

Decoupling Rendering and Display Using Fast Depth Image Based Rendering on the GPU

Julian Meder^{1,2(✉)} and Beat Brüderlin¹

¹ TU Ilmenau, Helmholtzplatz 5, 98693 Ilmenau, Germany
{Julian.Meder,Beat.Bruederlin}@tu-ilmenau.de

² 3DInteractive GmbH, Am Vogelherd 10, 98693 Ilmenau, Germany

Abstract. Feasible rendering of massive scenes with hardware capabilities and scene data size concurrently growing has been an active subject of research in recent decades. While past attempts showed realtime rendering on basic consumer workstations, frame rates can still be too low for the growing demand in the professional CAD industry and sectors like entertainment for stable high frame rates. The work described in this paper focuses on fulfilling this demand in the context of CAD software. A fast custom application of the Depth Image Based Rendering (DIBR) technique is used to construct a view synthesis system generating the required rate using past frames of a classical parallel renderer. Artifacts associated with this technique are addressed in a cost-effective manner.

1 Introduction

In the past decade great advances have been made in the field of culling and out-of-core techniques, enabling the interactive display of massive scenes with acceptable quality [1,9] on standard workstation hardware. Still, view dependent scene complexity combined with insufficient hardware capabilities result in frequently changing frame rates, often below the acceptable. This creates two main problems:

- **Bad viewing comfort:** A stable high frame rate is needed (60 Hz and above)
- **Input lag:** Ensuring a low user input delay prevents user irritation

Failing to comply with these requirements can have a profound impact on usability and productivity, for example motion sickness with virtual reality.

The main target applications for this paper are renderers of professional CAD. Based on the insufficient image sequence of a slow CAD renderer we generate a new one at the required frame rate. For this, Depth Image Based Rendering (DIBR) is adapted as a novel fast GPU application.

The following Sect. 2 briefly summarizes DIBR while the rest of this paper is organized as follows: Sect. 2.1 gives previous work in the field of DIBR. Section 2.2 describes possible and previously implemented DIBR applications on the GPU. Following a general description of our system in Sect. 3, the projection method and its' GPU implementation is detailed in Sect. 4. Section 5 shows the solutions to reduce artifacts. Finally, an empirical analysis is given in Sect. 6 and conclusions in Sect. 7.

2 Depth Image Based Rendering

The first appearance of the DIBR term can be found in the work of C. Fehn [7], while the previous common designation was 3D image warping [13, 18]. It involves the following pixel transformation:

$$\begin{aligned} p_t &= P_t R_t T_t T_s^{-1} R_s^{-1} P_s^{-1} p_s \\ p_t &= M_t M_s^{-1} p_s \end{aligned} \quad (1)$$

A pixel p_s is transformed from the source image to a target view screen space position p_t by inverting the source view camera’s projection P_s , view rotation R_s and translation T_s and applying the transformations of the target view’s camera.

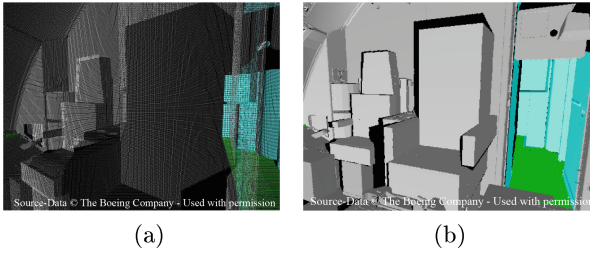


Fig. 1. Naive DIBR projections show artifacts: (a) The forward mapping problem and (b) information gaps (black)

DIBR suffers from a problem known as holes [17, 22], of which Fig. 1 shows two distinctions. The first results due to texture mapping from source to target space, which is not necessarily surjective, while the second occurs when scene geometry visible from the target view is occluded in the source view.

