SIGRAD 2017, pp. 1–7
M. Hayashi (Editors)

High-Quality Real-Time Depth-Image-Based-Rendering

J. Ogniewski[1]

[1]Linköping University, Linköping, Sweden, jenso@isy.liu.se

Abstract

With depth sensors becoming more and more common, and applications with varying viewpoints (e.g. virtual reality) becoming more and more popular, there is a growing demand for real-time depth-image-based-rendering algorithms that reach a high quality.

Starting from a quality-wise top performing depth-image-based-renderer, we develop a real-time version. Despite reaching a high quality as well, the new OpenGL-based renderer decreases runtime by (at least) 2 magnitudes. This was made possible by discovering similarities between forward-warping and mesh-based rendering, which enable us to remove the common parallelization bottleneck of competing memory access, and facilitated by the implementation of accurate yet fast algorithms for the different parts of the rendering pipeline.

We evaluated the proposed renderer using a publicly available dataset with ground-truth depth and camera data, that contains both rapid camera movements and rotations as well as complex scenes and is therefore challenging to project accurately.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms

1. Introduction

Depth-sensors become more and more common, and are integrated in more and more devices, e.g. Microsoft Kinect, Google Tango, Intel RealSense Smartphone, and HTC ONE M8. This enables new applications such as virtual reality, 360 degree video, frame interpolation in rendering [MMB97], rendering of *multi-view plus depth* (MVD) content for free viewpoint and 3D display [TLLG09] [Feh04], all using *depth-image-based-rendering (DIBR)*.

DIBR has been explored before (e.g. [PG10] and [YHL16]), albeit using different algorithms and different test-sequences than in this work. Here, we start with and benchmark against a renderer that was highly optimized for quality, and use the Sintel [BWSB12] datasets, which provide ground-truth values for depth and camera parameters (thus ensuring that all errors are introduced by the projection itself), as well as sequences with complex scenes and camera movement, which are challenging to project accurately.

In an earlier paper [OF17], we examined different forward-warping methods to develop a renderer maximizing quality. This was done by creating a flexible frame-work incorporating state-of-the-art methods as well as own novel ideas, and running an exhaustive semi-supervised automatic parameter search to estimate the optimal parameter and methods. Our final algorithm is using a forward warp technique called *splatting* [Sze11], a popular choice since this leads to a high preservation of details. However, its great disadvantage is its high computational complexity, which is even made worse by the fact that it is nontrivial to parallelize.

In this paper, we develop a real-time version of our renderer, while minimizing quality loss. This was enabled by discovering and exploiting similarities between forward warping and mesh-based projection, as well as implementing efficient, accurate algorithms for the different rendering steps.

The rest of the paper is organized as follows: section 2 introduces the original renderer as well as an optimized CPU version. Section 3 discusses the similarities between forward-warping and mesh-based projection as well as the different stages of the OpenGL rendering pipeline. Section 4 presents an evaluation and section 5 concludes the paper.

2. Quality optimized forward warping

In the following, we will only describe the methods which proved to be most beneficial. For a complete comparison of the different methods the reader is referred to our original paper [OF17].

