Real-time shader rendering of holographic stereograms

Citation: Smithwick, Quinn Y. J. et al. "Real-time shader rendering of holographic stereograms." Practical Holography XXIII: Materials and Applications. Ed. Hans I. Bjelkhagen & Raymond K. Kostuk. San Jose, CA, USA: SPIE, 2009. 723302-12. © 2009 SPIE

As Published: http://dx.doi.org/10.1117/12.808999

Publisher: Society of Photo-Optical Instrumentation Engineers

Persistent URL: http://hdl.handle.net/1721.1/53721

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Real-time shader rendering of holographic stereograms

Quinn Y. J. Smithwick, James Barabas, Daniel E. Smalley, and V. Michael Bove, Jr.
Object-Based Media Group, MIT Media Laboratory, Room E15-368, 20 Ames St., Cambridge, MA USA 02142-1308

ABSTRACT

Horizontal-parallax-only holographic stereograms of nearly SDTV resolution (336 pixels by 440 lines by 96 views) of textured and normal-mapped models (500 polygons) are rendered at interactive rates (10 frames/second) on a single dual-head commodity graphics processor for use on MIT's third-generation electro-holographic display. The holographic fringe pattern is computed by a diffraction specific holographic stereogram algorithm designed for efficient parallelized vector implementation using OpenGL and Cg vertex/fragment shaders. The algorithm concentrates on light-field reconstruction by holographic fringes rather than the computation of the interferometric process of creating the holographic fringes.

Keywords: synthetic holography, 3-D display, holographic video, computer graphics, graphics processors

1. INTRODUCTION

The Mark III display system [1] is the MIT Media Laboratory's third-generation electro-holographic display. It has been designed with cost and size in mind to make a consumer holo-video displays a reality. Inexpensive, interactive-rate rendering of holograms on electro-holographic displays is necessary before wide-scale adaptation of the technology and the possibility of many practical applications.

Computer generated holograms (CGH) are often computed using physically-based algorithms, such as interferometric methods for point clouds. Due to the many-to-many calculations for computing the light interaction between each point in the scene and each point in the hologram, the computational requirements are formidable. While work has been done on using dedicated hardware (FPGAs), recent research has explored the use of graphics processing units (GPUs) to take advantage of their parallelism. [2] Because they are typically point-based, interference-based algorithms must deal with occlusion and normal approximation for lighting and texturing. Hologram resolution is typically small (800 x 600) and the number of object points is typically low (60 points) for 15 fps. [2] Triangle-based Fresnel CGH has been proposed to handle these issues, but is not meant to be real-time. [3]

Alternatively, diffraction specific holographic stereogram algorithms concentrate on lightfield reconstruction by holographic fringes rather than the physical process of creating the holographic fringes. This technique uses precomputed basis fringes modulated by the scene's visual content. [4] Because this rendering technique is image-based, imaging synthetic or real scenes from multiple viewpoints, it handles texture, occlusion, and even transparency of continuous surfaces naturally. The precomputation and table look up of the basis fringes makes this technique fast.

In 2004, Bove *et al.* applied this technique using three synchronized GPUs with fixed graphics pipeline functionality. [5] Diffraction specific algorithms are readily applied to the GPU as they involve image-based multi-view rendering and multi-texture blending hologram construction. A synthetic scene was rendered from 140 viewpoints. Portions of these views modulated a basis fringe texture-mapped to a single polygon representing one one directional-view "holopoint." To produce a hologram consisting of a collection of holopoints, 383 overlapping modulated textured polygons were combined using texture blending and an accumulation buffer for each of 440 "hololines." The horizontal-parallax-only (HPO) hologram resolution was large (256K x 144) and for very low polygon count texture-mapped models (6 polygons) achieved a viewable resolution of 383 x 144 x 140 views at 30° view angle and 2 fps.

Practical Holography XXIII: Materials and Applications, edited by Hans I. Bjelkhagen, Raymond K. Kostuk,
Proc. of SPIE Vol. 7233, 723302 · © 2009 SPIE · CCC code: 0277-786X/09/$18 · doi: 10.1117/12.808999

2. GRAPHICS PROCESSING UNIT AND SHADERS

As discussed above, even with diffraction-specific approaches, CGH requires large amounts of computation. Fortunately, the processing power of GPUs is approaching teraflops, and in recent years has approximately doubled every six months. GPUs are literally parallel vector supercomputers on the desktop. Furthermore, the new programmability of the graphics pipeline provides flexibility and opportunities to compute efficient general purpose algorithms. However to take full advantage of their performance, the diffraction-specific CGH algorithm must be cast in a form amenable to parallel vector computation and to the capabilities of the GPU and programmable graphic pipeline.

OpenGL is an open-source cross-platform application program interface (API) for 3-D graphics rendering. The OpenGL graphics pipeline takes a 3-D object described by polygons, texture coordinates, and textures, and then renders to output a 2-D image. The pipeline acts as a stream processor in which data parallelism is exploited; that is, the same operations are performed on large numbers of the same type of data. Vertex operations are applied to per-vertex attributes to transform and illuminate the polygon vertices, and to clip the object to the view frustum. The transformed vertices are projected on the 2-D image plane, assembled into primitives (triangles/polygons), and rasterized such that vertex information is interpolated across the primitive into fragments. Per-fragment operations occur which may alter the color or drawing of the fragment depending on depth calculations or fragment location. The fragments are finally inserted into the frame buffer to become pixels.

