HAL
open science

GPU rendering for autostereoscopic displays

François de Sorbier, Vincent Nozick, Venceslas Biri

To cite this version:

François de Sorbier, Vincent Nozick, Venceslas Biri. GPU rendering for autostereoscopic displays. 4th International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08), Jun 2008, France. 10pp. hal-00622365

HAL Id: hal-00622365

https://hal.science/hal-00622365v1

Submitted on 12 Sep 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

GPU rendering for autostereoscopic displays

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

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

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

Abstract

In recent years, stereoscopic technology has advanced from stereoscopic to autostereoscopic displays. These latter family involves to display several views of a scene. In the case of real-time computer graphics images, the standard approach consists in rendering every view independently. This paper presents an alternative method to generate multiple views for autostereoscopic displays in a single rendering pass. Our algorithm is based on the fact that vertices properties remain the same from different viewpoints. Taking advantage of the latest generation of GPUs including geometry shaders, we propose a method that significantly speeds up the rendering process by duplicating and transforming incoming primitives for a defined set of views. Our method involves very few modifications to be used with a standard stereo device.

1. Introduction

Stereovision is a well known technique to enhance virtual reality devices involved in human immersion. In comparison to standard display, stereoscopic devices increase the visual immersion feeling. In recent years, stereoscopic technology has advanced from stereoscopic to autostereoscopic displays. The latter family does not requires to wear any glasses or specific device for the user. Moreover, current autostereoscopic screens can display much more than two images to provide an adequate rendering in several directions and to be adapted for multi-user purposes. This new technology introduces new challenges like data acquisition, rendering and data transfer. Concerning the data acquisition, two cases should be considered: the input images are captured from video cameras or generated by computer graphics methods. We will focus on the second case where the scene should be rendered several times. This multiple rendering usually implies redundancies and can sometimes prevents rendering in real-time.

This article presents a new method for generating computer graphics that renders the scene from multiple viewpoints through geometry shaders. The only difference between the rendered images from a same scene being the viewpoints, the following properties are preserved: geometry, object colour computations, and part of the illumination computations. Thereby we take advantage of GPU capabilities to compute vertex attributes only once and then clones the result to render the multiple views dedicated to the autostereoscopic display. The goal of our method is to significantly enhance the rendering frame rate with few additional operations by generating multiple views of a scene in a single rendering pass.

2. Autostereoscopic rendering

This section presents a brief survey of autostereoscopic displays and some existing solutions to generate multiple view images.

2.1. Multi-view generation methods

The goal of a stereovision system is to produce two images of a same scene from two slightly different viewpoints and to associate them to the right and left eyes. Anaglyph, polarisation or shutter [10] are popular methods used to achieve stereovision but involve using glasses that impairs the feeling of immersion. Nevertheless, recent research works on autostereoscopy added the following benefits to the former stereoscopy idea:

- stereo displays allow multi-user applications;
- users do not have to wear glasses;
- users receive adequate stereo images according to their position.

Dodgson [2] presents a variety of autostereoscopic displays. But among them only two major techniques pro-

viding multiple view displays are available from several famous companies such as Philips [12] or Sharp [1].

The technique proposed by [5] uses the parallax barrier to feed both eyes strips of images lighted according to a barrier mask. The second technique presented by [14] relies on lenticular sheets. Vertical lenses are set over a group of pixels and emit light from each "sub-pixel" in different directions (up to 64 as presented by Takaki [15]). Unlike the parallax barrier technology which induces a luminosity loss because of opaque areas, lenticular sheets preserve the wholeness of luminosity.

2.2. Multi-view generation methods

As mentioned above, the 3D data can come from video camera acquisition or computer graphics rendering, but in both cases, the data generation can become a problem. In the case of video camera acquisition, plugging several cameras on a computer leads to video-stream saturation. This problem can be solved using online video-based rendering method as proposed by Nozick and Saito [9].

