Computer generated holography using parallel commodity graphics hardware

Lukas Ahrenberg
Max-Planck-Institut für Informatik, Saarbrücken, Germany
ahrenberg@mpi-inf.mpg.de

Philip Benzie
Communications and Optical Engineering, University of Aberdeen, Scotland

Marcus Magnor
Institut für Computergraphik, Techische Universität Braunschweig, Germany

John Watson
Communications and Optical Engineering, University of Aberdeen, Scotland

Abstract: This paper presents a novel method for using programmable graphics hardware to generate fringe patterns for SLM-based holographic displays. The algorithm is designed to take the programming constraints imposed by the graphics hardware pipeline model into consideration, and scales linearly with the number of object points. In contrast to previous methods we do not have to use the Fresnel approximation. The technique can also be used on several graphics processors in parallel for further optimization. We achieve real-time frame rates for objects consisting of a few hundred points at a resolution of 960×600 pixels and over 10 frames per second for 1000 points.

OCIS codes: (090.0090) Holography; (090.1760) Computer holography; (090.2870) Holographic display.

References and links

1. M. Lucente, "Diffraction-Specific Fringe Computation for Electro-Holography," Ph. D. Thesis, Department of Electrical Engineering and Computer Science," Ph.D. thesis, Massachusetts Institute of Technology (1994).
2. J. Watlington, M. Lucente, C. Sparrell, V. Bove, and J. Tamitani, "A Hardware Architecture for Rapid Generation of Electro-Holographic Fringe Patterns," in *SPIE Practical Holography IX*, vol. 2406, 172–183 (1995).
3. M. Lucente and T. A. Galyean, "Rendering interactive holographic images," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, 387–394 (ACM Press, New York, NY, USA, 1995).
4. T. Ito, N. Masuda, K. Yoshimura, A. Shiraki, T. Shimobaba, and T. Sugie, "Special-purpose computer HORN-5 for a real-time electroholography," Opt. Express **13**, 1923–1932 (2005). URL http://www.opticsexpress.org/abstract.cfm?uri=OE-13-6-1923.
5. A. Ritter, J. Böttger, O. Deussen, M. König, and T. Strothotte, "Hardware-based rendering of full-parallax synthetic holograms," Appl. Opt. 1364–1369 (1999).
6. C. Petz and M. Magnor, "Fast Hologram Synthesis for 3D Geometry Models using Graphics Hardware," in *Practical Holography XVII and Holographic Materials IX*, 266–275 (SPIE, 2003).
7. J. V. Michael Bove, W. J. Plesniak, T. Quentmeyer, and J. Barabas, "Real-time holographic video images with commodity PC hardware," in *Proc. SPIE Stereoscopic Displays and Applications*, vol. 5664, 255–262 (SPIE).
8. N. Masuda, T. Ito, T. Tanaka, A. Shiraki, and T. Sugie, "Computer generated holography using a graphics processing unit," Opt. Express **14**, 603–608 (2006). URL http://www.opticsexpress.org/abstract.cfm?uri=OE-14-2-603.

9. M. Lucente, “Interactive Computation of holograms using a Look-up Table,” J. Electron. Imaging **2**(1), 28–34 (1993).
10. “Example movies,” URL http://www.mpi-inf.mpg.de/departments/irg3/ahrenberg/animations/holongpu.
11. “Stanford 3D Scanning Repository,” URL http://graphics.stanford.edu/data/3Dscanrep/.

1. Introduction

Recent advances in three dimensional display technologies have shifted attention to the possibility of true holographic displays. One possible way of realising this is through the use of Spatial Light Modulators (SLMs). The performance and resolution of SLMs are increasing rapidly, consequently, holographic displays of increasing spatial bandwidth product can be built. To drive these, Computer Generated Holograms (CGHs), rendered from point models are commonly used. However, generating a hologram from a set of point samples is a computationally intensive task. In this paper we propose a method that takes advantage of parallel graphic processing units (GPUs) to perform the computation. Although development of special purpose hardware for CGH rendering has been reported, this type of equipment is expensive and must be custom built. In contrast, graphic boards are readily available today, and are specially constructed to accelerate numerical operations frequently used in computer graphics. Thus, a parallel GPU system is an attractive alternative to both expensive custom-built hardware and to much slow CPU-based approaches.

