself. Any dynamic memory allocation present in the source program must be converted to static before a hardware version can be created, annulling one of the most convenient capabilities of C++ along with features such as `std::vector`. Also, type casting is limited in Vivado HLS, and it was unable to perform some casts for data at the top-level interface and within the functions, in one case requiring data from the top-level interface to be stored in a local register of the same data type before being cast. Language support in Vivado HLS lags far behind the current versions, and C++11 is only supported in a preliminary form, foregoing features such as template aliasing.

Some parts of the Vivado HLS design process are lacking in documentation. The

outcomes of pragma directives, a tool for modifying the hardware implementation, were unexpectedly dependent upon where in the code they were placed. Task scheduling, an important part of designing a fast architecture, was especially difficult to control and required peculiar workarounds to achieve, including when enabling parallelism between functional blocks to reduce latency. Vivado HLS is not intended for designs that require communication between parallel tasks, so this brings some limitations as to what type of architectures can be implemented with this tool. Some iterations of the design executed successfully in software simulation but generated non-functional hardware, leading to iterative guess-and-check design alterations until a working combination was discovered. Hardware instantiation and reuse of hardware blocks is difficult to control, so effort was also spent to reuse the memory and reprojection cores instead of instantiating redundant ones.

4.3 Memory Design

One of the key challenges of implementing this accelerator proved to be the memory system. A depth image of size $1,920\,\text{px} \times 1,080\,\text{px}$ contains 2,073,600 pixels. Each pixel contains three single-byte color channels and one four-byte depth channel for a total of over 14.5 MB of data. Since this is too large to store all at once on the accelerator, off-chip DRAM is used to store the data, which necessitates an interface between the accelerator and memory. One of the features that Vivado HLS provides is the ability to design a streaming processor that operates on a large set of data by passing elements through the processor in sequential order. This works well for image filters such as convolutions and per-pixel differences to execute with a low latency. Unfortunately, streaming is not suitable for positional timewarping, which requires a random access to memory that stems from the unpredictability of the depth values in the input image. Instead, this

accelerator must use an addressable bus that is called AXI Master in Vivado.

One design decision to be made is the bus width, or how much data to read or write with one transaction over the memory interface. In Vivado HLS, the width of the AXI Master data bus can be from 8 bits to 1,024 bits in width. The bus width also indicates the alignment and granularity of the data—if the bus is 64 bits wide, in order to change the value of byte 3 of element n, all 64 bits of n must be read, the value of the third byte must be written, and all 64 bits must then be written back to memory. Likewise, data stored in segments that do not divide into the bus width will be split across boundaries, e.g. accessing 48 bits at a time with a 64-bit bus leads to some 48-bit elements with 16 bits at one address and 32 bits at another, requiring two reads from the bus.

An attempt was made at accessing each pixel individually. This design was extremely slow on account of multiple (~ 6) clock cycles required to read from or write to the off-chip DRAM. Although a `DATA_PACK` pragma directive is available to combine structs into single elements over the memory interface interface, it results in a complete partitioning of the elements inside and an explosion of registers in the accelerator, which are expensive in hardware. To mitigate the performance overhead for memory accesses, two steps were taken. First, a wide bus width was chosen in order to read and write many pixels at once. Designs for 128-bit, 256-bit and 512-bit bus widths were evaluated, as well as a design with a 256-bit input bus and a 512-bit output bus. The design with 256-bit memory buses for both inputs and outputs resulted in the best performance-to-resource trade-off, resulting in $10.\overline{6}$ pixel colors or 8 pixel depths per transaction. Second, burst transactions were used. Burst transactions combine multiple reads or writes under one request, allowing for performance to approach one read or write per clock cycle for a high number of requests. To use burst transactions, data must be contiguous in memory. For image data stored in row-major order, like in this project, one row can be read in a burst, but reading from multiple rows requires a separate burst transaction for each row.