In the computer graphics case, the scene should be rendered for every different viewpoint which dramatically decreases the rendering frame rate and may prevent rendering in real-time. Several methods have been proposed to overcome the multi-pass rendering limitation for multi-view rendering of 3D information. A point-based rendering solution was proposed by Hübner *et al.* [4] using GPU to compute multi-view splatting, parametrised splat intersections and per-pixel ray-disk intersections in a single-pass. This method reaches 10 fps for 137k points in a 8-view configuration. To increase multi-view rendering performance, Hübner and Pajarola [3] present a direct volume rendering method based on 3D textures with GPU computations to generate multiple views in a single pass. These two solutions significantly decrease the computation time but are not suited for polygon based graphics.

An alternative solution has been proposed by Morvan *et al.* [8]: a single 2D image plus a depth map which are interrelated to display multiple views. Although the algorithm saves the bandwidth of data emitted to the system, it does an assessment over available data to fill the area's missing information of the new views and then reduces the content's truthfulness.

3. Algorithm outline

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.

3.1. Geometry shaders

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

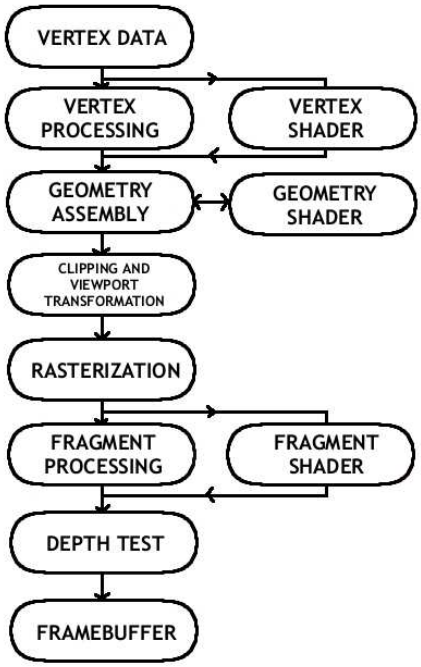

Figure 1. OpenGL pipeline with the fourth version of shader model.

The fourth version of shader model introduces a new kind of shader called geometry shaders [7]. The geometry shader stage is inserted in OpenGL pipeline just after the vertex shader stage and before the clipping transformations as depicted in Fig. 1. The main purpose of geometry shaders is to manipulate incoming single fixed primitives like points, lines or triangles. Thus operations on geometry such as duplication, transformations or adding vertices are now available on GPU. According to the input data, geometry shaders can generate multiple fixed primitives like points, line strip or triangle strip.

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

3.2. Our algorithm

The main purpose of our method is to speed up the multiple view rendering of a scene considering that vertex properties remain the same from a viewpoint to another. The new geometry shader stage takes place between the vertex

shader and fragment shader stages. An incoming primitive in the geometry shader stage can be manipulated and used to produce new primitives, thus saving extra-time required for vertex attribute computation in a basic multi-view process.

Our algorithm duplicates and transforms every rendered triangle according to the corresponding viewpoint. The resulting primitives are rasterized and transmitted to the fragment shader. Given N render-buffers associated to each of the N views, every received fragment is rendered into the correct buffer.

Rendering is performed into several textures to allow to create the correct textures' combining according to the autostereoscopic screen used. FrameBuffer Object (FBO) and Multiple Render Target (MRT) extensions are used to achieve this process. However since OpenGL implementation of FBO and MRT share a single depth buffer tied to the first draw buffer, it is not possible to use the common depth test to perform hidden-surface removal. So we need to apply our own depth test in fragment shader to discard undesirable fragments.
Our algorithm (Fig. 2) can be described as follows:

- compute modelview and projection matrices related to the N viewpoints
- set the middle viewpoint as the default viewpoint
- render the scene
  - process vertices
  - for each received primitive in the geometry shader
    - clone the primitive
    - perform modelview and projection transformations for the *N-1* extra viewpoints
    - emit new primitives to fragment shader
  - according to the viewpoint, render results into N separate textures using MRT
- compose the N textures according to the multi-view autostereoscopic display

Currently the painter's algorithm is used to perform the depth test even if results are well known to be not optimal. It exists two others solutions that could be applied to perform an efficient depth test.

The first one is to use one single texture that is shared between the multiple viewpoints that allow to use a single depth buffer. It takes advantage of the geometry shader to clip the triangles to the bounds of the area defined for a viewpoint and such a way there is no overlapping between the different renderings. However, using a single texture for every view strongly limits the number of generated images and their resolution. Moreover, this method may cause triangles to pop in and out because of the clipping.

