# Real-Time View Correction for Mobile Devices

Thomas Schöps, Martin R. Oswald, Pablo Speciale, Shuoran Yang, and Marc Pollefeys

**Abstract**— We present a real-time method for rendering novel virtual camera views from given RGB-D (color and depth) data of a different viewpoint. Missing color and depth information due to incomplete input or disocclusions is efficiently inpainted in a temporally consistent way. The inpainting takes the location of strong image gradients into account as likely depth discontinuities. We present our method in the context of a view correction system for mobile devices, and discuss how to obtain a screen-camera calibration and options for acquiring depth input. Our method has use cases in both augmented and virtual reality applications. We demonstrate the speed of our system and the visual quality of its results in multiple experiments in the paper as well as in the supplementary video.

**Index Terms**—View Correction, Depth Image Based Rendering (DIBR), Mobile Devices, Augmented Reality (AR)

## 1 Introduction

Virtual and augmented (or mixed) reality, VR and AR, are commonly believed to become more and more important in the near future. Google's Tango, for example, is an AR project which has enabled mobile devices to localize themselves accurately in their environment and perceive depth. AR applications on mobile devices typically overlay their content over the device's camera image directly, which they display on the screen. As a result, the displayed image does not match the user's perspective on the scene. See Fig. 1 (left) for an example, showing the misalignment between objects in the image and the real world. This detracts from the user experience and may be an obstacle for these applications' success; after all, smartphones only became as commonplace as they are today once they became intuitive to use.

VR headsets, on the other hand, completely block the user's view on the real world. However, it is often convenient to be able to view the real world without taking the headset off and putting it back on afterwards. The HTC Vive, for example, contains a camera for this purpose. In this scenario, the same issue arises: the camera's viewpoint does not match those of the user's eyes, leading to discomfort when displaying the camera image directly.

In this paper, we propose a real-time capable method to correct for such viewpoint changes, *i.e.*, for rendering the real-world scene from an arbitrary observer's viewpoint. We only use the image and depth information from the camera's point of view as input. Since it is often impossible to reconstruct scenes completely, this requires the plausible filling of unobserved regions. Fig. 1 (right) shows an example result of our method. Using this method could enable a "virtual frame" viewing mode for AR applications on mobile devices, and displaying correct views in AR applications on VR devices with cameras.

**Contributions.** We make the following contributions in this paper: i) We present a real-time capable pipeline for view correction in order to render a scene from arbitrary viewpoints while filling in missing information in a temporally consistent way. The pipeline's main component is a fast inpainting approach, proposed as an extension of [27]. ii) We describe methods for depth acquisition and system calibration, and demonstrate our system in a number of experiments.

## 2 Related Work

There is a large number of works on viewpoint interpolation, image synthesis, depth image based rendering, and depth inpainting. After discussing some early pioneering works, we focus on papers with either similar methods or a similar application setting to ours.

Two early works on viewpoint interpolation and synthesis are [9], as well as the pre- and post warping scheme by Seitz and Dyer [29].

Without view correction Proposed view correction

Fig. 1. Example application of the proposed method. The original camera view (left) is transformed to a different viewpoint (right) in order to match the user's perspective on the scene.

Zitnick *et al*. [34] use superpixel segmentation to make depth jumps align with edges in the input image during stereo depth estimation. Further processing steps like disparity smoothing and Bayesian matting finally lead to high quality view interpolations. Lipski *et al*. [22] combine view interpolation with temporal interpolation for multi-view video sequences. Linz *et al*. [21] cast the interpolation as a labeling problem to decide on a per-pixel basis which input image is sampled to generate the synthesized view.

All these works expect multi-view image input and hence focus on view interpolation rather than extrapolation. Moreover, these methods are not suitable for real-time processing. Although some of them allow for interactive exploration of scenes by view interpolation, they require a considerable amount of preprocessing.