2.1 Related Work

One popular field of applications of DIBR has been pseudo stereoscopic image generation. The work of Fehn included a simplified version of the general 3D warp by formulating the projection only in x-direction of the screen space [7] and a gaussian source depth blur to avoid holes. Zhang et al. proposed an asymmetrical gaussian filter creating fewer distortions in the target image [22], which Horng et al. improved by additionally rotating the kernel by local depth edge directions and constricting the filter to hole regions [11].

Since these filling methods proved to be unsuitable for the general projection case, Ndjiki-Nya et al. [16] and Xi et al. [21] employed a more complex method via Inpainting combined with a background model, except the approaches were not real time capable. The latter used a higher quality projection method involving a forward projection to target space of the source depth buffer, subsequent depth hole filling and finally a backward projection to sample the color image. This idea

has also been used by Oh et al. [17] and Zinger et al. [23]. Zinger et al. additionally accelerated the forward-backward projection by only backprojecting pixels which needed to be filled in the target depth image.

Image space triangulation for general warping was used by Fu et al., who constructed a topological ordering of the generated triangles to render without a z-buffer [8]. Mark et al. proposed an online method for free camera movement, which renders necessary reference frames from predicted future viewpoints [13]. Ghiletiuc et al. applied a cube map technique in an online variant, continually creating small cube map meshes based on the last known client camera on a server and sending them to a mobile client for conventional rendering [10]. Another mesh rendering variant was proposed by Didyk et al. who use a multi pass mesh refinement through a geometry shader to create pseudo stereo images [5].

2.2 GPU-Suitable DIBR

A straightforward way of implementing DIBR on a GPU is point rendering. Every pixel of the source image is assigned a vertex with a position derived from the pixel’s texture coordinates and the corresponding depth value, which is transformed by Eq. (1) in the vertex shader. Chang et al. utilize such a simple point rendering [2]. This method clearly shows the forward mapping problem, which may be reduced by employing splatting [20]. However, only quadratic and circular reconstruction kernels are currently directly hardware-supported, which either introduces new artifacts, due to inaccurate splatting forms, or decreased performance, when implementing more complex kernels.

A second GPU-adaptable DIBR method was introduced by Morvan et al. [15], who separated the projection (1). His first step, a source space homogeneous translation, is done first line- and then column-wise for easy neighbor interpolation, solving the forward mapping problem. However, this step can have a large memory footprint and needs to write an arbitrary amount of pixels, requiring implementation through GPGPU. In contrast, the second step, a perspective texture mapping, is directly supported by the traditional GPU-pipeline [15].

GPU DIBR projection can also be implemented by using triangle meshes. The method creating the basis for this paper has been used for example by Mark et al. [13]. Vertices are placed in the center of each pixel, with each quadruple of neighboring vertices being connected via two triangles. This avoids holes caused by forward mapping, but new problems are introduced as the assumption of one continuous surface over the whole image does not hold in general (see Sect. 5). The constructed mesh is rendered conventionally as a textured model (the texture being the source color image). Some publications therefore call these meshes impostors [4, 10].

3 General System Overview

Figure 2 depicts the two parallel independent processes resident on the same hardware: the classic renderer and the DIBR projector. View camera changes

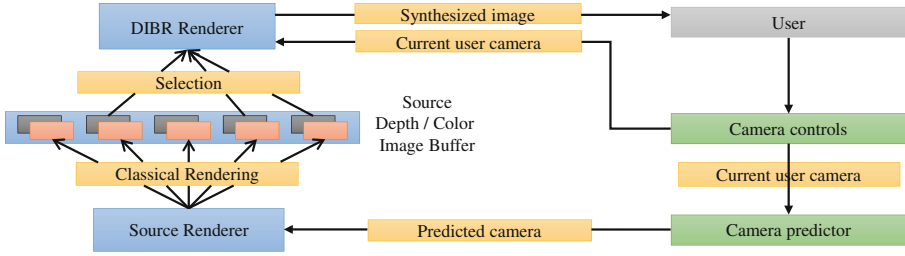


Fig. 2. The constructed system

are not directly reflected in the source renderer, but rather evaluated by a prediction algorithm generating the final cameras used for source rendering (Sect. 5.1). Rendered source frames are buffered for asynchronous retrieval by the DIBR target renderer, which uses a subset of buffered frames and the current view camera to synthesize an expected view image for the user. Consequently, the user input and the frame display are decoupled from the original renderer, ensuring low input delay and hiding varying frame rates of the source renderer.