The second solution will be available with next generation of graphic cards. The incoming specifications will allow to read and write into an active texture. The alpha value can be used in such a way as to store and read back depth values to perform our own depth test.

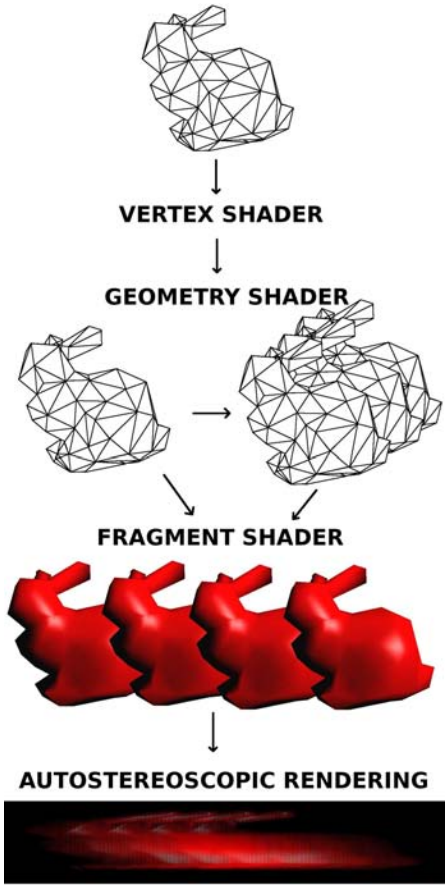

Figure 2. Outline of our method.

4. Implementation

This section describes the three main stages implementing our method.

4.1. Geometry duplication

As mentioned above, the duplication stage performed by the geometry shader is the key point of our method. N new primitives are generated according to the incoming primitive and the corresponding transformation matrices.
The following GLSL [13] code describes this process:

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

// maximum number of views
const int MAX_VIEW = 8;
// id transmitted to the fragment program
flat varying out float flag;

// number of rendered view
uniform int nview;
// reference view id
uniform int ref;
// views' projection matrices
uniform mat4 matrix[MAX_VIEW];

void main(void)
{
 // reference view processing
 flag = float(ref);
 // for each vertex of the triangle
 for(int i=0; i < 3; ++i){
  gl_Position =
   gl_ModelViewProjectionMatrix *
   gl_PositionIn[i];
  EmitVertex();
 }
 EndPrimitive();

 // additional views processing
 for(int v=0; v<nview; ++v){
  if(v!=ref){
   flag = float(v);
   for(int i=0; i<3; ++i){
    gl_Position = matrix[v] *
     gl_PositionIn[i];
    EmitVertex();
   }
   EndPrimitive();
  }
 }

}
```

The matrix array transmitted from the main program to the geometry shader contains for each viewpoint the pre-computed projection matrix multiplied by this viewpoint's modelview matrix. The variable *flag* is set with a different fixed value for each primitive. Then, in the next stage, fragments are rendered in the correct draw buffer according to this value.

4.2. Multi-view rendering

According to the value of the received variable *flag*, the fragment is rendered into the corresponding draw buffer. MRT constrains to render the fragment into each defined render buffer otherwise the result could be undefined. Thus if a fragment have to be written into a wrong buffer, its alpha value is set to zero and an alpha test is applied to discard these undesirable fragments.

Of course, since this operation arise at the end of the fragment program, this method is absolutely compatible with a normal use of the transparency with the alpha channel.

