GPU-Based Multi-view Rendering for Spatial-Multiplex Autostereoscopic displays

Jianli Luo, Kaihuai Qin, Yanxia Zhou, Miao Mao, Ruirui Li
Department of Computer Sciences, Tsinghua University, Beijing, 100084, China
qkh-dcs@tsinghua.edu.cn

***Abstract*—Spatial-multiplex autostereoscopic systems display images interleaved from multi-view to provide adequate viewing zone sections for multiple users. The traditional methods render multiple views in multiple passes. In this paper, we present a GPU-based multi-view render (GBMVR) for spatial-multiplex autostereoscopic displays. It generates multiple views as textures in only one pass through the geometry shader and then interleaves the images through the fragment shader in another pass. It is implemented non-invasively in Chromium so that most traditional OpenGL applications like the game Quake III, without any source-code modification and re-compiling, can run directly on the Chromium enhanced with GBMVR. Configured as the sort-first parallel rendering architecture, our method can also render images for high-resolution spatial-multiplex autostereoscopic systems that display high-resolution images with tiled projectors.**

Keywords-autostereoscopic; interleave; GPU; non-invasive

I. Introduction

Autostereoscopic displays provide immersive 3D perception without the need of glasses or specific devices. They can be used in various applications such as entertainments (games and movies), simulations and virtual reality. Latest spatial-multiplex autostereoscopic systems can display multiple views to provide adequate viewing zone sections for multiple users [1]. These views are either captured by an array of cameras [2] or generated by computer graphics methods [3]. There are data redundancies in the multi-view rendering that affect the rendering efficiency.

In this paper, we present a GPU-based multi-view rendering (GBMVR) method that renders multiple views in only one pass through the geometry shader for spatial-multiplex autostereoscopic displays. To make our approach widely applicable, we have implemented it non-invasively in Chromium [4] to intercept and manipulate OpenGL commands called by the application that is running on the Chromium enhanced with GBMVR. Thus, GBMVR can be applied to the existing single-view applications.

The remainder of this paper is organized as follows: In the second section the related previous works are reviewed; the GPU-based rendering is briefly described in the third section; some results are shown in the fourth section, and the paper ends with the conclusion section.

978-1-4244-5539-3/10/$26.00 ©2010 IEEE

II. Previous Work

A. *Autostereoscopic displays*

Since the turn of the 20th century, researchers have made great progress in autostereoscopic displays. Almost all recent autostereoscopic displays fall into two main groups: the spatial-multiplex display [5] and the multi-projector display [2]. The spatial-multiplex autostereoscopic systems display images that are interleaved from multiple views [6]. In the multi-projector autostereoscopic systems, each view is displayed by one or more projectors simultaneously.

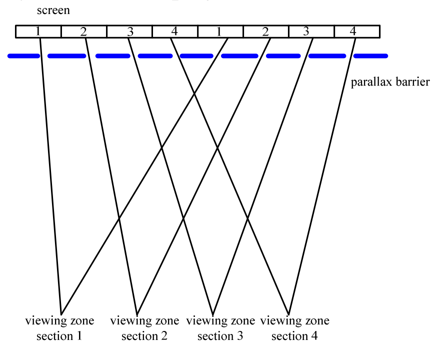

Figure 1. Viewing zone sections formed by parallax-barrier.

The spatial-multiplex display relies on the optical plates such as parallax-barriers to form the viewing zone sections [7]. The parallax-barrier is a raster sheet that consists of arrays of slots. In each of the viewing zone sections, only parts of pixels of the image laid behind the slots can be seen (see Fig. 1). The image is interleaved from multiple views according to a mask. The mask is a two-dimensional array whose size is as same as that of the interleaved image. Each element of the mask, *mask*[*i*][*j*], stores the index of the view. As shown in Fig.2, the pixel of the interleaved image in the *i*-th row and *j*-th column is got from the *mask*[*i*][*j*]-th view image.

