Accelerated Stereoscopic Rendering using GPU

François de Sorbier
Université Paris-Est
LABINFO-IGM
UMR CNRS 8049
fdesorbi@univ-mlv.fr

Vincent Nozick
Graduate School of Science and Technology,
Keio University, Japan
nozick@ozawa.ics.keio.ac.jp

Venceslas Biri
Université Paris-Est
LABINFO-IGM
UMR CNRS 8049
biri@univ-mlv.fr

ABSTRACT

This paper presents a new method to create a pair of stereoscopic images in one pass. Our algorithm takes advantage of the latest version of GPUs, including geometry shaders. Given the left viewpoint of a scene, primitives are duplicated and transformed to compute the right image. Hence this method saves the extra-time required to recompute attributes of the vertices for the second view in the traditional rendering pipeline. Even for very complex scenes, our method provides stereoscopic pair roughly twice faster than a traditional method and involves few additional implementations.

Keywords

Stereovision, GPU, Geometry Shaders, real-time computing.

1.INTRODUCTION

The unceasing enhancement of GPU (Graphics Processor Unit) is usually leaded by well defined real-time improvements. However, it is common to notice that such new capabilities are also used for unexpected applications, as shown in numerous articles [Eng06a, Fer04a]. This paper, dealing with stereoscopic rendering acceleration, belongs to this category. Indeed, stereovision techniques which significantly increase the immersion feeling, involve that they involve to render the scene twice, i.e. once for each eye. Therefore, the rendering process becomes nearly twice longer. Nevertheless, the emergence of commercial autostereoscopic displays [Dod05a] tends to proof that stereovision is a hot research topic.

This article presents a new method to create a stereoscopic pair by rendering the scene geometry only once. Our method is based on geometry shaders and can reduce the two traditional drawing passes to one. Indeed, the one and only difference between two stereoscopic images is the viewpoint: geometry, object colours computation, and sometimes illumination, remains the same. The main purpose of geometry shaders is to clone input primitives without requiring any additional process on the vertex attributes.

Our goal is to design an algorithm able to render the scene only once for any stereovision technique. Our method takes full profit of new technologies of graphic cards – vertex, fragment and geometry shaders, multiple render target and frame buffer object – but with few instructions on the vertex and fragment programs.

In the following parts, we introduce a brief reminder about how to generate stereoscopic images and also provide a survey about existing methods for stereoscopic rendering. Next, we describe our method and explain how this technique takes advantage of the latest GPU capabilities. Then we detail the implementation step by step. Finally, we present our experimental results showing that compared to traditional techniques, our method achieves nearly twice faster performances for equivalent visual results.

2.STEREOVISION

This section introduces the rudiments of stereoscopic pair computation and describes previous work related to stereovision rendering.

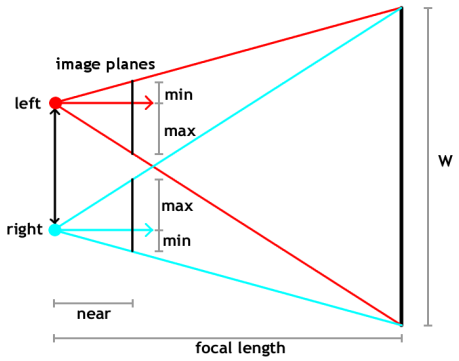

Figure 1: projection volume computation

Rudiments of stereovision

A stereovision system generates two images of a scene from two slightly different viewpoints associated to the left and right eyes. The most used and proper method to create stereo pairs is called "off-axis" [Bou99a]. This method induces that both viewpoints share the same image plane and use an asymmetric projection volume as depicted on Figure 1.

The projection parameters are computed as follows :

$$min = \frac{near}{focallenght}\left(\frac{W - dc}{2}\right)$$
$$max = \frac{near}{focallenght}\left(\frac{W + dc}{2}\right)$$

where W is the width of the focus plane and *focal length* – improperly - denotes the distance from the camera centre to the focus plane. The *dc* variable corresponds to the distance between left and right camera.

This method can be used with various restitution techniques such as anaglyph (red/cyan glasses), polarized glasses, active stereo displays, HMD, etc.

