# Depth Camera Based System for Auto-Stereoscopic Displays

François de Sorbier [1], Yuko Uematsu [2], Hideo Saito [3]

*Graduate School of Science and Technology, Keio University*
*3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Kanagawa, 223-8522, Japan*
[1] fdesorbi@hvrl.ics.keio.ac.jp
[2] yu-ko@hvrl.ics.keio.ac.jp
[3] saito@hvrl.ics.keio.ac.jp

***Abstract*—Stereoscopic displays are becoming very popular since more and more contents are now available. As an extension, auto-stereoscopic screens allow several users to watch stereoscopic images without wearing any glasses. For the moment, synthetized content are the easiest solutions to provide, in real-time, all the multiple input images required by such kind of technology. However, live videos are a very important issue in some fields like augmented reality applications, but remain difficult to be applied on auto-stereoscopic displays. In this paper, we present a system based on a depth camera and a color camera that are combined to produce the multiple input images in real-time. The result of this approach can be easily used with any kind of auto-stereoscopic screen.**

## I. Introduction

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 five to 64 input images [2] to display a single 3D frame. A filter, made of small lenses or thin 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.

In computer graphics, the input images can be easily generated [3]. For video streams, a basic solution consists in using as many cameras as needed input images [4], [5]. Even so, such approaches require to configure precisely the capture system, can be difficult to move or are designed for only one specific device.

To reduce the number of cameras, a solution using online video-based rendering method has been proposed by Nozick and Saito [6]. It takes advantage of the GPU to generate up to 16 views thanks to a capture system made of only four web-cameras. The images can be computed and displayed in interactive time (around 15 frames per second), but it works only in a limited area and needs to be configured precisely.

Another solution has been proposed by Fehn [7]. A single image with its corresponding depth information is used to generate images from new viewpoints. However, this kind of approach is mostly applied with video file since the depth map cannot be generated in real time. Some specific cameras [8] are able to capture videos with the corresponding depth-map, but are still prototypes, cannot process the data in real-time or have a very small resolution. Lee *et al.* have proposed a system to generate multiple 2D-Plus-Depth images by using several cameras and a single depth camera [9]. The depth camera is used to estimate an initial depth map for each color camera. The result is refined by applying a belief propagation method. The authors have improved this system by adding more depth cameras [10].

Our goal is to create a real-time multi-view rendering system designed for live video streams. Our approach is based on a depth camera that can provide depth information in real-time. A color camera is also added in order to capture the color information.

This paper is structured as follow. We start by giving an overview of our depth and color cameras based system that takes advantage of a 3D-mesh to correct the mismatching between the two viewpoints and to generate the images corresponding to the new viewpoints. In the next section, we describe our algorithm used to speed-up the multi-view rendering process of a mesh thanks to the GPU.

## II. Depth-Camera Based System

In this section, we describe our capture system composed of a depth camera and a color camera. We also explain the steps to generate the depth map corresponding to the viewpoint of the color camera by creating a mesh from the data transmitted by the depth camera.

### A. Details About our System

A depth camera is a device that captures and transmit the depth information corresponding to a real environment. The camera of our system is based on the Time Of Flight (TOF) technology [11] and generates the depth map of a scene in real time. For each pixel of the low resolution sensor of the camera ($176 \times 144$) a depth value, a 3D coordinate, a grayscale component and a confidence value are available. These values are computed according to the time required by an infrared light to go and return to the camera's sensor.

*MMSP'10, October 4-6, 2010, Saint-Malo, France.*
*978-1-4244-8112-5/10/$26.00 ©2010 IEEE*

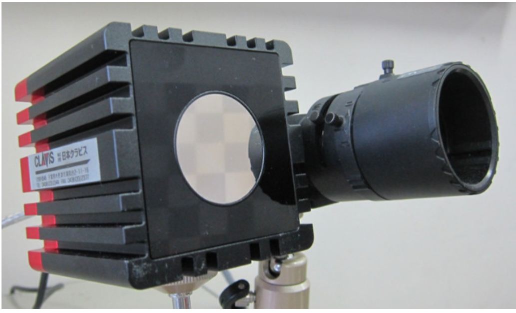

Fig. 1. A color camera is added in the system since our depth camera cannot capture the color information.