Holographic rendering hardware was presented in 1994 by researchers in the MIT Media group [1, 2, 3]. This implementation was reported to be approximately 50 times faster than a workstation of its day. Another hardware architecture, HORN, has been developed at the Department of Medical System Engineering at Chiba University [4]. Using a clever iterative recurrence formula, as well as an architecture allowing several boards to be coupled in parallel, the system is reported to be about 1000 times faster than a CPU implementation.

Computer generated holography using the so called fixed-function (non-programmable) pipeline on a graphics workstation was proposed in 1999 by Ritter et al. [5]. Petz and Magnor implemented a similar method on commodity graphics hardware [6] in 2003. These algorithms precompute the complex valued Fresnel zone-plate of an object point and store it as a texture. The CGH is then computed by repeatedly rendering scaled and translated versions of this texture for each object point while accumulating the result. Both techniques applied state-of-the-art graphics hardware which at that time did not allow for much programming freedom, hence the use of precomputed zone plates as approximations. In [7] Bove et al. presents two algorithms to compute horizontal parallax only stereograms using graphics hardware. The first method uses a hardware accumulation buffer, available on high end graphics cards, to compute stereograms, while the second is an interference-based approach using textures and blending to accumulate projection fringes. Both methods use fairly modern GPU functionality but are are designed for a fixed-function pipeline. In contrast to the above three papers we make use of the programmable GPU functionality and are not so limited, allowing for a more accurate algorithm to be implemented.

Very recently Masuda et. al presented a GPU implementation for CGH computation [8]. By implementing CGH calculations in a GPU-program that is executed once per output pixel (a so called fragment program), they managed to achieve video frame rates for a hologram image size of 800×600 pixels. However, their method is restricted by the number of GPU registers to a maximum of 100 object points. Although the method presented in this paper is implemented on similar hardware (their GPU was an nVIDIA GeForce 6600, our’s an nVIDIA GeForce 7800) the algorithms differ significantly. Our method does not suffer from a fixed maximum number of points as we have tailored the algorithm to adhere to the programming model imposed by the GPU. In order to speed up the computations by parallelizing multiple

GPUs for CGH rendering, we also use nVIDIA's Scalable Link Interface (SLI). This technique is also mentioned as a future improvement in [8] but not implemented.

This paper is organized as follows. In Sect. 2 we give a brief introduction to GPU architecture. In Sect. 3 we review the preliminaries for CGH computations, followed by a presentation of our implementation in Sect. 4. Sect. 5 presents results and performance tests, and we conclude our work in Sect. 6.

2. Programmable graphics hardware

Special-purpose graphics hardware was developed to perform the computationally intensive transformations needed by many of today's CAD and computer games applications. The GPU architecture is based around the *rendering pipeline*, a step-by-step sequential approach allows for a very fast, and at some stages, parallel hardware implementation. On modern GPUs the pipeline is also programmable. This allows software routines, so called shaders, to be downloaded to the graphics card and inserted to bypass fixed functionality at certain places in the pipeline. The programs can be written in a high-level language and use such features as loops and conditional branching. It is important to note however, that although the shaders are written in high level programming languages and allow for much freedom, they are still just part of the graphics pipeline, and have to process data in an input-output fashion. Thus, many problems and algorithms have to be reposed to be efficiently implemented on a GPU. Although this might seem restrictive from a programming point of view, it is also what makes the GPU a special-purpose processor and accounts for much of its increased computational efficiency over standard CPUs. Therefore, special care must be taken when using a GPU for general purpose problems. Complex or ill-posed shaders can clog the pipeline down and thus actually slow down the processing speed.

A huge class of algorithms, for instance integrals and sums, rely on repeated processing of the data, i.e. looping. This also includes the CGH algorithm. Whilst loops are supported within the newer shader programs, they can only be used in a rather restrictive manner than when programming a CPU. Due to compiler and hardware limitations it is only possible to loop a few times while performing operations that make heavy use of GPU registers. A different approach than to perform the loop inside a shader is to repose the problem so that it uses multiple passes of the entire graphics pipeline. That is, the result is computed by repeatedly running the entire graphics pipeline and summing up the results. One such pass will create some overhead, as many operations of the pipeline may have to be executed that are not needed for the computation. However it allows for many more sequential operations than a loop in a single shader. In Sect. 4 we will present how we use both in-shader looping and multiple passes to create a fast and extendable CGH rendering method.