Related work

A great effort has been provided concerning stereo pair compression, comfort of stereoscopic perception and stereoscopic device enhancement but few research has been done on stereoscopic rendering acceleration. Indeed, traditional stereoscopic rendering methods always require two passes to render a scene from both left and right viewpoints.

Adelson *et al.*[Ade92a] propose a stereoscopic ray-tracing algorithm that takes advantage of the coherence between the two viewpoints. This approach saves about 75% of computational time for the second view. However this method belong to the ray-tracer family and hence is not suited for real-time applications.

Nvidia has developed a stereoscopic driver [Nvi03a] that gives the capability to intercept programming instructions from DirectX or OpenGL and convert them into stereoscopic informations. In theory, this driver should work with any 3D application but some problems and incompatibilities still exist: applications must respect some developing rules to be compliant with the driver [Nvi06a]. Moreover this method implicitly performs two rendering passes and so provides no special optimisation.

3.PROPOSED METHOD

This section presents the geometry shader and shows how these new capabilities can be used to speed up stereoscopic rendering. We also explain how this method can be inserted in a classic rendering pipeline.

Geometry shaders capabilities

In recent years, vertex and pixel shaders have been widely used to speed up or improve rendering quality compared to CPU or older GPU.

The fourth version of shader model introduces a new kind of shader called geometry shaders [Lich07a]. The geometry-shader stage is inserted in OpenGL pipeline just after the vertex-shader stage and before the clipping transformations (Figure 2). The main purpose of geometry shaders is to generate new primitives. Indeed, geometry shaders' input is a single fixed primitive (point, line or triangle). Then, according to this input data, geometry shaders can generate multiple fixed primitives (point, line strip or triangle strip).

A vertex belonging to a new primitive is generated using *EmitVertex* function and finally emited with the *EndPrimitive* function.

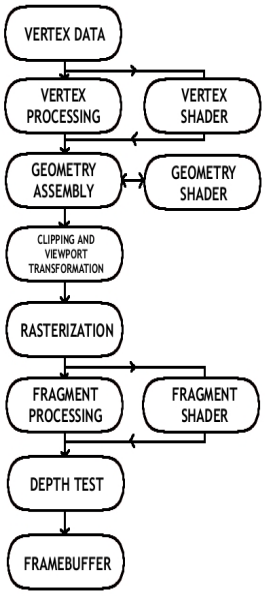

Figure 2: OpenGL programmable pipeline

Our rendering method

The fundamental point of our method is to use the duplication property provided by the geometry shader. Each primitive used for a single view is duplicated and transformed according to the new viewpoint. Moreover, using geometry shaders involves that only input vertices are processed by vertex shaders. Hence this method saves computation time for the duplicated primitives.

Our algorithm can be described as follows:

- compute left and right modelview and projection matrices
- render the scene from the left viewpoint
- perform vertices processing
- receive primitives in geometry shader from vertex shader
  - clone primitives
  - perform modelview and projection transformations to the left an right viewpoints
  - emit new primitive to fragment shader
- render results into two separate buffers using Multiple Render Target
- display stereo pair according to the stereoscopic restitution method

Of course, the two output images can be displayed using any stereoscopy restitution method.

The product of modelview and projection matrices are precomputed only once according to the user's eyes separation and the scene *focal length.* If these stereoscopic parameters change, new modelview and projection matrices should be computed consequently. Note that the product of the projection matrix by the current modelview matrix plus the eye separation translation vector for each primitive would consume extra-computing time since the same instruction would be repeated for every primitive.

4.IMPLEMENTATION

This part describes the three implementation stages required by our method. The first section explains the cloning stage performed by the geometry shaders. The next section describes the rendering process and the last section presents a brief description about how to display the stereo pair.

Cloning the geometry

As mentioned above, the key point of this method is the duplication stage performed by the geometry shader. Indeed, the geometry shader clones the primitives designated for the left viewpoint to the right viewpoint. The duplicated vertices are then transformed and projected on the right buffer while the initial vertices are projected on the left buffer.