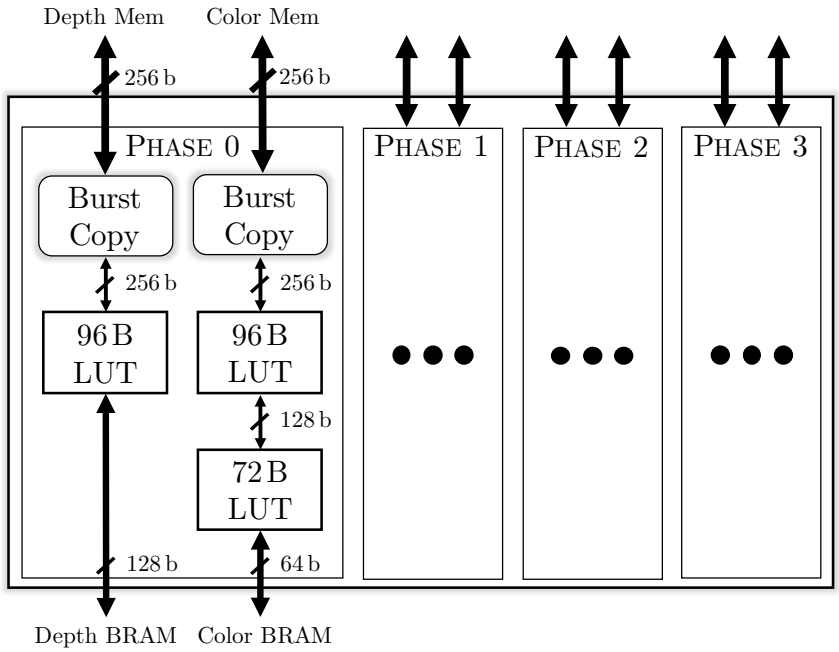


Figure 4.2: Memory architecture

In Vivado HLS, burst transactions write to and read from a block RAM (BRAM) on the host side with an equal size to the bus width. BRAM is a more efficient way to store a large amount of data than with registers. BRAM can be instantiated at any size in hardware, although there are complications in Vivado HLS if the size exceeds 1,024 bits. A design challenge with BRAM in this tool is its limitation to only two ports for reading and writing at any time. This means that if 32 pixels are stored in one 768-bit BRAM, only two can be read from or written to at once, creating a significant bottleneck for the algorithm logic. Another challenge is the high cost of dynamic bit accesses in the design, such as reading byte k from a 256-bit word. To maintain a high memory bandwidth while eliminating these bottlenecks, the data was first read into a 256-bit-wide BRAM and then copied into 24-bit and 32-bit BRAM arrays for color and depth, respectively. To split the color and depth arrays into separate BRAMs, the `ARRAY_PARTITION` pragma directive was used.

However, this still did not achieve optimal performance due to a bottleneck in the two BRAM ports of the 256-bit BRAM. To address this, intermediate BRAMs were used to copy data in multiple stages. When reading from memory, the data from a 256-bit BRAM is copied to two 128-bit BRAMs, which then copy to four 64-bit BRAMs, which in turn copy to the sized arrays. For write-back, the reverse is performed.

An obvious issue with a 256-bit bus width for the 24-bit pixel color elements is the data alignment, as 24 does not divide 256. In order to dynamically read from and write to memory in this fashion, a memory interface core was created with four phases to handle each alignment case. Each phase is instantiated separately in hardware. This serves to improve performance because of the expensive nature of dynamic bit addressing.

4.4 Algorithm Implementation

The positional timewarp algorithm is the core of the accelerator. To achieve a low latency, the throughput of this block should be as high as possible. If each pixel in a $1,920\,\text{px} \times 1,080\,\text{px}$ image was processed sequentially in one clock cycle each at a 100 MHz clock speed, the computation would take 20.7 ms. This is already slower than 60 frames per second, and it does not include the overhead required for memory I/O or a sufficient number of cycles to perform the reprojection and lens distortion compensation. For a sufficient throughput, some amount of concurrent processing is required. Note that for basic positional timewarping, any pixel's target destination could theoretically conflict with another, so the processing cannot be fully parallelized.

To start with, the input image is divided into tiles of size $8\,\text{px} \times 8\,\text{px}$. Because individual writes to the output image are too expensive, an on-chip BRAM output cache is used to buffer the values. As a gradient in depth values across the $8\,\text{px} \times 8\,\text{px}$ input tile will lead to an expansion or compression of their spatial distribution, they will probably

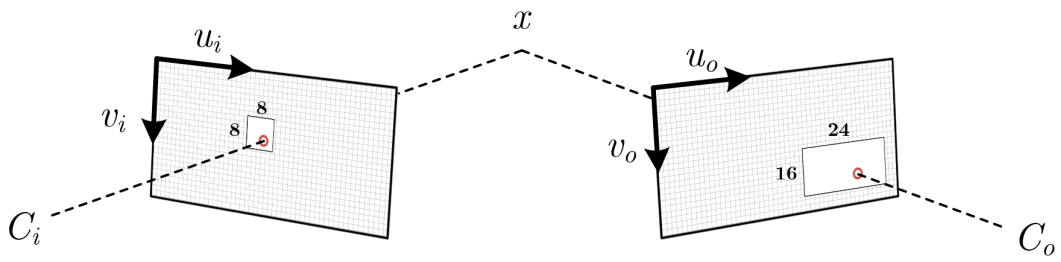


Figure 4.3: Reprojection with 8 px × 8 px input tile, 24 px × 16 px output cache

not warp cleanly to an 8 px × 8 px output tile. Thus, a larger output cache size is used. Heuristic analysis showed that 24 px × 16 px is a sufficient output tile size for most scenes.

When designing the reprojection logic, several steps were taken to increase efficiency. An advantage of custom digital hardware is that data sizes can be tailored to suit the application. For this algorithm, it is not necessary to use a 32-bit floating-point value because it is known that the input depth values are between 0.0 and 1.0. Fixed-point numbers have a static decimal range and fractional precision and lead to more efficient circuitry in most cases. Multiple depth precisions were tested, and it was found that a fixed-point representation with a 22-bit fractional precision and 4 decimal bits was the lowest precision to retain sufficient visual quality in the final image. When converting from normalized device coordinates, it was expected that a fixed-point representation would perform better with the division. However, floating-point division proved to be faster here, so a floating-point representation was retained for this step.

Before the reprojection, the depth pixel is converted into three-dimensional homogeneous coordinates. Also, the transformation matrix is fully partitioned into registers so that there are no port limitations when reading values. Since the reprojection is part of the critical path, it is important to make it as fast as possible. An efficient matrix multiplication template from the Vivado HLS library is used to multiply the fixed-point transformation matrix with the coordinate vector. Then, the radius and lens distortion

Algorithm 1: Reproject Tile

```
Result: 8 × 8 tile is reprojected
for v ← 0 to 7 do
    do in parallel
        for u ← 0 to 7 do
            d ←LoadDepthFromTile(u,v);
            {u_new, v_new, d_new}←Reproject(u, v, d);
            if {u_new, v_new} ∈ c ;                ▷ If cache contains (u_new,v_new)
            then
                isValid_uv ← 1
            else
                isValid_uv ← 0
            end
        end
    end
end
for v ← 0 to 7 do
    for u ← 0 to 7 do
        rgb ←LoadColorFromTile(u,v);
        do in parallel
            foreach {splat_x, splat_y} ∈ {[0,1], [0,1]} do
                d_c ←LoadDepthFromCache(u_new + splat_x, v_new + splat_y);
                if isValid_uv ∧ {u_new + splat_x, v_new + splat_y} ∈ c ∧ d_new < d_c then
                    WriteToCache(u_new + splat_x, v_new + splat_y, rgb, d_new);
                end
            end
        end
    end
end
```

compensation coefficient for each pixel is calculated and applied. The lens distortion is applied at this step so that only one depth test per pixel is necessary. Finally, the fixed-point homogeneous coordinates are converted into discretized screen coordinates.