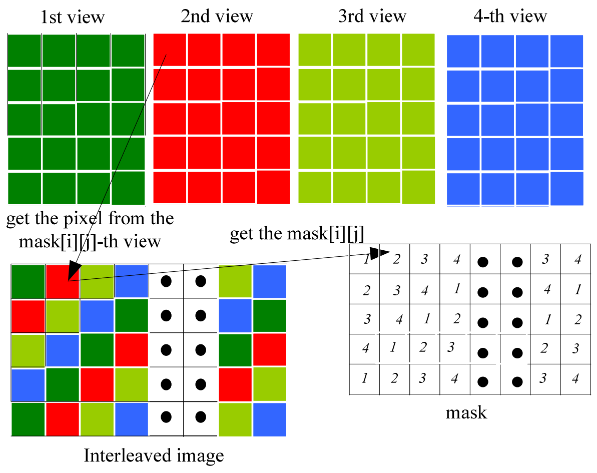

Figure 2. Image interleaved from four views

B. *GPU programming*

The latest nVidia graphics cards provide three programmable units: vertex shader, geometry shader and fragment shader. The geometry shader performs some math computing as the vertex shader. Specifically, the geometry shader can output multiple elements since it can generate new primitives. Thus it can handle one-input and multi-output problems.

GPUs are widely used to speed up the rendering in expensive computing applications. For example, the geometry shader is utilized to slice the projected tetrahedrons for 4D visualization [8]. Sorbier et al. presented a method to generate multiple views into multiple textures via the geometry shader in a single rendering pass [9]. However, their method is invasive and cannot support high-resolution spatial-multiplex autostereoscopic displays. Kooima et al. presented a GPU-based image-interleaving algorithm to interleave multiple views through the fragment shader. Their method need render two passes for the left and right views [10].

C. *Chromium*

Chromium is an open-source non-invasive system for interactive rendering on clusters of workstations [4]. It intercepts the OpenGL commands called by an application through app nodes. These intercepted OpenGL commands are processed by the SPU (Stream Processing Unit). Through SPUs, the developers can modify and replace any of the graphics commands called by the application. Furthermore, Chromium offers the sort-first rendering architecture for high-resolution displays [11].

III. MULTI-VIEW RENDERING

A. *GPU-based multi-view rendering*

GPU-based multi-view rendering methods render interleaved images in two pass. In the first pass, the GPU generates multiple views into a texture. In the second pass, the GPU interleaves an image from the texture through the fragment shader.

1) *GPU generates multi-view into a texture*

In our method, when the application computes primitives only for a single view, the GPU generates the primitives for all views through the geometry shader.

(a) Parallax.

The primitives of other views generated by the GPU are transformed from the input primitives in geometry shader. The transformation composes of the following steps [12]:

Step 1. View transformation. It shifts primitives along the x-axis. A configurable parameter, κ, which is the distance between the viewpoints, is used to control the transformation value.

Step 2. Parallax distribution transformation. It translates the projected image along the x axis to adjust the parallax distribution by transforming the projection matrix. In order to control the transformation value, we set a configurable parameter ξ, which is the depth of the focal plane.

Actually, there is only a horizontal parallax between the vertices in two views. According to the transformation, we can obtain the parallax $\Delta H(\Delta x, \Delta y, \Delta z)$ of a vertex as:

$$\begin{cases} \Delta x = A \times \kappa \times (1 - z/\xi) \\ \Delta y = 0 \\ \Delta z = 0 \end{cases},$$

where $A = (2 \times Z_n)/(r-l)$, r and l are the coordinates for the left- and right-vertical clipping planes, Z_n is the distance to the near-depth clipping planes, z is the depth of the 3D vertex.

(b) Pimitives generated for multi-view through geometry shader.

Different from Sorbier's method [9], our method renders multiple views into a texture. Since these rendered views are aligned side by side in the texture memory, the viewport is divided into multiple sub-windows along the x axis. The viewport of the i-th view is the i-th sub-window.

We assume the input primitve belongs to the first view. Based on the input primitive, the primitives of other views are generated as shown in Fig.3.

```
1.    for j from 1 to the number of the vertices in the input primitive
2.  {
3.        generate the j-th vertex of the i-th primitive;
4.              emit the j-th vertex of the i-th primitive with its attributes;
5.  }
6.  call Cg API restartStrip to start a new primitive;
```

Figure 3. The pseudo-code of generating the i-th primitive for the i-th view.