For early implementations of the GPU's graphics pipeline, the vertex and per-fragment operations had fixed functionalities. In recent implementations, the functionalities of the vertex and per-fragment operations have been programmable, allowing "shaders" to apply custom transformations and calculations to the vertices and fragments. Common uses for vertex shaders are to compute per-vertex lighting, custom perspective transformations, and texture coordinate remapping. Common uses for the fragment shaders are to compute per-fragment lighting, multi-texturing, and bump/normal mapping. New shader types continue to be introduced, such as the geometry shader, which acts on primitives rather than vertices and is capable of spawning new geometry. The vertex and fragment shaders run on stream processors in the GPU. Early implementations had a fixed number of dedicated vertex and fragment processors. Recent GPUs use unified shaders that can dynamically allocate themselves to vertex or fragment processing. Cg is nVidia's C-like language for shader software development.

To achieve high performance in GPUs, the following guidelines should be followed: [6][7][8]

- **Parallelism:** Computation must be independently performed on each fragment; a fragment's computation can not simultaneously depend on neighboring fragments.
- **Gather, not scatter:** The GPU is optimized for texture lookups (gathers) and cannot write to random framebuffer locations (scatters).
- **Data locality and coherence:** Computations should use sequential memory reads rather than random access memory reads for best memory performance.
- **Intrinsic classes and functions:** Use intrinsic classes (float4) and functions (dot, normalize). GPUs are generally vector processors, typically working with four component vectors (RGBA). Some newer GPU architectures are scalar and automatically convert vector calculations into scalar ones. It is still efficient to think in terms of vector operators for data coherence reasons. Intrinsic functions either map directly to GPU instructions or are highly optimized.
- **Single instruction multiple data (SIMD):** Multiple scalars that have the same operation applied to them can be packed into a single vector. A single function call operates on all the data in parallel. Packing and unpacking the scalars into and from the vector may be time-consuming. However, in some cases there is no overhead; the data can be directly computed in the correct format, [8] and functions can operate on the vector that ultimately result in a scalar (*e.g.* dot product).
- **Precomputation using table lookups:** Complicated functions that do not vary between iterations of the algorithm can be precomputed and stored in a texture. However, if there are a large number of texture lookups, the application may become memory-bandwidth limited when it may be better to perform a simple calculation rather than recall it from a lookup table.

- **Latency hiding and computational intensity:** Texture fetches take several cycles. This latency can be hidden if other computations are being performed simultaneously. It is desirable to have a high computational intensity – a high ratio of computation to texture fetches. That is, many computations should be performed on each texture value fetched to amortize the cost of the texture fetch.
- **Load balance:** If all stages in a pipeline aren't occupied, performance suffers.

3. HOLOGRAPHIC STEREOGRAM RENDERING

3.1. Approach

One method of diffraction-specific encoding regards the holographic stereogram as a summation of overlapping amplitude modulated chirp gratings. This amplitude modulated chirped grating on the hologram plane produces a set of view-directional emitters on an emitter plane. Each chirp focuses light to create a point emitter, while the angle-dependent brightnesses of the views are encoded in the amplitude modulation (Figure 1). The varying views provide 3-D parallax/disparity cues. Chirp gratings can overlap providing the ability to encode a large number of views over a large aperture without ghosting and view flipping.

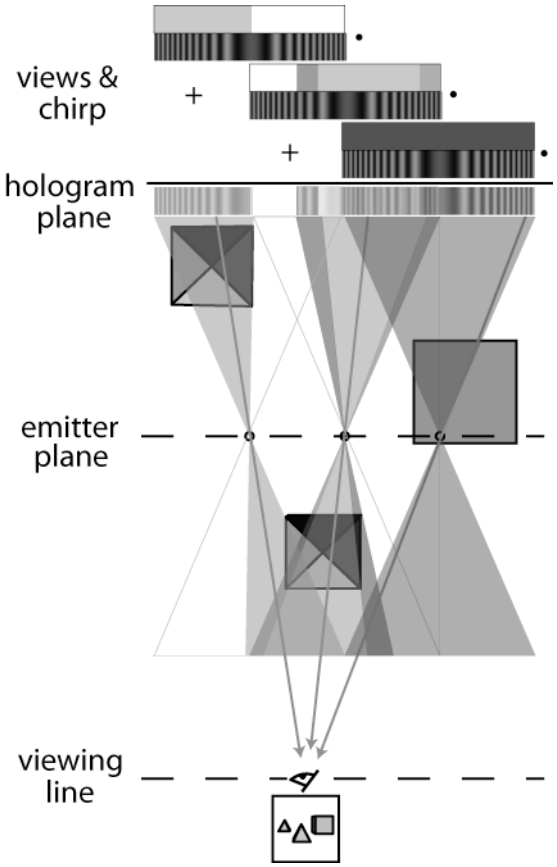

Fig. 1. Diffraction specific holographic stereogram: parallax views modulate chirps and sum to form hologram made up of view-dependent point emitters. Views are captured from emitter plane using a double frustum geometry.

Parallel vector computation of the hologram involves three main steps: 1) precomputing the chirp vectors into a chirp texture, 2) multi-view rendering of the scene directly into the modulation vectors stored in the modulation texture using a double frustum camera, and then 3) assembling the hologram by gathering chirp and modulation vectors via texture fetches and then performing a dot product.

The previous GPU implementation of the diffraction-specific holographic stereogram modulated each basis separately then summed them in the accumulation buffer. This is essentially a scalar scatter operation. In the application of data parallelism and vector processing, we change this to a vectorized gather operation in a fragment shader.