Other works expect depth input and focus on inpainting depth maps. In [19] a tensor voting approach is used to perform depth inpainting. [32] propose a hole filling approach for multi-view setups that especially accounts for inter-view consistency. Similar to [34], Buyssens *et al*. [5] use superpixels in order to use image edge information to guide the depth inpainting. Building upon [10], [6] present a method for depth inpainting, and an exemplar-based color inpainting making use of depth information. Luo *et al*. [24] focus on video processing and use temporal information to fill in disoccluded areas. The results are impressive, but numerous processing steps like background modeling, foreground segmentation, and motion compensation make the approach computationally expensive.

Impressive results have been achieved in real-time diminished reality using inpainting by Kawai *et al*. [17]. They assume a static background geometry composed of few local planes. This allows them to inpaint the planes once at the beginning such that at frame-rate only the results need to be rendered.

Several previous works have explored ways of performing view correction on mobile devices. In order to achieve fast runtimes and to perform dense view correction based on sparse SLAM keypoints, Tomioka *et al*. [30] optimize for a homography to approximate the correct transformation of the keypoints. As an advantage, the homography transformation is consistent for the whole image and thus does not

- *Thomas Schöps, Martin Oswald, Pablo Speciale, and Marc Pollefeys are with the Department of Computer Science, ETH Zürich. Marc Pollefeys is also with Microsoft. E-mail: firstname.lastname@inf.ethz.ch.*
- *Shuoran Yang is with Google. E-mail: shuorany@google.com.*

*Manuscript received 15 Mar. 2017; accepted 24 July 2017.*
*Date of publication 7 Aug. 2017; date of current version 29 Sept. 2017.*
*Recommended for acceptance by W. Broll, H. Regenbrecht, and J.E. Swan II.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TVCG.2017.2734379*

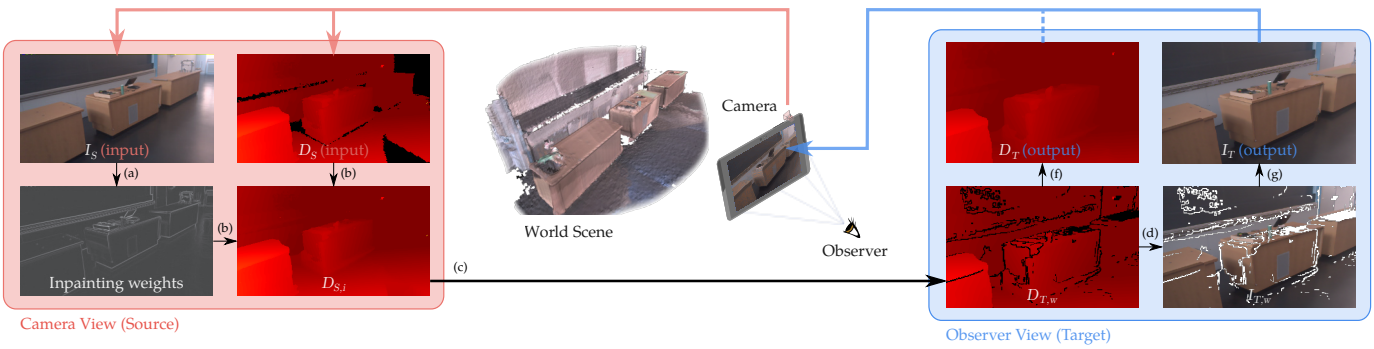


Fig. 2. **Pipeline overview**. The key idea of our approach is to use edges in the input image (source view) to intelligently interpolate depth values since image edges often coincide with depth discontinuities. The subsequent warping to the target view then requires additional interpolation (inpainting). **The individual steps are**: The mobile device records the RGB image $I_S$ and depth map $D_S$ as input. **(a)** Edges are extracted from $I_S$ to be used as inpainting weights. **(b)** $D_S$ is inpainted using these weights to get $D_{S,i}$. **(c)** $D_{S,i}$ is warped to the virtual target viewpoint corresponding to the view of the observer, resulting in $D_{T,w}$. **(d)** $I_S$ is projected onto $D_{T,w}$ to compute the initial warped image $I_{S,w}$. **(e)** The results of the previous iteration of the pipeline are reprojected into the target view to achieve temporal consistency (not shown in figure). **(f)** Inpainting is applied to $D_{T,w}$ to complete it. **(g)** Inpainting is applied to $I_{T,w}$ to complete it.