The $8\,\text{px} \times 8\,\text{px}$ tiles serve as inputs to the warping logic along with the transformation matrix used to calculate the reprojections. First, the reprojections for all 64 pixels are calculated. If the position is valid, a flag is set and the target location is saved into a partitioned BRAM array. Eight reprojection blocks are instantiated and subsequently pipelined to maximize throughput, so all 64 pixels are processed in about 64 clock cycles. Pipelining the reprojections in their own loop led to a lower initiation interval (i.e. the number of clock cycles between each iteration of the functional block) than grouping the reprojections with the depth tests because calculating the reprojections without committing them to memory is fully parallelizable. To enable the parallel processing in Vivado HLS, `DEPENDENCE` pragma directives were used to indicate the absence of data dependencies in the data arrays between iterations.

After this step, depth tests are performed to arbitrate which pixels are saved to the output image. Each valid reprojected pixel results in four writes according to the splatting technique described in the previous section. For each splatted pixel, a depth test is required to compare the new depth value with the one already present in the cache. Because the four splats act on separate targets, their depth tests are instantiated separately and run in parallel. To generate a working hardware design, the output array target elements were passed to the depth test cores as direct references instead of pointers to the beginning of the array. The depth test and splatting hardware is pipelined, and the resulting performance is about 270 clock cycles.

4.5 Top-Level Design

The top-level module defines the inputs and outputs of the hardware block and instantiates functional blocks inside. At this level, the memory interface cores and timewarping blocks are controlled. A tile-based design is used to scale the performance up for different image resolutions and latency targets.

Each tile present in the accelerator is able to read from the input image, reproject pixels and write to the output image independently. For each tile in the design, new memory interfaces are added; this avoids memory bus contention. To constrain the hardware cost of the memory interface cores, image data is loaded only at boundaries that are multiples of 8 in the x direction. For a memory bus width of 256 bits, There are four ways that 8 pixel colors of 192 bits can align within an element, which means only four separate cases to handle in hardware. The output cache uses a 24-pixel width and the same alignment constraint. Because the data is stored in column-major order, the y direction is unconstrained in addressing, hence a shorter 16-pixel output cache height.

Before reprojection begins, the transformation matrix is built from the user's current and previous head position (Formula 4.3). At the start of each loop iteration, the next $8\,\text{px} \times 8\,\text{px}$ input tile is loaded, and the output cache location is predicted. In parallel with this, the output cache from the previous iteration is written back to off-chip memory, hiding most of the clock cycles. To perform a motion-informed predictive output cache load, the upper-left corner pixel depth is sampled from the input tile, converted to homogeneous coordinates and multiplied with the transformation matrix. The resulting screen coordinates are adjusted to the closest tile boundary in the x direction. Then, 8 is subtracted from both dimension to find the upper-left corner of the desired output cache. If this location is outside of the domain of the output image, then no output data will be loaded.

Algorithm 2: Accelerator, single tile

```
Result: Input image is reprojected
for t_n ∈ N do
    do in parallel
        if loc_{n-1} ∈ I_o ;        ▷ If previous target is within image bounds
        then
            Writeback(c_{n-1}, loc_{n-1});
        end
        do
            Load(t_n);
            loc_n ← Predict(t_n);
        end;
    end
    if loc_n ∈ I_o then
        Load(c_n, loc_n);
        ReprojectTile(t_n, c_n, pos_i, pos_o);
    end
end
if loc_N ∈ I_o then
    Writeback(c_N, loc_N) ;                    ▷ Writeback the last tile
end
```

After both of these steps are completed in 80 clock cycles, the output cache loads the predicted location from off-chip memory in 79 clock cycles. Then, the input tile is reprojected, and valid pixels are saved to the output cache. Before beginning the reprojection step, a check must be made to ensure that no two output caches overlap, as this would cause memory contention; to reduce the chance of this occurring, the input tiles are widely distributed. For the last iteration, a write-back of the output cache data is performed without loading new input data.

In order to manipulate Vivado HLS into executing hardware blocks in parallel, wrapper functions were created around the function calls. In one case, multiple levels of wrapper functions were required to achieve the desired parallelism.

Chapter 5

Results and Discussion

This chapter presents the simulation results from accelerator implementations with 12, 16 and 20 tiles. The RTL representation of each design was simulated in Vivado to estimate the clock cycles and resource use of a hardware accelerator. After the results are shared, the performance, hardware usage, image quality, possible applications and areas for improvement are discussed.

5.1 Results

To evaluate this hardware accelerator concept, several hardware designs were created and simulated in order to observe the performance and resource usage. Designs with 12, 16 and 20 tiles were created to illustrate the trade-off between latency and resource use. The accelerator concept is flexible in the number of tiles that it can include, and these are only shown as examples. These designs were created to process $1,920\,\text{px} \times 1,080\,\text{px}$ stereo images, and an HTC Vive was used for position data and the lens distortion model. Vivado HLS was used to simulate the RTL designs. The clock frequency of each accelerator is 105 MHz.