Since, this camera is not able to capture any color information, another camera is necessary. Color is also an important cue for the perception of depth [12], [13], so we use a high resolution camera to compensate the low resolution of the depth camera. This color camera is added besides the depth camera as presented in Fig. 1.

These two cameras are not located at the same position, so their viewpoints are slightly different. Combining directly the depth information with the color image will result into a discordance between the depth perception and the displayed image. Therefore, a transformation is required to map the depth information onto the color image. It consists in creating a mesh from the data captured by the depth camera and projecting the result onto the color image. This process can be done only if the cameras are calibrated in the same coordinate system. This step is described in the following sub-section.

### *B. Calibration*

The pose estimation of depth and color cameras has to be defined in a same coordinate system in order to be able to project the mesh generated from the depth map onto the color image.

For each pixel of the depth image, the depth camera provides a corresponding 3D coordinate. This set of points is defined in a coordinate system wherein the depth camera's position is the origin. Following this statement, we also set the depth camera as the origin of our capture system. Thus, the calibration stage only requires to estimate the pose of the color camera in relation to the depth camera.

Using a set of 2D/3D correspondences is a common way to estimate the pose of a camera [14]. In our case, the 3D coordinates can be easily retrieved thanks to our depth camera.

First, 2D correspondences are found between the data from the depth camera and the color image by using a chessboard pattern or by clicking pixels. Three kinds of images provided by the depth camera are used and associated with the color image to define these correspondences as depicted in Fig. 2. The depth and gray-scale images are used to select relevant points, whereas the confidence map is used to check the validity of the depth value computed by the camera (white areas represent a high confidentiality).

Second, for each pixel of the images generated by the depth camera, a 3D coordinate exists. So, a list of 2D/3D correspondences between the color image and the 3D space can be created. OpenCV [15] is then used to compute the pose estimation of the camera based on these 2D/3D correspondences and intrinsic parameters of the camera.

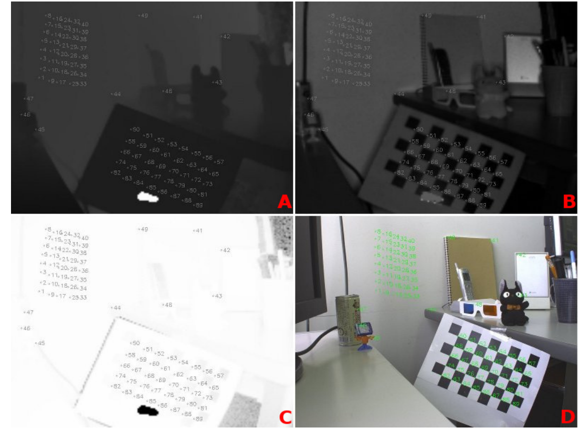


Fig. 2. Four images are used for the calibration. (A) the depth map, (B) the corresponding gray scale image, (C) the confidentiality map and (D) the color Image

### *C. Generating the New Depth Map*

In order to compute the depth map corresponding to the viewpoint of the color camera, our objective is to take advantage of a mesh generated from the 3D coordinates provided by the depth camera. This relation between the depth camera, the color camera and the mesh is depicted in Fig. 3. Using a computer graphics based approach can significantly reduce the processing time in comparison to a Depth Image Based Rendering (DIBR) approach.

The intrinsic and extrinsic parameters, computed during the calibration stage, are used to set up the position of the viewpoint for the rendering. The mesh, made of triangles and defined in the coordinate system of the depth camera, will then be observed from the viewpoint of the color camera. The mesh is finally rendered using this configuration.

