Multi-view Rendering using GPU for 3-D Displays

François de Sorbier
Graduate School of Science and Technology
Keio University,Japan
Email: fdesorbi@hvrl.ics.keio.ac.jp

Vincent Nozick
Université Paris-Est LABINFO-IGM
UMR CNRS 8049
France
Email: vnozick@univ-mlv.fr

Hideo Saito
Graduate School of Science and Technology
Keio University, Japan
Email: saito@hvrl.ics.keio.ac.jp

***Abstract*—Creating computer graphics based content for stereoscopic and auto-stereoscopic displays requires to render a scene several times from slightly different viewpoints. In that case, maintaining real-time rendering can be a difficult goal if the geometry reaches thousands of triangles. However, similarities exist among the vertices belonging to the different views like the texture, some transformations or parts of the lightning. In this paper, we present a single pass algorithm using the GPU that speeds-up the rendering of stereoscopic and multi-view images. The geometry is duplicated and transformed for the new viewpoints using a shader program, which avoid redundant operations on vertices.**

***Index Terms*—multi-view, 3D displays, stereoscopy, real-time, GPU.**

I. Introduction

Stereoscopy is a technique that enables to watch three dimensional images on a display thanks, most of the time, to specific glasses. It has many applications in various fields such as data visualization, virtual reality or entertainment because it tends to reproduce our visual perception and makes information easier to understand. In computer graphics, stereoscopic rendering consists in generating two images of a virtual environment from two slightly different viewpoints. In other words, it requires to render the geometry of the scene twice which can double the computational time. In such case, it can be difficult to maintain real-time rendering especially for applications like video-games that are complex in term of geometry and visual effects.

Auto-stereoscopy is a technology recently applied to LCD displays [1] that introduces the ability for one or several users to watch stereoscopic images without wearing any glasses. Depending on their characteristics, auto-stereoscopic displays require from 5 to 64 images [2] to display a single 3-D frame. A filter, made of small lenses or precision slits, is overlaid on the surface of the screen and ensures to emit each image in a specific direction. So, if the user is well located in front of the display, each eye can see a single specific image.

However, the important number of required input images makes rendering difficult to maintain in real-time compared to a single view rendering. We can state two facts in term of time for standard multi-view rendering. Firstly, data transfer from main memory to the graphic card is costly, especially when no specific data structure, like Vertex Buffer Objects, is used. Secondly, some operations on vertices remain the same from one view to another which mean there are redundant computations. Global transformations, parts of illumination calculation and texturing are identical for example.

GPU programming is now very popular because it can speed-up many algorithms thanks to an efficient parallelized architecture. Recently, shaders have been updated with a new feature named geometry shader (GS) that takes place between the vertex shader and the rasterization stages [3]. Geometry shader introduces the possibility to manipulate vertices of input primitives like points, lines or triangles before emitting the result to the rasterization and clipping stages. It becomes also possible to generate new primitives during this stage.

The goal of our approach is to exploit geometry shader to speedup the rendering process of stereoscopic or multi-view images. The ability of geometry shaders to duplicate input primitives allows rendering in a single pass. Multiple sending of the geometry to the graphic card are reduced to a single transfer. Moreover extra computation due to redundant operations is avoided since our algorithm take place after the vertex shader stage.

This paper is structured as follows. We start giving an overview of related and previous works, and then we present a description of our approach. In the next section we give details about the implementation of our algorithm. Finally, we present and discuss the results of our approach.

II. Previous Works

Several methods have been proposed to overcome the multi-pass rendering limitation for multi-view rendering of 3-D information. A point-based rendering solution was proposed by Hubner *et al.* [4] using GPU to compute multi-view splatting, parameterized 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, Hubner and Pajarola [5] 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 [6] has been proposed by Morvan *et al.* a single 2-D image plus a depth map that are interrelated to display multiple views. Although the algorithm reduces 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.

In 2008, de Sorbier *et al.* [7], [8] introduced a new single pass algorithm to render stereoscopic and multi-view images using the GPU. This approach is based on a geometry shader implementation and uses multiple rendering target extension (MRT) associated with frame-buffer object (FBO) to save results in distinct textures. Results show that, in some cases, the frame-rate can be twice faster than a multi-pass technique. However, MRT is limited to a single depth buffer shared by all the rendering targets. This restriction is minimized by sorting the triangles in a back to front order that increases computation time. Moreover, hardware constraints limit the number of output textures to eight while some auto-stereoscopic devices require nine viewpoints or more. Finally, it is difficult to integrate this algorithm in an existing application because existing shaders have to be rewritten.