Design	Clock Cycles	Megapixels/ Sec	Latency (ms)	Frames/Sec
12 tiles	1,626,151	134	15.43	64.7
16 tiles	1,220,231	179	11.58	86.4
20 tiles	976,676	223	9.27	107.9

Table 5.1: Performance for tile configurations

Design	BRAM	DSP	FF	LUT
12 tiles	2,502	2,449	515,397	1,169,539
16 tiles	3,334	3,241	683,913	1,555,829
20 tiles	4,166	4,033	852,439	1,942,125

Table 5.2: Resource utilizations in Vivado HLS

Table 5.1 shows the performances for the three designs. As expected, a larger design with more tiles leads to a lower latency. The 12-tile design results in a latency of 15.43 ms, which is close to the lowest latency that an accelerator for this task should have. At the other end, the 20-tile design exhibits a more competitive latency. This is not the same frame rate that a real system would exhibit with this accelerator because of the additional time that it takes to scan out the image from the frame buffer to the display. Also, the delay from gathering and processing sensor data to check the user's latest head position must be considered, especially if an effective prediction method is not used.

Table 5.2 shows the resource utilizations for the three designs in Vivado HLS. Figure 5.1 illustrates the resource utilizations normalized to the 12-tile design. The most significant use of resources according to Vivado is lookup tables (LUT), followed in order by BRAM, digital signal processors (DSP) and flip-flops (FF). Most of the lookup table utilization is due to the memory system. Much effort was spent to lower the hardware cost of maintaining a high off-chip memory bandwidth, but the hardware cost remains high in that respect. A high BRAM utilizations is due to the on-chip buffering of image

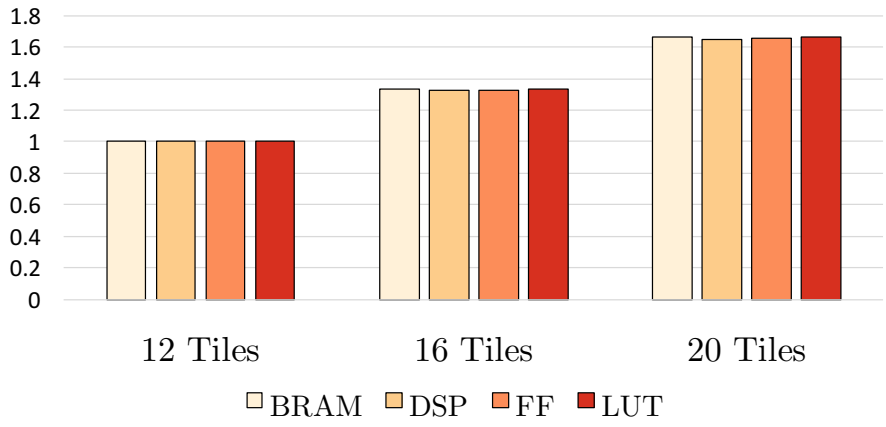


Figure 5.1: Resource utilizations, normalized to 12-tile design

data. To decrease this, tile and cache sizes could be reduced in the y direction.

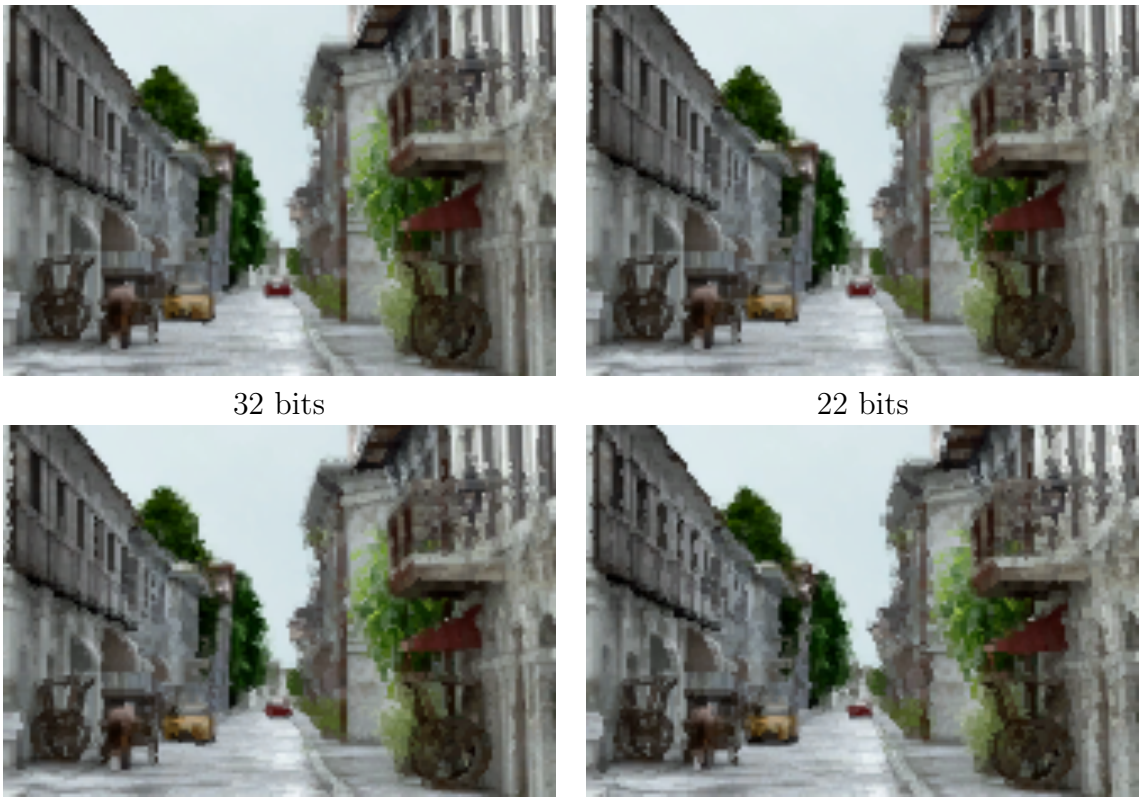
To evaluate image quality under what could pass for a real use case in virtual reality, proper head position data should be used to provide the correct amount of image artifacts. An average human walk speed of $1.386\,\mathrm{m\,s^{-1}}$ was selected to represent a lateral movement [56]. Lateral translations result in the largest disocclusions, and these movements are usually slower than when moving forward, so this figure was thought to err on the side of caution in illustrating disocclusion.