introduce local distortions. However, in non-planar scenes a homography is not sufficient to correct the view. Baričević *et al*. [2, 3] present a view correction system using a stereo camera. They developed a fast semi-dense stereo approach and use it together with a gradient-based inpainting method for image-based rendering with geometrically correct warping. However, despite using a workstation with one [2] or two [3] GPUs, the system runs with 16 FPS only on average on this hardware (with the slowest component being the image-based rendering), making it too expensive for current mobile devices.

In sum, a large amount of advanced inpainting techniques have been proposed. However, only few of them are real-time capable and provide a good trade-off between result quality and runtime. For instance, [1] achieves real-time performance in depth image based rendering for 3D-TV. However, in this setting the target view is always close to the input view and only very small disocclusions need to be filled, which in this case is done by a very simple heuristic that does not perform well on larger inpainting regions. With the proposed approach we try to find a good trade-off between the quality of the synthesized images and the required processing time, allowing for real-time application on modern mobile devices.

## 3 View Correction

The input to our method is a single image $I_S$ and a potentially incomplete depth map $D_S$, both recorded at the same *source* camera view $S$. The goal is to generate a complete and plausible virtual image $I_T$, as well as a corresponding complete depth map $D_T$, as they would be seen from the given *target* camera view $T$. An accurate and complete output depth map $D_T$ is crucial for augmented reality applications to be able to account for occlusions of virtual objects by the real world environment. Our processing pipeline consists of the following steps, which are also visualized in Fig. 2:

(a) **Compute inpainting weights** from image $I_S$.
(b) **Inpaint** $D_S$ to get a complete depth map $D_{S,i}$ in $S$.
(c) **Reproject** $D_{S,i}$ to $T$ to get a warped depth map $D_{T,w}$ with disocclusions, and warped inpainting weights.
(d) **Reproject** $I_S$ to $T$ using $D_{T,w}$ to get an incomplete warped image $I_{T,w}$.
(e) **Reproject** $D_T$ **and** $I_T$ from the previous iteration of the pipeline to $D_{T,w}$ and $I_{T,w}$ to achieve temporal consistency.
(f) **Inpaint** $D_{T,w}$ to get the complete depth map $D_T$.
(g) **Inpaint** $I_{S,w}$ to get the complete image $I_T$.

Since inpainting is an important component which is used in several stages of our pipeline, we next present the fast inpainting method used in our algorithm before we describe each of the pipeline steps in detail.

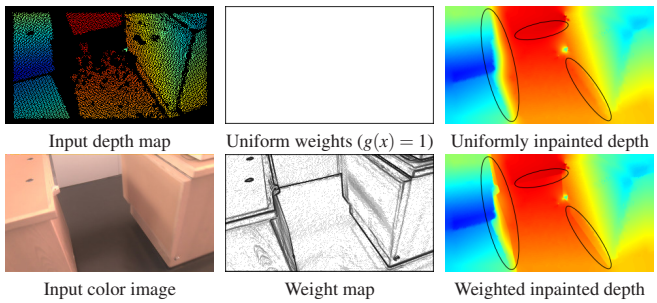


Fig. 3. Uniform vs. weighted depth map inpainting. The edge information helps to align depth discontinuities with object boundaries if they have a different color than their surroundings.

### 3.1 Fast Inpainting

Inpainting is required to plausibly fill in missing information in areas which are occluded or cannot be measured. This both applies to color information and to depth information, which we use for image reprojection and also provide as an output of the method, such that it can be used for occlusion testing.

