Pacific Graphics 2016
E. Grinspun, B. Bickel, and Y. Dobashi
(Guest Editors)

*Volume 35 (2016), Number 7*

# Proxy-guided Image-based Rendering for Mobile Devices

Bernhard Reinert[1] Johannes Kopf[2,4] Tobias Ritschel[3] Eduardo Cuervo[2] David Chu[2,5] Hans-Peter Seidel[1]

MPI Informatik[1] Microsoft Research[2] University College London[3] now at Facebook[4] now at Google[5]

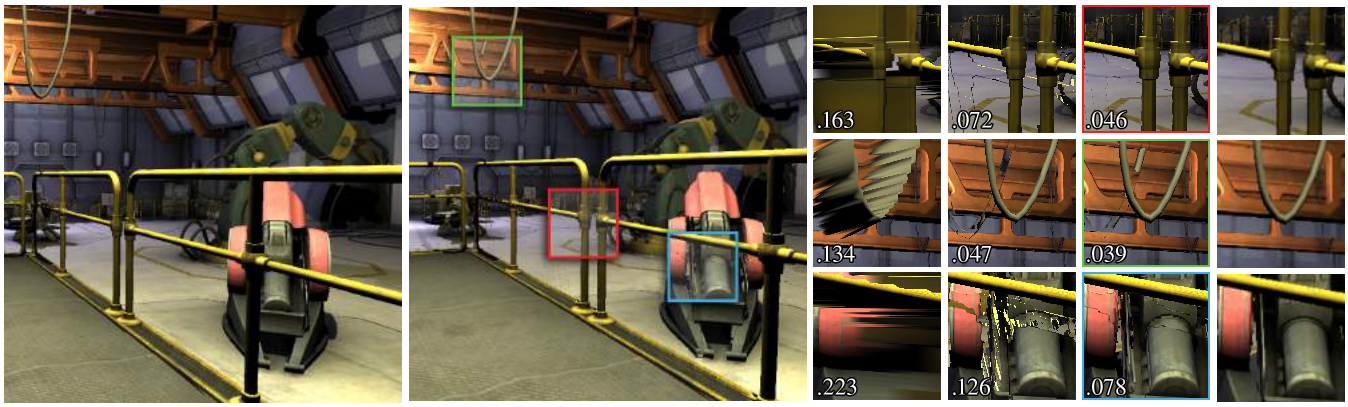



**Figure 1:** *Our new image-based rendering algorithm warps original views (with depth, not shown) to produce novel views. It performs as fast as the fastest competitors but maintains much higher visual quality for larger displacements. The insets compare against other common approaches and show the RGB-DSSIM difference. Frame rates are computed on an an Intel Compute Stick at a resolution of 1280×720.*

**Abstract**

*VR headsets and hand-held devices are not powerful enough to render complex scenes in real-time. A server can take on the rendering task, but network latency prohibits a good user experience. We present a new image-based rendering (IBR) architecture for masking the latency. It runs in real-time even on very weak mobile devices, supports modern game engine graphics, and maintains high visual quality even for large view displacements. We propose a novel server-side* dual-view *representation that leverages an optimally-placed extra view and depth peeling to provide the client with coverage for filling disocclusion holes. This representation is directly rendered in a novel wide-angle projection with favorable directional parameterization. A new client-side IBR algorithm uses a pre-transmitted level-of-detail proxy with an encaging simplification and depth-carving to maintain highly complex geometric detail. We demonstrate our approach with typical VR / mobile gaming applications running on mobile hardware. Our technique compares favorably to competing approaches according to perceptual and numerical comparisons.*

## 1. Introduction

After going through decades of neglect 2016 might be the year in which virtual reality (VR) will finally happen, backed by large investments from major corporations. Widespread interest is driven by the promise of countless applications ranging from video gaming to education, training, industrial and architectural design, art, therapy etc. But any system that wants to provide a *truly* immersive VR experience needs to excel in three mutually conflicting areas—*visual fidelity:* the virtual environments should look beautiful, detailed, and life-like, but this requires substantial computational resources; *responsiveness:* any head motion must be reflected as quickly as possible in visual feedback to prevent motion sickness, so the system needs to provide high refresh rate and low latency; and *mobility:* users ought to move *untethered* in physical space, free to explore the virtual world.

It is challenging to fulfill all three requirements simultaneously, and, no current VR platform meets more than two of these three objectives (Figure 2). Head-mounted display (HMD) systems like the Oculus Rift or HTC Vive can provide high quality graphics, but only when tethered (with a thick and short cable) to a high-end computer

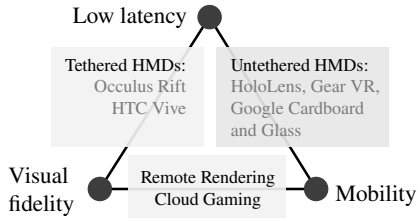



**Figure 2:** *Landscape of current VR systems*

with a powerful GPU, thus, limiting the users' mobility to narrow head movement. *Untethered* HMDs, such as the Samsung Gear VR or Google Cardboard, allow the user to freely move around while wearing the device, but they do not provide translational position tracking and the mobile GPUs force compromises on rendering quality. The Microsoft HoloLens does provide positional tracking, but the rendering performance is still limited. Wearable mobile GPUs are likely to remain limited due to battery power and thermal dissipation constraints. It is possible to improve the graphics quality of untethered HMDs by offloading the rendering task to a remote server, e.g. a Wi-Fi-connected desktop computer or a cloud machine. However, this introduces unacceptable latency for VR, up to 100 ms for Wi-Fi and in excess of 200 ms for wide-area connections.

There has been previous work on lowering the perceived latency for remote rendering. The Oculus Mobile SDK contains a function called "Asynchronous TimeWarp" [Ocu15] that updates a rendered frame at presentation time with a simple global transformation that models the interim head rotation. The same technique can also be used to avoid visual judder from dropped frames by temporally upsampling low rendered frame rates to higher screen refresh rates. The technique works well in the context of rotational-only tracking (i.e. a world at infinity), but does not model parallax, and thus cannot handle translational head motion and near-field objects (Figure 10). It is notably absent from the Oculus *PC* SDK.

State-of-the-art mobile cloud gaming systems such as Outatime [LCC*15] hide latency by transmitting both color and depth maps for each frame, which are then forward-warped using points [CW93], coarse grids [MMB97], or quad-trees [DRE*10]. These simple image-based rendering (IBR) techniques suffer from several problems: (1) mobile GPUs are not powerful enough to render many primitives; however, subsampled depth meshes cause a variety of rendering artifacts, e. g., problems with thin structures and poorly localized depth discontinuities (Figure 1); (2) depth maps do not store connectivity information, so heuristics need to be used to decide whether neighboring samples are part of the same surface or a disocclusions (i. e., hole or tear) should open up; (3) only pixels that exist in the reference viewpoint can be warped correctly. Disoccluded pixels need to be inpainted, which causes distracting visual artifacts. In this paper, we present a new method that is designed to avoid the problems mentioned above and achieve all three earlier mentioned objectives—visual quality, responsiveness, and mobility—simultaneously.