Figure 5.2 shows one half of a positionally-timewarped frame with a $1.386\,\mathrm{m\,s^{-1}}$ lateral velocity to the left. The spikes at the corners are due to the experimental lens distortion model and are not representative of what appears in HMDs. In a real HMD, these areas would remain hidden from the user's view by the lens distortion. Figure 5.3 shows the detail of image artifacts due to disocclusion. The tree branches have a high spatial frequency in the reference depth image, leading to black areas appearing on their left side.

Figure 5.4 shows the effect of fixed-point depth value precision on image quality. The images are a zoomed-in sample of a scene looking in the distance. The source data uses 32

Figure 5.2: Positional timewarp at 60 frames per second for one eye

Figure 5.3: Disocclusion artifact detail



32 bits 22 bits

20 bits 16 bits

Figure 5.4: Effect of fixed-point precision on image quality

Precision	BRAM	DSP	FF	LUT	Clock Cycles	Latency (ms)
`float`	2,502	3,481	693,724	1,168,447	1,783,841	16.93
22-bit fixed	2,502	2,449	515,397	1,169,539	1,626,151	15.43
Δ	0%	−29.6%	−25.7%	$\sim 0\%$	−8.8%	−1.5 ms

Table 5.3: Effects of precision on hardware resources and performance

Source data Quad splatting

Figure 5.5: Effect of pixel splatting on image quality

bits of precision for the depth channel. When this is reduced, the image clarity decreases slowly until 21 bits of precision, at which it begins increasing more rapidly. A precision of 22 bits was found to have the best image quality-to-performance trade-off. Table 5.3 shows the hardware resources and performance trade-offs. A lower precision results in significant savings in hardware resources and performance.

Figure 5.5 shows a reference image compared to splatting after a small positional timewarp, enlarged for detail. This type of image reconstruction is efficient in hardware but results in a slight reduction in image quality. This effect is discussed in the next section.

5.2 Discussion

This type of accelerator provides higher performance when more hardware is used. Higher performance is required when image resolution or frame rate increases. These targets are different depending on the virtual reality system in question. The 12-tile design is the smallest in consideration because of the 17 ms of latency [20] required to deliver a high-quality experience. Given the initial hardware investment of including the minimum number of tiles, it may make sense to add a few more tiles to achieve a significantly lower latency.

This accelerator is envisioned to be a part of a mobile virtual reality device, such as a smartphone. Because the performance of this platform is lower than personal computer- and console-tethered virtual reality systems, it has more to gain from an accelerator like this. Something to keep in mind when choosing the timewarping accelerator latency is the time it takes to render a frame on the GPU. An accelerator latency of 15 ms does not guarantee a perceived latency of 15 ms for every frame or the ability to timewarp asynchronously—for an application running at 60 frames per second, this probably does not leave enough time before the timewarp process begins for the GPU to render the frame. In this case, the accelerator can be used to inject frames when the GPU drops them.

To perform reliable asynchronous timewarping and lower the perceived latency of each frame, a larger accelerator is required. The 20-tile example can deliver a stream of frames at 107 per second as the GPU renders new ones at a slower pace. Any animation in the scene will remain static until it is updated by the GPU. To avoid a situation where scene animation jerks and stutters as the GPU's frame rate oscillates with respect to the accelerator's, the GPU's frame rate should be limited to a factor of the accelerator's, but not so low as to make the user uncomfortable from a low rate of animation.

Regarding the high lookup table usage of the example designs, the high-bandwidth random-access memory interface was a difficult piece of hardware to design in Vivado HLS. Taking another route with the hardware design process like writing HDL for a full-custom algorithm core and using an external memory interface IP may result in a design with a smaller footprint.

The latencies presented here assume that the head position data is valid as of the time that the accelerator begins processing. Sensor data incurs a delay during recording, transmitting to the CPU and processing, so some amount of motion prediction in software is required to ensure that it is still valid after the delay. Fortunately, prediction can be used to lower the perceived latency by predicting what the user's head position will be at the time that the frame will appear on the display. The shorter the prediction time interval, the more accurate the prediction can be [57], particularly during acceleration.

Foveated rendering could further reduce the latency of the accelerator. This is an optimization to the graphics rendering process that lowers the image quality in areas outside of the primary focal point of the image [58, 59]. It is particularly well-suited for HMDs because their use of lenses limits the ability to perceive the full pixel resolution of the display to the center of each eye's image. When eye tracking is added to HMDs, it will become even more powerful. Foveated rendering could also be applied to positional timewarping. For example, pixels away from the center of each eye's image could be reprojected with a much lower fixed-point precision, and the depth values could be tiled together to reproject pixels in groups using fewer clock cycles. Another consideration is to discard every other pixel from tiles with target locations in the compressed areas of the image where the distortion coefficient is higher, leading to fewer unnecessary re-projections and depth tests. Because additional capabilities require additional hardware, the hardware resources would likely remain the same or increase with these changes.