4 On-the-Fly Tessellation Reprojection

One critical observation about the triangle mesh method of Mark et al. [13] is its' high rendering cost. Given the pixel width m and height n of a source image, the amount of triangles needed is $2(m-1)(n-1)$. We reduce this cost by dynamically tessellating the source image space for every DIBR projection. The higher efficiency of this method is shown in Sect. 6.1.

Contrary to the geometry shader method of Didyk et al. [5] a single pass tessellation shader is used. It carries more potential for concurrency, as generated vertices are processed in parallel after fixed-function tessellation, which is not possible in a geometry shader. Also, the multi-pass transform feedback has a higher performance and memory access overhead than using hardware-accelerated single pass tessellation.

4.1 Mesh Layout Considerations

The original mesh structure used by Mark et al. [13] exhibited the problem of deleting a half-pixel edge at the image's border and on edges between surfaces of different depth (see Sect. 6.2 for a comparison). To avoid this problem, every source pixel is treated as a small surface by placing the vertices on the pixel edges instead of the center. The starting mesh before tessellation however is a coarse set of $\lceil \frac{m \times n}{t_{max}^2} \rceil$ patches, $m \times n$ being the source image resolution and t_{max} the tessellation factor used.

As our approach is single pass, a binary decision whether to tessellate a patch is made. When tessellated, a coarse starting mesh with few large patches could

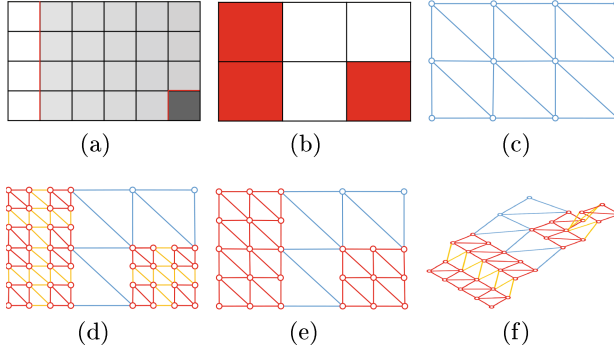


Fig. 3. (a) Edge regions of the depth map are (b) queried from the edge mipmap of a level corresponding to the (c) patch size. (d) Intermediate triangles are (e) quasi degenerated through vertex positioning. (f) A resulting projection

introduce many unnecessary triangles. Contrarily, many small patches are costly. We empirically chose a tessellation factor t_{max} of 16 for our target hardware (listed in Sect. 6).

Surfaces with largely differing source depth values need to be disjoint in the generated mesh to avoid artifacts (see Sect. 5). Hardware tessellation however produces connected triangle grids from patches. For a quasi-separation, further rows and columns of intermediate grids are introduced, which are discarded later, modifying the required tessellation factor to $2t_{max} - 1$.

4.2 Depth Guided Subdivision

For reduced memory bandwidth usage the coarse starting mesh is stored only as an indexed quad set in GPU memory, while the source image space positions are generated during vertex processing (vertex shader) from the vertex ids.

To decide on patch subdivision Didyk et al. [5] use a disparity measure trivially calculated from the depth image, as their stereoscopic projection case only involves camera translation on the local x-axis. Such a measure is however not trivial to calculate for free camera movement, as the whole pixel region in question would need to be fully projected first to evaluate the pixel displacement.

Instead, we create a simple impostor by tessellating regions of high depth variance. We precalculate an edge image via a thresholded laplace operator on the depth image and use its' mipmap level corresponding to the coarse mesh ($\lceil \log_2 t_{max} \rceil$) to evaluate a present edge in a patch's image region (see Fig. 3).

