Real-Time Depth-Image-Based Rendering for 3DTV Using OpenCL

Roberto Gerson de Albuquerque Azevedo, Fernando Ismério, Alberto Barbosa Raposo, and Luiz Fernando Gomes Soares

Department of Informatics, Pontifical Catholic University of Rio de Janeiro, PUC-Rio
Rio de Janeiro, RJ, Brazil
{razevedo,fismerio,abraposo,lfgs}@inf.puc-rio.br

Abstract. This paper proposes a real-time (performance of at least 30 fps for full-HD video) Depth-Image-based Rendering (DIBR) approach for stereoscopic 3DTV using OpenCL. Many stereoscopic 3DTV, multi-view, and Free-viewpoint TV (FTV) technologies have been based on DIBR, aiming at saving bandwidth, and to allow for user adaptation in the client-side. Unlike related work, this paper uses OpenCL for all the DIBR steps, including the re-projection in the virtual views (which is commonly performed using OpenGL, even when implemented in GPGPUs). The use of OpenCL-only can, in some cases, outperform the OpenGL z-testing performance. Two execution models have been implemented (per-line parallel, and per-pixel parallel) and tested against standard video-plus-depth test sequences to show the approach performance.

Keywords: DIBR, OpenCL, GPGPU, Stereoscopic, 3DTV.

1 Introduction

3D video is often considered one of the major upcoming innovations in video technologies [1]. Compared to 2D-only videos, the 3D video bandwidth requirement is huge to the point that current broadcast and IP networks are not prepared for it. To reduce the amount of necessary data in 3D video transmission, a common approach is to use video-plus-depth representation [2] and Depth-Image-Based Rendering (DIBR) [3] at the client-side to generate additional views. Since DIBR has the potential to reduce the amount of transmitted data [2], it has been a key part of 3DTV [4] and free-viewpoint TV (FTV) [5] technologies.

Nevertheless, DIBR has some drawbacks, such as producing unpleasant to see holes and ghost artifacts in the generated views. Such problems have attracted a lot of research in the past few years, aiming at filling the holes, and improving the quality of the generated views. Some approaches focus on pre-filtering the depth map, which can reduce the amount of produced holes, while others propose complex algorithms, based on in-painting [6] techniques. One of the main drawbacks of early DIBR implementations is, however, that they are not focused on achieving real-time performance. MPEG View Synthesis Reference Software (VSRS) [7], for instance, takes nearly one second to render one frame with 1024x768 resolution [8]. A promising approach to achieve real-time performance for DIBR is its implementation using highly-parallel architectures, such as Graphical Processing Units (GPUs) [9]. By real-time performance, we mean at least 30 frames per second for rendering a full-HD (1920x1080 pixels) video.

Aiming at achieving real-time performance in the generation of a stereoscopic pair from original color and depth data, this paper proposes a DIBR implementation purely based on OpenCL [10]. OpenCL is chosen because of its portability and availability for PCs and mobile devices. Even though the current implementation has been tested only in off-the-shelf GPUs, because it uses OpenCL it can also conceptually run in other multicore processing units, such as CPUs, hybrids of CPUs and GPUs, etc. Some performance tests are presented in the paper showing that the implementation achieves real-time requirements.

The remainder of the paper is organized as follows: Section 2 presents an overview of view synthesis based on DIBR. Section 3 discusses some related work. Section 4 introduces the parallel DIBR approach in generating stereo pairs, and its implementation using OpenCL. Section 5 presents some evaluations of the proposed implementation. Finally, Section 6 is reserved for conclusions and future work.

2 DIBR Overview

Conceptually, view synthesis based on DIBR can be divided into three main steps: *depth map preprocessing*; *3D warping*; and *hole filling*.