Our technique comprises new server-side and client-side algorithms. We propose a novel *dual-view* representation for the server-rendered frames, that comprises the primary and an extra view, rendered from an optimally placed camera offset and using a depth peeling technique to avoid redundancies. It leverages a novel wide-angle projection with full hemisphere coverage and favorable directional parameterization.

A new client-side IBR algorithm replaces the slow forward-warping of previous methods with faster backward-warping, enabled by delivering a level-of-detail (LoD) scene representation to the client ahead of time. The combination of a special *encaging* LoD simplification and a depth-carving pixel shader enables precise reproduction of highly detailed silhouettes. The pixel shader also inpaints residual disocclusions without extra performance cost.

Our technique is fast enough to run on very modest client hardware and can extrapolate high-quality novel views even for large view changes. It supports temporally upsampling low and/or fluctuating server frame rates to constant high client frame rates. In addition to VR, our techniques are also applicable to regular (non-VR) cloud-based gaming.

We tested an end-to-end prototype implementation of our algorithms on different mobile systems including an extremely low-powered Intel Compute Stick (i.e. a fat-finger sized computer that plugs into any HDMI port). Our results compare favorably to competing approaches in a perceptual user study as well as in numeric evaluations. An important limitation of our base system is that it only supports static scenes. However, in Section 6.3, we discuss three extensions for handling dynamic content with different trade-offs.

## 2. Related work

In this work we are interested in masking network latency in a remote rendering setup for interactive untethered VR or mobile cloud gaming applications. Shi and Hsu [SH15] provide a survey on the state-of-the-art in interactive remote rendering. One solution is image-based rendering (IBR), which describes the concept of synthesizing novel views of a 3D scene from one or multiple existing pre-rendered (or captured) views [CW93]. The idea is to extrapolate images instead of transferring them across a network for remote rendering. In this paper, we propose an improved IBR approach that is custom-tailored to the requirements of remote rendering.

The most related work to our effort is the Outatime system [LCC*15], which shares a similar goal, though they only discuss cloud gaming (which may use more powerful GPUs and has lower requirements on target frame rate). Their key contribution is a speculative execution engine: the server predicts future user actions from the past behavior and recent tendencies, and renders multiple frames ahead of time for different possible outcomes. The frames are then sent to the client which selects the one that is most relevant to the actual user input that has happened since. As a result, the frames are often close to what the client ought to see, though, slight mispredictions still have to be compensated for. To that end, Outatime uses a very simple IBR approach, single-view coarse mesh-based forward warping, which shows poor performance on our target devices and leads to a variety of artifacts.

The new IBR algorithm presented in this paper could be used as a drop-in replacement to achieve better performance and higher visual fidelity in a system like Outatime. For simplicity's sake we tested

our algorithm in a standalone fashion, i.e., we don't use any form of user input prediction or speculative execution and our server simply renders views for the last known user state. In the following we will discuss different flavors of IBR in the context of this scenario.

**Ray space** It is possible to perform IBR purely in ray space without the need for any geometric scene information [LH96,GGSC96]. This is attractive if such data is not available, e.g. for captured scenes. However, since we are rendering 3D models, we can trivially pass 3D scene information to the IBR algorithm. Utilizing it generally leads to more accurate results.

**Forward warping** Most of the earlier IBR algorithms rely on *forward* warping to deform the input images before blending them [Wol98]. In their seminal paper, Chen and Williams [CW93] propose to represent the scene using pairs of color and depth maps. The depth information is used to *independently* project and splat each pixel. Since the splats are not connected, small holes can form in the output, and the massive amount of rendering primitives is prohibitively slow on our target GPUs. Resulting warp quality can be improved by predicting its outcome on the server and transmitting adaptively compressed residuals [YN00, BG04]. Forward warping was used for remote rendering on mobile devices [CG02], but remains of low fidelity and speed without addressing its specific requirements. Follow-on work resolved the lack of connectivity by using fine grid-meshes with micro polygons [MMB97]. Continuous meshes produce long and skinny "rubber sheet" triangles at depth discontinuities that connect the edge of the foreground object with the background, although heuristics may be able to introduce tears in these cases. Fine meshes still employ a prohibitively large number of rendering primitives. Better performance is achieved by downscaling the meshes [LCC*15]. However, this has a degrading effect on visual fidelity, since thin structures cannot be well represented and depth discontinuities cannot be localized. Quadtrees [DRE*10] provide a more adaptive way of reducing the number of primitives. Novel views of a quadtree can also be produced by ray-tracing [WPS*15]. Our experiments show that while quadtrees work well for moderately complex scenes, the highly detailed scenes used in modern graphics engines require too many subdivisions, and, hence, incur encoding, transmission and processing overhead that does not fit the simple GPUs of mobile clients. In this paper, we advocate using simplified 3D geometry instead of simplified 2D depth geometry as in quadtree-based IBR.

**Multiple layers** An improvement in quality can be achieved by rendering not just the first visible surface, but multiple layers [SGHS98]. Our approach uses a related representation: the server generates dual-view pairs, comprising a primary view for visible surfaces and an extra view for surfaces that are occluded in the primary view.

**Backward warping** Instead of forward-projecting points in the input image to the output space, another option is to start with a pixel in the output image and search for a location in the source image that projects there [YTS*11, BMS*12]. This search can be done in a simple iterative procedure in a pixel shader, which leads to very fast performance. However, as our experiments show, this approach is critically dependent on good initialization, especially for large view offsets and at depth discontinuities, where the results might be noisy (see Section 6.2).

**Proxy geometry** Another kind of backward-warp algorithms renders approximate 3D proxy representations of the scene objects, and gathers color samples by projecting the resulting fragments into the input images [DTM96, BBM*01]. Our method belongs to this category. Our client-side algorithm renders a dramatically simplified pre-transmitted level-of-detail representation of the scene to efficiently gather color and depth from the input views, and uses the depth information to refine silhouette boundaries as well as resolve occlusion ordering.

**Phase-based** Working in phase space does not require depth information, is fast and simple and allows for specular and transparent surfaces [DSAF*13]. Regrettably, it can only be applied to small baselines, preventing its use for novel-view synthesis of practical head motions in an interactive application such as a computer game.