III. Multi-View Rendering Algorithm

In this section, we present a GPU-based multi-view rendering algorithm that takes advantage of similarities of vertices among each view and of the geometry shader to speed-up the rendering process.

A. Overview

By studying the concept of multi-view rendering [9], we can state that some characteristics remain the same from one view to another. Position of vertices is unchanged in the referential of the scene meaning that the same transformation is shared over the viewpoints. Likewise, texture coordinates, light vector and normal are identical for a given vertex. So, each characteristic independent of the viewpoint might be computed only once in order to increase the performances. Then, each vertex should be duplicated and shifted according to a given viewpoint.

In that sense, shaders [10] provide useful functionalities to merge some operations and duplicate only relevant data. A vertex shader is designed to apply several independent processing on each vertex, while a geometry shader is dedicated to handle primitives. In particular, this GPU stage allows to create or remove vertices, to emit new primitives and to apply transformations. This stage takes place just after the vertex shader.

B. Our Approach

1) Single texture based approach: Previous works [7], [8] on GPU-based multi-view rendering demonstrate that it is possible to considerably speed-up this kind of process. However, this algorithm is difficult to implement and require many modifications in the original code. The result of this approach is rendered into several textures (limited to eight) thanks to FBO and MRT extensions. One consequence is that an algorithm like the painter algorithm have to used instead of the standard depth test of the OpenGL pipeline. It can generate artifacts or increase computational time.

Our approach have been designed to overcome all these issues. The solution is to render all the views in one texture instead of several. In that case, the number of views depends on the maximal resolution of the texture and the resolution of the generated views, but remains higher compared to the previous approaches. Moreover, the depth test can now be used without any restriction. Finally, only few modifications are required and are mainly located in the geometry shader code.

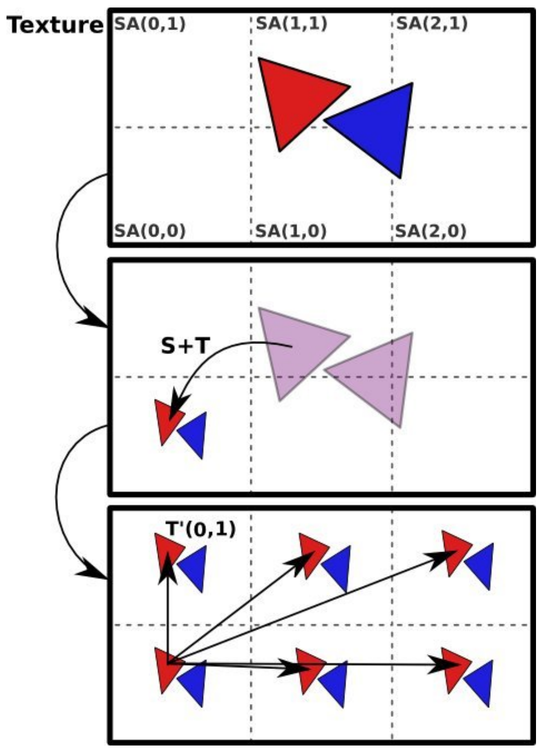

Fig. 1. Description of the transformations required to duplicate the geometry using a single texture.

In this approach, the different views have to be organized over the surface of our single texture. Each single space occupied by a view is named sub-area and defined as $SA(i, j)$ where i and j are the coordinates along the horizontal and vertical axis as depicted in Fig. 1. We defined the 2-D vector $\mathbf{NV}$ as the number of sub-areas along each axis. Of course, the total number of rendered views must be lower than the number of sub-areas. For example, five views with a resolution of (w, h) will spread over five sub-areas $SA(i, j)$ where $0 \leq i \leq 2$ and $0 \leq j \leq 1$. So the number of sub-areas is then $\mathbf{NV} = (3, 2)$, the final resolution of the texture is $(3 \times w, 2 \times h)$ and one sub-area will not be filled.