In the *depth map preprocessing* step, depth map is pre-processed aiming at reducing the large number of dis-occlusions that are usually generated by the 3D warping (the second step). Moreover, depth-map preprocessing also helps in reducing the noise usually present in captured depth maps, decreasing the warping error. Depending on the used algorithm, depth map preprocessing is also able to reduce the computational complexity of the hole filling (third step). As an example, some work use Gaussian filters to pre-process the depth map to completely remove the holes generated by 3D warping, making the hole filling step unnecessary. Such approach, however, has some drawbacks, such as creating non-natural geometric distortions in the generated virtual views. Other proposed techniques include: asymmetric Gaussian filter, bilateral filter; and adaptive filters; which can be implemented using parallel architectures through convolution approaches. Another alternative to improve DIBR performance is to perform depth map pre-filtering at the sender-side to improve the DIBR performance.

After depth map preprocessing, *3D warping* is used to project the pixels of the reference color image to the virtual image plane. Given a reference color image, the corresponding per-pixel depth map, and the camera parameters, DIBR warps the pixels of the reference texture image to the virtual image plane in two steps. First, it back projects the 2D pixels on the image plane to 3D points on the world's coordinate system. Second, it re-projects each 3D point on the target image plane. A complete formulation of DIBR can be found in [3] and [7]. In this paper, however, we are mainly interested in the generation of the stereoscopic-pair from the original texture and depth images, which is actually a simplification of the DIBR equation. Fig. 1 shows the virtual camera setups used in this paper DIBR approach. It follows a parallel camera setup, which, unlike convergent camera setups, does not generate vertical disparity.

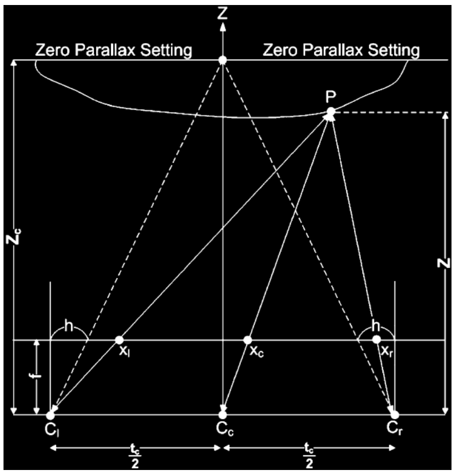

Fig. 1. Camera setup for rendering stereoscopic views (virtual views) from a central color image and depth information [11]

From Fig. 1, it is possible to derive that the pixels positions *(x_c, y)*, *(x_l, y)*, and *(x_r, y)* of the original (C_c), the virtual left view (C_l), and the virtual right view (C_r) respectively, are related by:

$$\begin{cases} x_l = x_c + \frac{t_c f}{2Z(x_c,y)} + h \\ x_r = x_c - \frac{t_c f}{2Z(x_c,y)} - h \end{cases} \tag{1}$$

where: f is the focal length; t_c is the baseline distance from the left virtual view (C_l) to the right virtual view (C_r); and h depends on the selected convergence distance (Z_c), and is given by:

$$h = -\frac{t_c f}{2Z_c} \tag{2}$$

When using gray images to represent depth information, each pixel value usually ranges from 0 to 255. The original depth distance must then be quantized in this range. Instead of a linear quantization, a common approach is to use a non-linear quantization that considers human factors and improves the perceived depth [12], such as:

$$Z = \frac{1}{\frac{1}{Z_{min}}\left(\frac{z}{255}\right) + \frac{1}{Z_{max}}\left(1 - \frac{z}{255}\right)}; z \in [0, \ldots, 255] \tag{3}$$

where Z is the depth of the real scene; Z_{min} and Z_{max} denote the nearest distance and the furthest distance, respectively; and z is the gray-scale depth value of the depth image.

After the 3D warping step, the empty pixels in the target view (named *holes* and *cracks*) can be filled by using neighboring pixel information in the *hole filling* step. Hole filling for 3D video is a very active research area. Many techniques have been proposed, such as: nearest neighbor, simple interpolation, depth-aided interpolation, in-painting, and depth-aided in-painting. Some of them are suitable to be implemented in parallel architectures, such as the nearest neighbor, and the simple interpolation. However, more complex approaches, such as those based on in-painting techniques [6], are not easily suitable for parallel or real-time implementations.