A commonly used inpainting technique is Total Variation (TV) inpainting [8]. This method is able to take 2D cues about possible locations of depth discontinuities into account that can considerably improve the quality of the results (see Fig. 3). The idea behind this method is to run a diffusion process in order to fill in missing values, in which the existing values are used as boundary conditions. Moreover, with a simple weighting scheme, depth discontinuities are favored at locations with large gradient in the input image, because depth changes often occur along color changes in the image.

This can be formulated as an optimization problem, minimizing the total variation of the result image: Given a real valued image $I : \Omega \to \mathbb{R}$ with image domain $\Omega$, the goal is to fill in a region $\Gamma \subset \Omega$ in which the image values are not known. The inpainted image $J^* : \Omega \to \mathbb{R}$ can be computed as the minimizer of the following energy:

$$J^* = \arg\min_J \int_\Omega g(\mathbf{x}) \, \|\nabla J\|_\varepsilon \, d\mathbf{x} \qquad \text{subject to} \quad J = I\big|_{\Omega\setminus\Gamma} \tag{1}$$

where $g : \Omega \to [0, 1]$ is a weight function which can either favor or penalize jumps of $J$ at certain locations $\mathbf{x}$ [4]. Here, $\|\cdot\|_\varepsilon$ denotes the Huber-TV norm [7] which is defined as a hybrid of quadratic regularization to smooth noisy values smaller than $\varepsilon \in \mathbb{R}_{\geq 0}$ and an $L1$ regularization to be robust to outliers:

$$\|\mathbf{x}\|_\varepsilon := \begin{cases} \frac{1}{2\varepsilon}|x|_2^2 & \text{if } |\mathbf{x}| \le \varepsilon \\ |\mathbf{x}| - \frac{\varepsilon}{2} & \text{otherwise} \end{cases} \quad (2)$$

The limiting case $\varepsilon = 0$ corresponds to classical total variation regularization, while $\varepsilon \to \infty$ corresponds to quadratic regularization.

This optimization can be solved with the preconditioned first-order primal-dual algorithm from [28]. However, even with an optimized CUDA-based implementation, in our experiments the achieved runtime performance was not sufficient for real-time performance on mobile devices if the inpainting regions were large. Therefore, we chose to employ a simpler method which approximates this TV inpainting.

**Convolution-based Inpainting.** In order to obtain an approximate solution to (1) which can be computed much faster, we propose an extension of [27]. This method leverages the fact that energy (1) with purely quadratic regularization ($\varepsilon \to \infty$) corresponds to solving a linear heat equation whose solution can be computed much more efficiently via convolution [13, p.47].

Since our goal is to maintain discontinuities with the help of the weight function $g(\cdot)$, we modify the convolution kernel by pixel-wise multiplications with the underlying diffusion weights $g$. Hence, we compute an approximate solution to (1) via the following weighted convolution:

$$J(\mathbf{x}) \approx (K * gI)(x,y) = \int_\Omega \frac{K(a,b)\, g(x-a,y-b)}{Z}\, I(x-a,y-b)\, da\, db \quad (3)$$

where $Z$ ensures proper normalization of the weighted kernel and $K$ is the discretized Gaussian-inspired $3 \times 3$ kernel proposed in [27] as

$$K = \begin{bmatrix} a & b & a \\ b & 0 & b \\ a & b & a \end{bmatrix} \qquad \text{with } a = 0.073235,\ b = 0.176765\ . \quad (4)$$

Only pixels in the inpainting domain $\Gamma$ are recomputed by using the image values of the full domain $\Omega$. The convolution operation is repeated until the maximum change of a pixel's value falls below a termination threshold or a maximum number of iterations is reached. The weight directly influences the diffusion of information from a given pixel; in the extreme case, with a weight of zero, a pixel will have no influence on the values of its neighbors.