To produce the hologram, each chirp is multiplied by its view modulation information, then all the overlapping chirps are added together point by point. This computation of the final value of each hologram pixel is a dot product of a chirp vector with a modulation vector. Four component vectors (float4) are an intrinsic class and the dot product of two four-component vectors is an intrinsic function in Cg. If we choose to have the number of overlapping chirps be a multiple of four, we can efficiently use the four-component vector. Dot products would be done on pairs of four component vectors, then the results summed to produce the final hologram point's value.

In order to parallelize this operation, each point in the hologram plane must be able to compute its value independently from every other point in the hologram plane. Each pixel in the hologram needs to know its position and view number in each of several overlapping chirps. The chirps' positions select what is packed into the chirp vector (the values of the overlapping chirps at that point), and the chirps' view numbers select what is packed into the modulation vector (the brightness of the corresponding view produced by the overlapping chirps at that point). With careful attention, the parameters and data format can be chosen to remove the overhead of packing the data in vector form and of random access memory calls.

To achieve an efficient data format, we must first determine key parameters by following a simple procedure: equally divide hologram plane into abutting chirps, pick the number of overlapping chirps per abutting chirp (preferably divisible by four so they fill four-component vectors), then pick the number of views per chirp.

3.2. Basis functions (chirp vector assembly)

A chirp vector is scene independent but does rely on its position on the hololine. If the stereogram is horizontal parallax-only (HPO), individual hololines are also independent of each other. Only the chirp texture on one hololine needs to be computed. Since they do not depend on the scene, the chirp texture can be precomputed and stored as a 1-D texture to be used as a lookup table (or wrapped to form a 2-D texture if there are GPU limits on texture size). Texture values are natively four-component vectors (RGBA or xyzw), so four overlapping chirps can be stored in one texel. If there are more than four overlapping chirps, the remaining chirps can be stored in groups of four as RGBA texels vertically tiled to make a 2-D texture. See Figure 2a.

3.3. Multi-view rendering (modulation vector assembly)

An emitter's brightness from a particular viewing direction is adjusted by amplitude-modulating some section of a chirp. By projecting the view ray back through the emitter to the hologram plane, the chirp and particular chirp section contributing to the appearance of that emitter's view can be found. The modulation is found by rendering the scene from many viewpoints.

In previous work, modulation had been captured from the observer's viewpoint using a shearing-recentering camera. [5] Many captured pixels weren't used because they were outside the display's field of view; the angles are too steep to be reproduced by the display.

Our new approach reduces the number of unused rendered pixels by using a virtual camera that captures the scene from the emitter's viewpoints; the centers of projection of the captured images are positioned at the emitter locations. Since scene objects may straddle the emitter plane, the camera must have a double frustum to capture viewpoints both in front of and behind the emitter plane. With a double frustum, every pixel captured is used. Double frustum cameras have previously been implemented using two opposite facing cameras for full-parallax holographic rendering but not for HPO real-time displays. They also require a pseudoscopic camera, conjugate lighting, flipped normals, and front face culling. [9] We have derived a new projection matrix for a HPO double frustum camera in which the center of projection is in front of the virtual camera position. This new projection matrix is implemented in the vertex shader.

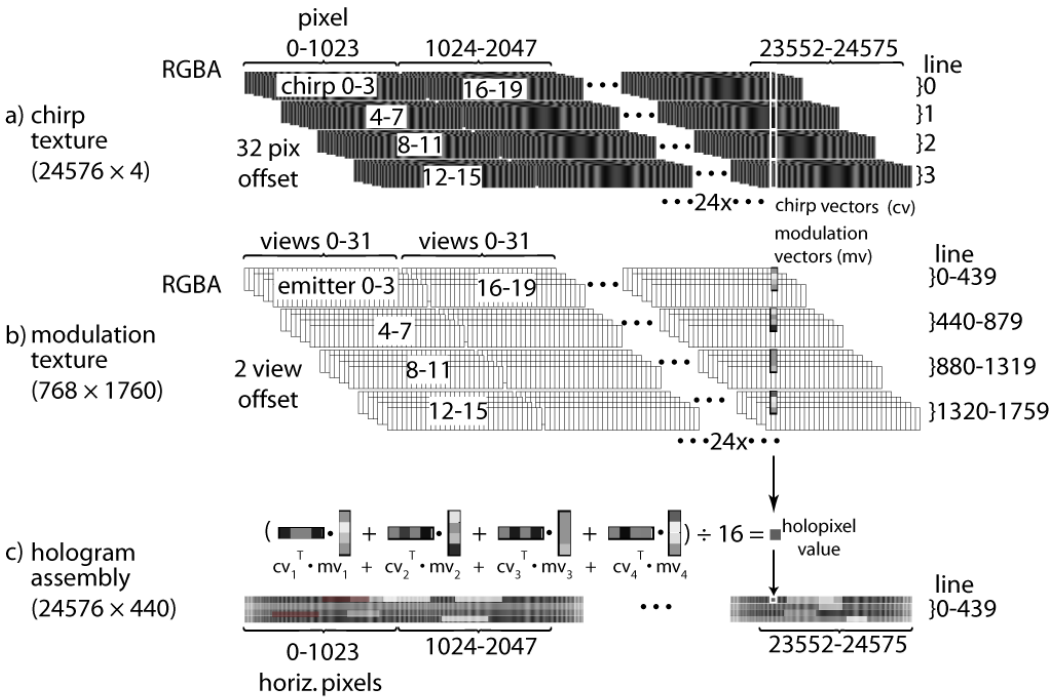

Fig. 2. a) and b): Format of chirp and modulation textures (with 16 chirp overlap, 32 abutting chirps, 1024 holopoints/chirp, 32 views/emitter). c): Parallel vectorized hologram assembly (four chirp and four modulation texture grabs, four dot products shown).