3. The CGH model

Using the real bipolar model for a computer generated hologram [9], we have the following expression for an intensity distribution

$$I(x,y) = \sum_{j=1}^{N} a_j \cos\left(\frac{2\pi}{\lambda}\sqrt{(x-x_j)^2+(y-y_j)^2+z_j^2}\right). \tag{1}$$

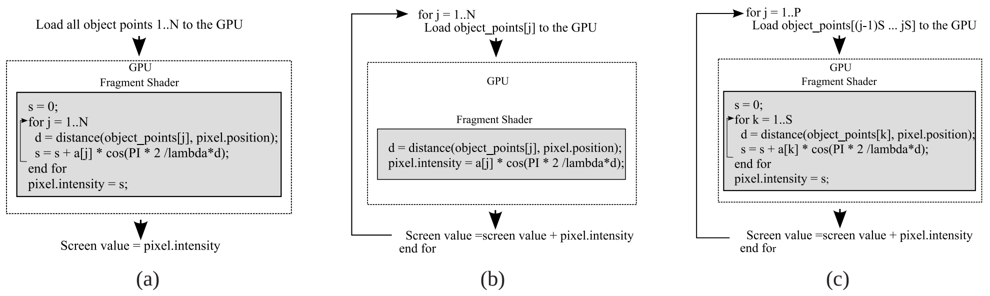
I is the intensity in the hologram plane at $z=0$, N the number of object points and λ the wavelength. (x_j, y_j, z_j) is the object point coordinates and a_j the intensities.

A further common simplification of (1) is to use the Fresnel approximation, i.e, the object size is assumed much smaller than the distance between the object and the hologram plane; $(x_{\mathrm{max}}, y_{\mathrm{max}}) \ll z_{\mathrm{object}}$. This approximation is made for numerical reasons since square roots can be computationally expensive. However, as distance computations occur frequently in computer graphics algorithms it is reasonable to assume that GPUs are optimized for these operations. This is also confirmed by practical experimentation, and thus we work directly with Eq. 1 and do not have to restrict ourselves to the case of Fresnel diffraction.

4. Implementation

In this section we will discuss how we implement (1) given the programming constraints discussed in Sect. 2. The type of computation hardware used is an nVIDIA GeForce 7800 GTX graphics card, and a Brillian 1080 reflective spatial light modulator for display. The SLM has a maximum resolution of 1920×1200 elements and a pixel pitch of 8.1 μm. The software was implemented using the OpenGL graphics library and the GPU shaders written in OpenGL Shading Language (GLSL).

Analyzing the CGH equation in the previous section, it is clear that a summation of N terms is needed for each pixel of the SLM. A standard CPU implementation would solve this by a simple loop. On a GPU this would lead to an algorithm like the one outlined in Fig. 1(a) performed for every fragment in the pipeline. Due to the restrictions discussed in Sect. 2, a somewhat different strategy is necessary. Although loops are available on modern graphics hardware, the size is restricted to a few hundred iterations. This is too few for anything but the simplest objects. An alternative, used frequently in computer graphics before looping in shaders was available, is to use multiple rendering passes and a functionality known as *blending*. Using blending, the intensity, as output from sequential executions of the graphics pipeline to the screen image, is combined using a linear operation. This means that the pipeline is run once for every term in the sum and the result is accumulated, see Fig. 1(b). However, this method also has drawbacks as there is in general an overhead associated with running multiple passes of the pipeline. For a small number of points it is thus faster to perform the operations in a shader program.



```
Load all object points 1..N to the GPU
s = 0;
for j = 1..N
  d = distance(object_points[j], pixel.position);
  s = s + a[j] * cos(PI * 2 /lambda*d);
end for
pixel.intensity = s;
Screen value = pixel.intensity
```

```
for j = 1..N
  Load object_points[j] to the GPU
  d = distance(object_points[j], pixel.position);
  pixel.intensity = a[j] * cos(PI * 2 /lambda*d);
  Screen value =screen value + pixel.intensity
end for
```

```
for j = 1..P
  Load object_points[(j-1)S ... jS] to the GPU
  s = 0;
  for k = 1..S
    d = distance(object_points[k], pixel.position);
    s = s + a[k] * cos(PI * 2 /lambda*d);
  end for
  pixel.intensity = s;
  Screen value =screen value + pixel.intensity
end for
```

Fig. 1. Three different algorithmic layouts for computing the distribution from N object points on a GPU. The fragment shader is called once for every pixel. (a) Process all points in one pass. Loop in shader. (b) Process one point per pass. Multi-pass. (c) Process $N = PS$ points. P passes and S summations in shader.