We further modify the algorithm to give zero weight in the convolution to pixels which are pixels to be inpainted and have not received any update from a boundary pixel yet. The maximum number of iterations is set to the maximum of the width and height of the image to be inpainted. This ensures that the algorithm is able to propagate information over the maximum distance (from one corner of the image to the other), while no traces of the initialization will be left over in the result. It is important to note that the weighted convolution approach approximates the quadratic penalization and the solution is different from the minimizer of problem (1). A comparison is shown later in Fig. 8. Nevertheless, the introduced weighting preserves depth discontinuities along image edges which is an important feature for the visual output quality. Due to its simplicity, this algorithm can be implemented to run very efficiently on a GPU and is therefore well suited for our purposes.

### 3.2 Processing Pipeline

In this section, we explain the steps of our approach in detail.

**(a), (b): Weighted source frame depth inpainting.** For each pixel in the input image $I_S$, we first compute the inpainting weights $g(\cdot)$ from the corresponding gradient magnitudes of the input image. For this processing step, the weight function $g : \Omega \to \mathbb{R}_{\ge 0}$ is defined as

$$g(\mathbf{x}) = \left(1 + \alpha \|\nabla I_S(\mathbf{x})\|_2\right)^{-1}\ , \quad (5)$$

with $\alpha = \frac{50}{255}$. This parameter represents the belief about the likelihood of depth discontinuities depending on the image gradient magnitude. The best value is scene-dependent, and we chose the value empirically. As the next step, we perform an inpainting step on the input depth image $D_S$ using these weights. The effect of the weighting function $g$ in this step is demonstrated in Fig. 3.

On the first sight, this first inpainting step may seem unnecessary since both the depth map and image could also be inpainted after reprojecting them to the target view $T$ only, skipping this first inpainting iteration. The reason for doing a first inpainting step in the input camera view is that the the edges in the image from this view usually give strong hints about the possible locations of depth discontinuities. This information cannot be fully warped to another view, since depth estimates may not be known for all pixels. Especially for very sparse input depth maps this information is valuable for computing accurate depth maps of the scene.

Note that in the scenario of view correction for a VR headset which has to provide one output image per eye, these initial processing steps which operate in the source frame can be done only once for both output images. However, if two camera views are available which better match the positions of the eyes, separately running the complete pipeline for each camera can lead to higher quality since less disocclusions will occur when warping the depth maps to the eyes' views.

**(c), (d): Depth and image reprojection.** The previous step of the algorithm provides a dense depth map $D_{S,i}$ for the source frame. This step uses this intermediate result to compute an initial partial depth map $D_{T,w}$ and color image $I_{T,w}$ for the target view.

First, we generate a triangle mesh from the inpainted depth map $D_{S,i}$ in order to have a continuous representation of the surface. This enables to render it in the target view without potentially leaving holes. We thereby proceed as follows: for each pixel from the inpainted depth map, we generate a vertex at the unprojected 3D point corresponding to the pixel. For each square of four adjacent pixels in the inpainted depth map, we determine whether there is a depth jump at this location in the depth image by testing whether the depth difference between any two adjacent vertices among them is larger than 7 cm. This parameter depends on the depth range and noise magnitude of the input depth maps and was chosen empirically to give good results for our scenarios. If this test recognizes a depth jump, we do not create any triangles for these pixels. Otherwise, we create two triangles to connect the pixels' vertices, which in the image projection cover the square spanned up by the pixels. The final triangle mesh is then rendered as a depth map in the target frame to produce an initial partial target depth map $D_{T,w}$.

We use this intermediate result to also reproject the input RGB image to the target view. For each pixel having a depth value in $D_{T,w}$, we reproject the corresponding 3D point onto the input image $I_S$ and look up the color value at the projected pixel position. This value is entered at the corresponding pixel into the initial target frame color image $I_{T,w}$.

**(e): Reprojection of previous result.** Treating every input frame independently will lead to flickering artifacts since the inpainting results are very sensitive to the pixel values of the inpainting region borders. Those borders are likely to change frequently due to camera movements and image noise, for example. It is therefore important to ensure temporally consistent inpainting to avoid creating distracting artifacts.