We render the object from a number of viewpoints equal to the number of emitters distributed along a hololine. The horizontal resolution of each rendering is equal to the number of views. The vertical resolution of each rendering is equal to the number of hololines; each captures the view information for a column of emitters. We store four separate horizontal views into the RGBA components of each pixel. The modulation information can be rendered directly to a texture for use by the hologram rendering step. This can be done with frame buffer objects to avoid the slow process of drawing to a frame buffer then copying the contents to a new texture.

When rendering into the modulation texture, the views for the first emitters are color-masked so we render only into the red channel. The next rendering's viewport is shifted horizontally in the texture by a number of pixels equal to the number of views between adjacent chirps. It is also color-masked to be rendered in only the green channel. The following two renderings are horizontally shifted and color-masked to the blue and alpha channels respectively. Subsequent draws follow the same pattern of horizontal shifting and RGBA color-masking, but are also shifted vertically in the 2-D texture by an integer multiple of the number of hololines so that renderings are tiled vertically. This is repeated for the number of overlapping chirps supported at each location (discussion of numerical parameters like this appears in section 5 of this paper). Then the process repeats starting at the top of the texture, so the next rendering is to the right adjacent to the first rendering. See Figure 2b. The multi-viewpoint rendering stage's fragment shader allows us to keep track of the destination vector component (color masking) and allows us to render directly to the alpha channel when needed.

This rendering procedure and data format were chosen to enable sequential data access while assembling the hologram. The view vectors are directly rendered in the appropriate vector format without the overhead of collecting views from different texture locations.

This is an image-based algorithm. As such, each rendering can be visually complex using texture and lighting effects. Computer gaming graphics with real-time interaction are moving away from geometric complexity with large polygon models and towards pixel complexity on low polygon models using texture mapping, normal mapping, and so on to achieve high visual impact at lower computational expense. The per-fragment lighting, texture mapping, and normal mapping are done in the fragment shader.

3.4. Hologram rendering (chirp and modulation dot product)

To produce the final hologram, a single screen-aligned polygon is created. The texture coordinates of the polygon are in normalized coordinates between [0-1] in both the horizontal and vertical directions. Each fragment produced by that poylgon has an interpolated *(u,v)* texture coordinate. Using this *(u,v)* texture coordinate, the fragment shader fetches the chirp vectors and their corresponding modulation vectors from their respective textures. The chirp vectors and modulation vectors are in the same format in normalized coordinates. The fragment shader then performs dot products on corresponding pairs of four component vectors then sums the results. This sum is normalized by the number of overlapping chirps to produce the final hologram point's value. See Figure 2c. This process uses high precision floats so it should avoid any overflow problems from adding multiple chirps, which was a problem in earlier diffraction-specific algorithms. It also doesn't require a multipass algorithm or an accumulation buffer.

4. ADDITIONAL ISSUES

4.1. Double frustum

The use of a double frustum allowed us to render the scenes directly to a format conducive to rapid efficient retrieval in vector processing. The double frustum renders the scene from the viewpoint of the emitters; however objects may straddle the emitter plane. This allows images to appear to float in space and straddle or appear in front of the holographic display.

We derived a double frustum perspective matrix in which the horizontal center of projection is located in front of the viewpoint. As we are performing HPO rendering, the perspective matrix will be astigmatic; it will have a different center of projection in the horizontal and vertical direction. Horizontally, the center of projection is located at the emitter plane. Vertically, the center of projection is located at the viewer line so the objects appear to have the correct vertical perspective. In our implementation we collect views not aligned with the optical axis, therefore we need to support virtual cameras with asymmetric (skewed) view frustums. We derived an asymmetric astigmatic double frustum perspective matrix:

$$Q = \begin{bmatrix} \frac{2(f+p)}{X_{right}-X_{left}} & 0 & \frac{X_{right}+X_{left}}{X_{right}-X_{left}} & \frac{X_{right}+X_{left}}{X_{right}-X_{left}}p \\ 0 & -\frac{2p}{Y_{top}-Y_{bot}} & 0 & 0 \\ 0 & 0 & \frac{f+n+2p}{f-n} & \frac{2fn+p(f+n)}{f-n} \\ 0 & 0 & 1 & -p \end{bmatrix}$$

f : far clip plane — X_{left} : left clip plane
n : near clip plane — X_{right} : right clip plane
p : center of proj — Y_{top} : top clip plane
Y_{bott} : bottom clip plane

(1)

There are two centers of projection but only one perspective divide by w is used in the standard 4 x 4 homogenous perspective matrix. We can premultiply the Y components by w_1/w_2, so when the w_1 division is performed, only the $1/w_2$ remains. This can be done using the vertex shader.