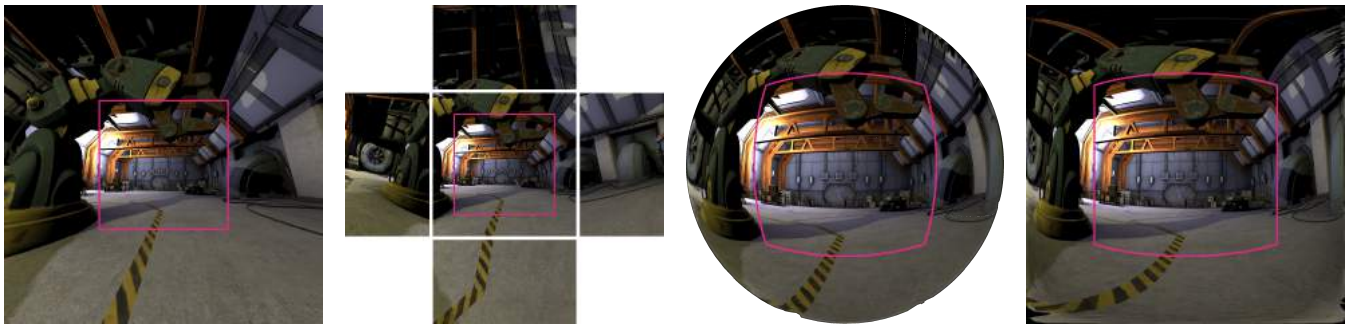
**Non-Lambertian scenes** Most algorithms discussed in this section—including ours—tacitly assume diffuse surfaces that have the same appearance from all viewing directions. Some recent algorithms [SKG*12, LRR*14] extended IBR toward supporting some non-Lambertian effects, e.g., specularities and reflections. Many of these ideas could potentially be incorporated into our method, although we leave this for future work.

## 3. Overview

Since our client devices have neither the power to run a modern game engine nor to render high-fidelity graphics, they merely transmit the user input events to a high-end network-connected computer, which executes the game engine and streams high-fidelity "views" back to the client, each consisting of color and depth maps, as well as camera pose parameters.

The key technical problems are in dealing with network latency, and low or fluctuating server frame rates. As in previous approaches [LCC*15], we run an IBR algorithm on the client to mask the latency by warping the latest received frame to account for interim camera pose changes. There are substantial technical difficulties in this approach: (1) due to the high latency, the camera position and orientation can have changed significantly since the server rendered the last frame, so our IBR algorithm has to be able to maintain high visual fidelity even in the presence of strong induced parallax; (2) while many previous temporal upsampling algorithms generate novel in-between views by *interpolating* past and future frames, our system operates in real-time, so we cannot access future information and have to *extrapolate* our novel views purely from past frames; (3) we target light-weight mobile devices with low-powered GPUs, so our algorithm needs to be very fast. Our method comprises both server- and client-side algorithms that are carefully designed to deal with these challenges.

In the next two sections, we first present our novel server-side generated dual-view representation (Section 4) followed by our novel client-side proxy-based IBR algorithm (Section 5).



(a) Perspective (b) Partial cube map (c) Hemispherical fisheye (d) Final projection

**Figure 3:** *Comparison of different projection methods. The magenta line indicates a view with horizontal and vertical* $70^\circ$ *FOV. A favorable projection should have sufficient pixel density inside the line and slowly reduce pixel density in the periphery.*

## 4. Dual-view Representation

In a simple IBR architecture, a server might stream regular depth-augmented color images to the client, which only cover the visible surfaces seen from the last known client pose. This is not ideal, however, because they do not provide the client with information to deal with disocclusions, i.e. previously unseen parts of the scene that come into view due to pose changes.

We distinguish between two types of motion that induce disocclusions: *translational motion* induces parallax and uncovers parts of the scene that were previously occluded by other objects closer to the camera, while *rotational motion* does not induce parallax but uncovers previously out-of-frame directions. We introduce a new view representation that is designed to help the client handle both situations.

Rotational disocclusions are easily dealt with by providing the client with wider field-of-view (FOV) imagery. However, it is tricky to do this efficiently, both in terms of storage and rendering performance. We propose a $180^\circ$ wide-angle projection that can be directly rendered to in a single pass and that concentrates samples where they are most useful (Section 4.1).

To alleviate translational client motion, we generate an extra view that provides some coverage of surfaces that are occluded in the primary view, and use a depth peeling technique to avoid storing redundant surfaces, i.e. ones that are already visible in the primary view (Section 4.2).

### 4.1. Wide-angle Projection

As discussed before, generating views that are wider than the client FOV allows compensating some rotational motion. Since the maximal rotation rate is unbounded, only transmitting a full $360^\circ$ sphere would be completely safe. However, in practice it is acceptable to leave a few pixels missing during rapid rotations [Ocu15], so we settle on transmitting a $180^\circ$ hemisphere per view.

Standard linear perspective projections distribute samples unevenly across the viewing plane: the lowest density is in the image center, while highest in the periphery. This is the exact opposite of the ideal situation, since the center parts are more likely to be used in novel views than the image edges when moving forward, and fast rotation masks a slight undersampling of peripheral regions. Moreover, beyond ca. $90^\circ$, perspective projections become increasingly distorted (Figure 3a).

Partial cube maps (Figure 3b) have been used to alleviate this problem [MMB97, BP06, LCC*15]. However, cube maps are expensive to generate since each cube face requires a separate rendering pass, or a slow geometry shader needs to be employed to replicates primitives into multiple output render buffers. In addition, they still allocate a larger-than-necessary proportion of samples to peripheral parts of the image, and, thus, excessive image sizes are necessary for acceptable image quality.

We are interested in a projection where most samples concentrate in the center (i. e., where they are most useful) and the density gradually decays towards the periphery. A hemispherical fisheye projection has this property (Figure 3c). It is constructed as follows: Let $p$ be a 3D point in camera coordinates, i.e., already transformed by the model-view matrix. We first project $p$ onto the unit sphere, and then orthographically project it on the view plane (by dropping the $z$ component). Formally, the texture coordinates $(t_x, t_y) \in [-1,1]^2$ are obtained as $t_x = p_x/\|p\|$ and $t_y = p_y/\|p\|$.

This projection has the slight disadvantage that the corners of the image remain unused (Figure 3c). While this does not incur a significant network bandwidth cost, since the empty areas compress efficiently, it still amounts to a somewhat poor utilization of texture space. We use a slightly modified projection that resolves this problem by horizontally stretching each row to fill the full height of the texture (Figure 3d):

$$t'_x = \frac{t_x}{\sqrt{1-t_y^2}}, \quad t'_y = t_y. \tag{1}$$

We could have alternatively stretched vertically. However, since panning rotations occur more frequently, we opted for filling the corners with samples that are most useful for this case. We also considered fisheye projections that fill the corners with samples from behind the camera. However, these seemed less useful for novel view generation than extra samples obtained by the modified projection in Eq. 1.

This wide-angle projection is not linear. However, we can still efficiently render the scene directly in a single pass by computing Eq. 1

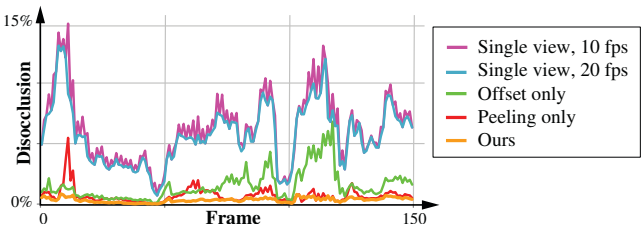



**Figure 4:** *Fraction of missing pixels during a walkthrough of the* VIKING VILLAGE *as seen in Figure 5.*

in the vertex shader instead of applying the traditional projection matrix, and clipping against near and far clipping *spheres* instead of planes. Straight lines in world-space turn into curves in a fisheye projection, hence, triangles that cover a large screen-space area need to be subdivided to account for this non-linearity. To this end, we employ a tessellation shader that adaptively subdivides triangles based on their screen-space size to cover less than a pixel.