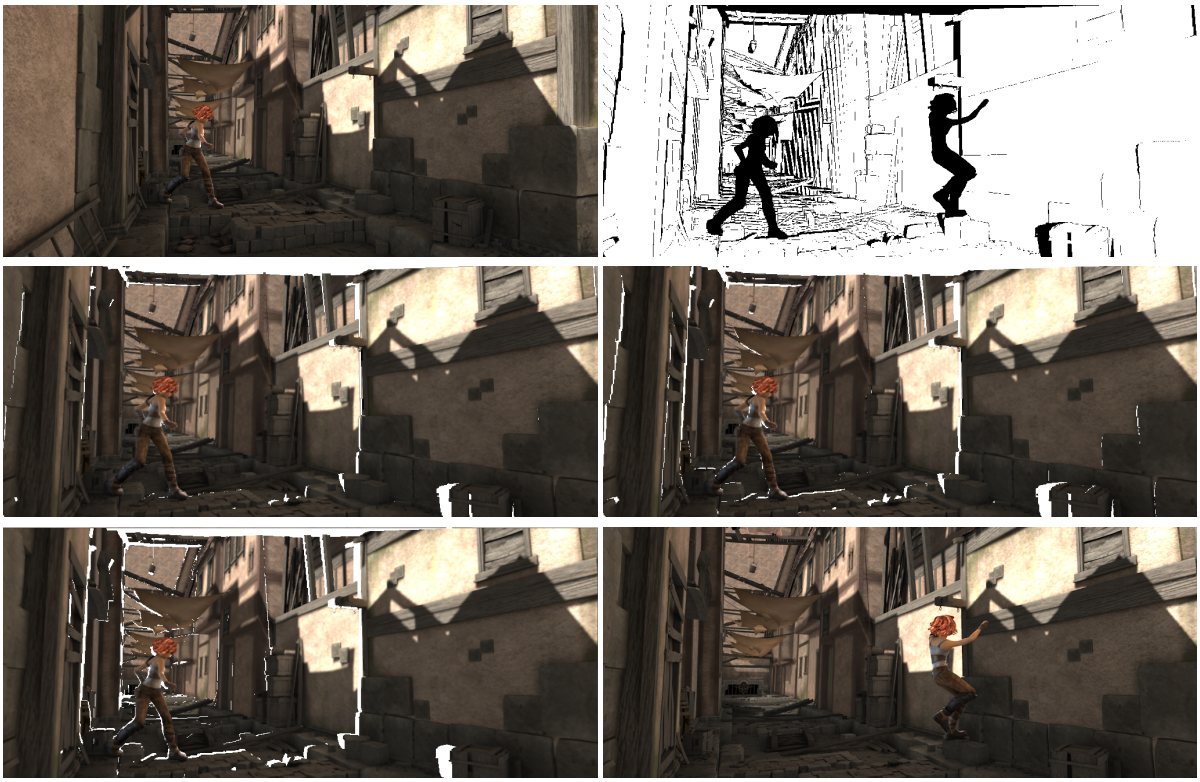

Figure 1: *Example projection (taken from the alley2 sequence):*
Top row: input image (left), mask image as used for quality measurement (right)
2nd row: projection using the CPU versions (from frame 1 to frame 25 of the sequence), original (left) and optimized (right)
3rd row: projection using the OpenGL version (from frame 1 to frame 25 of the sequence) (left), and ground-truth frame 25 of the sequence (right)

In forward warping, the points of the input frame are splatted across a neighborhood in the target frame. We call the resulting points candidate points. In many cases, several candidate points compete for the same pixel in the target frame. These are merged using agglomerative clustering [MLS14, SV14]: two candidate points will be merged if their distance in both depth and color is small enough, using initial weights based on the distance of the projected candidate point to the center of the pixel that is currently colored. The weights are summed up, to give candidates with a higher number of original points a higher weight in consecutive mergings. If another candidate point is added to the same cluster, the summed-up weight means that the same result is received as if a weighted average of all points of the cluster would have been calculated, using the initial weights. In every step, only the two points/clusters are merged that are closest to each other, and the process is stopped when this minimal distance is higher than a predetermined threshold. Then, the point/cluster is selected which is nearest to the camera; the accumulated weights are considered in the decision as well.

To counter artifacts we discovered during our work, we introduced two extensions:

1. *Edge suppression*: which removes anti-aliased pixels at the edge of objects, which otherwise lead to visible lines in the output frame.
2. *Scale adaptive kernels*: we adapt the kernel size of the splatting algorithm taking local scale change into account, using a similar method as described in section 3.1.

Also, we use an internal upscale during the splatting process, of 3 in both width and length, and a Gaussian filter with overlapping neighborhoods for the downscale, see also 3.3.
Since we did an exhaustive evaluation of different methods and parameters we are confident that the final set-up lead to overall best quality results, and thus we use the same in all following implementations (if not stated otherwise) and concentrate on reducing the runtime.

2.1. CPU-optimized version

The disadvantage of the derived projection algorithm is its high computational complexity. To reduce it, we heavily

optimized the code. Among other things, this included the change of all parameters to nearby values that were a power of two, as well as replacing exponential functions by functions of the form $(1-(\frac{d}{k_s})^2)^n$, where k_s is the kernel size, d the distance (e.g. to the center of the kernel), and n an integer chosen to match the original function, and which lead to an exponent that can easily be replaced by a few multiplications. This was done e.g. for the weight calculation of the agglomerative clustering algorithm, and is demonstrated in figure 2. Also, saving the candidate points to an intermediate data structure and merging them according to agglomerative clustering is done in the same step. This simplified the computation, but leads to slightly different results, since in some cases different points are merged, or even not merged at all.
While the optimized version reached a high speed-up, it is not high enough for real-time applications. Thus, we developed a OpenGL-based version.