The view frustum cannot contain the center of projection (COP) singularity. To avoid the inclusion of the singularity, two renderings must be taken: one with the far clipping plane slightly before the horizontal COP (h-COP), and another with the near clipping plane slightly after the horizontal COP. Another option is to move the frustum vertex slightly behind the h-COP with the far plane at the h-COP, then take another picture with the frustum vertex slightly in front of the h-COP with the near plane at the h-COP. This option is similar to Halle's solution of overlapping near clipping planes at the COP to avoid an imaging gap between the near plane and the COP of the double-frustum camera. [9] The use of overlapping clipping planes at the COP also means the COP is not a point, but an arbitrarily small area or aperture, but this generally produces no noticeable imaging problems. [9] Since vertices do flip as they go through the COP,

the definition of vertex ordering (CW vs. CCW) must be changed for the two pictures, which is done with a simple OpenGL command.

The use of an astigmatic double frustum made it possible to render directly to a format amenable to vector processing. Unfortunately, the transformation is only done per-vertex. Pixel values are linearly interpolated between vertices in screen space. For astigmatic perspective, straight lines in world space may map to curved lines in screen space. With linear interpolation between vertices, the object may appear distorted in screen space. Hyperbolic interpolation (which takes into account the depth of the vertex) is the correct form of interpolation to use here. While hyperbolic interpolation is used for texture interpolation, linear interpolation is used for the rasterization of primitives. For highly tessellated objects, this won't be a problem, as there will be very few fragments between vertices. Also, in these relatively low resolution renderings, the possible effects should not be too visible. We did not perceive any distortion due to the astigmatic perspective during initial tests using a rudimentary lightfield viewer to reconstruct the scenes from the viewer's vantage point.

4.2. Normal mapping

The use of an image-based rendering algorithm has many advantages; for example, occlusion and lighting effects are automatically handled. The algorithm directly fits into the 3-D graphics pipeline and allows the algorithms, architectures, and hardware advancements used in the graphics field to be leveraged. The visual complexity of the hologram can be easily enhanced using standard 3-D graphics techniques. Shadows, refraction, particle effects, and so on can be readily added, although at the cost of speed.

We use texture and normal mapping to improve the visual appearance of our rendered objects. Normal mapping maps the normals of a high polygon model to a *(u,v)* mapped normal texture of a low polygon model. The low polygon model is rendered using per-pixel lighting, and rather than interpolated normals, the normals from the normal map are used. Surface color is also computed using a *(u,v)* mapped texture map and lighting model. The low polygon object will respond similarly to lighting conditions as the high polygon model does. The silhouette of the object will appear to be that of the low polygon model.

Since we are currently presenting a monochromatic hologram, the texture map needs only one channel. The normal map requires three channels, one for each component of the normal *(x,y,z)*. OpenGL textures are natively handled as RGBA. Therefore, we can combine the normal and texture map into a single map, with the normal components in the RGB channels, and the texture store in the A channel. This allows us to use and load a single texture for both texture and normal mapping.

We implemented the texturing and normal mapping with the fragment shader.

5. IMPLEMENTATION

5.1. Specifics

When implementing the vectorized diffraction-specific holographic stereogram algorithm for use with the Mark III holo-video display, the capabilities and limitations of the video drivers, graphics card and RF electronics must be taken into account. The maximum viewport dimensions of the video drivers, the six channel baseband outputs of the graphics card, and the use of I/Q quadrature chirps needed by the RF electronics [10] are the three factors which greatly affect modifications to the hologram algorithm.

The Mark III holo-video display currently is driven by a 2.13GHz Dell Precision 390 PC tower with a Quadro 4500FX (circa 2006), running Ubuntu. The 1.2GHz bandwidth of a Quadro 4500FX graphics card creates an 80mm x 60mm HPO hologram with a resolution of 336 pixels x 440 lines x 96 views, 30fps refresh, and 24° FOV. A dual head graphics card provides six 200MHz baseband channels (two sets of RGB) from 400MHz RAMDACs. 167MHz of each of those channels is encoded with view/frequency multiplexed holographic information, and a 33MHz guardband to allow for gentle filtering of reconstruction images produced by the RAMDACs.

The current design has three adjacent 8° viewzones each produced by one of three 400MHz precomputed Weaver I/Q pairs generated by the graphics card. [10] Chirps are double sided sine/cosine quadrature pairs. Head 1 RG channels produce the sine/cosine pair of zone 1. Head 1 B channel produces the sine of zone 2. Head 2 RG channels produce the sine/cosine pair of zone 3. Head 2 B channel produces cosine of zone 2. The baseband signals are upconverted to their appropriate frequency ranges so the combination of abutting bands forms a contiguous 1 GHz signal from 200-1200MHz.

For this hologram, each channel has 440 visible hololines and 24576 total samples per hololine. We divide the hololine into 24 abutting chirps with 1024 pixels per chirp. We choose 16 overlapping chirps per abutting chirp. We choose 32 views per chirp, giving us two views between adjacent chirps. All views change together every 32 pixels on the hololine. To tally the number of emitters, we must remember that there are three viewzones, so three subsequent chirps in separate video channels contribute to the total field of view of each emitter. Accounting for three complete viewzones per emitter, since there are 24 abutting chirps and 16 overlapping chirps per abutting chirp, there are a total of (24 – 3) x 16 = 336 emitters.

When used with normal video monitors (as opposed to what we're doing) the maximum viewport dimensions for nVidia's Gen 7 series cards such as the Quadro 4500FX are 4096 x 4096 pixels. The holo-video line was chosen to be evenly divisible by 4096, so that the 24576 pixels per hololine could be split into six video lines. This leads to a 4096 x 2640 (440 x 6 = 2640) pixel six channel hologram. For Gen 7 series cards, we were unable to configure nVidia's TwinView multi-monitor drivers for a single window spanning two monitors with both having hardware acceleration. We instead must run two X-Windows servers, each driving a separate 4080 x 3386 (viewable pixels) window on a different monitor, with 16 horizontal pixels used as horizontal blanking, and 764 vertical lines being used as vertical blanking and sync.