Since our method is intended for augmented reality applications, it is safe to assume that the relative pose of the previous processed image to the current image is known in the system. We therefore take the result of the previous algorithm iteration, *i.e.*, $D_T$ and $I_T$, and reproject it into $D_{T,w}$ and $I_{T,w}$ of the current algorithm iteration by simple point cloud rendering. In doing so, we discard all points which project to pixels in $D_{T,w}$ and $I_{T,w}$ that were already initialized in the previous algorithm step. This gives new measurements precedence over propagated old results.

If the previous result of the algorithm was correct, this algorithm step will also geometrically correctly reproject it to the new image. However, in practice the inpainting will deviate from the real world since it cannot always guess correctly, and in addition the scene geometry and colors may change, for example due to specular reflections. Naively reprojecting the previous result may keep wrong values in the image, preventing them from being replaced by improved results later. We therefore add two modifications to this procedure: First, we apply slight blurring to $I_T$ during this reprojection step. This avoids the preservation of distracting high-frequency artifacts. Second, we only reproject every $k$-th pixel in $D_T$ and $I_T$. This allows the inpainting re-

sults to get updated with new information gradually. The parameter $k$ trades off temporal consistency vs. the ability to correct errors using new data. According to our experiments the choice of this parameter is not critical to the algorithm; we used $k = 4$ for all experiments.

We remark that for this algorithm step either the exposure settings of the camera should be fixed, or potential changes should be accounted for if known, to prevent mixing differently exposed images.

**(f), (g): Weighted target frame depth and color inpainting.** Due to disocclusions or unobserved regions, $D_{T,w}$ and $I_{T,w}$ can still contain holes. Thus, we perform a second inpainting step to fill such holes, which is now performed in the target frame.

Disocclusions will typically uncover surfaces in the background. We thus want to prefer inpainting values of the background in these places instead of diffusing the values of the foreground. To facilitate this we again use weighted inpainting. The inpainting weights are calculated jointly with the warped depth map in step (c) as follows: For each vertex of the triangle mesh created from the inpainted source depth map $D_{S,i}$, we determine whether any of the vertices in its 4-neighborhood defined on the pixel grid has a larger depth than that of the center pixel by at least the depth jump threshold. If so, we classify this vertex as a foreground boundary vertex. Simultaneously to the depth map rendering of the reprojection step (c), we also render an intensity image of the same mesh where the intensity of each vertex is set to 255 if it is a foreground boundary vertex, and zero otherwise. The resulting rendered pixel intensities are binarized by thresholding (indicating whether there is a depth jump at a pixel or not), and remapped to the minimum and maximum inpainting weight given by Eq. 5, respectively, to define $g(x)$ for these inpainting steps. See Fig. 10 for examples of the weights computed by this method.

## 4 System Integration

In this section, we show how our view correction method defined in Sec. 3 can be integrated into a complete system for mobile devices.

### 4.1 Depth Map Acquisition

Our approach requires depth maps $D_S$ as input in order to reproject images from the source view to the target view. In this work, we evaluate two different options for obtaining depth maps. The first is directly using images returned by a depth sensor (*c.f.* Sec. 4.1.2). The second is creating a 3D reconstruction in the background while the application is running, and rendering it as a depth map when required (*c.f.* Sec. 4.1.3). In both cases, it is necessary to have estimates of the camera poses, either for relating the last obtained depth map to the current camera pose or to fuse depth maps into the reconstruction. Therefore, first we shortly describe the camera pose tracking approach we use.

#### 4.1.1 Camera Pose Estimation

In order to determine the camera poses of the mobile device in real-time, we use the motion tracking ability of Google Tango (based on [15]). The use of accelerometer measurements in this approach makes the absolute scale of the trajectory observable. Although the implementation we use is specific to Tango devices, we note that very similar results for visual-inertial odometry in terms of odometry drift have been reported in [31], which is able to run on standard mobile devices.