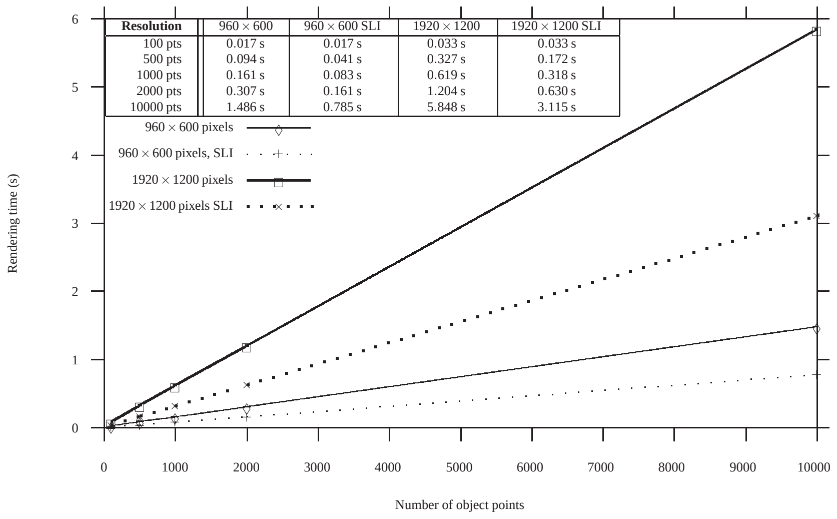
We combine both methods to create an algorithm that can handle an arbitrary number of points while still taking advantage of the GPU for hardware acceleration. Figure 1(c) depicts our algorithm. In short, for a set of N object points we perform S summations in the shader while performing P passes so that $N = SP$. This allows us to find a balance between the number of operations performed in the shader for each pass and the number of passes, thus optimizing performance.

The implementation works with 16-bit floating point precision which can be handled completely in hardware by modern graphics processors. Thus, the method is not as prone to numerical errors due to fix-point 8-bit representation as earlier methods. An extra fragment shader scales the intensities to 256 levels to address the SLM after the computations.

Scalable Link Interface (SLI) is a technology that allows several graphics boards to be connected in parallel in order to increase rendering speeds. This technology automatically distributes the rendering tasks over the GPUs and the load balancing is handled at the driver level. We have confirmed that our implementation works in single GPU mode as well as on a two GPU SLI-equipped system.

5. Results

We have tested our software running on a GeForce 7800 GTX PC connected to a Brillian 1080 SLM. The PC was equipped with an extra GeForce 7800 GTX board for dual GPU SLI rendering. For optical reconstruction we used a 10 mW, 633 nm laser. The maximum resolution of the SLM is 1920 $\times$ 1200 pixels. We have performed tests at both full, and half, 920 $\times$ 600, resolution. The test objects used had from 100 to 10000 points corresponding to a size of approximately 1.2 cm, and were calculated at a distance of 20 cm from the SLM.



Resolution	960 × 600	960 × 600 SLI	1920 × 1200	1920 × 1200 SLI
100 pts	0.017 s	0.017 s	0.033 s	0.033 s
500 pts	0.094 s	0.041 s	0.327 s	0.172 s
1000 pts	0.161 s	0.083 s	0.619 s	0.318 s
2000 pts	0.307 s	0.161 s	1.204 s	0.630 s
10000 pts	1.486 s	0.785 s	5.848 s	3.115 s

Fig. 2. Rendering times 100 to 10000 points for two SLM resolutions using single and dual GPUs. The rendering time scale almost perfectly linear for 100 to 10000 points. Using the SLI setup doubles the performance, with the exception of the 100 point case where the GPU does the computation in just one pass and no efficiency is gained by parallelization.