A Modeline describes the timing signals and resolution of the graphics card output. A custom Modeline is designed to make a 4080 x 3386@29.9fps visible window. The Modeline is designed to minimize horizontal and vertical front/back porches and syncs while still having an 80% vertical duty cycle to accommodate the flyback time of the Mark III display's vertical mirror. Small, unavoidable sync/blanking intervals within a hololine result in partial missing views, but our experience with the same situation in earlier work shows that these gaps aren't generally perceptible. [4]

The holographic rendering program first precomputes the chirp functions for one hololine and stores them in a texture. The texture is 4096 pixels wide to match the window's total horizontal pixels. The texture is 6 x 4 x 2 = 48 pixels tall because each hololine is split into six video lines and there are 16 overlapping chirps tiled into four vertical sections; sine and cosine pairs are tiled vertically as well.

The holographic renderer loads the .obj model and its corresponding texture and normal maps in 32 bit .bmp format. The .obj model is converted into a drawlist of OpenGL draw commands. The model is free to translate or rotate programatically. For tests described in this paper, a 10122 polygon .obj model of the Stanford Bunny [11] was decimated to 500 polygons. The decimated model was *(u,v)* unwrapped using Wings3D and a custom texture was applied in Photoshop. The normals of the 10122 polygon model were burned into a normal map of the 500 polygon version using nVidia's meLODy. Figures 3a and 3b compare interpolated-normal versus normal-mapped rendered models. The texture map and normal maps were combined to form a single combo tex/norm map as a 32 bit .bmp image.

The renderer renders three arrays of 336 emitters at 32 x 440 pixels with an asymmetric frustum, one for each viewzone. The renderer uses a frame buffer object to render direct-to-texture. Each viewzone is rendered separately with its appropriate double frustum projection matrix and tiled horizontally. For each emitter, the scene renders with per pixel lighting, texture mapping, and normal mapping. The renderer offsets each rendering horizontally in the texture by two pixels from the previous rendering. We render in sets of 16, matching the number of overlapping emitters. Each rendering in a subset of four is rendered to RGBA. Each subset is offset 440 pixels vertically. When the set of 16 completes, we start again at the top of the texture. At this point, the rendering will be offset 32 pixels horizontally and the renderings will abut.

To create the final hologram, a single polygon is drawn using screen-aligned coordinates. The texture coordinates map directly to the chirp's texture map once they are horizontally scaled by 4080/4096 in the vertex shader to account for only visible pixels. Due to the formatting of the textures, a single texel grab has the correct components in the RGBA channels. Four texel grabs along the same column provide the 16 components needed for the amplitude modulation.