The following GLSL [Ros06a] code describes this process:

```
#version 120
#extension GL_EXT_geometry_shader4 : enable
#extension GL_EXT_gpu_shader4 : enable

varying float flag;
uniform mat4 matrix;

void main(void)
{
 flag = 0.0;
 for(int i =0; i < 3; ++i){
  gl_Position =
    gl_ModelViewProjectionMatrix *
    gl_PositionIn[i];
  EmitVertex();
 }
 EndPrimitive();

 flag = 1.0;
 for(int i =0; i < 3; ++i){
  gl_Position = matrix * gl_PositionIn[i];
  EmitVertex();
 }
 EndPrimitive();
}
```

In this program, the uniform variable `matrix` corresponds to the right image transformation matrix, product of the modelview matrix and the projection matrix of the right viewpoint. The left transformation matrix can be directly read as a built-in uniform variable set in the main program as the product of the modelview matrix with the projection matrix. The variable `flag` indicates whether the created vertex belongs to the left or to the right view. Note that the product of the projection matrix by the current modelview matrix plus the eye separation translation vector for each primitive would consume extra-computing time since the same instruction would be repeated for every primitive.

Rendering process

In the fragment shader stage, the incoming fragments should be sorted out depending on the variable `flag` set in the geometry shader. This flag is set to zero if the fragment belongs to the left viewpoint and different of zero otherwise.

```
#version 110
#extension GL_ARB_draw_buffers : enable

varying float flag;

void main()
{
  vec4 color = (1.0);
  if(flag==0.0){
    gl_FragData[0] = color;
     gl_FragData[1] = vec4(0.0);
  }else{
     gl_FragData[0] = vec4(0.0);
    gl_FragData[1] = color;
  }
}
```

This sample program corresponds to the fragment shader operations for right and left image. To perform this stage, we use both Multiple Render Targets (MRT) and FrameBuffer Object (FBO). MRT can render a scene in multiple buffers while the FBO can directly render the result in a texture.

Unfortunately that using MRT and FBO involves some constraints:

1. FBO and MRT share a common buffer for rendering tests such as depth and alpha tests even if there are multiple colour buffer targets.
2. MRT involves that every fragment is rendered in the two buffers. Otherwise, the result would be undefined.

Sharing the depth buffer for the right and left viewpoints rendering, which is a consequence of the first constraint, means that some fragments could be discarded when they should not. Our approach to solve this problem is to disable the depth test. We are then force to use the painter's algorithm as a substitute to the depth test to solve the visibility problem.

The second remark induces that the two primitives created by the geometry shader are independent. Using MRT involves that every fragment must be drawn on the two buffers (Figure 4). Hence a fragment that does not belong to the proper buffer indicated by the flag should be discarded. The elimination of an undesirable fragment can be performed by setting its colour to black and its alpha value to zero. The blending function should be enabled and depth test disabled in the main software program, thus the fragment will not be displayed. The function used for blending is:

$$Red = R_{src} \times A_{src} + R_{dest} \times (1 - A_{src})$$
$$Green = G_{src} \times A_{src} + G_{dest} \times (1 - A_{src})$$
$$Blue = B_{src} \times A_{src} + B_{dest} \times (1 - A_{src})$$
$$Alpha = A_{src} \times A_{src} + A_{dest} \times (1 - A_{src})$$

Displaying a stereo pair

The rendering process generates two separate textures corresponding to the left and the right viewpoints. According to the stereoscopic device used, the user should eventually modify the stereo pair.

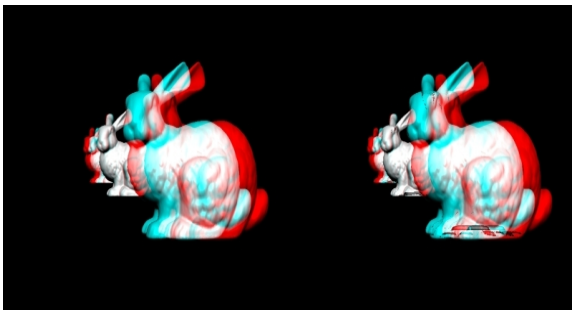

Figure 3: On the left the traditional stereo pair result. On the right the same scene rendered with our method.