Concerning the final visual quality, in-painting techniques usually perform better. Subjective tests comparing hole filling techniques have been carried out by Vázquez *et al.* [13] and Azzari *et al.* [14]. Vázquez *et al.* endorse that "hole filling using the background pixels rather than the foreground ones as the dis-occluded areas is more reasonable by the definition of dis-occlusion". Based on this assumption our approach for hole filling takes into account the depth of neighbor pixels, in order to fill the holes, as it is detailed in Section 4.

3 Related Work

In DIBR approaches there is always a tradeoff between the quality of the produced images and the algorithm performance. Although DIBR approaches for 3DTV have been proposed since 2004 [3], the first implementations were not able to achieve real-time performance for a full-HD video.

More recently, there have been efforts, as this work, able to achieve real-time, or almost real-time performance for DIBR. A common employed technique has been the use of FPGAs (Field Programming Gate Array) [15] [16]. On the other hand, Lee et al. [17] presents a DIBR approach using GLSL (OpenGL Shading Language) for object rendering. In [18], Rogmans *et al.* report a complete system for real-time stereo correspondence, disparity estimation, refinements, and warping, also based on GLSL. Compared with the FPGAs approaches, the use of GPU is easier to program and more flexible, simplifying the test of different algorithms for the different steps of DIBR.

With the advent of GPGPUs (General Programming GPUs) in the last years, there has been a growing interest in using such a higher-level APIs to implement DIBR. For instance, Shin *et al.* [19], Shin and Ho [20], Rogmans *et al.* [21], Wang *et al.* [22], Xu *et al.* [23], Do *et al.* [24], and Zheng *et al.* [25] use CUDA for depth-preprocessing and for back-projection of pixels onto the 3D world. For the re-projection of the pixel on the virtual view, they usually use OpenGL. This can be mainly explained because when using OpenGL re-implementing *z-testing* is not needed. Similar to these works, this paper also recognizes the importance of using GPGPU to achieve real-time performance for DIBR. We have implemented DIBR using OpenCL (a non-proprietary alternative to CUDA). However, unlike the aforementioned work, and motivated by the work of Gunther *et al.* [26] (in which it is shown an approach to outperform the OpenGL performance purely using OpenCL) we have implemented all the DIBR steps using OpenCL, as detailed in the next section.

4 DIBR Implementation Using OpenCL

In general, depth map preprocessing and 3D warping are pixel-based, and have the potential to be implemented in parallel architectures. On the other hand, hole filling algorithms, mainly the ones based on in-painting techniques, are computing intensive tasks, which make them, up to now, impractical for real-time applications. As a consequence, simpler hole filling strategies are used to allow real-time performance. In special, simpler hole filling approaches can be used in the generation of a stereoscopic pair, since the holes in virtual views are usually not large, preserving good quality virtual views.

Fig. 2 shows our DIBR approach using OpenCL. The texture and the depth frame are first decoded by the CPU. A table with the horizontal movement for each possible depth is also pre-calculated by the CPU. Then, the decoded frames and the pre-calculated horizontal movements are sent to the GPU, where the DIBR process begins. The GPU DIBR implementation is based on two OpenCL kernels, related to the common DIBR steps previously mentioned: 3D warping and hole filling. We assume that the depth map does not need to be pre-processed in the client-side, since this task can be performed in the sender-side.

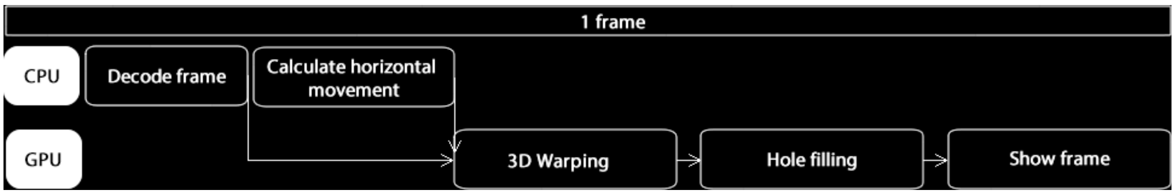

Fig. 2. DIBR Implementation using OpenCL.