To generate the j-th vertex of the i-th primitive, P^{ij}, froms the j-th vertex of the input primitive, P^{j}, and the GPU performs the following steps:

Step 1. Calculate the coordinates of P^{ij}.

P^{ij} is the sum of P^{j} and its parallax. To render the i-th view into the i-th sub-window, P^{ij} must be pre-transformed at the x-axis. Thus, it is calculated as:

$$\begin{cases} P_x^{ij} = ((P_x^{j} + \Delta x \times i) + (2 \times i + 1.0 - N)) / N \\ P_y^{ij} = P_y^{j} \\ P_z^{ij} = P_z^{j} \end{cases},$$

where, (Px^{ij}, Py^{ij}, Pz^{ij}, Pw^{ij}) is the homogeneous coordinates of P^{ij}, and (Px^{j}, Py^{j}, Pz^{j}, Pw^{j}) the homogeneous coordinates of P^{j}.

Step 2. Calculate the clipping distances for P^{ij}.

Since all views share a common view-frustum set by the single-view OpenGL application, some primitives may be drawn into their neighboring sub-windows without being clipped. Thus, for every vertex, it need calculate and set the clipping distances between the vertex and the left/right edges of its sub-window. The rasterization stage in the graphics pipeline will clip the primitives according to the clipping distances if the user-defined clipping planes are enabled. The two clipping distances of the vertex are calculated as:

$$\begin{cases} d_0^{ij} = P_x^{ij} + (1 - 2.0 \times i / N) \times P_w^{ij} \\ d_1^{ij} = -P_x^{ij} + (2.0 \times (i+1) / N - 1) \times P_w^{ij} \end{cases},$$

where d_0^{ij} and d_1^{ij} are the clipping distances from P^{ij} to its left and right edges of its sub-window.

(c) Render multiple views into a texture.

Through the geometry shader, the GPU renders multiple views into the frame buffe, and the application copies the multiple views into the bound texture memory before calling the OpenGL API *glSwapBuffers*.

2) Interleave the image from the texture through the fragment shader in the second pass.

```
void main(
        float2  win_pos: TEXCOORD0, //input texture coordinate
        out float4 color: COLOR, //the color of the current pixel

        //the texture rendered in the first pass
        uniform sampler2D multi-view,
        uniform sampler2D mask)
{
    //calculate the texture coordinate
   float4 v = text2D(mask, win_pos);

   //sample the color from the texture
   color = tex2D(multi-view, float2(v.r, v.g));
   return;
}
```

Figure 4. The pesudo code of interleaving.

To speed up the computation, the values stored in the mask are pre-transferred into the texture coordinates. Hence, the pixel colors of the image are sampled from the texture with the texture coordinates. The interleaving process of the fragment shader is described in Fig. 4.

B. *GPU-based multi-view rendering*

By combining the GPU-based multi-view rendering method with Chromium, GBMVR is composed of five kinds of SPUs: the GPUSMRender SPU, the warp SPU, the standard pack SPU, the standard tilesort SPU and the standard render SPU. The developed GPUSMRender SPU encapsulates the GPU-based multi-view rendering method while the warp SPU performs the geometry calibration [14] and the luminance calibrations [15, 16] for the sort-first parallel rendering. The pack SPU transfers the geometry to its downstream server node while the standard tilesort SPU dispatches the geometry data to its downstream server nodes. The render SPU sends all the geometries to the system's OpenGL library for rendering.

1) Configuration

GBMVR can be configured to support the spatial-multiplex autostereoscopic display in a PC (see Fig. 5 (a)). In Fig. 5 (a), there is an app node and a server node. The app node intercepts the geometry and transfers it to the server node. The server node renders interleaved images with the GPU-based multi-view rendering method.

In addition, GBMVR can be configured to support the high-resolution spatial-multiplex autostereoscopic display among PC cluster (see Fig. 5 (b)). In Fig. 5(b), there is an app node and several server nodes. The app node intercepts the geometry data. The tilesort SPU does a sort-first partition of the geometry data and sends the data to multiple downstream server nodes. Each server node renders a subset of the total large interleaved image for the connected projectors. The warp SPU guarantees to make the images displayed by the projectors seamless and uniform-luminance.