2) Geometry Transformation: In OpenGL, the *modelview* matrix M is used to transform the geometry into the coordinate system of the camera. It refers to the extrinsic parameters of the camera, while the OpenGL *projection* matrix P refers to the intrinsic parameters. The rendering is then achieved by multiplying each vertex, first with the modelview matrix and after with the projection matrix.

The goal of our approach is to duplicate and shift the input geometry using the GPU because similarities exist among the vertices of the different views. One similarity is the modelview matrix, so we need to apply it only one time on each vertex. The view shifting can be performed after using the modelview matrix on each vertex and will consist in a simple translation on x axis depending on the value of the eye separation distance Δ. This translation for the k^{th} generated view will then be defined as a vector $\mathbf{T_v}(k) = (k \times \Delta, 0, 0)$.

Since our rendering context is a single texture then the input triangles will be transformed and projected on the overall surface of the texture. So we have to apply extra operations on the triangles to transform them to fit the bottom-left sub-area of the texture. All the input views share a common image plan, so the transformation of the triangles is a 2-D operation composed of a scaling $\mathbf{S}$ and a translation $\mathbf{T}$ that are applied after the normalized OpenGL projection P of vertices. $\mathbf{S}$ and $\mathbf{T}$ are then defined as:

$$\mathbf{S} = \frac{1}{\mathbf{NV}} \qquad (1)$$
$$\mathbf{T} = -1 + \frac{1}{\mathbf{NV}}$$

By applying the transformations of Eq. 1, the triangles will be located in the sub-area $SA(0,0)$ in bottom-left. One more translation T_{SA} is then required to move the duplicated and shifted triangles into their respective sub-areas.

$$T_{SA}(i,j) = \frac{1}{\mathbf{NV}} \times 2 \times (i,j) \qquad (2)$$

The full process is depicted in Fig. 1. Using Eq. 1 and 2, the process that transforms an input vertex V_{in} into the sub-area $SA(i,j)$ for the view k can be summed up as :

$$V_{SA(i,j)} = T_{SA}(i,j) + TSP \times (T_v(k) + M \times V_{in}) \qquad (3)$$

with $TSP = T \times S \times P$.

3) Clipping: The clipping stage consists into eliminating triangles or part of triangles that are not visible in the rendering area. This process takes place after the rasterization, thus after the geometry shader. So, no data will be missing when the triangles will be shifted in the geometry shader. However, it induces that some triangles will overlap several sub-areas or will be rendered into an incorrect sub-area instead of being eliminated. This problem is presented in Fig. 2. It means that a specific clipping have to be applied just after the geometry shader stage and will depend on each sub-area borders .

Our clipping is based on the OpenGL user's defined clipping using a distance value. In the geometry shader, we compute the distance between each vertex and the four borders of the sub-area it should belong. If the distance is negative for one vertex, then the triangle associated with that vertex will be automatically clipped by OpenGL.

IV. IMPLEMETATION

This section describes the implementation of our algorithm using OpenGL 2.1 and GLSL 1.2. The result of our method is saved in a texture using the Frame Buffer Object extension.

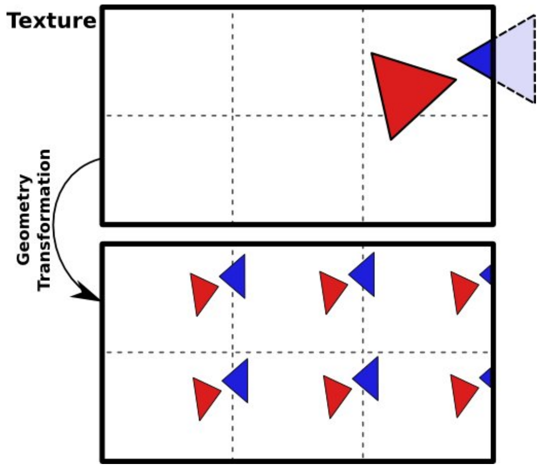

Fig. 2. Some triangles can overlap two different sub-area. A specific clipping is then required to eliminate undesirable pixels.