```
#version 120
#extension GL_ARB_draw_buffers : enable

flat varying in float flag;
uniform int nview;

void main()
{

  // Any fragment shader source code
  // like illumination, texture...

  for(int i=0;i<nview;++i){
   if(float(i)==flag)
    gl_FragData[i]=vec4(colorfrag.rgb,1.0);
   else
    gl_FragData[i]=vec4(0.0);
  }
}
```

Results are rendered in textures linked to the N colour buffers using the framebuffer objects extension. However, at the present time, FBO limits the number of useable render buffers to eight which also limits the number of views per pass to eight. Waiting for an increase of the available buffers, a simple solution to render more than eight views is to perform several passes using multiple framebuffer objects. Such a way we are able to render up to 12 views in real time. A 8-image generation result is depicted on Figure 3

4.3. Final image generation

The rendering process generates N textures tied to the N viewpoints. According to the stereoscopic device used, these textures have to be composed to produce the correct final image. Figure 4 depicts two existing pixel arrangements for lenticular sheet-based system. To speed up the computational time of this stage, the GPU is once again required. More information can be found in [6]. Figure 5 presents a ready-to-use image.

5. Results and discussions

We tested our method on PC Intel core2 duo 2,40GHz with a Nvidia GeForce 8800 GTS graphic card under Linux

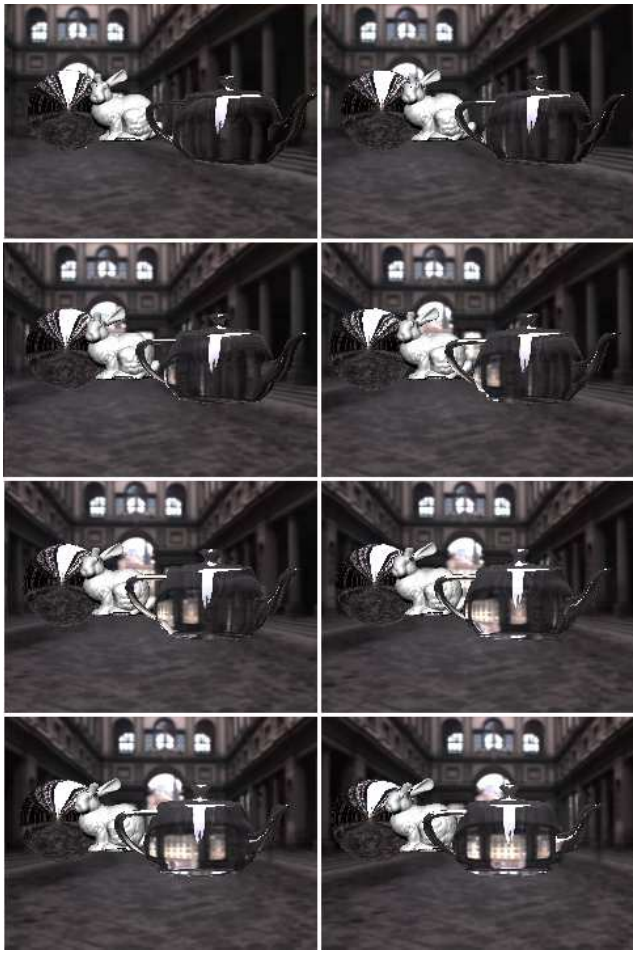

Figure 3. The 8-generated views.

system. Usually autostereoscopic screens using lenticular sheets technology have a 1920×1080 resolution. for our tests we choose to use the maximum of available images in one single pass i.e. 8 images related to different viewpoints. Consequently the resolution of each generated image is 240×180 (keeping a 4:3 ratio). Of course, our method is also well suited for much higher resolution. Table 1 presents our results obtained with our method and compares it to the traditional multi-pass process.

Number of views	1	2	4	8
Multi-pass method (*fps*)	157	80	38	19
Our method (*fps*)	157	141	70	24

Table 1. Frame rates obtained while rendering a scene with about 80.000 triangles.

These results show that our method speeds up the rendering process especially for a small number of additional

R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B	R	G	B
0			1			2			3			4			5			0		
0			1			2			3			4			5			0		
0			1			2			3			4			5			0		
0			1			2			3			4			5			0		

R	G	B	R	G	B	R	G	B	R	G	B
0	2	4	0	2	4	0	2	4	0	2	4
5	1	3	5	1	3	5	1	3	5	1	3
4	0	2	4	0	2	4	0	2	4	0	2
3	5	1	3	5	1	3	5	1	3	5	1

Figure 4. Examples of pixel arrangements with normal lenticular system and slanted lenticular system for 6 viewpoints. Each number denotes an image corresponding to a specific viewpoint and each RGB triplet is a pixel.