4.1 Pre-calculating Horizontal Movements

In the first step of the DIBR approach, the final horizontal movement of each possible depth [0, 255] is pre-calculated, which allows for avoiding redundant calculation and for improving the performance of the 3D warping kernel, similarly to the look-up table approach of [11]. As shown in formulas (1) and (2), the horizontal movement depends only on the camera parameters, and should be updated only when such parameters changes (*e.g.*, following user preferences). The amount of horizontal movement is calculated as follows:

$$H(z) = fx\left(\frac{1}{Z(v)} - \frac{1}{Z_c}\right) \qquad (4)$$

This step could also be calculated by the GPU, but as the table contains 255 values and should be updated only when camera parameters (or user preferences) change, its implementation by the CPU does not incur in much overhead. These pre-calculated horizontal values are sent, together with the color frame and the depth frame, to the 3D warping kernel. The amount of horizontal movement table is also sent to the GPU as constant memory, which allows further optimizations.

4.2 3D Warping in Parallel

The 3D warping kernel receives the original view color frame, the depth frame, and the pre-calculated depth-to-horizontal movement table, and shifts the pixels, for left- and right-views, according to the horizontal movement table. The 3D warping kernel output is the virtual left- and right-views. One of the main problems of the warping algorithm is that multiple pixels can be mapped to the same final virtual view position. When this happens, it should be assured that only the pixel closest to the viewer will get drawn on the final virtual view. Therefore, the central problem to calculate the final pixel positions in DIBR parallel implementations is to ensure a thread-safe depth test.

As a special case of DIBR, in parallel camera setups such as the one of Fig. 1, the pixels will only shift horizontally, and, as consequence, only pixels that are in the same line can be mapped to the same final position. Thus, one way to avoid thread-safe depth test is to parallelize only the computation of entire lines. This means that each kernel thread execution must be responsible to calculate all the pixels of a same line of a frame. This approach, which we call the *per-line parallel*, has been implemented and tested against the more general *per-pixel parallel* kernel execution.

In the per-pixel parallel model, the 3D warping kernel runs for each pixel. Each kernel thread execution is responsible to warp a pixel to its final position. In this approach, the aforementioned concurrency problems (see Fig. 3) can happen. Thus, to ensure a thread-safe depth test, we have used a global depth buffer, shared over all computing units of the GPU; *i.e.,* a buffer that is in the OpenCL global memory.

If the kernel only had access and render the depth buffer, atomic operations, using *atomic_min* OpenCL function, would be enough to guarantee thread-safe access. However, after depth-testing, the access to the color buffer is also needed. Since there is no built-in function in OpenCL to guarantee such a thread-safe access to more than one memory position, we have implemented a semaphore-like synchronization mechanism, and have added an additional *lock* buffer to inform if a certain pixel position is locked or not. Before reading or writing to depth or color buffers, the 3D warping kernel must first acquire lock of the corresponding pixel position. We use *atomic_cmpxchg* operation to implement such lock feature.

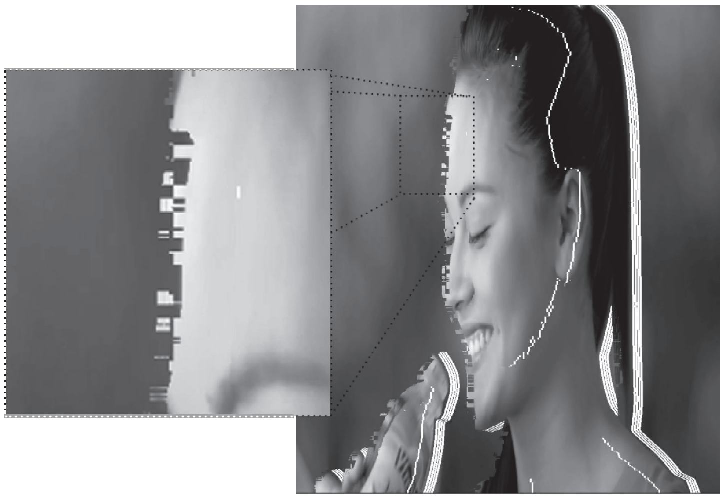

Fig. 3. Example of warping multiple pixels to the same target position without a thread-safe access to the depth buffer (the frame is shown without hole filling)[1]