2) GBMVR SPU

GBMVR SPU is developed to support the GPU-based multi-view rendering method in Chromium. This SPU re-implements the *glBegin* and *glSwapBuffers* functions.

Fig. 6 shows the pseudo code of the *glBegin* function. It enables the user-defined clipping planes first; and then binds the geometry program; next calls the Cg API function *cgSetParameterValue1f* [13] to transmit the parameters of κ and ξ to the geometry program; finally, passes the function *glBegin* to the next SPU.

Fig. 7 is the pseudo code of *glSwapBuffers*. First, it enables the fragment shader and disables the geometry shader. Second, it binds a texture and copies the multiple views from the frame buffer to the texture's memory. Third, it draws a rectangle to let the fragment shader interleave the image from the texture. Fourth, it passes *glSwapBuffers* to the next SPU. Finally, it enables the geometry shader and disables the fragment shader to start the next frame rendering.

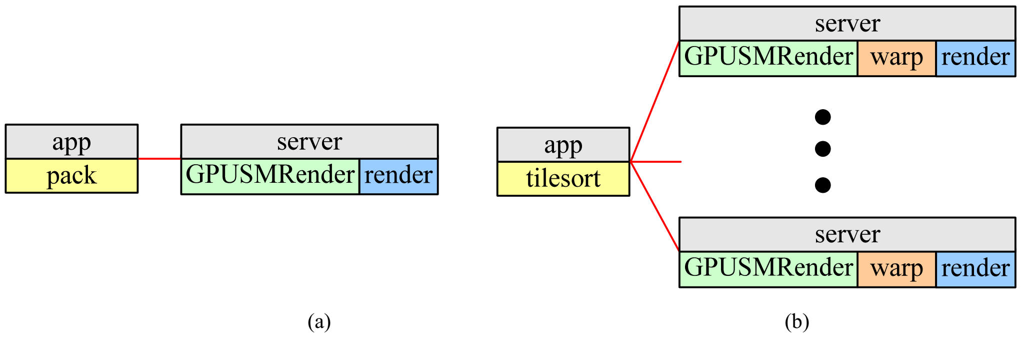

Figure 5. (a) The configuration of GBMVR for the spatial-multiplex autostereoscopic display; (b) The configuration of GBMVR for the high-resolution spatial-multiplex autostereoscopic display.

```
void glBegin (GLenum mode )
{
    glEnable(GL CLIP PLANE0);
    glEnable(GL CLIP PLANE1);
    call function cgGLBindProgram to bind the geometry program.
    cgSetParameterValue1f(eyeOffset,1, $\kappa$ );
    cgSetParameterValue1f(zeroParallaxPos,1, $\xi$ );
    nextSPU.glBegin(mode);
}
```

Figure 6. The pesudo code of the *glBegin* function.

```
void glSwapBuffers( GLint window, GLint flags )
{
    cgGLEnableProfile(cg_fprofile);
    cgGLDisableProfile(cg_gprofile);
    glBindTexture(multi-view);
    glCopyTexSubImage2D( . . . );
    glBegin(GL_QUADS);
    call glvertex and glTexCoord2D to draw a rectangle.
    glEnd();
    nextspu.SwapBuffers(window, flags );
    cgGLEnableProfile(cg_gprofile);
    cgGLDisableProfile(cg_fprofile);
}
```

Figure 7. The pesudo code of the *glSwapBuffers* function.

IV. Results

Fig. 8 shows the images interleaved from the multiple views of the game Quake III. Fig. 9 shows a high-resolution image. The image is displayed on a 3.6×1.6 m^2 screen by 12 tiled projectors. The screen is built for our multi-projector autostereoscopic display. The image is blurred because the lenticular sheets in the screen diffuse the image.