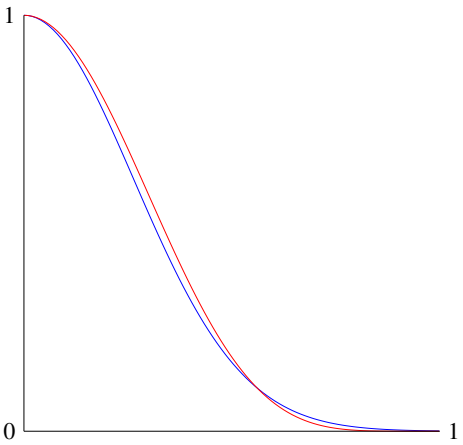

Figure 2: *Replacing one function with a similar, less computational complex one: $e^{-0.8*(\frac{d}{k_s})^2}$ (blue), $(1-(\frac{d}{k_s})^2)^6$ (red). The x-axis shows the normalized distance $\frac{d}{k_s}$. These are the functions used in the agglomerative clustering steps: the one presented in blue is used in the original renderer, the one presented in red otherwise.*

3. From forward-warping to mesh-based projection

Forward-warping is extremely difficult to implement efficiently on a GPU, since it requires parallel writing to and modifying the same memory address. However, we noticed how forward warping using agglomerative clustering can be emulated by mesh-based projection:
The main reason for the high quality of forward warping lies is that several candidate points are taken into account when coloring a pixel. As discovered earlier, agglomerative clustering leads to the highest quality in forward warping, and the idea behind agglomerative clustering is to cluster candidate points together which are likely to lie in the same neighborhood of the same object. Thus, in the ideal case different clusters are derived, where each one belongs to one specific neighborhood on one specific object, and the most likely cluster is selected for the pixel in question.
Instead, a mesh-based renderer can read this neighborhood from the input texture, and calculate the final color using a Gaussian filter on this neighborhood. This filter emulates the agglomerative clustering merging process, by calculating the weights in a similar way, however only taking the distances to the center of the kernel into account. For a calculation of the color distance we would first need to determine which color the pixel is most likely to have, which is difficult to achieve accurately in a limited computation time, and therefore omitted here. Care has however to be taken that all texture values belong to the same object.
Also, both the scale-adaptive kernel and edge suppression are included naturally, the latter because points on the border between objects will not be connected by the meshing algorithm, and thus the anti-aliased color will spread in a much more limited area. However, this will also lead to more holes (as can be seen in figure 1), even if the downsampling does cancel this out to a certain degree, since only one pixel needs to be set in the neighborhood used for coloring a pixel. The rest of the missing data can be easily filled in using a simple hole-filling algorithm, e.g. *hierarchical hole-filling* [SR10].
In the following, we take a closer look at the different pipeline stages of the OpenGL version.

3.1. Meshing

Creating high quality objects and meshes from 3D depth maps has extracted a lot of attention from the research community in recent years, an example is of course [NIH*11]. However, most approaches use several depth maps for the mesh (an exception is e.g. [KPL05]), and concentrate on single objects rather than whole scenes. Here, we are interested in constructing one or several mesh(es) for the whole scene including several objects (e.g. the girl and the house in figure 1) whose number and positions are unknown from a single depth map, to allow for real-time rendering with a low latency. Also, in our application the scene may contain moving objects (see even the girl in figure 1), which is something that still has to be explored using depth map-fusion techniques. On the other hand, we only calculate the connections of the mesh rather than also refining the vertex-positions (as is often done in meshing algorithms), and assume that this is handled by an earlier depth map refining step, such as e.g. [WLC15]. Also, for reasons of computational complexity, we assume that a point may only be connected to points it is directly neighboring in the depth map. Thus, whenever the term neighborhood is used in the following, it is refering to 3x3 neighborhoods in the depth map.
The trade-off necessary in most meshing methods is trying to connect as many points belonging to the same object as possible, while creating as few connections between different objects as possible, which is demonstrated in figure 3. We