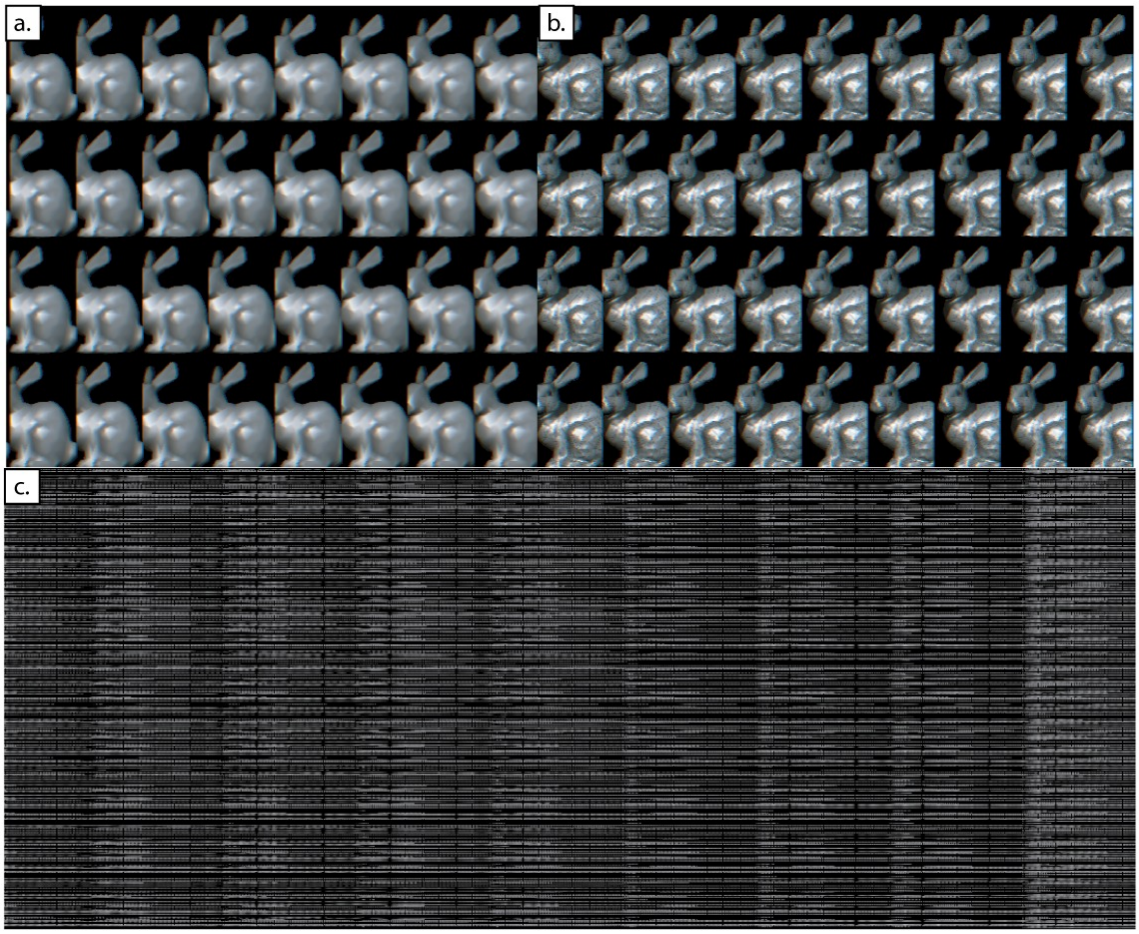

Fig. 3. a): Multiview rendering of 500 polygon model with no texturing. b) Rendering with texture-normal mapping. c) Portion of corresponding holographic stereogram data pattern.

Likewise, four pixel grabs in the precomputed chirp lookup table provide the corresponding 16 components of the chirp basis function. Four dot products and three sums produce the final output pixel value. This is done in the fragment shader.

Two programs run concurrently using two separate X-Windows servers and windows, one for each head. We assume the heads update at same time since they are on the same card. A 336 pixel x 440 line x 96 views hologram runs at 10 fps (two heads) for a texture mapped model with 500 polygons with per-pixel normal mapped lighting (equivalent to a 10122 polygon model). See Figures 3b and c. If the modulation texture is prerendered, or the multi-view rendering and modulation vector assembly is performed once and the texture reused, the hologram is constructed at 15 fps.

5.2. Discussion

The previous implementation of the diffraction-specific holographic stereogram algorithm on a GPU achieved two frames/second on an eight vertex model using three video cards. The total output bandwidth was equal to the 1GHz bandwidth of the current implementation. From a rough perspective, based on the number of polygons, frame rate, and number of GPUs, we have increased the performance by over a factor of 1000 using only one graphics card and imple-

menting normal-mapping and per-pixel lighting for greater visual effect. This can not be purely attributed to the rapid increase in GPU computational power, as we would expect a rough factor of 16 performance increase for the two year gap between the graphics cards. By recasting the algorithm into a vector parallelized form that could be implemented using vertex and fragment shaders, we have further increased the efficiency of the algorithm to achieve interactive update rates for modest size models.

Although we have tried to adhere to general principles to achieve high performance in GPUs, ultimately the program needs to be profiled to find its bottlenecks and optimize its performance. Although we have not performed profiling yet, we believe we are currently fill-rate limited; there are just too many pixels to draw. Because we have a set number of pixels to render in order to compute the modulation and construct the hologram, the only option to break the fill-rate bottleneck is to have a GPU with a larger fill-rate.

Fortunately, GPUs continue rapidly to increase in speed – every six months they roughly double in performance – and at the time of this writing our GPU is over two years old. The Quadro 4500FX has a texel fill rate of about 11 Gtexels/second. The current top-of-the-line nVidia GeForce 280 has a texel fill rate of 48 Gtexels/second. We feel this alone will probably allow us to achieve at least interactive rates (about 10 fps) and perhaps realtime rates (30 fps) on texture/normal mapped models ranging in the 1.5K polygon range, which is typical for a single model in PS2 generation 3D games. We also expect real-time frame rates when using pre-rendered modulation textures.

Further modifications can also improve the efficiency of our algorithm and better align it to the general principles we described above. For instance, in assembling the hologram, four chirp texture calls, four modulation texture calls and four dot products are required. We envisioned storing the precomputed chirp functions in textures to use as lookup tables as a way increase the efficiency of the algorithm. However, due to the latency from texture fetches, it may be more efficient to compute the chirps while the four modulation texels are fetched, thus hiding the texture fetch latency, though at a possible increased computational cost.

Multi-view rendering is likely another area that could be optimized, especially when larger models are used. Each time the model is rendered, the model vertices must pass through the pipeline, even though they do not change from view to view. Geometry shaders can duplicate primitives and apply a different view transform (Eqn. 1) to each, thus achieving multi-view rendering in a single pass. [12] Unfortunately, current geometry shaders have a limit on the number of new primitives that can be spawned, which may limit the usability of this technique in our case. Each view is also rendered to a different render-buffer. At the present time, there are only a maximum of eight render buffers allowed.

We will need to profile the algorithm to determine if the bottlenecks occur during any of the above processes to determine if these possible solutions would be beneficial.

6. CONCLUSIONS AND FUTURE WORK

The era of interactive holographic displays is upon us. We have developed a parallel-vector CGH algorithm for commodity hardware which can render holographic stereograms of useable dimensions and large view numbers from modest sized models at interactive frame rates. The use of image-based multi-view rendering, especially with texture mapping and normal-mapped per-pixel lighting, has given us compelling visual complexity. Implementation using a single commodity graphics card and standard 3-D APIs brings this capability to the average user and opens up new possibilities for interactive holographic applications.

Currently we are running two programs, one for each head. The maximum viewport dimensions are limited to 4096 x 4096 textures in 7 Series nVidia GPUs. Since the introduction of 8 Series nVidia GPUs, the maximum viewport dimensions have increased in size to 8192 x 8192 and allow one program with one window spread over two 4096-pixel-wide screens. We are making minor modifications to the fragment shader for hologram assembly, so the hologram for the first head would be displayed on the right side of the window and the hologram for the second head would be displayed on the left side of the window. Since both heads are under control of one program, the second viewzone would only need to be rendered once.