views. Since the geometry shader implementation is not effective yet, we notice that performance deviation result for higher number of images is not as high as expected. Indeed, we have observed that adding just one varying variable between the geometry and the fragment shader can induce a severe drop of performance (for instance, switching from 13 to 14 floats doubles the rendering time in the 4-view case). According to the opinion of the computer graphic community, bad performances of the first geometry shaders release with high amount of vertices seem to be well known [11]. However forthcoming generation of graphic cards will probably improve geometry shaders efficiency.

Our method was also tested with different kinds of rendering process to evaluate their impact on performances. Table 2 depicts the results.

Number of views	2	4	6	8
No illumination (fps)	326	168	80	60
Texture (fps)	326	167	80	60
Per-vertex illumination (fps)	160	83	28	24
Per-pixel illumination (fps)	160	83	28	23

Table 2. Results presenting our method with different kinds of rendering with 80.000 triangles.

We can notice that results of per-vertex illumination and per-pixel illumination are similar. This is due to the capability of latest GPU to distribute resources between the fragment shader and the vertex shader.

The goal of the last test was to increase the number of rendered views (more than 8) and to check results with varying number of passes including more or less views to render.

Number of passes	1	2	3	4	6
Views per pass	12	6	4	3	2
Results (*fps*)	N/A	14	15	23	21

Table 3. Results outlined by our method for 12 views using varying number of passes. In this configuration, the traditional method reaches 14 fps.

Table 3 shows that currently, a mix between multi-pass and geometry shader provides the best results, with a significant enhancement compared to the traditional multi-view generation method. We can especially notice good performance when rendering three views per pass. Indeed, the current version of geometry shader is efficient with a low number of primitives to generate and balance the passes needed to render the multiple views.
In addition to require only few changes according to a traditional rendering process, benefits of our method under 8 views can be summed up as follows :

- $nview - 1 \times nb_primitives \times nb_vertex_shader_op$ operations saved on vertex attribute computation;
- data are rendered only once;
- less data exchange between CPU and GPU;

Finally to resolve the single shared depth-buffer limitation, we expect that a separate depth-buffer for each rendering buffer of the MRT will be available on the next graphic cards generation or that reading and writing in a active texture will be possible.

6. Conclusion