Figure 3: *Wrongly connected mesh (left): connections over object boundaries (causes stripes) and missing connections (causes black spots). Our meshing algorithm was able to remove most of these artifacts (right).(detail from the temple2 sequence)*

found that the following algorithm worked very well, while being comparably inexpensive to compute:
We start by creating the input vertexes for the mesh: for every value in the input depth map one point is projected to 3D-space, using the position in the depth map, the depth-value as well as the camera parameters of the input frame.
In the next step, we determine which points should be connected to which of its neighbors. We use a spheroid-approximation for that, to allow for different geometric changes in perpendicular directions. To estimate the spheroid, we calculate the difference vectors from the central point to each of its 8 nearest neighbors in the depth map, using the projected 3D positions. We select the difference vector with the smallest length (i.e. the one originating from the nearest neighbor of the central point), and calculate which of the remaining difference vectors are the most perpendicular to this difference vector, and select the two most perpendicular, taking care that they point in (approximately) opposite directions. We again select the one of these two with the smallest length. The length of this difference vector, and the length of the difference vector we selected first, are then used as the radii of the spheroid. Rather than estimating a spheroid directly, we calculate the absolute value of the dot product of each of the remaining difference vectors to the ones selected as representing the radii, and use the results as blending weights for the respective radii to derive a local radii, one for each of the remaining difference vectors. This local radius is then multiplied by a predetermined factor (2.425 was selected based on experimental results). If the resulting local radii is greater than the length of the corresponding difference vector, the neighbor used for the calculation of this difference vector is considered to be connected. Two exceptions were made in this method: 1. if one of the two radii is smaller than a predetermined factor (0.1 was selected based on experimental results) it will be set to this factor, and 2. if the radii of a neighbor is greater than the maximal depth-range found in the depth map, divided by a predetermined factor (81.25 proved to lead to good results), it will not be connected. These two selections both maximize the number of correct connections and minimize the number of false connections, see also figure 3. We save the distances of each neighbor, divided by the local radii, where the sign determines whether or not the neighbor should be connected.
From the connections, edges are calculated in the next step. An edge is created if both points have positive connections. The absolute value of both connections is added up and saved; an edge is indicated by saving it as a positive value, otherwise it is saved as a negative value.
Finally, the edges are used to create the triangles used for the mesh-based rendering. For this, always 4 directly neighboring points are considered. If they are connected on at least 3 of the 4 horizontal and vertical edges, and at least one of the diagonal edges, two triangles are created connecting the two points. Out of the possible two connections, we select the one using the diagonal with the lowest (absolute) edge value. If the 4 points are only connected in one horizontal and one vertical edge, one triangle will be created if the corresponding diagonal edge is positive as well.

3.2. Agglomerative Clustering

We do the agglomerative clustering emulation in a two step approach: during the actual point projection we save the texture coordinates rather than a color. In the second step, we use the distance between the texture coordinates of the center pixel to the texture coordinates of its 8 neighbors (multiplied by the width respectively the height of the texture) to determine the scale in x- and y-direction. The respective smallest distances are used, limited to a maximal value of 2.5. The kernel size for the Gaussian filter is then determined in the following way: if any of the two coordinates is equal or smaller than 0.5, only the two nearest pixels will be used for this direction. If it is greater than 0.5 but smaller or equal to 1.5, the kernel size will be set to 3 in this direction, if it is greater it will be set to 5. Higher kernel sizes did not lead to significant increase in quality, but lead to a high increase in the runtime, in all likelihood due to the much higher demand of memory and the higher amount of memory accesses. Then, we calculate the final color by running a kernel over this neighborhood. The weights for each color is calculated using the function presented in red in figure 2, and the distance in x- and y-direction are normalized by the scale in this direction. If the distance is larger than the scale in