The rendering process will also automatically generate a depth map but the computed z-values are not linear due to a

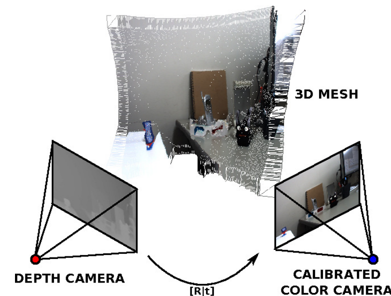


Fig. 3. The goal is to use a mesh obtained thanks to the data of the depth camera to compute the depth map corresponding to the viewpoint of the color camera.

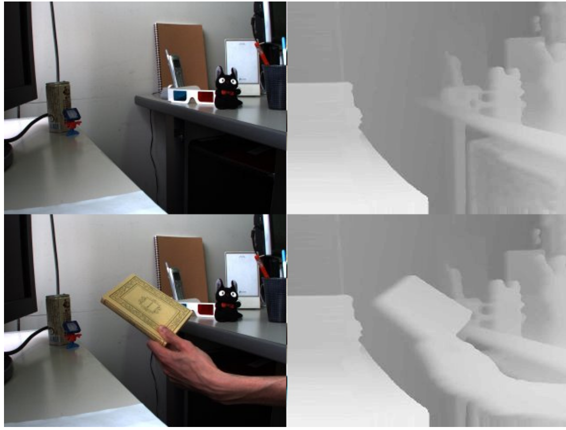

Fig. 4. Results presenting color images with their corresponding depth-map generated thanks to our approach.

non-linear transformation. A GPU-based program is then used to convert this depth map into a metric depth information.

In our approach, occlusions are reduced because we are using only one mesh in which all the points are connected with triangles. So occlusions are replaced by an automatic interpolation between two different depths. However, depth data can also be missing on the border of the mesh. Our solution is to extrude the borders of the mesh as presented in the Fig. 3 but will generate flat areas in the depth map. Examples of results are presented in Fig. 4.

A matching between the color image and the depth map is presented in Fig. 5 and shows that errors are mainly located in flat areas previously described, in areas close from the depth camera, on specular objects and on occlusion areas.

With our approach, the depth map corresponding to the color camera's viewpoint is generated in real time. The rendering time is only limited by the frame-rate of the cameras.

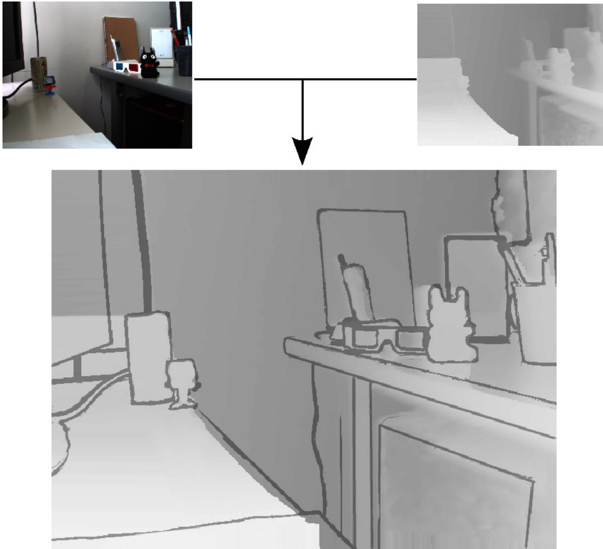

Fig. 5. Comparison between the edges color image and its corresponding depth-map. Errors are mainly located in occlusions and close areas. Especially, the screen on the left part is not visible from the depth camera, so depth information doesn't match in this area.

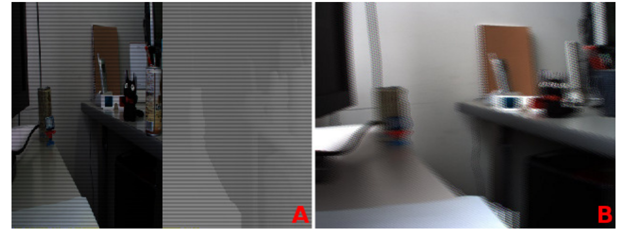


Fig. 6. Two input image formats for auto-stereoscopic displays. (A) Philips' 2D-plus-depth format. (B) Tridelity's sub-pixel alignment.