For a point of comparison, van Waveren's 2016 study on asynchronous timewarp

performance can be referred to [24]. This study implemented and compared rotational asynchronous timewarp performance on a variety of processors, including a laptop CPU, laptop GPU, smartphone CPU, smartphone GPU and a smartphone DSP. A $1,920\,\text{px} \times 1,080\,\text{px}$ image was used for each processor.

An important distinction is that van Waveren's study was only for rotational timewarping, while this thesis implements positional timewarping. Rotational timewarping does not require a depth component in the frame, and it is fully parallelizable. Basic positional timewarping requires a depth check for every pixel, introducing a sequential component into the algorithm. Another difference is that the processors in this study run anywhere from 500 MHz to 2.6 GHz, whereas the example designs in the thesis are clocked at only 105 MHz. Power in integrated circuits is proportional to $\frac{1}{2}CV^2f$, so given that these accelerators should be no larger than the processors in question and use the same voltage, it is expected that the accelerators here would use less power for the same task, although any difference in data movement energy between the accelerator and memory must also be considered. A lower power use means less heat dissipation, which is a boon for mobile virtual reality devices with positional tracking that must perform all of the processing near the user's body [10].

Of particular note are the results for the smartphone processors. The 2.6 GHz smartphone CPU can complete a rotational timewarp in 7.4 ms to 11.3 ms, while the 500 MHz smartphone GPU can do the same in 1.9 ms. Unlike an accelerator, these processors play other important roles in the system, and they can only perform so many tasks at once. If the more demanding positional timewarp were to execute, it would take away more execution time from these processors, leaving less time for other tasks in the system to complete. An accelerator removes this burden and allows them to carry on with the other tasks required of an interactive virtual reality system.

It is worth noting that an increase in clock speed will lead to a proportional loss in

accelerator latency, but it is not so simple as turning up the phase-locked loop. A shorter clock period leaves less time for signals to travel from one part of the chip to another. Faster signals are difficult to route and sometimes result in design rule complications that must be addressed. Eventually, increasing the clock speed will require an increase in voltage, leading to an exponential growth in power use. This avenue was not explored in this work.

One weaknesses in this work stems from a small loss of detail in the warped image (Figure 5.5). The type of image reconstruction used, splatting, results in a slightly lower image fidelity than higher-quality methods. Depending on the positional change, the data may not be aligned in the exact same way to preserve fine detail. A more advanced image reconstruction method such as meshes would be required to approach the level of quality of the source image, likely combined with foveated rendering to minimize the impact on latency.

Also, this work does not consider the additional effort required to correct for chromatic aberration of the lenses. For the best user experience, colors toward the edges of each eye's image should be redistributed to produce an accurate color at the user's retina to compensate for the varying refraction in the lens.

Finally, this work does not address the problem of animation, only updating the user's head position relative to the virtual world. If a positional timewarp accelerator was relied on to generate new frames in place of the GPU, no motion would occur in the virtual environment other than the user's local head movement. Any moving objects in the virtual world would remain static, and relative motion, such as scenery moving past as the user drives a vehicle, would not occur. An idea for updating animation is mentioned in Section 6.2, Future Work. Adding this ability to an accelerator would enable a much higher visual quality while the mobile GPU is decoupled from the display and relying on the accelerator for frame timing.

Chapter 6

Conclusion and Future Work

6.1 Summary of Contributions

This thesis proposed a hardware accelerator for positional timewarping on mobile virtual reality devices. The accelerator accepts a rendered frame and depth image and produces an updated frame corresponding to the user's head position and orientation. The accelerator is compatible with existing deferred rendering engines for minimal modification of the software structure. Its execution time is directly proportional to the image resolution and is agnostic of the scene complexity. The accelerator's size can be adjusted to meet the latency requirement for a given image resolution. It can be integrated into a system-on-chip or fabricated as a separate chip.

Three RTL examples are designed and simulated to show the performance potential of this accelerator architecture. The designs provide respective latencies of 15.43 ms, 11.58 ms and 9.27 ms, and hardware-performance trade-offs are presented.

The 20-tile example design can deliver a stream of frames at 107 per second as the GPU renders new ones at a slower pace. Any animation in the scene will remain static until it is updated by the GPU. Although the visual side effects in the resulting images

stemming from the lack of animation updates may be insufficiently few to entirely disregard the GPU's frame rate, the accelerator can still improve the end-to-end positional latency and is also capable of substituting the GPU in the case of dropped frames.

This accelerator helps to improve frame rate and latency in mobile virtual reality systems, two important metrics for measuring the quality of a virtual reality experience. The frame rate is the frequency with which new frames are presented on the display. The latency is the time it takes from when the user's head position is sampled to when the user sees the corresponding view appear on the display. Both of these areas are currently in need of improvement in order for virtual reality systems to mimic reality, especially for performance- and energy-limited mobile virtual reality devices.

6.2 Future Work

There are a number of visual quality improvements that would benefit this accelerator design. Adding hole filling capability, for one, would provide a better visual quality by reducing artifacts due to positional changes. The less severe the artifacts are, the more comfortable of an experience the accelerator can provide in-between GPU reference frames. Also, using a bilinear mesh interpolation instead of pixel splatting would likely also result in better visual quality.

A major improvement to the accelerator would come from enabling animation updates. A potential method is to use optical flow data for the reference frame, such as a motion vector map [39]. With this data, animation could be updated in each frame by adding an extra calculation in the reproject step to account for the additional vector offset, incurring little additional computational overhead but requiring more data to be read from memory. With animation updates, the GPU could confidently assume a low frame rate and render at a higher visual quality while the accelerator asynchronously

delivers frames with both rotational, positional and animation updates. One caveat is that the mobile GPU must be capable of generating optical flow data for each frame.