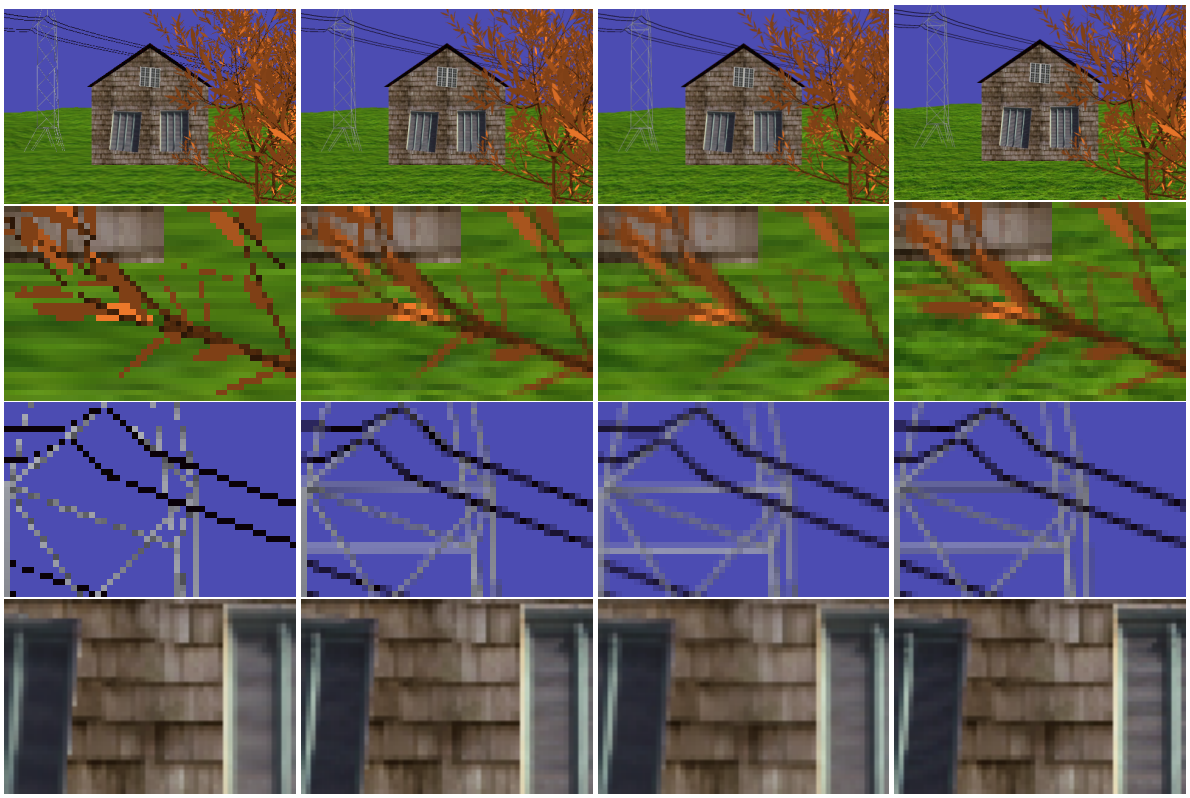

Figure 4: *Example of anti-aliasing techniques. From left to right: without anti-aliasing, 2x2 upscale using 2x2 neighborhood averages for the downscale, 2x2 upscale using 3x3 Gaussian kernels for the downscale, 4x4 upscale using 4x4 neighborhood averages for the downscale; the top row is showing the complete images, the other rows zoomed-in details*

one direction, the color-value is discarded. Also, we use the edges calculated earlier to discard color-values belonging to points not connected to the pixel we are currently coloring.

3.3. Downsampling

As in the CPU versions, an internal image upscaled by 3 in both width and height was used, as well as a Gaussian 5x5 kernel for downsampling. This proved to be best during our parameter estimation of the projection framework. However, instead of calculating the weights using an exponential function as used in the original CPU approach, we use predetermined power-of-2 weights for decreased computational complexity (as used in the CPU optimized version as well). We demonstrate the differences of different downsampling methods in figure 4, with the example application of full screen anti-aliasing *FSAA*. Anti-aliasing is a related application, where the internal upscale is applied for similar reasons as in our DIBR approach. We chose this example to emphasize differences, an thus make them more visible. In figure 4, a Gaussian kernel with overlapping neighborhoods is compared to using averages of a non-overlapping neighborhood, which is the most popular choice for FSAA due to its simple implementation. The overlapping neighborhoods introduces a slight blur, but overall leads to results with a similar visual quality as averaging methods with higher internal resolution, and which also uses more memory accesses (16 vs 9 comparing the 4x4 averaging downsample with the 3x3 Gaussian kernel) when calculating the final color in the output image. Here, the downscale has the additional advantage of filling pixels that will not be written to otherwise, since only one pixel in the neighborhood of the upscaled image needs to be set to set a pixel in the output image.

4. Evaluation

For evaluation, we compare projected images to ground-truth images to accurately measure the projection performance of the different DIBR methods. We use the Sintel test-sequences [BWSB12] for that. These provide both ground-truth depth and camera poses. Access to ground-truth data is crucial to ensure that all noise and artifacts are introduced by the projection algorithms rather than by inaccurate or noisy input data. The sequences we selected were *sleeping2*, *alley2*, *temple2*, *bamboo1* as well as *mountain1* (see figure 5). We choose these sequences since they contain moderate to

sleeping2 alley2 temple2 bamboo1 mountain1

Figure 5: *Selected Sintel sequences.*

high camera motion, but only few moving objects, which are currently not handled by our projection. Note that some of the sequences contain highly complex scenes and are therefore difficult to project accurately. The size of the textures and depth maps used were 1024x432.