A. The Vertex Shader

The goal of the vertex shader is to centralize the common operations from one view to another one. Eq. 3 shows that the transformation matrix M (*MODELVIEW* matrix) is similar for each vertex. So each vertex can be multiplied with that matrix during the vertex shader stage. In the same way, normals, texture coordinates, color, can be defined only one time for each vertex.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

flat varying ivec2 SA;
uniform vec2 NV;
uniform int numberofviews;
uniform float eyesep;
vec2 T = -1.0+1.0/NV;
mat4x4 TSP = mat4x4(1.0/NV.x,0.0,0.0,0.0,
                    0.0,1.0/NV.y,0.0,0.0,
                    0.0,0.0,1.0,0.0,
                    T.x,T.y,0.0,1.0)*gl_ProjectionMatrix;
vec4 Tv = vec4(-float(numberofviews*0.5)*eyesep,0,0,0);

void main(void){
  if(mod(numberofviews,2)==0) Tv.x += eyesep*0.5;
  for(int k=0;k<numberofviews;++k){
    SA.x = k%int(NV.x);
    SA.y = int(floor(k/NV.x));
    for(int i=0; i<3; ++i){
          vec4 tmp = TSP*(Tv+gl_PositionIn[i]);
          vec2 coeff =  2.0*NV*tmp.w;
          gl_ClipDistance[0]=tmp.x+tmp.w;
          gl_ClipDistance[1]=coeff.x-(tmp.x+tmp.w);
          gl_ClipDistance[2]=tmp.y+tmp.w;
          gl_ClipDistance[3]=coeff.y-(tmp.y+tmp.w);
          gl_Position = tmp;
          gl_Position.xy += SA/NV*tmp.w*2.0;
          EmitVertex();
    }
    EndPrimitive();
    Tv.x += eyesep;
  }
}
```

Listing 1. One possible code for the geometry shader

B. The Geometry Shader

In the geometry shader, we apply the transformations presented in the previous section. The code corresponding to this step is introduced in listing 1. For each sub-area, vertices of the input triangles are duplicated and translated according to the corresponding viewpoint. The result is multiplied with the OpenGL projection matrix, translated, and scaled to fit the subareas. All this operations must in homogeneous coordinates to correspond with OpenGL matrix.

Since transformations are applied in camera's reference system, positioning the viewpoint corresponds to a simple translation on x axis based on the eye separation distance $eyesep$. The value Tv is used to define this translation.

To avoid a sub-area to overlap another one, we apply the clipping at the geometry shader stage. It consists in defining the distance from the top, left, right and bottom borders of the corresponding sub-area. OpenGL clipping will be applied if one of the distances is negative. Of course, this process requires to activate user clipping planes in the OpenGL program.

C. The Pixel Shader

The main advantage of our approach is that the pixel shader does not require any modification. Since our clipping is a standard operation of the pipeline, it only needs to be set up in the geometry shader.

So we can applied any kind of per-pixel operation like illumination per pixel like in Figure 3. Moreover, multi-pass rendering algorithm are still available like in Figure 4 which depicted a two-pass toon shading with border. And finally, Figure 5 presents a simple texturing of surfaces.