Vertex Image space positions after fixed function tessellation are assigned

$$v_{ij} = v_{00} + \left(\frac{\lceil 0.5 i \rceil}{m}, \frac{\lceil 0.5 j \rceil}{n} \right), \quad (2)$$

with v_{00} being the lower left edge vertex of the patch, $i, j \in [0; 2 t_{max} - 1]$ the vertices' relative ids in the patch and m, n the width and height of the source image. This quasi degenerates the intermediate triangles (Fig. 3(e)).

Algorithm 1. Vertex positioning

```

function VERTEX_POSITION( $v_{00}, i, j, m, n, T$ )
   $v_{ij} \leftarrow v_{00} + \left( \frac{\lfloor 0.5 i \rfloor}{m}, \frac{\lfloor 0.5 j \rfloor}{n} \right)$ 
  if  $edge(v_{ij}) > 0$  then
     $v'_{ij} \leftarrow v_{ij} + \left( \frac{0.5 - i \bmod 2}{m}, \frac{0.5 - j \bmod 2}{n} \right)$ 
     $N \leftarrow \{v_{ij} + w \mid w = (x, y), x \in \{-\frac{1}{2m}, \frac{1}{2m}\}, y \in \{-\frac{1}{2n}, \frac{1}{2n}\}\}$ 
     $N' \leftarrow \{w \in N \mid |depth(w) - depth(v'_{ij})| < T\}$ 
     $d_{ij} \leftarrow \frac{\sum_{w \in N'} depth(w)}{|N'|}$ 
  else
     $d_{ij} \leftarrow depth(v_{ij})$ 
  end if
  return  $(v_{ij}, d_{ij})$ 
end function

```

Vertices not residing on depth edges are assigned linearly interpolated depth values fetched directly at their image position. For edge vertices this would create invalid depth values [10] and thus a simple reconstruction similar to [21] is employed by averaging all neighboring pixel depths that are closer than a threshold T to the depth of the reference pixel, given by

$$v'_{ij} = v_{ij} + \left(\frac{0.5 - i \bmod 2}{m}, \frac{0.5 - j \bmod 2}{n} \right). \quad (3)$$

The full vertex positioning is summarized in Algorithm 1. Resulting vertices are transformed using Eq. (1) to finally obtain the desired reprojection.

4.3 Mesh Reduction

To lower the cost of the added intermediary triangles in Sect. 4.1 the global tessellation amount can be reduced. To exploit present edge mipmaps the amount of subdivisions and the assumed source image size during vertex positioning is simply decreased by a power of two, i.e. $t'_{max} = \lfloor \frac{t_{max}}{2^i} \rfloor$ and $(m', n') = (\lfloor \frac{m}{2^i} \rfloor, \lfloor \frac{n}{2^i} \rfloor)$ and the initial coarse mesh constructed accordingly.

The results of this procedure are exemplified by the reprojections in Fig. 4. A color bleeding effect can be seen on some object boundaries, which is induced

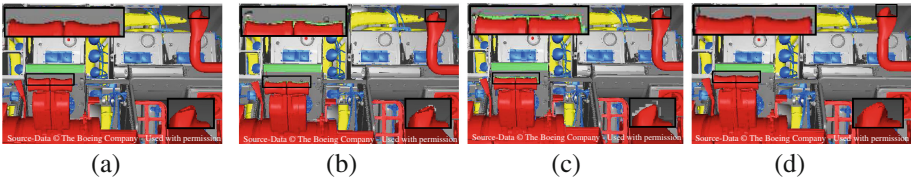


Fig. 4. Downsampling levels (a) 0, (b) 1, (c) 2, and (d) 1 with artifact suppression (artifacts in enlarged frames) (Color figure online)

through a mismatch between the real source pixel depth and the mipmapped depth of the impostor geometry. This effect can be suppressed by slightly shifting the texture coordinates away from an edge (see Fig. 4(d)). General image quality is preserved by still accessing the full resolution source color image.