For example, using anaglyph glasses involves to mix red/cyan filtered images (figure 3) as describes in the following pseudo-code:

- disable depth test
- enable blending with additive function
- set colour mask to red
- render left texture on a fullscreen quad
- set colour mask to cyan
- render right texture to fullscreen quad

Polarized or active stereo systems do not require such process, each image should be send to the appropriate video output.

5.EXPERIMENTAL RESULTS

This section presents our experimental results. Then we analyse our system limitations and propose solutions to solve them.

Results

We tested our method on PC Intel core 2 duo 2,40GHz with a Nvidia GeForce 8800 GTX graphic card.

We performed our tests on a test scene displaying numerous Stanford bunny models [Sta93a] and using various graphical effects such as lighting. We set the screen resolution to 1024x768.

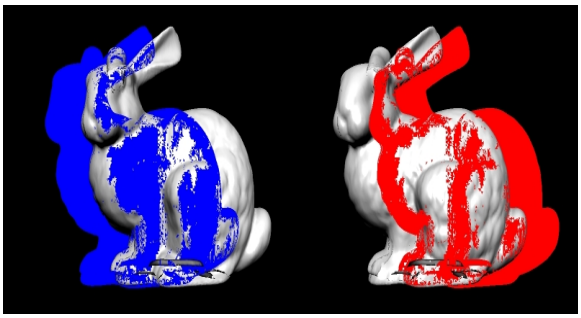

Figure 4: Errors due to the obligation to render a fragment in the two buffers

	Traditional stereoscopy [Bou99a] (fps)		Our method (fps)	
Number of triangles	70000	210000	70000	210000
Flat rendering	163	55	304	108
Vertex lighting	83	28	158	54
Shader lighting	82	27	156	54
Number of triangles	350000	630000	350000	630000
Flat rendering	33	18	65	36
Vertex lighting	16	9	33	19
Shader lighting	16	9	33	18

Table 1: Comparative statement

Table 1 shows that our method is more effective in every situation. Our method is especially effective when the rendering method requires a large amount of processing operations on vertices. The tests also show that for a scene containing an important number of vertices, our method achieves a 95 to 100% gain with a shader lighting method.

Limits and solution

Even if this method is effective to render stereo pairs in real time based on complex geometry, this system presents some limitations, especially those mentioned on section 4.2.

Waiting for a separate depth-buffer for each rendering buffer of the MRT, we propose to use the painter's algorithm as a substitution of the depth test. All objects should be sorted and drawn in a back-to-front order according to the “camera” position in the scene. This solution will provide a correct rendering except for some well known particular cases relative to the painter's algorithm. However, some depth artefacts will occurred with concave objects (Figure 5).

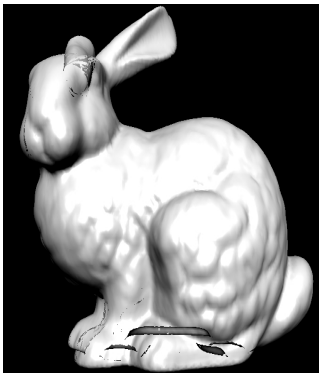

Figure 5: Artefacts due to concave objects properties

These concave objects can be correctly displayed by rendering the left viewpoint scene in a depth texture. Then during the fragment shaders stage, the depth values should be read from the depth texture to decide whether a fragment should be discarded or not. The right depth value computation is based on the fact that the left and right cameras share the same image plane. Thus, the object depth remains constant between the two views.

The depth-map-based method can be summarized as follows :

- in geometry shaders, transmit transformed vertices coordinates of the left viewpoint primitives to the fragment shaders
- if fragment belong to the right viewpoints
  - read coordinates value coming from the geometry shader
  - apply viewport clipping transformations
  - if coordinates exist in depth map
    - compare the depth map value and the fragment z-value
    - according to this z-test, accept or discard the fragment
  - else accept the incoming fragment

Nevertheless, for the right viewpoint, in some very specific configurations depth values read from the depth-map will not match with the z-value expected. This could also display some artefacts on the right view rendering.

Table 2 shows that even if the depth map computation is time consuming, this method is still faster or at least equivalent to the traditional rendering method. Moreover, this technique becomes especially effective for the methods requiring high computation from the vertex shaders (lighting shader used in figure 6 for example).