V. Results

We experimented our algorithm on a bi-Xeon 2,5Ghz running Linux. The graphic card is a nVIDIA GeForce GTX 285 with 1Go memory. The algorithm was tested using different kind of models with various numbers of triangles and graphical effects. The resolution of each view is 1024×768 . No special data structure like Vertex Buffer Objects was used.

Figure 6 presents the performances obtained using our approach compared to the standard multi-pass rendering. We evaluate the results over different number of triangles and views. If the scene is made of one thousand triangles then we notice that performances of our algorithm are similar or, in case of two views, slightly worst compare to the normal one.

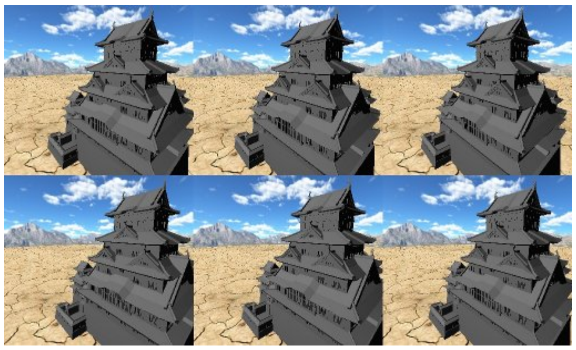
Fig. 3. Himeji Castle, 6 views. 35,200 triangles. Lighting per pixel.

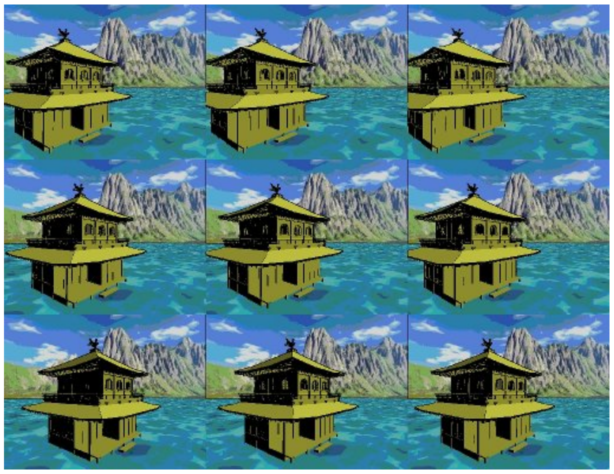
Fig. 4. Kyoto Golden Pavilion, 9 views. 23,400 triangles. Toon shading effect with borders emphasis.

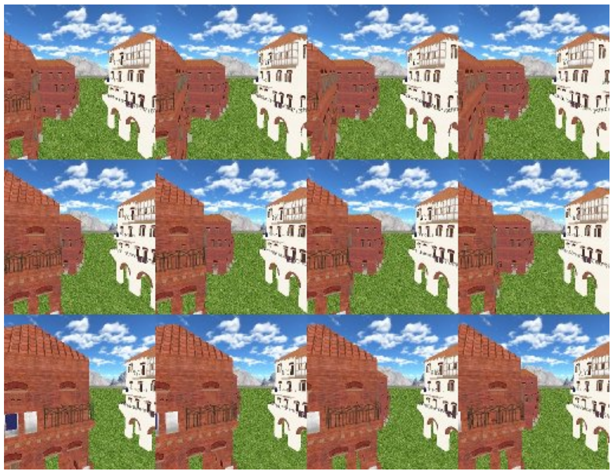
Fig. 5. Rome from City Engine (procedural.com), 12 views. 86,300 triangles. Texturing with shaders.

In all other analyzed conditions, performances of our multi-view rendering are around twice better than the standard multi-view rendering. Rendering with four view-points shows that our results can be three time better for more than 5000 triangles. Especially in this case, the differences between two and four views are small.

Performances are closely dependent of the number of input primitives. For low number of triangles, our approach is less effective than the standard one because the number of OpenGL drawing calls does not exceed the transfer capabilities from the main memory to the graphic card.

Results are similar for two and four views. So, in Figure 7, we analyze our rendering algorithm with one to 17 view-points and 10000 triangles. In the first fourth cases, performances are quite similar then after, an important drop in frame-rate occurs until 13 view-points. Finally, performances seem to become stable again. We think that under a given amount of data, a

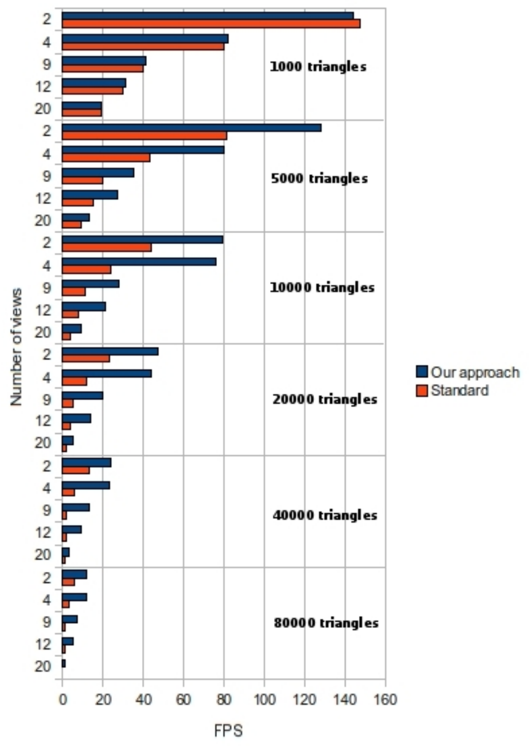

Fig. 6. Performances of our approach compared to the standard one. Tests are applied with various numbers of views and triangles.

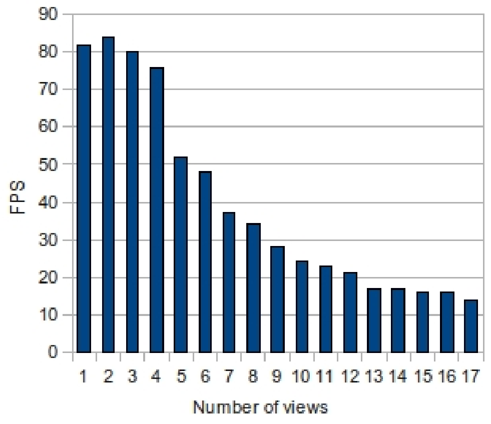

Fig. 7. Frame-rate of our approach on a scene with 10,000 triangles in function of the number of views.

geometry shader can parallelize operations but it will become a bottleneck in the other case.

The frame-rate for one view is less than the one for two views. This is because our algorithm apply some operations that are useful for multi-view rendering but make no sense for a single view.

VI. Conclusion