5 Artifacts

Artifacts typically occurring in mesh based DIBR applications are so called skins [10] or rubber-sheets [4, 13], which are triangles spanning depth edge regions, creating false surfaces. The projection method of Sect. 4 already removes them implicitly by discarding the intermediate triangles, creating holes. Additional holes can occur on edges between patches of different tessellation strength (the T-Junction problem [5]). Other artifacts are created by dynamic lighting.

5.1 Hole Filling Through Multi-camera Look-Ahead-Rendering

Different methods for hole filling have already been proposed in literature: Pseudo stereo synthesizers like [7, 11, 21, 22] rely on some form of Inpainting approach, which are either clearly noticeable when applied to larger hole areas or too costly for interactive display. Other works use a combination of multiple sources from either different views (multi-camera composition) [2, 3, 6] or layered images from the same view (depth peeling) [14, 18].

To avoid increasing time differences between consecutive source frames, we did not use depth peeling. Rather, a cost-efficient multi-camera composition is achieved by buffering and projecting a set of past source frames.

For the buffering to be effective, predicted source cameras are used rather than the current user camera. Mark et al. already experimented with a similar setup using simulated prediction through random jittering of a known camera path [13]. The prediction is based on a simple linear extrapolation shown in Fig. 5, evidently not accurately reflecting a possible non-linear camera path, but sufficient if the time distance between cameras is small [13].

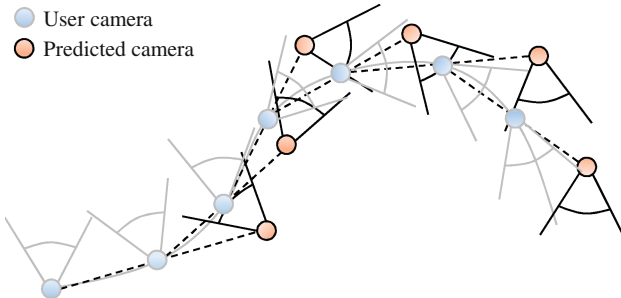


Fig. 5. Linear camera prediction

5.2 Additional Redundancy

Projecting all buffered source images for one synthetic frame is not cost-efficient, due to increasingly distant views carrying decreasingly useful information. For quick selection of a meaningful subset the following distance measure is applied:

$$\text{dist}(c_s, c_t, r_s, r_t) = \frac{1 + \|c_s - c_t\|^2}{0.5(r_s \circ r_t + 1)} - 1, \quad (4)$$

the conservative assumption being decreasing image quality with increasing square euclidean distance of source and target camera positions c_i (pixel areas change proportionally) and with increasing divergence of the cameras' normalized view directions r_i (\circ being the dot product).

In addition, the source frames are rendered with a larger field of view (FOV) compared to the target camera's, providing interception of holes caused by camera rotation. As a consequence, the source frames need to have a higher resolution to avoid blur in the projections.

5.3 Avoiding Lighting Artifacts

Time-varying lighting like specularities or from animated lights can also cause artifacts: Already shaded scene portions from different source images create noticeable wrong shading in the synthesized image (see Sect. 6.2 for an example). Deferred shading via G-Buffer rendering [19] was implemented as a remedy.

Algorithm 2. Deferred lighting of a target pixel

```

function INDIRECT _ GBUFFER _ LIGHTING( $p_{xy}, sid, light$ )
  if  $stencil(p_{xy}) \neq sid$  then
    return ▷ pixel unwritten or belonging to different source
  else
     $p'_{xy} \leftarrow target(p_{xy})$  ▷ coordinates from DIBR pass
     $f_{gbuffer} \leftarrow source[sid](p'_{xy})$  ▷ source G-Buffer pixel
    return  $apply\_lighting(f_{gbuffer}, light)$ 
  end if
end function

```

To keep the footprint of the DIBR renderer small we avoid heavy GPU memory write access by using an indirect approach shown in Algorithm 2: For each target pixel, only texture coordinates referencing the source G-Buffer and a stencil buffer containing the source's id are written. Deferred shading then applies each light for each source excluding unrelated target pixels by early stencil testing against the source id and fetching the required source G-Buffer pixels via the texture coordinates.