This article presents a GPU-based method to generate multiple views designed for autostereoscopic displays. This method takes advantage of the geometry shaders to duplicate and transform primitives to the desired viewpoints. Our method significantly speeds up the rendering despite the constraints due to the limited number of views and a single shared depth buffer. With very few modifications on this method, future specification of the MRT and FBO including separate depth buffer would enhance the rendering frame rate and make the implementation trivial. Contrary to traditional multi-pass methods, vertex attributes are computed only once. Thus, our method saves computational time and then can speed up the multiple view rendering. Finally our method is easy to adapt on existing stereoscopic system.

As a future work, we intend to merge multiple-view rendering with the composition process described on chapter 4.3.

7. Acknowledgement

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

References

[1] 3d lcds. *http://sharp-world.com/products/device/about/technology/lcd-03.html*, 2006.
[2] N. A. Dodgson. Autostereoscopic 3d displays. 38(8):31–36, 2005.
[3] T. Hübner and R. Pajarola. Single-pass multi-view volume rendering. In *IADIS*, 2007.
[4] T. Hübner, Y. Zhang, and R. Pajarola. Multi-view point splatting. In *GRAPHITE*, pages 285–294, 2006.
[5] H. E. Ives. A camera for making parallax panoramagrams. In *Journal of the Optical Society of America*, number 17, pages 435–439, 1928.
[6] R. Kooima, T. Peterka, J. Girado, G. Jinghua, D. Sandin, and T. DeFanti. A gpu sub-pixel algorithm for autostereoscopic virtual reality. In *IEEE Virtual Reality 2007*, pages 131–138, 2007.
[7] B. Lichtenbelt and P. Brown. *EXT_gpu_shader4 Extensions Specifications*. NVIDIA, 2007.
[8] Y. Morvan, D. Farin, and P. H. N. de With. Joint depth/-texture bit-allocation for multi-view video compression. In *Picture Coding Symposium (PCS), to appear*, 2007.
[9] V. Nozick and H. Saito. Multiple view computation for multi-stereoscopic display. In *IEEE Pacific-Rim Symposium on image amd video Technology (PSIVT 2007)*, volume 4872, pages 399–412, 2007.
[10] T. Okoshi. *Three-dimensional imaging techniques*. Academic Press, 1977.
[11] S. Patidar, S. Bhattacharjee, J. M. Singh, and P. J. Narayanan. Exploiting the shader model 4.0 architecture. *Technical Report IIIT Hyderabad*, 2006.
[12] Philips. Wowvx for amazing viewing experiences. *Philips 3D solutions*, 2006.
[13] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2006.
[14] G. Saitoh, T. Suzuki, T. Abe, and K. Ebina. Lenticular sheet, rear-projection screen or tv using the same, and fabrication method for said lenticualr sheet. *U.S. Patent 5870224*, 1999.
[15] Y. Takaki. High-density directional display for generating natural three-dimensional images. In *Proceedings of the IEEE*, volume 94, pages 654–663, 2006.

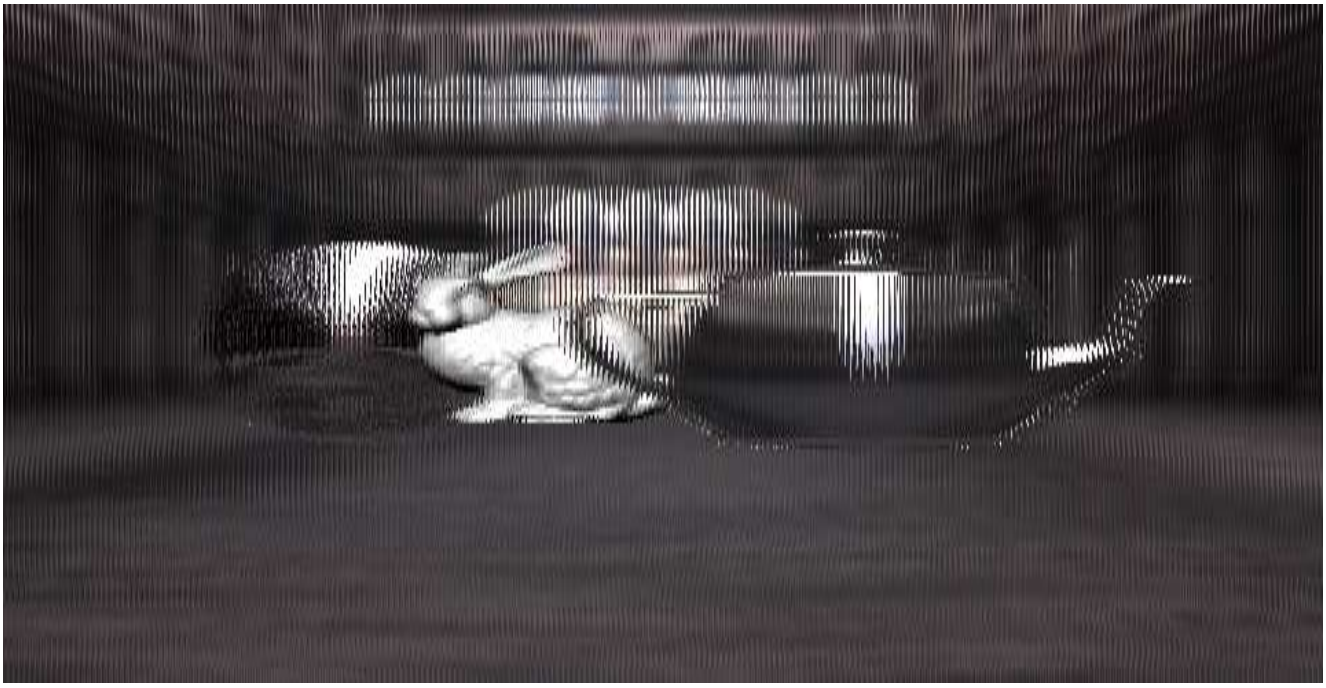

Figure 5. A 4-images composition for a lenticular sheet system with focus on the Stanford Bunny model.