All sequences are provided with two different texture sets: clean as well as final, where the final sequences include more accurate lighting and effects such as blur, which are omitted in the clean sequences. We selected the clean sequences since the difference between different projection algorithms is more pronounced there due to a higher level of detail, which is lessened by the effects added to the final sequences.

For each projection algorithm and sequence, we projected from the first and the last frame of each sequence to all other frames of the sequence, then measured the differences between the projected and the ground-truth frame in both PSNR and multi-scale SSIM [WSB03]. The results are presented in figure 7. To reach a high accuracy, we removed pixels that are occluded in the input frames as well as those containing moving objects from the measurements. This was done by using mask images, which were created beforehand. All points were projected from the input frames to each of the respective target frames. The calculaded position was rounded up and down in both the x- and y-coordinate, and the resulting 2×2 regions were set in the mask. Before that, the depth of the projected point was compared to the depth found in the depth map of the target frame for each of the 4 pixels, and only the pixels were set where the difference between the two depth-values is comparably small, to remove moving objects from the measurements. An example mask is given in figure 1, where also example images are presented from the different projection methods.

The CPU used was an Intel Xeon E5-1607 running at 3 Ghz with 8 Gbyte of memory, and the GPU used was a GeForce GTX 770 with 2 Gbyte of memory. Timing results are given in table 1. The reason why the original CPU-based renderer performs so poorly in some sequence is due to the adaptive kernel sizes. Changes in local scale (due to camera-movement or rotation) lead to large splatting kernels and thus to an inflation of candidate points that need to be considered. In the OpenGL version, the meshing step (as described in 3.1) takes up most of the time, up to 70%, followed by the rendering and the agglomerative clustering (as described in 3.2) with ca. 20% of the time. If the same depth map is to be reused, the meshing step might be omitted in consecutive frames, thus reducing the runtime drastically.

The differences in the results between the CPU versions lie mainly in the different merging process (see also 2.1).

Sequence	Projected from	CPU, original	CPU, optimized	OpenGL
Sleeping 2	1	3559	999	10.9
	49	2516	1081	10.5
Alley 2	1	2248	918	10.0
	49	1713	789	9.6
Temple 2	1	24724	1009	9.7
	49	5492	898	10.4
Bamboo 1	1	4079	949	11.2
	49	5852	1051	11.3
Mountain 1	1	34963	951	10.0
	49	4166	1113	10.5
average		8931	976	10.4

Table 1: *Timing results in ms. Average results are given for the projection to each frame of the sequence from frame 1 and frame 49 respectively.*

As suspected, the OpenGL-version leads to a lesser quality in most sequences, in some cases however it performed better. The reason for this lies in the difference how the projection works. Mesh-based projection uses triangles, which can take a nearly arbitrary form in the target frame, e.g. a line segment not aligned with any of the image axes. The forward warping however always projects to a rectangle. If this rectangle contains the whole aforementioned line-segment, the connected points will project to a multitude of pixels in the target frame they are not supposed to project to, which will be punished in our paramater estimation algorithm. Therefore the two points will be connected in the mesh-based projection, but not in the forward warping. This leads to artifacts were background-objects can shine through foreground objects whose points are not connected, as demonstrated in figure 6. On the other hand, in some cases the agglomerative clustering of the CPU version might use a cluster which is not the one nearest to the camera, but has a higher number of contributing candidate points. This is not realized in our mesh-based projection approach, and would require a modification of the OpenGL pipeline, which in all likelihood would increase the runtime. However, this is probably also one of the main reasons why the CPU versions reach a higher measured quality in most sequences.