6 Analysis

The method described previously has been implemented on a Windows 7 \times 64 platform in C++ and OpenGL, with the executing hardware including an Intel Core i7-3820 CPU and a NVIDIA GeForce GTX 650 GPU. Two CAD scenes were used for evaluation: a 777 plane model with approx. 300 million triangles of The Boeing Company and an $8 \times 8 \times 8$ replicated Powerplant model of the University of North Carolina yielding approx. 6.5 billion triangles. 3DInteractive’s VGR renderer [1] was used as the source renderer.

6.1 Rendering Time

Both renderers were executed simultaneously during scene flythrough. Two sources were used for each synthesis with a source FOV of 110° and a target FOV of 90° (the source images were approx $1.43 \times$ the target image size), with $1 \times$ mesh downsampling enabled. Methods described in Sect. 2.2 were also implemented on the GPU for comparison.

Table 1 shows Morvan’s method to be unsuitable for this paper’s target application. It cannot leverage part of the GPU’s units, as the GPGPU execution is confined to the shaders and every pixel is projected individually. The highly tessellated triangle net of Mark et al. also suffers from the latter problem, making it unsuitable for high resolutions. In contrast, the point rendering fares better due to a very simple pipeline (no skin detection and removal, no interpolation), but its’ complexity is also directly influenced by the source pixel count.

Our method leverages more GPU functionality through tessellation and has a lower overhead in image regions of low depth variance. As a consequence the frame computation times are lower than all other methods on the test hardware.

Table 1. Flythrough average frame times (in ms), with the results of this paper’s method marked in bold

Scene	Renderer	Target resolution			
		1280×720	1366×768	1600×900	1920×1080
777	Morvan et al.	34.2	38.5	53.1	76.6
	Point	8.7	10.2	13.4	16.3
	Mark et al.	11.8	14.0	19.6	30.2
	Proposed	4.5	5.0	6.7	9.3
	Source	27.7	33.8	37.8	40.0
Powerplant	Morvan et al.	36.5	40.7	55.3	79.0
	Point	9.3	11.1	14.1	16.3
	Mark et al.	23.1	27.6	43.2	84.2
	Proposed	8.2	9.2	11.8	15.5
	Source	36.4	41.1	48.6	56.1