[1] Image created using video sequence available from Triaxes (`http://www.triaxes.com`).

4.3 Hole Filling in Parallel

The last OpenCL kernel, the *hole filling*, is responsible for filling the holes that appear after 3D warping. As previously mentioned, due to the real-time execution requirement, it is not possible to use costly in-painting-based approaches. Therefore, a simpler heuristic is used.

Our implementation searches for colored pixels in the hole neighborhood but that are in the same line of the hole. It then selects the ones with lowest depth values, *i.e.*, the background pixels, and fills the hole with a weighted average of the neighborhood pixels. The weight is based on the distance from the pixel to the hole. This approach takes into account that pixels belonging to the background are probably the most correct ones to fill dis-occlusion holes. Fig. 4 shows an example of a frame before and after hole filling kernel execution.

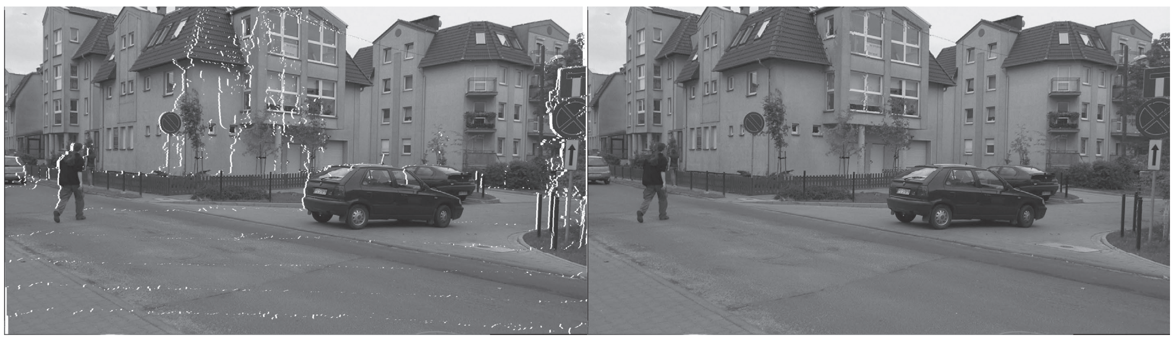

Fig. 4. Example of a frame before (left) and after (right) hole filling kernel execution

5 Evaluation

In order to test our proposal, we have run it for two video-plus-depth test sequences provided by Poznan University [27]. Table 1 shows the results of the execution on a laptop computer with a Core i7 CPU, 8 GB of RAM, and an Nvidia GeForce GT 740M GPU. For comparison purposes, the table shows the average frame per-second achieved by the per-line parallel kernel execution, and the per-pixel parallel kernel execution, discussed in the previous section. The stages that are taken into account to calculate Table 1 results are: computation of the horizontal movement table (when necessary); sending frames to GPU; 3D warping; hole filling; and frame display. Decoding is not taken into account. The global work-group size for each test is the size of the final images, while the local work group sizes are 9 and 16x9 for the per-line and per-pixel kernel execution models, respectively.

As can be noted in Table 1, the DIBR using OpenCL reaches real-time performance for full-HD video, and almost real-time performance for stereoscopic full-HD (2x horizontal resolution of full-HD), when running in the per-pixel parallel execution kernel. The per-line parallel execution kernel has the advantage of not explicitly requiring concurrent synchronization mechanisms. However, runs slower than the per-pixel parallel execution kernel. A possible explanation for the slower performance of the per-line parallel execution kernel is that the higher granularity of the data partition fails to fully exploit the parallel capabilities of the GPU configuration.