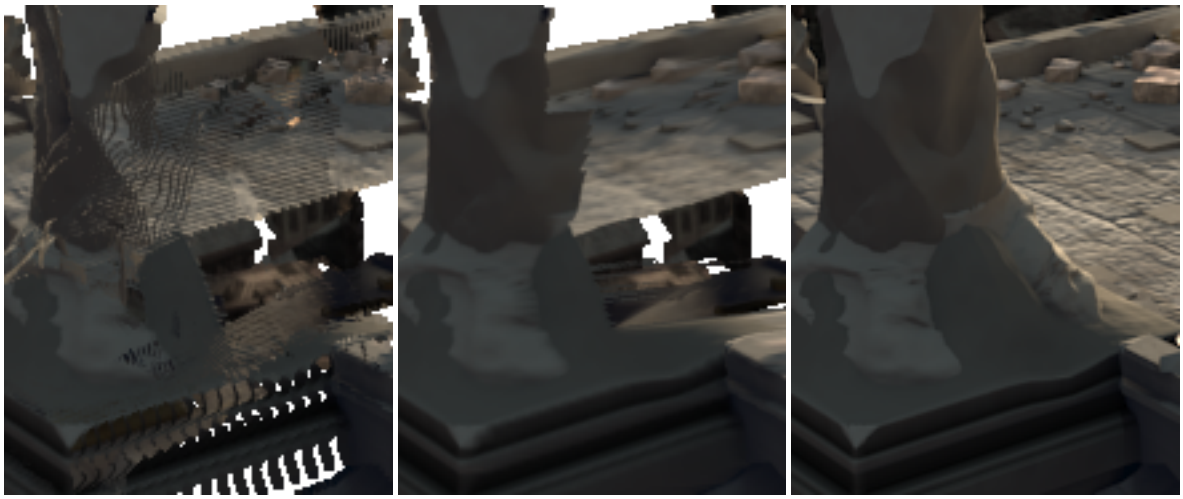

Figure 6: *Artifacts due to missing connectivity (detail from the temple2 sequence):*
CPU version (left) with background objects shining through foreground; OpenGL version (middle): no artifacts, the partly missing foreground objects are due to occlusion in the input frame; ground-truth image (right)
This is a particular difficult projection, where the camera was rotated by nearly 90 degrees between input and output frame

5. Conclusions

We developed a real-time method for DIBR, based on a computationally complex, but quality-maximized renderer. This was done by exploiting similarities between forward warping and mesh-based projection. Despite sharing these similarities, in practice they show different behavior, meaning that it might be possible to optimize the real-time renderer further, in quality as well as in runtime. Furthermore, in real-world applications depth map and camera parameters will contain noise and inaccuracies, whose effects on the projection still have to be determined.

References

[BWSB12] BUTLER D., WULFF J., STANLEY G., BLACK M.: A naturalistic open source movie for optical flow evaluation. In *Proceedings of European Conference on Computer Vision* (2012). 1, 5

[Feh04] FEHN C.: Depth-image-based rendering, compression and transmission for a new approach on 3D-TV. In *Stereoscopic Displays and Virtual Reality Systems* (2004), SPIE. 1

[KPL05] KIM S.-M., PARK J.-C., LEE K. H.: Depth-image based full 3d modeling using trilinear interpolation and distance transform. In *Proceedings of the 2005 International Conference on Augmented Tele-existence* (2005), ICAT '05. 3

[MLS14] MALL R., LANGONE R., SUYKENS J.: Agglomerative hierarchical kernel spectral data clustering. In *IEEE Symposium on Computational Intelligence and Data Mining* (2014). 2

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3d warping. In *Proceedings of 1997 Symposium on Interactive 3D Graphics* (1997). 1

[NIH*11] NEWCOMBE R. A., IZADI S., HILLIGES O., MOLYNEAUX D., KIM D., DAVISON A. J., KOHI P., SHOTTON J., HODGES S., FITZGIBBON A.: Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality* (2011). 3

[OF17] OGNIEWSKI J., FORSSÉN P.-E.: Pushing the limits for view prediction in video coding. In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 4: VISAPP, (VISIGRAPP 2017)* (2017). 1

[PG10] PALOMO C., GATTASS M.: An efficient algorithm for depth image rendering. In *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry* (2010). 1

[SR10] SOLH M., REGIB G. A.: Hierarchical hole-filling(HHF): Depth image based rendering without depth map filtering for 3D-TV. In *IEEE International Workshop on Multimedia and Signal Processing* (2010). 3

[SV14] SCALZO M., VELIPASALAR S.: Agglomerative clustering for feature point grouping. In *IEEE International Conference on Image Processing (ICIP)* (2014). 2

[Sze11] SZELISKI R.: *Computer Vision: Algorithms and Applications*. Springer Verlag London, 2011. 1

[TLLG09] TIAN D., LAI P.-L., LOPEZ P., GOMILA C.: View synthesis techniques for 3D video. In *Proceedings of SPIE Applications of Digital Image Processing* (2009), SPIE. 1

[WLC15] WANG C., LIN Z., CHAN S.: Depth map restoration and upsampling for kinect v2 based on ir-depth consistency and joint adaptive kernel regression. In *IEEE International Symposium onCircuits and Systems (ISCAS)* (2015). 3

[WSB03] WANG Z., SIMONCELLI E. P., BOVIK A. C.: Multi-scale structural similarity for image quality assessment. In *37th IEEE Asilomar Conference on Signals, Systems and Computers* (2003). 6