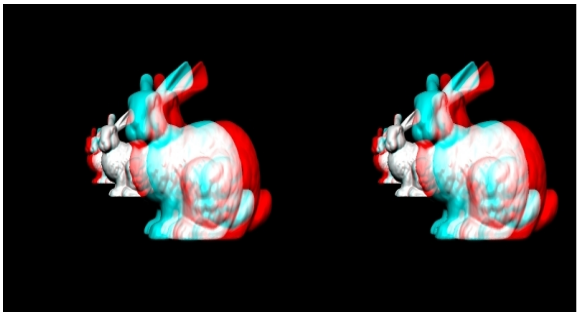

Figure 6: On the left the traditional stereoscopy algorithm. On the right our method including the depth-map.

	Traditional stereoscopy (fps)		Our method (fps)	
Number of triangles	70000	210000	70000	210000
Flat rendering	163	55	158	55
Vertex lighting	83	28	107	36
Shader lighting	82	27	105	35
Number of triangles	350000	630000	350000	630000
Flat rendering	33	18	33	18
Vertex lighting	16	9	22	12
Shader lighting	16	9	22	12

Table 2: Results using depth-map

6.CONCLUSION

This article presents a stereoscopic GPU-based method computing a stereo pair in one-pass. This method takes advantage of the geometry shader to duplicate and transform primitives from left to right viewpoint. Rendering, done using Multiple Render Target, can be used with any stereoscopic restitution method. Contrary to traditional stereoscopic methods, vertices attributes are not computed a second time for the second view. Thus, our method saves computational time and then can render stereo pairs from scene containing a high level of detail with a negligible frame rate decrease.

This system presents some limitations mainly due to the Multiple Render Target depth buffer management. However we propose an optimisation of our method including a depth-map computation. We hope that in the future, two separate depth buffers will be available for MRT what will definitively speed up our algorithm.

Concerning future works, we plane to experiment our method on autostereoscopic displays. These kind of screens require at least a half-dozen stereo images of the same scene.

7.ACKNOWLEDGMENT

This work has been partly supported by “Foundation of Technology Supporting theCreation of Digital Media Contents” project (CREST, JST), Japan.

8.REFERENCES

[Ade92a] S. J. Adelson, L. F. Hodges. *Stereoscopic ray-tracing*. The Visual computer, vol.10, n. 3, pp.127-144, 1993.

[Bou99a] P. Bourke. *Calculating Stereo Pairs*. http://local.wasp.uwa.edu.au/~pbourke /projection/stereorender/, 1999.

[Dod05a] N. A. Dodgson. *Autostereoscopic 3D Displays*. Computer, Volume 38, Issue 8, 2005.

[Eng06a] W. Engel. *ShaderX5: Advanced Rendering Techniques*. Charles River Media, ISBN-13 978-1584504993, 2006.

[Fer04a] R. Fernando, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, ISBN-13 978-032122832, 2004.

[Lich07a] B. Lichtenbelt, P. Brown, *EXT_gpu_shader4 Extensions Specifications*. NVIDIA, 2007.

[Nvi03a] NVIDIA. *Technical Brief : 3D Stereo*. TB-00252-001_v02, 2003.

[Nvi06a] NVIDIA. *GPU Programming Guide Version 2.5.0*. pp. 69-75, 2006.

[Oko77a] T. Okoshi. *Three-Dimensional Imaging Techniques*. Academic Press, 1977.

[Ros06a] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, ISBN 0-321-33489-2, 2006.

[Sta93a] *The Stanford 3D Scanning Repositry.* http://graphics.stanford.edu/data/3Dscanrep/, 1993.

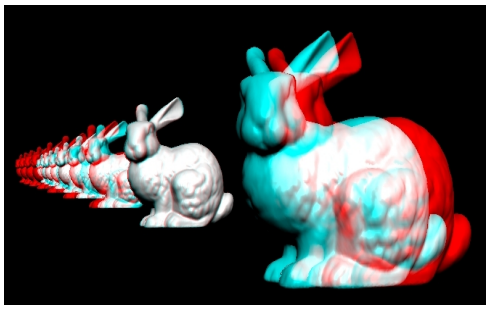

Figure 7: A result of our stereoscopic algorithm using GPU