We have presented an algorithm to generate stereoscopic and multi-view images using the GPU in a single pass. We take advantage of the geometry shader to speed-up the rendering process by duplicating the geometry on the graphic card and avoiding redundant computations. Our algorithm overcomes the limitations introduced in previous works, such as shared depth buffer and restrictions on the number of output-views. Moreover, the implementation of this approach is now simplified.

We generate all the views on a single texture which requires only one depth buffer. We explained the transformations applied on the input triangles to duplicate and spread them over the single texture. We also introduced a solution based user's defined clipping plane to easily resolve the clipping problems when a duplicated triangle overlaps two views.

The results showed that performances of our approach vary according to the number of triangles and views. The algorithm is efficient when it processes more than 1000 triangles otherwise, benefits of the geometry shaders are under-exploited. We also noticed that the frame-rate is quite similar to render two, three or four views but decreases while rendering more views because geometry shader becomes a bottleneck. But our approach always remains better than the standard multi-pass rendering. Results are two times higher or even three times for example in case of four views rendering.

In future, we can expect better results using our approach since the reason of the main limitation is the hardware bottleneck at the geometry shader stage. Future graphic cards should be able to be fully compliant with our multiple-view rendering algorithm.

Acknowledgment

Part of the work presented in this paper was supported by the FY2009 Postdoctoral Fellowship for Foreign Researchers from the Japan Society for Promotion of Science (JSPS) and by the National Institute of Information and Communications Technology (NICT).

References

[1] N. A. Dodgson, "Autostereoscopic 3d displays," *Computer*, vol. 38, no. 8, pp. 31–36, 2005.
[2] Y. Takaki, "High-density directional display for generating natural three-dimensional images," in *Proceedings of the IEEE*, vol. 94, no. 3, 2006, pp. 654–663.
[3] B. Lichtenbel and P. Brown, "Ext_gpu_shader4 extensions specifications," *NVIDIA*, 2007.
[4] T. Hübner, Y. Zhang, and R. Pajarola, "Multi-view point splatting," in *GRAPHITE*, 2006, pp. 285–294.
[5] T. Hübner and R. Pajarola, "Single-pass multi-view volume rendering," in *IADIS*, 2007.
[6] 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.
[7] F. de Sorbier, V. Nozick, and V. Biri, "Accelerated stereoscopic rendering using gpu," in *16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2008 (WSCG'08)*, ser. ISBN 978-80-86943-16-9, Feb. 2008. [Online]. Available: http://wscg.zcu.cz/wscg2008/wscg2008.htm
[8] F. de Sorbier, V. Nozick, and V. Biri, "Gpu rendering for autostereoscopic displays," in *4th International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08)*, Jun. 2008, electronic version (7 pp.).
[9] N. Dodgson, "Autostereoscopic 3D displays," *Computer*, vol. 38, no. 8, pp. 31–36, 2005.
[10] R. J. Rost, *OpenGL(R) Shading Language (3rd Edition)*. Addison-Wesley Professional, July 2009.