Table 1. Performance of the two implemented DIBR approaches (per-line parallel and per-pixel parallel) using OpenCL

Test Sequence	Size	Per-line parallel (fps)	Per-pixel parallel (fps)
Poznan_Street	1280x720 (HD)	40.6	83.4
	2560x720 (Stereo HD)	19.3	45.4
	1920x1080 (Full-HD)	**17.7**	**39.8**
	3840x1080 (Stereo Full-HD)	8.6	21.1
Poznan_Hall2	1280x720 (HD)	41.6	89.5
	2560x720 (Stereo HD)	19.9	48.6
	1920x1080 (Full-HD)	**18.1**	**42.7**
	3840x1080 (Stereo Full-HD)	8.7	22.3

6 Conclusions

This paper proposes a DIBR implementation using OpenCL aiming at achieving real-time performance for full-HD stereoscopic video. Unlike previous related work, our DIBR approach relies completely on OpenCL. Two approaches – one based on a per-line parallel kernel execution avoiding concurrency problems, and another based on a per-pixel parallel execution, which requires the explicit use of OpenCL synchronization mechanisms – have been implemented and tested in off-the-self GPUs. Tests with standard video-plus-depth sequences show that with the per-pixel parallel approach it is possible to achieve a performance of 40 fps for full-HD videos; a performance better than real-time rates (30 fps). In contrast, the per-line parallel approach achieves performance of approximately 18 fps. These results are similar to other approaches that do not rely completely in GPGPU language, but use OpenGL for the re-projection step.

Future work include the use of GPU to decode color and depth frames, the support to multiview-plus-depth, and the use of arbitrary (possibly with movement) camera configurations. When using arbitrary camera configurations, mainly the ones with wide baseline, more advanced hole filling approaches, possibly based on in-painting techniques, will be necessary. As a consequence, how to efficiently map such algorithms, which are not easily parallelizable, to the GPGPU must also to be handled.

References

1. Dufaux, F., Pesquet-Popescu, B., Cagnazzo, M. (eds.): Emerging technologies for 3D video: creation, coding, transmission, and rendering. John Wiley & Sons Inc., Chichester (2013)
2. Müller, K., Merkle, P., Wiegand, T.: 3-D Video Representation Using Depth Maps. Proc. IEEE 99, 643–656 (2011)