### D. Application to Auto-Stereoscopic Displays

Content for auto-stereoscopic displays [16], [17] can be divided into two categories. The first one is the *2D plus Depth* format [7] that only requires a color image and its corresponding depth map. The second one consists in generating and mixing the different views before displaying the result.

The Philips' display [18] use an integrated rendering algorithm based on the *2D plus Depth* technology. The input image format is a disparity map interleaved with the color image as depicted in Fig. 6. Our approach can easily provide the color image with its corresponding depth map. Depth information is easily and quickly transformed into a disparity and mixed with the color image map by using the GPU to fit the requirement of the *2D plus Depth* format.

Other displays like [19] use a sub-pixel alignment of several input images into a single one (Fig. 6). In our approach, a 3D mesh reduces significantly the complexity of the multi-view rendering because we can use the same stereoscopic rendering algorithm than in computer graphics. It consists in translating the viewpoint along a specific axis according to the eye separation distance and the view direction. However, the color information has to be associated with the mesh to be correctly displayed when the mesh will be rendered from a new viewpoint.

A solution is to map the color image on the mesh by using a projective texture technique [20] that generates automatically the texture coordinates on the mesh. This method requires to define a projective texture matrix as follows:

$$\mathbf{M}_{texture} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix} \times \mathbf{M}_{projection} \times \mathbf{M}_{modelview} \quad (1)$$

Where $\mathbf{M}_{projection}$ and $\mathbf{M}_{modelview}$ are matrices defined with the converted intrinsic and extrinsic parameters of the color camera. The texture coordinates are obtained by multiplying each vertex of the mesh by this matrix using the GPU for example. Then, the color information will be correctly displayed, even if the viewpoint is modified.

However, multi-view rendering is often considered as a slow process since each view requires a specific rendering pass. Computational time can be very high, especially when the geometry is made of thousands of triangles (25000 in our

case). In the next section, we present a GPU-based multi-view rendering algorithm applied to speed-up that process.

## III. Multi-View Rendering Algorithm

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

### A. Overview

By studying the concept of multi-view rendering [1], we can state that some characteristics remain the same from one view to another. Position of vertexes 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 [21] 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 vertexes, 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 [22], [23] 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.

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. 7. We defined the 2D 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

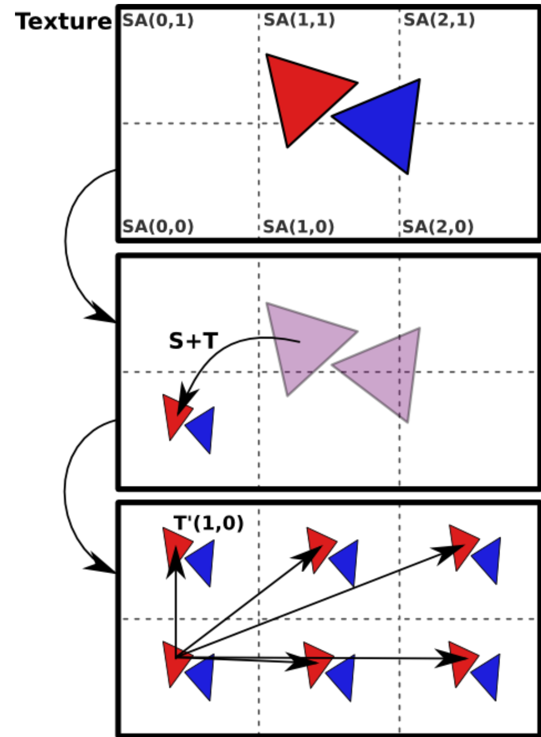


Fig. 7. Description of the transformations required to duplicate the geometry using a single texture.