Figure 2 presents a table and a graph of the times in seconds to render an interference pattern using our software for a few test sets. Note that the rendering times scales almost perfectly linearly with the number of points for both resolutions. With the exception of the object set consisting of just 100 points, where the GPU can do the computations in just one pass, using the SLI setup with two graphic boards effectively doubles the performance. Moreover, as the number of output pixels is quadrupled from the lower 960 $\times$ 600 to the higher 1920 $\times$ 1200 resolution, so is the rendering time. This demonstrates that the algorithm is suitable for graphics hardware. The rendering time is directly dependent on the number of input and output points. There is no extra overhead associated with increasing either the number of points or the resolution of the SLM. As can be seen from Fig. 2 we achieve interactive frame rates for fairly dense sampled objects. Two thousand points are rendered in just 0.6 seconds at full SLM resolution, and in less than 0.2 seconds at half resolution. Smaller objects, in the order of a few hundred points, can be rendered at real time frame rates of 30 frames per second and above. Figure 3 shows a photographs of a off-axis reconstructions. The models have 200, 1800 and 8000 points respectively, and render times of 15, 2 and 0.4 frames per second on a 1920×1200 SLM.

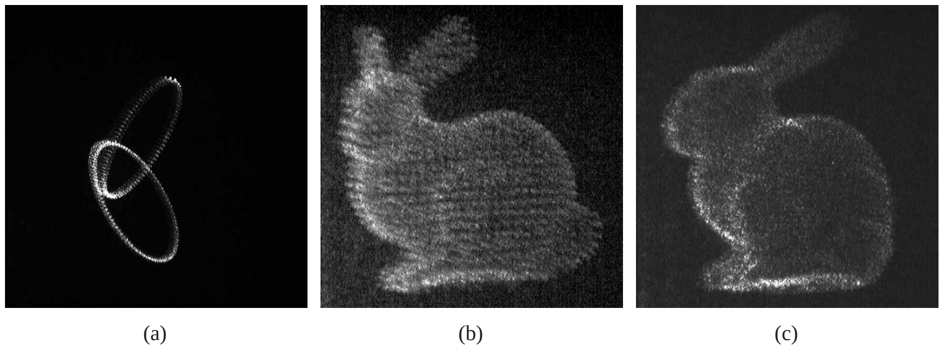


Fig. 3. Off-axis reconstruction. The models has 200 (a), 1800 (b) and 8000 (c) points. Movies of the reconstruction are available at [10]

6. Conclusions and future work

We have presented a novel method for using a commodity graphics processor to generate holographic patterns for SLM-based holographic displays. The algorithm is designed to fit the pipelined model of the GPU and takes high advantage of the parallel architecture. Interactive rates of 10 frames per second are achieved for one thousand object points at a resolution of 960×600 pixels. We have shown that the rendering time increases linearly with the number of points and the SLM pixel count. Thus, the algorithm fits the programming architecture imposed by the hardware, and can be expected to perform proportionally better with future generations of GPUs. We have also shown that the performance can be doubled by using two graphic boards in a parallel configuration. Todays SLI standard allows for up to four GPUs to be configured in this way, which potentially would quadruple the speed.

Future research on this project will consider color, occlusion, more complex primitives and display quality. Since GPUs are designed to work with multi-channel color data, it should be trivial to adopt our program to render three patterns, each one for a different wavelength, without additional computational load. These patterns could then be multiplexed to create a color hologram. Most CGH algorithms usually ignore the problem of occlusion due to the associated computational complexity. However, recently approximate algorithms for similar problems in computer graphics have been proposed. We plan to investigate whether any of these can be adopted to suit CGH rendering. We would also like to look into using of more complex primitives than points to represent objects. Finally, the ability to discern between distinct object points is dependent upon a number of factors such as the point spread, speckle noise, aberration and quantization noise introduced by the display system [1]. These are important factors in determining the effectiveness of our display system, we plan to address the visual and actual resolution of the display in future research.

Acknowledgment

We would like to thank our reviewers for their thoughtful comments and suggestions and also acknowledge Stanford University [11] from where we obtained the bunny model This work is supported by the EC within FP6 under Grant 511568 with the acronym 3DTV.