[YHL16] YAO L., HAN Y., LI X.: Virtual viewpoint synthesis using cuda acceleration. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology* (2016). 1

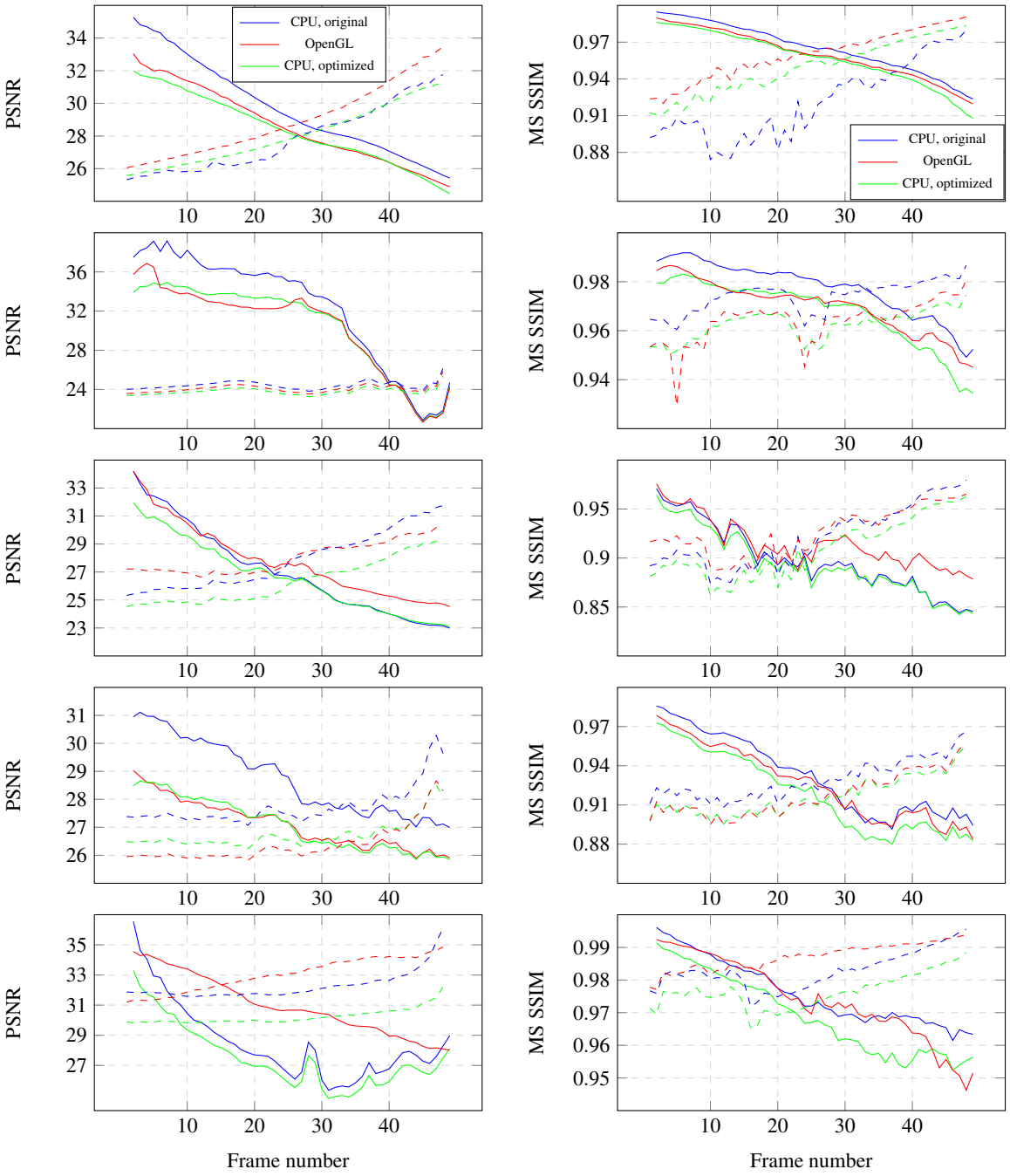

Figure 7: *Measured PSNR (a), left) and MS SSIM (b), right) between the projected images and the original images of the sequences:*
From top to bottom: alley, bamboo, mountain, sleeping and temple. Both projections from frame 1 (continuous lines) and from frame 49 (dashed lines) are shown, for each of the different DIBR methods.
Note that the curves are ordered according to their performance in the legend, the curves with the highest values are mentioned first.