We could increase the number of horizontal emitters from 336 to 672 emitters. In the modulation texture, there are currently two views between adjacent chirps. By rendering views offset one pixel apart, we could double the number of horizontal emitters per hololine. By doing this we could achieve a 672 x 440 x 96 views display – approximately standard definition TV (SDTV) resolution.

The multi-view rendering can be prerendered and the modulation texture precomputed offline (and possibly compressed). The modulation texture is fairly efficient being only slightly larger than all the views required to reconstruct the hologram; and thus could serve as a standard file-format for holo-video images. Storing the holo-video fringe pattern itself would be wasteful, since the fringe pattern is highly redundant. The modulation textures could then be loaded (and possibly decompressed), the number only limited by texture memory, which is in the gigabyte range in today's cards. Only the hologram assembly would be required at runtime; it currently runs at 15 fps, but could easily perform faster on newer cards with higher fill-rates. The precomputed modulation texture could also hold real-world images from camera arrays or other lightfield cameras that had been processed to adhere to the format requirements.

Full color holograms have been produced by a modified version of MIT's first generation holo-video display, the Mark I. Modifying the diffraction-specific algorithm to produce color holograms for the Mark III is straightforward. The color would be line sequentially multiplexed, so the first hololine would be the red channel, the second hololine the green channel, and the third hololine the blue channel. The use of line-sequential color multiplexing would avoid the rainbow tearing that occurs for animated objects in displays using frame-sequential color multiplexing. The texture and material properties of the scene would need to be changed appropriately for each color rendered. The chirps would have to be modified so the different wavelengths diffract through the same angle. The hardware would also need to be synchronized alternately to flash red, green and blue light synchronized to the color the hololine was displaying.

Modifying the diffraction-specific algorithm to produce full-parallax holographic stereograms for the Mark III is also straightforward. Although the Mark III uses a 2-D light modulator, the horizontal axis is used for holographic rendering and the vertical axis is used for Scophony descanning. Because of this, the current implementation is inherently HPO. One way to make a full-parallax scene is to multiplex the vertical views in the hololines and use a horizontally oriented lenticular array to disperse the views vertically.

A diffraction-specific algorithm could also produce a full-parallax holographic stereogram for 2-D light modulators by using 2-D zone plate basis functions instead of 1-D chirps, rendering the scene using a double frustum camera from a 2-D array of emitter positions rather than from a 1-D line, then performing summations of dot products for zone plate vectors and modulation vectors. The view rendering could still be directly rendered to a texture in the appropriate format, although the texture might need to be a 3-D texture or a massively tiled 2-D texture.

7. ACKNOWLEDGMENTS

The authors would like to acknowledge Michael Halle for his insightful correspondence on astigmatic/anamorphic rendering. This work has been supported by the Digital Life, Things That Think, and CELab consortia and the Center for Future Storytelling at the MIT Media Laboratory.

REFERENCES

1. D. Smalley, Q. Smithwick, and V. M. Bove, Jr., "Holographic Video Display Based on Guided-Wave Acousto-Optic Devices," *Proc. SPIE Practical Holography XXI,* 6488, 2007.
2. N. Masuda, *et al.*, "Computer Generated Holography using a Graphics Processing Unit," *Optics Express*, 14(2), pp. 603-608, 2006.
3. I. Hanak, M. Janda, and V. Skala, "Full-Parallax Hologram Synthesis of Triangular Meshes using a Graphical Processing Unit," *Proc. IEEE 3DTV Conference*, pp. 1-4, 2007.
4. W. Plesniak, M. Halle, V. M. Bove, Jr., J. Barabas, and R. Pappu, "Reconfigurable Image Projection Holograms," *Optical Engineering*, 45(11), 2006.

5. V. M. Bove, Jr., W. J. Plesniak, T. Quentmeyer, and J. Barabas, "Real-Time Holographic Video Images with Commodity PC Hardware," *Proc. SPIE Stereoscopic Displays and Applications,* 5664A, 2005.
6. D. H. Quynh, "Graphics Hardware and Shaders," http://www.cs.stevens.edu/~quynh/courses/cs537-notes/hardware_shaders.ppt, 2006.
7. C0DE517E blog, "How the GPU works – Part 3 (optimization)," http://c0de517e.blogspot.com/2008/04/how-gpu-works-part-3.html, April 24, 2008.
8. I. Buck, "Taking the Plunge into GPU Computing," *GPU Gems*, http://http.developer.nVidia.com/GPUGems2/gpugems2_chapter32.html
9. M. Halle, "Multiple viewpoint rendering," *Proc. 25th Annual Conference on Computer Graphics and Interactive Techniques,* pp. 243-254, 1998.
10. Q. Y. J. Smithwick, D. E. Smalley, V. M. Bove, Jr., and J. Barabas, "Progress in Holographic Video Displays Based on Guided-Wave Acousto-Optic Devices," *Proc. SPIE Practical Holography XXII,* 6912, 2008.
11. Stanford 3D Scanning Repository, http://www-graphics.stanford.edu/data/3Dscanrep/
12. F. Sorbier, V. Nozick, and V. Biri, "GPU Rendering for Autostereoscopic Displays," *Proc. Fourth International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT)*, 2008.