As far as performance goes, the average execution time could increase by grouping pixels with common depths and warping them together [60]. This could cut down on the number of reprojections and the number of depth tests, but care must be taken to avoid additional visual artifacts.

In the hardware realm, the memory cores in this design comprise most of the resources, so any steps to bring those down would be beneficial. One step could be to design the accelerator at a lower level than Vivado HLS to improve the efficiency of and control over the hardware.

Finally, this accelerator concept is not inherently limited to virtual reality systems. It could also be applied to augmented reality and mixed reality devices, augmented reality being somewhat a superset of virtual reality. A key difference with the performance requirements of these devices is that the real-world baseline for how responsive the display needs to be is always there, making a low latency crucial for blending virtual objects with the real world. As long as the display remains pixel-addressable, the graphics rendering process itself should remain similar enough to allow for a positional timewarp to update objects with respect to the user's position and orientation.

Bibliography

[1] P. Rubin, *The inside story of Oculus Rift and how virtual reality became reality*, *Wired* **22.06** (June, 2014).

[2] "Oculus Rift — Oculus." `https://www.oculus.com/rift/`, 2017.

[3] "VIVE — discover virtual reality beyond imagination." `https://www.vive.com/us/`, 2017.

[4] "PlayStation VR – virtual reality headset for PS4." `https://www.playstation.com/en-us/explore/playstation-vr/`, 2017.

[5] "Samsung Gear VR with controller." `http://www.samsung.com/global/galaxy/gear-vr/`, 2017.

[6] M. Korolov, "Report: 98% of VR headsets sold this year are for mobile phones." `http://www.hypergridbusiness.com/2016/11/report-98-of-vr-headsets-sold-this-year-are-for-mobile-phones/`, 2016.

[7] "Daydream - standalone VR." `https://vr.google.com/daydream/standalonevr`, 2017.

[8] O. Makerspace, "Clube Maker Realidade Virtual." `https://flic.kr/p/Untu5b`, 2007. Licensed under a Creative Commons Attribution-ShareAlike 2.0 Generic (CC BY-SA 2.0).

[9] "The thermal efficiency behind smartphone trends — Qualcomm." `https://www.qualcomm.com/news/onq/2013/10/09/thermal-efficiency-snapdragon-processors-under-screen-and-behind-trends`, 2013.

[10] "Google: Positional tracking "solved", but heat still a problem for VR." `https://uploadvr.com/inside-out-google-solve-tracking/`, Nov., 2016.

[11] G. Bishop, W. R. Mark, and L. McMillan, *Post-rendering 3D warping*, in *Proceedings of 1997 Symposium on Interactive 3D Graphics, Providence, RI, April 27-30, 1997, pp. 7-16*, (Providence, RI), pp. 7–16, Apr., 1997.

[12] M. Antonov, "Asynchronous timewarp examined." https://developer3.oculus.com/blog/asynchronous-timewarp-examined/, Mar., 2015.

[13] "Asynchronous timewarp." https://developer.oculus.com/documentation/mobilesdk/latest/concepts/mobile-timewarp-overview/, 2017.

[14] "OpenGL." https://www.opengl.org, 2017.

[15] W. Hunt, "Virtual reality: The next great graphics revolution." http://www.highperformancegraphics.org/2015/program/, Nov., 2015. High-Performance Graphics 2015.

[16] B. Lang, "Oculus CTO: Improved Gear VR visuals part of a brand new Oculus runtime system, details improvements." http://www.roadtovr.com/oculus-cto-improved-gear-vr-visuals-part-brand-new-oculus-runtime-system-details-improvements/, June, 2017.

[17] "Console latency: Exploring video game input lag." https://displaylag.com/console-latency-exploring-video-game-input-lag/, May, 2015.

[18] T. J. Buker, D. A. Vincenzi, and J. E. Deaton, *The effect of apparent latency on simulator sickness while using a see-through helmet-mounted display: Reducing apparent latency with predictive compensation*, *Human Factors* **54** (Apr., 2012) 235–249.

[19] M. Slater, B. Lotto, M. M. Arnold, and M. V. Sanchez-Vives, *How we experience immersive virtual environments: the concept of presence and its measurement*, *Anuario de psicologa/The UB Journal of psychology* **40** (2009), no. 2 193–210.

[20] B. D. Adelstein, T. G. Lee, and S. R. Ellis, *Head tracking latency in virtual environments: psychophysics and a model*, in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 47, pp. 2083–2087, SAGE Publications, 2003.

[21] J. J. Jerald, *Scene-motion- and latency-perception thresholds for head-mounted displays*. PhD thesis, 2009.

[22] P. Rubin, "Review: Samsung Gear VR." https://www.wired.com/2015/11/review-samsung-gear-vr/, Nov., 2015.

[23] P. Merkle, H. Brust, K. Dix, Y. Wang, and A. Smolic, *Adaptation and optimization of coding algorithms for mobile 3DTV*, *Mobile3DTV Project* (2008), no. 216503 55.

[24] J. M. P. van Waveren, *The asynchronous time warp for virtual reality on consumer hardware*, pp. 37–46, ACM Press, 2016.

[25] "NVIDIA VRWorks." `https://developer.nvidia.com/vrworks`, Feb., 2016.

[26] "Virtual reality with AMD LiquidVR technology." `https://www.amd.com/en/technologies/vr`, 2017.

[27] "CUDA." `http://www.nvidia.com/object/cuda_home_new.html`, 2017.

[28] Intel Corporation, *Intel Atom Processor Z3600 and Z3700 Series*, Dec., 2014. Rev. 003.

[29] L. McMillan and G. Bishop, *Head-tracked stereoscopic display using image warping*, in *IS&T/SPIE's Symposium on Electronic Imaging: Science & Technology*, pp. 21–30, International Society for Optics and Photonics, 1995.