### 4.2. Dual-view Generation

Translational disocclusion is another source of missing regions that is more difficult to deal with than rotational disocclusion, because the holes occur not only at image boundaries but spread across the entire image, and they uncover areas unseen from the original vantage point.

If there is network latency between server and client the amount of disocclusion can be reduced by anticipating future client motion, and rendering frames from the predicted location instead of the last known location. We experimented with using a Kalman filter to predict client motion, however, the reaction time to acceleration changes commonly caused mispredictions, which momentarily increased the amount of disocclusions. The Outatime system [LCC*15] is more successful with a similar strategy because their server renders and transmits *multiple* frames for different motion hypothesis, out of which the client then picks the most suitable one, which in turn however increases the required bandwidth.

Some disocclusion holes can be filled in using extra views from different vantage points. One potential source are views that have been transmitted in the past. However, we found that they usually do not provide sufficient coverage to fill disocclusion holes. Instead, we generate and transmit a *dual-view*, consisting of a *primary view* rendered from the last known client location and an *extra view* rendered from an offset location to provide extra visibility coverage. Both views are rendered at the same point in game time, and we store the extra view at quarter resolution (i.e., half image dimensions).

The extra view should ideally not contain redundant coverage of surfaces that are already visible in the primary view. However, it is difficult to selectively transmit only the unique sub-parts of the extra view efficiently. Instead, we use a *depth peeling* technique [SGHS98] to prevent inclusion of redundant samples in the extra view to begin with. In a pixel shader, we project every extra view fragment into the primary view. Here, a z-buffer comparison determines if the fragment is visible in the primary view, i.e. redundant, in which case it is discarded.

Finding the optimal placement of the extra view is a difficult visibility problem, which is impractical to solve in real-time. Instead, we precomputed a heuristic, *constant* offset by analyzing long user traces in multiple scenes (Figure 4 shows an example). We excluded sideways translation and rotation from the optimization, since we did not want to introduce horizontal bias. We used exhaustive search to determine the optimal setting for the remaining vertical and forward/backward translation as well as the tilt rotation angle. This procedure yielded a pose offset 1.5 m in front, 1.8 m above (Unity game engine units), and 15° tilted down relative to the primary view, which we heuristically used to produce all results shown in this paper. We have also experimented with generating the extra view pose using a Kalman filter, but this yielded a less optimal result than the constant offset.

Figure 5 illustrates different variants of the dual-view representation, and shows how using both camera offset and depth peeling in the extra view produces the most complete novel view. Figure 4 expands the comparison and plots the fraction of missing pixels for each frame in a 150 frames long walkthrough sequence. As can be seen the combination of both, depth peeling and a constant camera offset, leads to the lowest number of undefined pixels.

For transmission, linear depth units are converted into disparity (inverse depth) units, quantized to 8 bits. Using disparity instead of depth reserves more precision for depth details in nearby geometry, which moves noticeably when the view position changes and allocates less precision for distant objects, such as the sky, that move little in response to user motion. Finally, the pair of primary and extra color and depth images are placed in a rectangular layout and encoded in a single H.264 stream.

## 5. Proxy-based IBR Algorithm

The client receives the stream of dual-view images with a certain delay and at a potentially lower frame rate than desired. Its task is to then synthesize using IBR what the scene would look like from the new pose it has moved to since the server rendered the last frame. Since the network latency can be significant, the method has to be able to handle large view changes. Since it runs on a weak mobile device and we target high output frame rates, it needs be very fast.

Most IBR methods that are suitable for real-time applications use some form of depth map-based forward warping, i.e., pixel splatting [CW93], grid-mesh warping [MMB97], or quad-tree warping [DRE*10]. Iterative image warping [YTS*11, BMS*12] is a notable exception, but it does not provide sufficient quality for our application; see the discussion in Section 6.2. The primary problem with these methods is that they require a prohibitive number of primitives to represent detailed geometry and perform very slowly on our target devices (Section 6.2). To improve the performance the primitive count can be decreased, i. e., by subsampling the depth maps. This however has a degrading effect on visual quality. In particular, geometric details cannot be accurately represented anymore and depth discontinuities become poorly localized (cf. Figure 1a in the supplemental material). Another problem is that depth maps do not store connectivity information. Whether neighboring samples belong to the same surface and hence should be conntected can only be decided using heuristics that can fail and produce erroneous