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 vertexes 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 $\Delta$. 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 2D 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:

$$\mathbf{S} = \frac{1}{\mathbf{NV}} \tag{2}$$
$$\mathbf{T} = -1 + \frac{1}{\mathbf{NV}}$$

By applying the transformations of Eq. 2, 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.

$$T_{SA}(i, j) = \frac{1}{\mathbf{NV}} \times 2 \times (i, j) \tag{3}$$

The full process is depicted in Fig. 7. Using Eq. 2 and 3, 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 :

$$V_{SA(i,j)} = T_{SA}(i, j) + TSP \times (T_v(k) + M \times V_{in}) \tag{4}$$

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 rasterisation, 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. 8. 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.

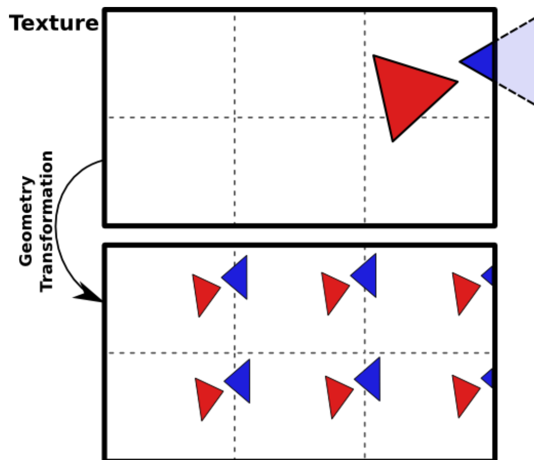


Fig. 8. Some triangles can overlap two different sub-area. A specific clipping is then required to eliminate undesirable pixels.

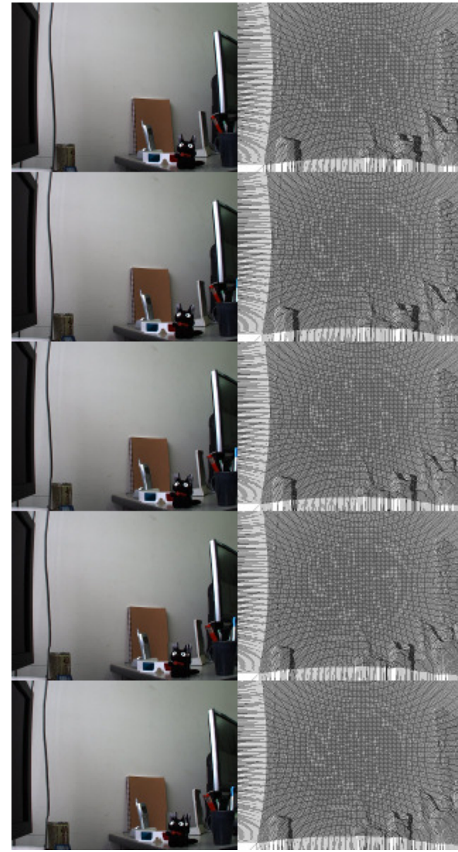

Fig. 9. Set of five images generated with our GPU based multi-view approach. For each view, the corresponding transformed mesh is presented.

### C. 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. Our depth mesh is made of 25000 triangles ($176 \times 144$). The resolution of each generated view is $1024 \times 768$.

For one view the frame-rate is around 51 frame/sec. Applying a standard multi-pass rendering for five views, the frame-rate drops to 23 frame/sec. Using our approach, we can increase our frame-rate to 38 frame/sec. An example of result of our approach is presented in Fig. 9.

## IV. Conclusion

In this paper, we have presented a capture system for autostereoscopic displays based on a depth camera. Since the depth camera is not able to capture color information, we added a color camera at a slightly different position. An algorithm using a 3D mesh is applied to match the color and depth information and produce the input views required by the autostereoscopic displays.