We test our method on PCs with Intel Core i7 920 2.67 GHz CPUs and nVidia GeForce 9600GT GPUs. To rule out the communication expense of Chromium, we compare the GPU-based multi-view rendering method with the traditional multi-pass rendering method in a PC. Both of them render the 3D models of a thinker (0.79 million triangles) with different number of views. The frame rates are listed in Table 1. Obviously, the GPU-based multi-view rendering method is faster than the other. Our method improves the performance greatly when the number of views is equal to 2. When the number of views is greater than 4, there is no obviously difference between the two methods. The reason is that the computing power of the geometry shader is limited.

TABLE I. Frame rates of two methods while rendering 3D models of a thinker.

Rendering method	1 view	2 views	4 views	8 views
our method	137.6	115.2	47.5	20.1
multi-pass method	137.6	68.6	34.6	17.2

Furthermore, we compare GBMVR with Annen's method [3] among 7 PCs. Both of them render the game Quake III with 2 views and 4 views in turn. The frame rates are listed in Table 2. Obviously, GBMVR is much faster than Annen's method. The reason is due to:

(a) Annen's method renders multiple views with multiple passes;

(b) To render *K* views, Annen's method must transfer the commands of a frame from the app node to every server node *K* times while our method does only once.

TABLE II. Frame rates of two methods while running Quake III.

Rendering method	2views	4 views
GBMVR	15.2	12.6
Annen's method	5.2	3.4

V. Conclusions

In this paper, we present a GPU-based multi-view rendering for spatial-multiplex autostereoscopic displays. GBMVR renders multiple views into a texture through the geometry shader in only one pass, but the traditional method does it in multiple passes. Most traditional OpenGL applications (e.g., some games like Quake III) can run directly on our system without any source-code modification or re-compiling. Furthermore, GBMVR can support high-resolution spatial-multiplex autostereoscopic displays.

Acknowledgment

This work is supported by the National High Technology Research and Development Program ("863" Program) of China (No.2006AA01Z326).

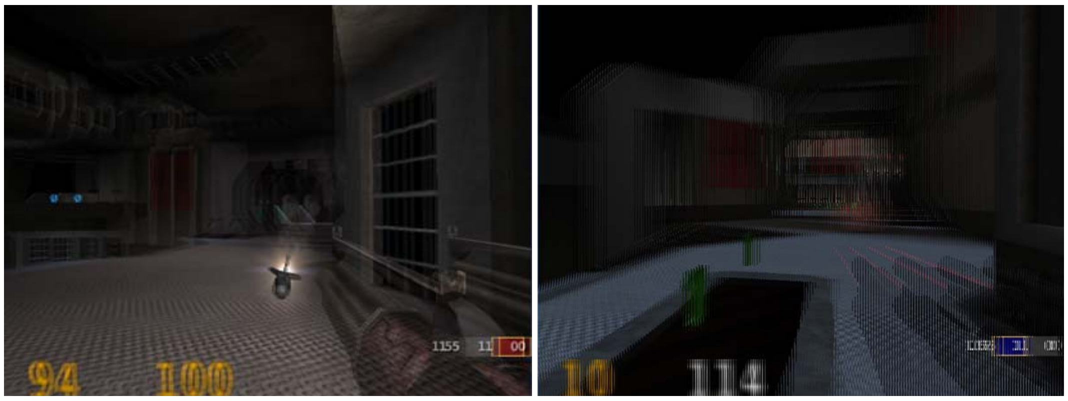

Figure 8. (a) An image interleaved from 2 views of the game of Quake III by GBMVR in a PC.
(b) An image interleaved from 4 views of the game of Quake III by GBMVR in a PC.

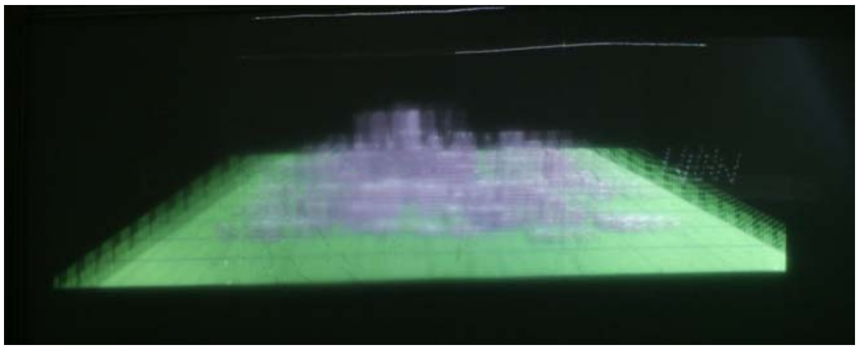

Figure 9. A high-resolution image interleaved from 4 views by GBMVR among 7 PCs.