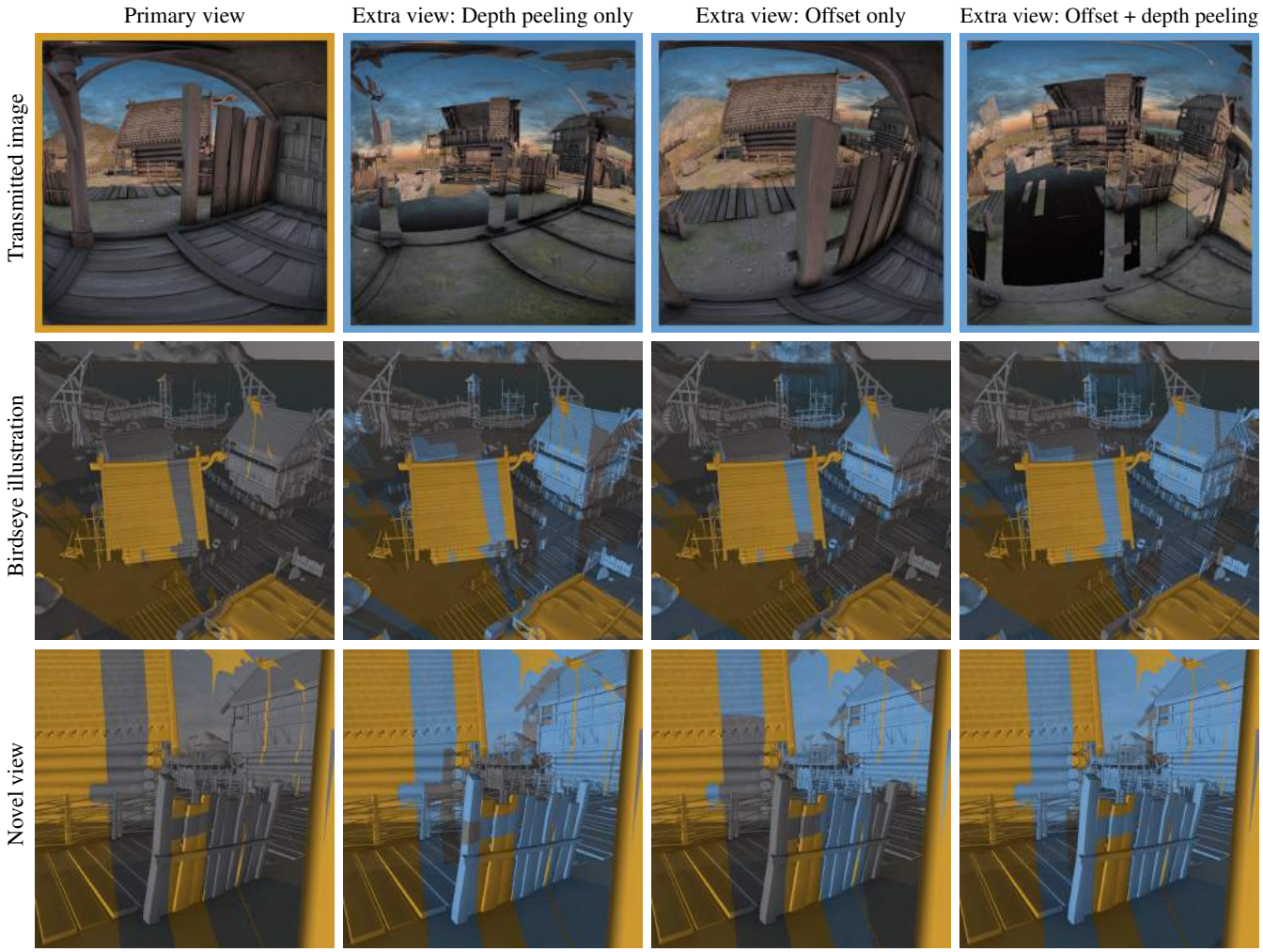


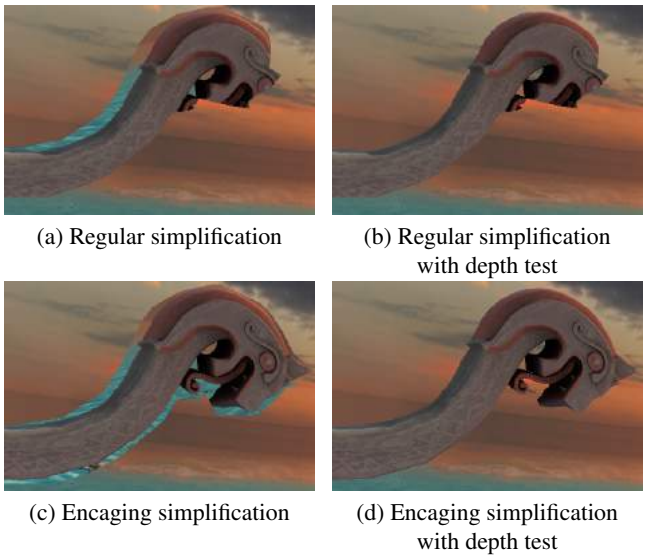

**Figure 5:** *Analysis of dual-view sampling: The first column shows an image of the scene from the primary view, while the other columns show images of three increasingly refined approaches to producing the extra view: only depth peeling, only offsetting and finally both simultaneously. The first row shows images taken from the respective view in each column, the second row shows the world from an bird's eye-view, and the third row shows a novel view. In the second and third rows visibility is color-coded: gray surface points are not seen by any of the two views, orange/blue points are only seen by the primary or extra view, respectively. Note, that due to depth peeling, no surfaces are ever seen by both views. The bottom row shows the variants adding coverage : in the right-most novel view almost all disocclusions have been filled in.*

tears or long "rubber sheet" triangles. In this section, we propose a new IBR method that avoids these problems, produces high quality results and runs fast on our target devices.

In particular, we use a geometry proxy. Assuming that our scene geometry is mostly static as well as entirely and precisely known in advance, enables us to generate and transmit a level-of-detail (LoD) representation of the whole scene in advance to the client. When rendering a novel viewpoint we adjust the LoD to achieve a *screen-space* geometric complexity that is roughly constant and dramatically decimated compared to the original geometry, so it requires only relatively few primitives and renders fast on our target devices. The simplified geometry serves as scene proxy [DTM96, BBM*01]. In a pixel shader we back-project fragments into both source views (primary and extra), gather color samples, and compute an output color as described below. Implementation details of the LoD generation and rendering are described in Section 5.1.

Since we use a geometry-adaptive simplification, major depth discontinuities remain well-localized. However, silhouettes are only approximated with simple outlines, causing some artifacts when a foreground object with intricate detail is projected onto such a proxy: a fraction of background samples projects erroneously on the foreground (Figure 6a). To remove these erroneous samples we use the transmitted depth maps of our dual view representation (Section 4) in a pixel shader as follows.

Let $f_\mathrm{p}, f_\mathrm{e}$ be the distances between a fragment's world position and the camera positions of the primary and extra view, respectively, and let $d_\mathrm{p}, d_\mathrm{e}$ be the values sampled from the respective depth maps



(a) Regular simplification (b) Regular simplification with depth test

(c) Encaging simplification (d) Encaging simplification with depth test

**Figure 6:** (a) *Common mesh simplification causes background to be projected onto foreground (neck of figure) as well as foreground to be cut off (missing nose).* (b) *Depth testing removes erroneous background, but foreground details remain lost.* (c) *Using an encaging simplification guarantees to never remove foreground details.* (d) *Combining both techniques results in a near-perfect image.*

(a) No hole filling (b) 2D Pull-push (c) Our hole filling

**Figure 7:** *Comparison of different hole-filling strategies.*

at the back-projected location of the fragment. We compute depth errors for both views,

$$e_p = \left| \frac{d_p - f_p}{f_p} \right|, \quad e_e = \left| \frac{d_e - f_e}{f_e} \right|, \tag{2}$$

and successively test both errors against a threshold of $\tau_{depth} = 0.1$. If one of these depth errors is smaller than $\tau_{depth}$, we use the color of the respective view, otherwise we discard the fragment. While this algorithm successfully removes erroneous background samples on the forground object it cannot recover foreground details that are located outside of the simplified outline (Figure 6b). We fix this problem by modifying the LoD decimation to produce a *strictly encaging* simplification, i.e., one that surrounds the original geometry without intersecting it [SVJ15] (Figure 6c, Section 5.1). This guarantees that the shader described above never clips foreground samples and is able to precisely cut out the original detailed silhouette (Figure 6d).

Usually, a few isolated disocclusions remain that cannot be filled from either of the two views (Figure 7a). Most real-time IBR implementations use the pull-push algorithm [GGSC96] to inpaint these smoothly (Figure 7b), since it can efficiently be implemented by creating and collapsing an image pyramid on the GPU. Our experiments showed, however, that this roughly halves the frame rate, which pushes us well below 30 frames per second on our weakest devices. As an alternative, we modify the shader above so it does not discard fragments anymore to achieve hole inpainting with zero performance overhead. Instead of filling gaps with a smooth membrane it fills with misplaced texture (Figure 7c). However, just changing the shader to always accept all fragments and use the color from the lower-error view does not work, because it eliminates the ability to carve out the original silhouettes. Instead, we "deprioritize" fragments that would have been discarded by rewriting their $z$-value behind any fragments within the depth threshold. Pseudocode for this algorithm can be found in the supplemental material.