We also presented a GPU-based multi-view rendering algorithm to speed-up the rendering process from several viewpoints. Similarities exists among the vertexes from one view

to another like position, color, normal. So we take advantage of the geometry shader to compute these properties only one time and to operate a duplication and transformation of the geometry for each viewpoint. This approach requires only few modifications compare to the original single view rendering code. Finally, our method is only limited by the frame-rate of the cameras (30 images per second). Results have been applied with success on two auto-stereoscopic displays using two different technologies.

For the moment, our approach is based on a single mesh and a projective texture. The consequence is that the image quality is decreased in occlusion areas. We plan to use a separate our generated mesh into several layers and fill the missing color information thanks to an in-painting method.

## V. 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). This work is partially supported in part by a Grant-in-Aid for the GCOE for high-Level Global Cooperation for Leading-Edge Platform on Access Spaces from MEXT, Japan. This research is partially supported by National Institute of Information and Communications Technology, Japan.

## References


[1] N. 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," *Proceedings of the IEEE*, vol. 94, no. 3, pp. 654–663, 2006.

[3] M. Halle, "Autostereoscopic displays and computer graphics," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 104.

[4] 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 (TOG)*, vol. 23, no. 3, pp. 814–824, 2004.

[5] (2010) 3d-cambox website. [Online]. Available: http://www.3dtvsolutions.com/en/

[6] V. Nozick and H. Saito, "Multiple view computation for multi-stereoscopic display," in *IEEE Pacific-Rim Symposium on image amd video Technology (PSIVT 2007*, vol. 4872, 2007, pp. 399–412.

[7] C. Fehn, "A 3D-TV approach using depth-image-based rendering (DIBR)," in *Proc. of VIIP*, vol. 3, 2003.

[8] A. Redert, M. de Beeck, C. Fehn, W. IJsselsteijn, M. Pollefeys, L. Van Gool, E. Ofek, I. Sexton, and P. Surman, "ATTEST: Advanced three-dimensional television system technologies," 2002.

[9] E. Lee, Y. Kang, Y. Ho, and Y. Jung, "3-D video generation using hybrid camera system," in *Proceedings of the 2nd International Conference on Immersive Telecommunications*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, p. 5.

[10] Y. Kang, E. Lee, and Y. Ho, "Multi-depth camera system for 3D video generation," in *International Workshop on Advanced Image Technology*, 2010, p. 44.

[11] (2010) Mesa Imaging website. [Online]. Available: http://www.mesa-imaging.ch/

[12] T. Troscianko, R. Montagnon, J. L. Clerc, E. Malbert, and P.-L. Chanteau, "The role of colour as a monocular depth cue," *Vision Research*, vol. 31, no. 11, pp. 1923 – 1929, 1991.

[13] L. Meesters, W. IJsselsteijn, and P. Seuntiens, "A survey of perceptual evaluations and requirements of three-dimensional TV," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 3, pp. 381–391, 2004.

[14] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, ISBN: 0521540518, 2004.

[15] (2010) OpenCV website. [Online]. Available: http://opencv.willowgarage.com/wiki/

[16] J. Eichenlaub, "Progress in autostereoscopic display technology at Dimension Technologies Inc." in *Proceedings of SPIE*, vol. 1457, 1991, p. 290.

[17] K. Matsumoto and T. Honda, "Research of 3D display using anamorphic optics," in *Proceedings of SPIE*, vol. 3012, 1997, p. 199.

[18] Philips, "Wowvx for amazing viewing experiences," *Philips 3D solutions*, 2006.

[19] (2010) Tridelity website. [Online]. Available: http://www.tridelity.de/

[20] C. Everitt, "Projective texture mapping," *White paper, NVidia Corporation*, 2001.

[21] R. J. Rost, *OpenGL(R) Shading Language (3rd Edition)*. Addison-Wesley Professional, July 2009.

[22] 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

[23] ——, "Gpu rendering for autostereoscopic displays," in *4th International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT'08)*, Jun. 2008, electronic version (7 pp.).