REFERENCES

[1] B. Lee, H. Hong, J. Park, H. Park, H. Shin, I. Jung, “Multi view autostereoscopic display of 36view using an ultra-high resolution LCD”, Proc. of SPIE, San Jose, CA, USA, 2007, vol. 6490, pp.1-8.

[2] W. Matusik, and H. Pfister, “3D TV: a scalable system for real-time acquisition, transmission and autostereoscopic display of dynamic scenes”, ACM Transactions on Graphics, 2004, 23(3): pp.814-824.

[3] T. Annen, W. Matusik, H. Pfister, H. Seidel, and M. Zwicker, “Distributed rendering for multi-view parallax displays”, in Proc. of SPIE Stereoscopic Displays and Virtual Reality Systems, San Jose, CA, USA, 2006, pp.5962P-1--5962P-10.

[4] G. Humphreys, M. Houston M, R. Ng, et al, “Chromium: a stream-processing framework for interactive rendering on clusters”, ACM Transactions on Graphics (TOG), 2002, 21(3), pp. 693-702.

[5] K. Perlin, S. Paxia, J. Kollin, “An Autostereoscopic Display”. In Proceedings of ACM SIGGRAPH 2000, Computers GraphicsProceedings, Annual Conference Series, 2000, pp.319-326.

[6] f. Knocke, R. de Jongh, and M. Rromel, “Concept, design and analysis of a large format autostereoscopic display system”, in Proc. Of SPIE, 2005, vol. 5962, pp.5962P-1~5962P-10.

[7] J. B. Eichenlaub, “A lightweight, compact 2d/3d autostereoscopic lcd backlight for games,monitor, and notebook applications”, In Proc. Of SPIE. San Jose, California, USA, 1998, vol.3295, pp.180-185

[8] A. Chu, C. Fu, A. J. Hanson, and P. Heng, “GL4D: A GPU-based Architecture for Interactive 4D Visualization”, IEEE transactions on visualization and computer graphics, 2009,15(6),pp.(1587-1594

[9] F. D. Sorbier, V. Nozick, and V. Biri, “GPU rendering for autostereoscopic displays”. In 3DPVT'08, the Fourth International Symposium on 3D Data Processing, Visualization and Transmission, Atlanta, GA, USA, 2008, pp.1-7.

[10] R. L. Kooima, T. Peterka, J. I. Girado, J. Ge, D. J. Sandin, and Thomas A. DeFanti, “A gpu sub-pixel algorithm for autostereoscopic virtual reality”, In IEEE Virtual Reality Conference, Charlotte, North Carolina, USA, 2007, pp.131-137.

[11] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan, “Distributed rendering for scalable displays”, IEEE Supercomputing 2000, October 2000.

[12] Stereographics Corporation. http://cs.unc.edu/Research/stc/FAQs/Stereo /stereo-handbook.pdf P35. 1997 mmodation and convergence difference in stereoscopic three-dimensional displays by using correcti.

[13] http://developer.download.nvidia.com/cg/Cg_2.0/2.0.0012/Cg-2.0/Jan2008/ReferenceManual.pdf

[14] R. Raskar, M. S. Brown, R.Yang, W. Chen, and G. Welch, et al., “Multi-projector displays using camera-based registration”, in Proc. Of IEEE Visualization , 1999, pp.161-168.

[15] A. Majumder, and R. Stevens, “LAM: luminance attenuation map for photometric uniformity in projection based displays”, in Proc. Of ACM Virtual Reality and Software Technology, 2002, pp.147-154.

[16] M. S. Brown, and B. Seales, “ A Practical and Flexible Tiled Display System “, in Proceedings of the 10th Pacific Conference on Computer Graphics and Applications , 2002, pp.194-203.