### 5.1. Level-of-detail Implementation Details

There is extensive literature on geometry simplification [Lue97] and level-of-detail techniques [DFKP05]. Modern game engine implementations of these algorithms are highly sophisticated and tuned for maximum performance. As this work is mostly interested in building a proof-of-concept system in reasonable time, we compromised on an extremely simplistic LoD implementation, described below. We would like to stress that we do not claim any contribution in this section, and merely include the description for completeness. Any practical system should be built on top of existing game engines and utilize their built-in LoD systems, for achieving best performance and quality.

Our test scenes are downloaded from public 3D scene repositories. They are composed of individual objects, potentially instanced multiple times throughout the scene. For each object, we precompute three LoD representations: the original geometry, a simplified mesh, and a bounding box (Figure 8). At runtime we cull objects against the view frustum and select the appropriate LoD based on distance thresholds.

Most original meshes in our test scenes are not watertight and contain self-intersections, tiny gaps, and other defects. We tried a variety of existing simplification methods, in particular, PolyMender [†] [Ju04], Poisson surface reconstruction [‡] [KH13], and the Blender decimation modifier [§], but they did not handle the mesh defects satisfactorily. So, we implemented a simple and robust alternative. We first voxelize each object into a $128^3$ array. This closes tiny gaps in the original mesh. Next, we remove big internal holes by flood-filling from the outside and inverting the selection. Finally, we use the marching cubes algorithm [LC87] to produce a watertight mesh, which we simplify using the Blender decimate modifier to 0.5% of the original number of triangles.

The recent Nested Cages algorithm [SVJ15] is able to produce strictly encaging simplifications. However, source code is not available. Fortunately, the voxelization described above already tends to produce slightly enlarged meshes. We detect any violating vertices

† http://www.cs.wustl.edu/~taoju/code
‡ http://www.cs.jhu.edu/~misha/Code/PoissonRecon
§ https://www.blender.org

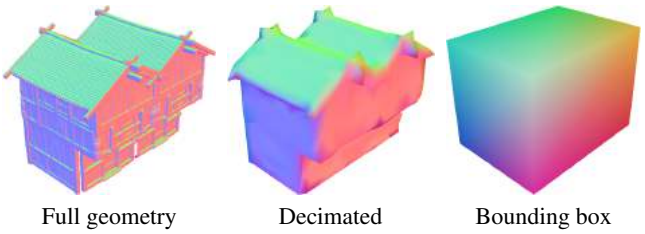



**Figure 8:** *LoD levels used in our implementation.*

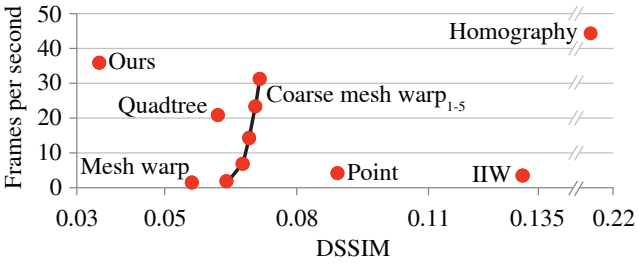



**Figure 9:** *Performance/quality tradeoff for different methods.*

in the simplification, and simply push them outwards. This happens rarely, and worked well in our experiments.

## 6. Results and Evaluation

### 6.1. Implementation Details

A prototype of our system was implemented in C++ using the OpenGL API. We ran our server-side algorithms on an Intel Xeon E5645 (2.40 GHz) CPU with 24 GB RAM and an NVidia GTX680 GPU. Our client side algorithms were tested on several different machines that allow for mobile usage ranging from an Intel Compute Stick STCK1A32WFC over an HP Stream Mini 200-010 and an HP Pavilion Mini 300-030 to a Microsoft Surface 3 Pro. Our algorithm was tested on three scenes (cf. Figure 2 in the supplemental material). We implemented our own rendering engine to generate the dual-views on the server as it offered more control over the rendered images as the build-in Unity rendering engine. For a 720p client resolution the primary view size is set to $2048\times2048$, for 1080p client resolution the primary view size is set to $3072\times3072$. To get a similar sample density in the center of the original view with a FOV of 70°, the resolution of each dimension of the output view needs to be multiplied with a factor of ca. 1.6 to get the resolution of the input view. When storing the dual view at 10 fps for 720p resolution it requires a bandwidth of 2,516 mbps uncompressed and 2,910 kbps when compressed with H.264.

### 6.2. Comparisons

We compared our algorithm against a variety of existing algorithms, which in turn have a varity of parameters described below. In the supplementary material we provide more extensive comparisons. Figure 9 shows average timings and DSSIM quality measures for 20 snapshots of the ROBOT LAB scene. All timings are measured on an Intel Compute Stick, for a more detailed analysis including other devices please see the supplemental material. Figures 1 and 10 show the visual quality of our method in comparison to other methods. In all of our experiments we used a simulated latency of 128 ms and the server produced a video stream with 10 frames per second.

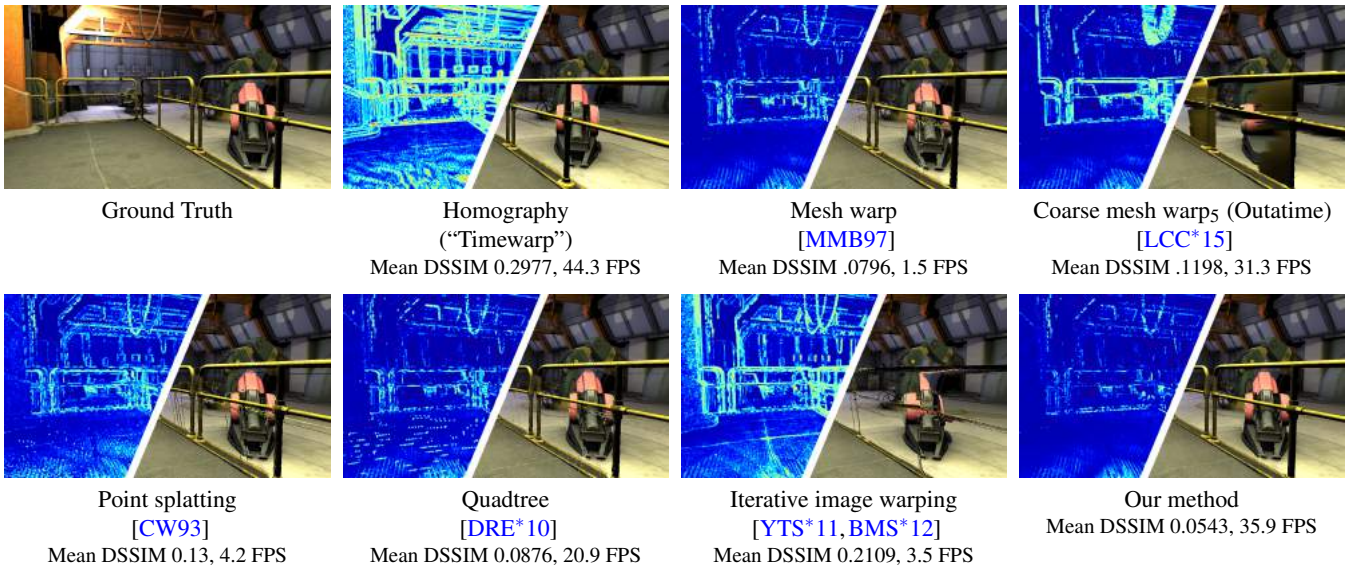
All methods use the same input, i. e., the color and quantized depth values generated by our dual-view generation (Section 4). Per-pixel depth values and the view matrix are used by all methods to reconstruct the world position of each pixel of the input view which in turn can be used to reproject the pixel into the new view, resulting in a forward flow field for each view. We did not use any additional hole filling for any of the algorithms described, as this step substantially decreased the performance on the target hardware and is orthogonal for all methods. Next, we will discuss the details of alternative techniques:

**Homography** as recommend in the Oculus SDK [Ocu15] assumes that all pixels have constant, fixed depth value allowing to compute the backward flow for each pixel in closed form. While this approach is capable of correcting rotational movements is fails to correctly handle disparities induced by view translation, leading to visible jumps when inputs views are updated. Homography only uses the first of our dual views.

**Point Splatting** by Chen and Williams [CW93] transforms a single point primitive for each pixel of the input view image to be drawn in the output view. To close small holes the point size is set to cover two pixels of the final screen size.

**Mesh Warp** uses the same transformation as point splatting, but assumes implicit connectivity between the pixels, i. e., renders triangles between neighboring pixels. A naïve implementation renders all triangles regardless if its vertices make up a connected surface or not, leading to rubber sheet-like triangles at depth discontinuities. To increase the performance, the pixels can be subsampled, i. e., instead of rendering a full set of vertices for all pixels only for each $n$-th pixel a vertex can be emitted [LCC*15]. Similar to the connectivity heuristic described in [MMB97] stretched triangles can be split into two triangles: one with the minimal and one with the maximal depth value of all three vertices. Instead of using a position and a normal buffer to determine the connectivity, our decision is based solely on the depth buffer as transmitting an extra normal or connectivity buffer is not feasible in our streaming setup. We tested different setups: one setup in which we used both views, split the triangles at depth discontinuities and emit one vertex for each pixel, denoted as *Mesh warp*. The second choice uses only the first view, without splitting the triangles and with a subsampled depth map resulting in an increased performance for different subsample factors $s$ denoted as *Coarse mesh warp*$_s$.

**Quadtree Warping** combines groups of neighboring pixels to form quads that can be warped together to reduce the number of primitives that need to be drawn. Since computation of the quadtree requires powerful parallel hardware only the server can generate the quadtree. As the client movement is unknown to the server at construction time, the disparity measure used to construct the quadtree [DRE*10] cannot be used in our case; instead we group pixels based on their



Ground Truth

Homography ("Timewarp")
Mean DSSIM 0.2977, 44.3 FPS

Mesh warp [MMB97]
Mean DSSIM .0796, 1.5 FPS

Coarse mesh warp$_5$ (Outatime) [LCC*15]
Mean DSSIM .1198, 31.3 FPS

Point splatting [CW93]
Mean DSSIM 0.13, 4.2 FPS

Quadtree [DRE*10]
Mean DSSIM 0.0876, 20.9 FPS

Iterative image warping [YTS*11, BMS*12]
Mean DSSIM 0.2109, 3.5 FPS

Our method
Mean DSSIM 0.0543, 35.9 FPS

**Figure 10:** *Numerical analysis. We used the DSSIM metric to asses the visual quality of the results of different methods to ground truth.*

depth values to construct the quadtree. To combine multiple views, we used the same triangle split as done in mesh warping, as rendering each view into different framebuffers and combining them based on their stretch factor as described in [DRE*10] yielded inferior results both in terms of speed and quality on our target hardware. Quadtrees reduce the number of primitives that need to be drawn to roughly 10 % of the full number of primitives in our scenes, but produces small gaps in between quads at level boundaries.

**Iterative image warping** [YTS*11, BMS*12] strives to invert the given forward flow for each of the pixels to obtain the desired backward flow that can be used for simple texture lookups with a cheap iterative optimization per pixel. While this algorithm works great for small and known camera offsets it heavily relies on a good initialization, especially for pixels with multiple solutions (occluding objects). While for certain camera configurations simple heuristics apply, for a generic camera motion as in our case the initialization needs to be done i. e., using a quadtree, resulting in a worse performance than the initialization method itself while not significantly improving the quality. Additionally, the different projections of our input and output views (spherical and perspective projection) cause the unscaled iteration to diverge quickly, requiring the correction vectors to be damped, which in turn leads to an increase of number of iterations. For our experiments we used 10 iterations with a damping factor of 90%.

### 6.3. Perceptual Evaluation

We assessed the effectiveness of our approach in a perceptual experiment, where it is compared to alternative approaches in terms of visual fidelity for a remote rendering scenario. 30 subjects were asked to compare the result of our approach and a competing approach in a two-alternative forced choice task. The results can be seen in Figure 11 and are discussed in the supplemental material.

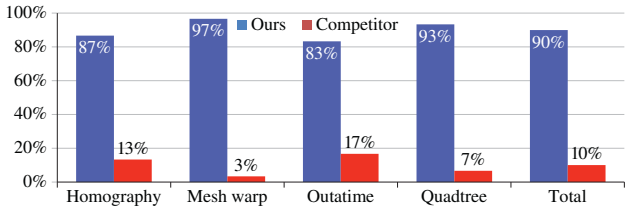



**Figure 11:** *Perceptual evaluation: Mean preference of our method* (blue) *over competitors* (red) *in percentage (all $p < .0001$).*

## 7. Discussion

As can be seen, e. g., in Figure 10 and the submission video, some limitations and artifacts remain. Their first and main source are remaining disocclusions not covered in either of the two views (cf. Section 4). These disocclusions could trivially be overcome by sending more than two views. However, making the view offset and depth peeling parameters viewport-adaptive might provide the same quality with no additional bandwith overhead. A second source of remaining artifacts stems from the usage of screen-space ambient occlusion (SSAO) [BSD08] employed to speed up our server-side computations. SSAO varies with different screen content, i. e., for different viewports, and the position offset of the second view (cf. Section 4.2) leads to dissimilar SSAO in our dual views, which in turn leads to minor but perceivable artifacts at view boundaries.

Our method shares a number of limitations with most IBR algorithms. One limitation of our method is the weak support for dynamic scenes. We implemented three different fallbacks for supporting certain kinds of dynamics, each having different advantages and drawbacks, further discussed in the supplemental material. What is more, the client-side algorithm renders all surfaces with diffuse shading. Strongly view-dependent effects, such as reflections or specularities, therefore, might lead to artifacts whose severity depends on the amount of view extrapolation. In practice, we found

these often acceptable, however. The last paragraph in Section 2 discusses recent advances in IBR that could potentially be incorporated into our method to alleviate these problems. Another limitation is that the transmitted depth maps contain only a single depth per pixel. This causes problems with semi-transparent surfaces, such as glass windows.