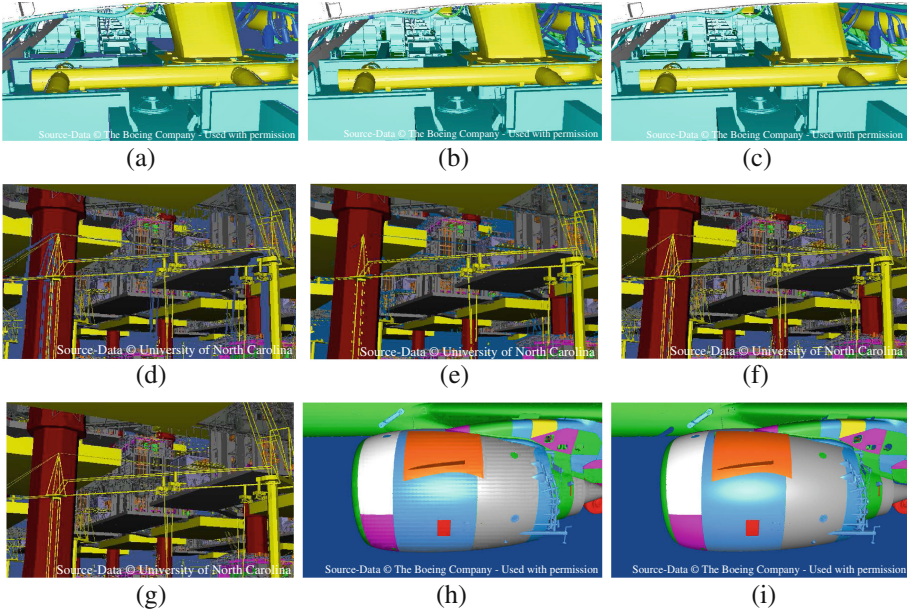


Fig. 6. Image quality during a camera flight through the (a, b, c) 777 Boeing and the (d, e, f, g) Powerplant scene. (a, d) Without hole filling, noticeable information gaps occur, while (b, f) being greatly reduced with one additional projection. Contrary to (e) [13], edge regions are not diminished. Plausible reconstructions are reached in comparison with the (c, g) ground truth. (h) Artifacts from view-dependent shading of multiple different sources are (i) removed when using the proposed deferred shading approach

However, the per-primitive complexity is higher compared to the point rendering resulting in roughly the same frame times when the whole image shows high depth variance, which is largely the case for the Powerplant scene.

6.2 Image Quality

As can be seen from Fig. 6(a) and (d), a single DIBR projection does not suffice for an acceptable synthesis, even when using the predictor. In contrast, the proposed hole filling already shows great improvement when using two projections. Also, the method does not exhibit the diminished borders of Mark et al.s’ original full triangle mesh technique and thus retains small pixel regions.

Figures 6(h) and (i) show an example of the approach of Sect. 5.3. Artifacts can be seen when combining view-dependent shading from multiple different sources, while the deferred shaded approach shows plausible results.

6.3 Drawbacks and Remaining Problems

A significant downside to the new method is the source buffers' GPU memory consumption: Ten buffered G-Buffers (each with a layout of three 8bit RGBA, one 16bit RGB and one 24bit depth buffer) take around 420 MB at 1920×1080 . We aim to use compression and material tables in the future.

For acceptable results a minimum source frame rate is required. 15 Hz or above seemed sufficient to produce little to no remaining holes in the synthetic images. The results were still acceptable around 5 to 10 Hz, while 5 Hz and below showed too sparse updates and many information gaps in the perceived scene.

Only static scenes are currently supported. Animated parts need to be rendered separately for correct display. An additional step in the DIBR projection equation handling the animation transformations could be a solution.

7 Conclusion and Future Work

During the course of this paper it has been shown, that DIBR can be efficiently implemented via the modern standard GPU pipeline using a novel implementation by adaptive hardware accelerated tessellation. The system design decoupling the user input and display from the source renderer proved to be a viable solution for the high frame rate and responsiveness problems listed in Sect. 1.

Subsequent work will focus on improving the method: The simple predictor will be replaced by a more sophisticated approach realizing a non linear prediction. Providing more redundancy via depth peeling will be tested, as there exist efficient one pass techniques [12]. Applying the static mesh downsampling of Sect. 4.3 dynamically promises further performance.

Other endeavors will include porting the described system to mobile devices. The main challenge will be the significantly lower graphic capabilities or the lack of support of required features used in this paper.

References

1. Brüderlin, B., Heyer, M., Pfützner, S.: Interviews3D: a platform for interactive handling of massive data sets. *IEEE Comput. Graph. Appl.* **27**(6), 48–59 (2007)
2. Chang, C.-F., Ger, S.-H.: Enhancing 3D graphics on mobile devices by image-based rendering. In: Chen, Y.-C., Chang, L.-W., Hsu, C.-T. (eds.) *PCM 2002*. LNCS, vol. 2532, pp. 1105–1111. Springer, Heidelberg (2002). doi:[10.1007/3-540-36228-2_137](https://doi.org/10.1007/3-540-36228-2_137)
3. Darsa, L., Costa Silva, B., Varshney, A.: Navigating static environments using image-space simplification and morphing. In: *Proceedings of the 1997 Symposium on Interactive 3D Graphics*. I3D 1997, pp. 25–ff. ACM, New York (1997)
4. Decoret, X., Schaufler, G., Sillion, F., Dorsey, J.: Multi-layered impostors for accelerated rendering. In: *EUROGRAPHICS 1999*, vol. 18 (1999)
5. Didyk, P., Ritschel, T., Eisemann, E., Myszkowski, K., Seidel, H.P.: Adaptive image-space stereo view synthesis. In: *Vision, Modeling and Visualization Workshop*, pp. 299–306, Siegen, Germany (2010)