3. Fehn, C.: Depth-image-based rendering (DIBR), compression, and transmission for a new approach on 3D-TV. In: Proc SPIE, vol. 5291, pp. 93–104 (2004)
4. Zhu, C., Zhao, Y., Yu, L., Tanimoto, M. (eds.): 3D-TV System with Depth-Image-Based Rendering. Springer, New York (2013)
5. Tanimoto, M.: FTV: Free-viewpoint Television. Signal Process. Image Commun. 27, 555–570 (2012)
6. Guillemot, C., Le Meur, O.: Image Inpainting: Overview and Recent Advances. IEEE Signal Process. Mag. 31, 127–144 (2014)
7. Tian, D., Lai, P.-L., Lopez, P., Gomila, C.: View synthesis techniques for 3D video. In: Proc SPIE, vol. 7443, pp. 74430T–74430T–11 (2009)
8. Zhao, Y., Zhu, C., Yu, L.: Virtual View Synthesis and Artifact Reduction Techniques. In: Zhu, C., Zhao, Y., Yu, L., Tanimoto, M. (eds.) 3D-TV System with Depth-Image-Based Rendering, pp. 145–167. Springer, New York (2013)
9. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 80–113 (2007)
10. Munshi, A.: OpenCL programming guide. Addison-Wesley, Upper Saddle River (2012)
11. Park, Y.K., Jung, K., Oh, Y., Lee, S., Kim, J.K., Lee, G., Lee, H., Yun, K., Hur, N., Kim, J.: Depth-image-based rendering for 3DTV service over T-DMB. Signal Process: Image Commun. 24, 122–136 (2009)
12. Fan, Y.-C., Chi, T.-C.: The Novel Non-Hole-Filling Approach of Depth Image Based Rendering. In: 3DTV Conference: The True Vision-Capture. Transmission and Display of 3D Video, pp. 325–328. IEEE (2008)
13. Vázquez, C., Tam, W.J., Speranza, F.: Stereoscopic imaging: filling disoccluded areas in depth image-based rendering. In: Javidi, B., Okano, F., Son, J.-Y. (eds.) Proc. SPIE, p. 63920D–63920D–12 (2006)
14. Azzari, L., Battisti, F., Gotchev, A.: Comparative analysis of occlusion-filling techniques in depth image-based rendering for 3D videos. In: Proceedings of the 3rd Workshop on Mobile Video Delivery, pp. 57–62. ACM (2010)
15. Chen, W.-Y., Chang, Y.-L., Chiu, H.-K., Chien, S.-Y., Chen, L.-G.: Real-time depth image based rendering hardware accelerator for advanced three dimensional television system. In: 2006 IEEE International Conference on Multimedia and Expo., pp. 2069–2072. IEEE (2006)
16. Bondarev, E., Zinger, S., De With, P.H.N.: Performance-efficient architecture for free-viewpoint 3DTV receiver. In: 2010 Digest of Technical Papers International Conference on Consumer Electronics (ICCE), pp. 65–66. IEEE (2010)
17. Lee, M.-H., Park, I.K.: Accelerating depth image-based rendering using GPU. In: Gunsel, B., Jain, A.K., Tekalp, A.M., Sankur, B. (eds.) MRCS 2006. LNCS, vol. 4105, pp. 562–569. Springer, Heidelberg (2006)
18. Rogmans, S., Lu, J., Lafruit, G.: A scalable end-to-end optimized real-time image-based rendering framework on graphics hardware. In: 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video, pp. 129–132. IEEE (2008)
19. Shin, H.-C., Kim, Y.-J., Park, H., Park, J.-I.: Fast view synthesis using GPU for 3D display. IEEE Trans. on Consum. Electron. 54, 2068–2076 (2008)
20. Shin, I., Ho, Y.: GPU Parallel Programming for Real-time Stereoscopic Video Generation. In: International Conference on Electronics, Information, and Communication, pp. 315–318 (2010)

21. Rogmans, S., Dumont, M., Lafruit, G., Bekaert, P.: Migrating real-time depth image-based rendering from traditional to next-gen GPGPU. In: 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video, pp. 1–4. IEEE (2009)
22. Wang, L.-H., Zhang, J., Yao, S.-J., Li, D.-X., Zhang, M.: GPU Based Implementation of 3DTV System. In: Sixth International Conference on Image and Graphics (ICIG), pp. 847–851. IEEE (2011)
23. Xu, K., Ji, X., Wang, R., Dai, Q.: Parallel implementation of depth-image-based rendering. In: IC3D, pp. 1–4. IEEE (2011)
24. Do, L., Bravo, G., Zinger, S., de With, P.H.: GPU-accelerated Real-time Free-viewpoint DIBR for 3DTV. IEEE Trans. on. Consum. Electron 58, 633–640 (2012)
25. Zheng, Z., An, P., Zhao, B., Zhang, Z.: Real-Time Rendering Based on GPU for Binocular Stereo System. In: Zhang, W., Yang, X., Xu, Z., An, P., Liu, Q., Lu, Y. (eds.) IFTC 2012. CCIS, vol. 331, pp. 492–499. Springer, Heidelberg (2012)
26. Günther, C., Kanzok, T., Linsen, L., Rosenthal, P.: A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds. J. WSCG 21, 153–162 (2013)
27. Domañski, M., Grajek, T., Klimaszewski, K., Kurc, M., Stankiewicz, O., Stankowski, J., Wegner, K.: Poznan multiview video test sequences and camera parameters. ISOIEC JTC1SC29WG11 MPEG. M17050 (2009)