## 8. Conclusion

We have presented a new proxy-guided IBR architecture that runs fast on very weak mobile devices, supports modern game engine graphics with intricate geometric detail, and maintains high visual quality even for large view displacements.

We propose a novel *dual-view* representation. The extra view is placed at an optimal camera offset and leverages depth peeling to avoid redundancy and equip the client with maximal surface coverage for filling disocclusions. We render both views directly into a novel wide-angle projection that covers a full hemisphere, and, unlike standard projections, concentrates resolution in the image center, where it is most needed. We transmit the video-encoded dual-view stream to the client, where it arrives with network delay and a lower-than-desired frame rate. On the client we run a new proxy-guided IBR algorithm to mask the network latency and generate novel views at a high frame rate. Our algorithm utilizes a pre-computed and pre-transmitted LoD representation to quickly render a 3D scene proxy. We use a special *encaging* LoD simplification and a depth-carving pixel shader to maintain highly complex silhouette details without compromising the frame rate. The pixel shader also inpaints residual disocclusions without extra performance cost. We have extensively analyzed and compared our algorithm with a variety of competing techniques. A numeric DSSIM analysis and a perceptual user study show that our technique compares favorably.

There are numerous interesting avenues for future work. Our base technique only supports static scenes. However, we have described and prototyped three extensions for dynamic content, that each have different characteristics w.r.t. what type of motions are supported, performance, delay, and visual quality. Further analysis and improvements in this area seem very valuable. It would also be interesting to combine our method with recent advances on supporting non-Lambertian scenes in IBR algorithms.

## References


[BBM*01] BUEHLER C., BOSSE M., MCMILLAN L., GORTLER S., COHEN M.: Unstructured lumigraph rendering. In *Proc. SIGGRAPH* (2001), pp. 425–432. 3, 6

[BG04] BAO P., GOURLAY D.: Remote walkthrough over mobile networks using 3-D image warping and streaming. *IEEE Proc. Vision, Image and Signal Processing 151*, 4 (2004), 329–36. 3

[BMS*12] BOWLES H., MITCHELL K., SUMNER R. W., MOORE J., GROSS M.: Iterative image warping. *Comp. Graph. Forum (Proc. Eurographics) 31*, 2pt1 (2012), 237–246. 3, 5, 9

[BP06] BOUKERCHE A., PAZZI R. W. N.: Remote rendering and streaming of progressive panoramas for mobile devices. In *Proc. MM '06* (2006), pp. 691–4. 4

[BSD08] BAVOIL L., SAINZ M., DIMITROV R.: Image-space horizon-based ambient occlusion. In *ACM SIGGRAPH 2008 Talks* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 22:1–22:1. 9

[CG02] CHANG C.-F., GER S.-H.: Enhancing 3D graphics on mobile devices by image-based rendering. In *Proc. PCM* (2002). 3

[CW93] CHEN S. E., WILLIAMS L.: View interpolation for image synthesis. In *SIGGRAPH* (1993), pp. 279–88. 2, 3, 5, 8, 9

[DFKP05] DE FLORIANI L., KOBBELT L., PUPPO E.: A survey on data structures for level-of-detail models. *Advances in Multiresolution for Geometric Modelling* (2005), 49–74. 7

[DRE*10] DIDYK P., RITSCHEL T., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Adaptive image-space stereo view synthesis. In *Proc. VMV* (2010), pp. 299–306. 2, 3, 5, 8, 9

[DSAF*13] DIDYK P., SITTHI-AMORN P., FREEMAN W. T., DURAND F., MATUSIK W.: Joint view expansion and filtering for automultiscopic 3D displays. *Proc. SIGGRAPH Asia 32*, 6 (2013). 3

[DTM96] DEBEVEC P. E., TAYLOR C. J., MALIK J.: Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH* (1996), pp. 11–20. 3, 6

[GGSC96] GORTLER S. J., GRZESZCZUK R., SZELISKI R., COHEN M. F.: The lumigraph. *Proc. SIGGRAPH* (1996), 43–54. 3, 7

[Ju04] JU T.: Robust repair of polygonal models. *ACM Trans. Graph. (Proc. SIGGRAPH 2004) 23*, 3 (2004), 888–895. 7

[KH13] KAZHDAN M., HOPPE H.: Screened poisson surface reconstruction. *ACM Trans. Graph. 32*, 3 (2013), article no. 29. 7

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 163–169. 7

[LCC*15] LEE K., CHU D., CUERVO E., KOPF J., DEGTYAREV Y., GRIZAN S., WOLMAN A., FLINN J.: Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. MobiSys* (2015), pp. 151–65. 2, 3, 4, 5, 8, 9

[LH96] LEVOY M., HANRAHAN P.: Light field rendering. *Proc. SIGGRAPH* (1996), 31–42. 3

[LRR*14] LOCHMANN G., REINERT B., RITSCHEL T., MÜLLER S., SEIDEL H.-P.: Real-time reflective and refractive novel-view synthesis. In *Proc. VMV* (2014), pp. 9–16. 3

[Lue97] LUEBKE D.: *A Survey of Polygonal Simplification Algorithms*. Tech. rep., UNC CS TR97-045, 1997. 7

[MMB97] MARK W. R., MCMILLAN L., BISHOP G.: Post-rendering 3D warping. In *Proc. i3D* (1997). 2, 3, 4, 5, 8, 9

[Ocu15] OCULUS VR: Oculus mobile SDK documentation, 2015. 2, 4, 8

[SGHS98] SHADE J., GORTLER S., HE L.-W., SZELISKI R.: Layered depth images. In *Proc. SIGGRAPH* (1998), pp. 231–42. 3, 5

[SH15] SHI S., HSU C.-H.: A survey of interactive remote rendering systems. *ACM Comput. Surv. 47*, 4 (2015), 57:1–57:29. 2

[SKG*12] SINHA S. N., KOPF J., GOESELE M., SCHARSTEIN D., SZELISKI R.: Image-based rendering for scenes with reflections. *ACM Trans. Graph. 31*, 4 (2012). 3

[SVJ15] SACHT L., VOUGA E., JACOBSON A.: Nested cages. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 34*, 6 (2015). 7

[Wol98] WOLBERG G.: Image morphing: a survey. *The visual computer 14*, 8 (1998). 3

[WPS*15] WIDMER S., PAJAK D., SCHULZ A., PULLI K., KAUTZ J., GOESELE M., LUEBKE D.: An adaptive acceleration structure for screen-space ray tracing. In *Proc. HPG* (2015), pp. 67–76. 3

[YN00] YOON I., NEUMANN U.: Web-based remote renderingwith IBRAC (image-based rendering acceleration and compression). In *Comp. Graph. Forum* (2000), vol. 19, pp. 321–30. 3

[YTS*11] YANG L., TSE Y.-C., SANDER P. V., LAWRENCE J., NEHAB D., HOPPE H., WILKINS C. L.: Image-based bidirectional scene reprojection. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 30*, 6 (2011). 3, 5, 9