[30] L. McMillan, *An Image-Based Approach to Three-Dimensional Computer Graphics*. PhD thesis, University of North Carolina, Chapel Hill, Chapel Hill, NC, 1997.

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

[32] A. Schollmeyer, S. Schneegans, S. Beck, A. Steed, and B. Froehlich, *Efficient hybrid image warping for high frame-rate stereoscopic rendering*, *IEEE Transactions on Visualization and Computer Graphics* **23** (Apr., 2017) 1332–1341.

[33] M. Regan and R. Pose, *Priority rendering with a virtual reality address recalculation pipeline*, (Orlando, Florida), ACM, July, 1994.

[34] P. Bao and D. Gourlay, *Superview 3D image warping for visibility gap errors*, *IEEE Transactions on Consumer Electronics* **49** (2003), no. 1 177–182.

[35] P. Bao and D. Gourlay, *Low bandwidth remote rendering using 3D image warping*, in *Visual Information Engineering, 2003. VIE 2003. International Conference on*, pp. 61–64, IET, 2003.

[36] P. Bao, D. Gourlay, and Y. Li, *Deep compression of remotely rendered views*, *IEEE Transactions on Multimedia* **8** (June, 2006) 444–456.

[37] W. Pasman, A. van der Schaaf, R. Lagendijk, and F. Jansen, *Accurate overlaying for mobile augmented reality*, *Computers & Graphics* **23** (1999), no. 6 875 – 881.

[38] W. Xiaochuan and L. Xiaohui, *Viewpoint-predicting-based remote rendering on mobile devices using multiple depth images*, pp. 216–223, IEEE, Oct., 2015.

[39] F. Smit, R. van Liere, and B. Froehlich, *A programmable display layer for virtual reality system architectures*, *IEEE Transactions on Visualization and Computer Graphics* **16** (Jan., 2010) 28–42.

[40] P. Lincoln, A. Blate, M. Singh, T. Whitted, A. State, A. Lastra, and H. Fuchs, *From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality*, *IEEE Transactions on Visualization and Computer Graphics* **22** (Apr., 2016) 1367–1376.

[41] P. Lincoln, A. Blate, M. Singh, A. State, M. C. Whitton, T. Whitted, and H. Fuchs, *Scene-adaptive high dynamic range display for low latency augmented reality*, pp. 1–7, ACM Press, 2017.

[42] S. Friston, A. Steed, S. Tilbury, and G. Gaydadjiev, *Construction and evaluation of an ultra low latency frameless renderer for VR*, *IEEE Transactions on Visualization and Computer Graphics* **22** (Apr., 2016) 1377–1386.

[43] M. Weier, T. Roth, E. Kruijff, A. Hinkenjann, A. Prard-Gayot, P. Slusallek, and Y. Li, *Foveated real-time ray tracing for head-mounted displays*, *Computer Graphics Forum* **35** (2016), no. 7 289–298.

[44] A. Leiby, "SteamVR beta updated (1477423729)." `https://steamcommunity.com/games/250820/announcements/detail/599369548909298226`, Oct., 2016.

[45] A. Leiby, "Asynchronous reprojection." `https://steamcommunity.com/app/250820/discussions/0/341537388320793951/#c341537388325283591`, Oct., 2016.

[46] D. Beeler, E. Hutchins, and P. Pedriana, "Asynchronous spacewarp." `https://developer.oculus.com/blog/asynchronous-spacewarp/`, Nov., 2016.

[47] P. Pedriana, "Under the hood of the Rift SDK building for touch." `https://www.youtube.com/watch?v=eAl2l_1KfqQ&t=20m39s`, Oct., 2016.

[48] M. Marinkovic, "Asynchronous spacewarp: Making great VR experiences more accessible than ever before." `http://radeon.com/en-us/asynchronous-space-warp/`, Oct., 2016.

[49] "Adreno graphics processing units." `https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu`, 2017.

[50] C. Hanel, B. Weyers, B. Hentschel, and T. W. Kuhlen, *Visual quality adjustment for volume rendering in a head-tracked virtual environment*, *IEEE Transactions on Visualization and Computer Graphics* **22** (Apr., 2016) 1472–1481.

[51] R. Azuma, *Tracking requirements for augmented reality*, *Commun. ACM* **36** (July, 1993) 50–51.

[52] H. E. Bedell, J. Tong, and M. Aydin, *The perception of motion smear during eye and head movements*, *Vision Research* **50** (2010), no. 24 2692 – 2701. Perception and Action: Part I.

[53] D. C. Brown, *Decentering distortion of lenses*, *Photometric Engineering* **32** (1966), no. 3 444–462.

[54] "Vivado High-Level Synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2017.

[55] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, *High-level synthesis for FPGAs: From prototyping to deployment*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **30** (2011), no. 4 473–491.

[56] R. V. Levine and A. Norenzayan, *The pace of life in 31 countries*, *Journal of Cross-Cultural Psychology* **30** (1999), no. 2 178–205, [http://dx.doi.org/10.1177/0022022199030002003].

[57] R. T. Azuma, *Predictive tracking for augmented reality*. PhD thesis, University of North Carolina, Chapel Hill, 1995.

[58] T. M. Henry and B. Ravikumar, *Foveated texture mapping with JPEG2000 compression*, in *Image Processing, 2001. Proceedings. 2001 International Conference on*, vol. 3, pp. 832–835, IEEE, 2001.

[59] *Foveated 3D Graphics*, ACM SIGGRAPH Asia, Nov., 2012.

[60] A. Dayal, C. Woolley, B. Watson, and D. Luebke, *Adaptive frameless rendering*, in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, 2005.