6. Do, L., Zinger, S., Morvan, Y., de With, P.: Quality improving techniques in DIBR for free-viewpoint video. In: 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2009, pp. 1–4, May 2009
7. Fehn, C.: A 3D-TV approach using depth-image-based rendering (DIBR). In: Proceedings of Visualization, Imaging and Image Processing, pp. 482–487 (2003)
8. Fu, C., Wong, T., Heng, P.: Triangle-based view interpolation without depth-buffering. *J. Graph. Tools* **3**, 13–31 (1998)
9. Funkhouser, T.A., Séquin, C.H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH 1993, pp. 247–254. ACM, New York (1993)
10. Ghiletiuc, J., Färber, M., Brüderlin, B.: Real-time remote rendering of large 3D models on smartphones using multi-layered impostors. In: Proceedings of the 6th International Conference on Computer Vision/Computer Graphics Collaboration Techniques and Applications. MIRAGE 2013, pp. 14: 1–14: 8. ACM, New York (2013)
11. Horng, Y.R., Tseng, Y.C., Chang, T.S.: Stereoscopic images generation with directional Gaussian filter. In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2650–2653 (2010)
12. Liu, F., Huang, M.C., Liu, X.H., Wu, E.H.: Single pass depth peeling via CUDA rasterizer. In: SIGGRAPH 2009: Talks. SIGGRAPH 2009, p. 79: 1. ACM, New York (2009)
13. Mark, W.R., McMillan, L., Bishop, G.: Post-rendering 3D warping. In: Proceedings of the 1997 Symposium on Interactive 3D Graphics. I3D 1997, pp. 7–ff. ACM, New York (1997)
14. Max, N., Ohsaki, K.: Rendering trees from precomputed z-buffer views. In: Eurographics Rendering Workshop, pp. 45–54 (1995)
15. Morvan, Y., Farin, D., de With, P.: System architecture for free-viewpoint video and 3D-TV. *IEEE Trans. Consum. Electron.* **54**(2), 925–932 (2008)
16. Ndjiki-Nya, P., Koppel, M., Doshkov, D., Lakshman, H., Merkle, P., Müller, K., Wiegand, T.: Depth image-based rendering with advanced texture synthesis for 3-D video. *IEEE Trans. Multimedia* **13**(3), 453–465 (2011)
17. Oh, K.J., Yea, S., Ho, Y.S.: Hole filling method using depth based in-painting for view synthesis in free viewpoint television and 3-D video. In: Picture Coding Symposium, 2009. PCS 2009, pp. 1–4, May 2009
18. Popescu, V., Lastra, A., Aliaga, D., Neto, M.D.O.: Efficient warping for architectural walkthroughs using layered depth images. In: *IEEE Visualization*, pp. 211–215 (1998)
19. Saito, T., Takahashi, T.: Comprehensible rendering of 3-D shapes. In: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH 1990, pp. 197–206. ACM, New York (1990)
20. Westover, L.: Footprint evaluation for volume rendering. In: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH 1990, pp. 367–376. ACM, New York (1990)
21. Xi, M., Wang, L.H., Yang, Q.Q., Li, D.X., Zhang, M.: Depth-image-based rendering with spatial and temporal texture synthesis for 3DTV. *EURASIP J. Image Video Process.* **2013**(1), 1–18 (2013)
22. Zhang, L., Tam, W.J.: Stereoscopic image generation based on depth images for 3D TV. *IEEE Trans. Broadcast.* **51**(2), 191–199 (2005)
23. Zinger, S., Do, L., de With, P.H.N.: Free-viewpoint depth image based rendering. *J. Vis. Commun. Image Represent.* **21**(5–6), 533–541 (2010)