#### 4.1.2 Depth from Sensor

Our approach is able to directly use the depth maps obtained from a depth sensor, even if they are sparse (*c.f.* Fig. 5). Using this direct type of depth map acquisition, *i.e.*, without temporal integration, has the advantage that no potentially outdated information is used. Furthermore, it presents the limiting case in the sense that directly after startup, or after moving into a new area, only one depth map is available.

#### 4.1.3 Depth from 3D Reconstruction

As an alternative to directly using depth sensor readings, we perform 3D reconstruction to accumulate the raw sensor estimates to gain more complete and accurate information. We use the 3D reconstruction implementation of Google Tango [18] for this task. It is based on Truncated Signed Distance Function (TSDF) fusion [11] in a volume with voxel hashing [26]. The isosurface of the TSDF corresponding to the real-world surface is extracted using Marching Cubes [23]. Having a reconstructed mesh and the camera pose of a new source image $I_S$, a matching depth map $D_S$ can then be generated by rendering the mesh.

While [18] has been implemented for Tango devices, other fast implementations for mobile devices have been proposed, for example [16]. Note that our view correction method only requires the rendered depth map at the current camera pose. Thus it is easily possible to use other 3D reconstruction methods with our approach.

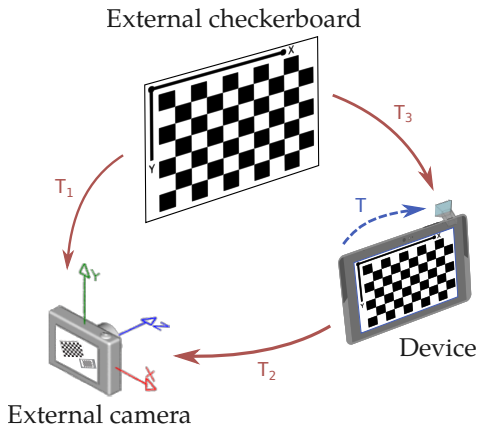


Fig. 4. Red arrows represent the extrinsics from the checkerboard to the camera coordinate systems. The transformation $T$, the screen-camera calibration, can be computed from those extrinsics.

### 4.2 Observer Position

The proper placement of the virtual target view is crucial for accurate view correction. This first requires to calibrate the relation between the device's camera (which defines the source frame for $D_S$ and $I_S$) and the device's screen (which is observed by the user, whose eyes eventually define the target frame for $D_T$ and $I_T$). In this section, we first describe the screen calibration method we used, and then give options for determining the user's eye positions.

#### 4.2.1 Screen-Camera Calibration

We estimated the extrinsic calibration between the screen and the back camera of a mobile device as in [2]. This is illustrated in Fig. 4.

The method requires two checkerboards: an external one representing the world coordinate system, and a separate checkerboard displayed on the device's screen. The device's back camera captures pictures of the external checkerboard. At the same time, an external camera is used to capture pictures which show both checkerboards simultaneously: the external one, and the one on the screen. Following usual calibration procedures [14, 33], for both cameras the relative pose to each visible checkerboard can be determined from those pictures, denoted $T_1$, $T_2$, and $T_3$ in Fig. 4. Using this notation, the screen-camera calibration $T$ can be expressed as:

$$T = T_3 \cdot T_1^{-1} \cdot T_2$$

For determining the metric screen corner positions from this calibration, we use the squares of the displayed checkerboard. Its square size is known in pixels, and therefore also in metric units on the display from the display's pixels per inch (PPI) specification. Similar screen-camera calibration methods have been presented in [12, 20].

#### 4.2.2 Eye tracking

The user's eye positions must be known to determine the target frame for the view correction. For the scenario of a VR headset, the position depends on the geometry between lenses, screen and the user's eyes. However, it can be calibrated once and remains largely static afterward. For the purposes of this paper, we assume that a static position is known in this case.

For the scenario of a device whose screen moves relative to the user's eyes, the user's eyes need to be tracked. Unfortunately, our test

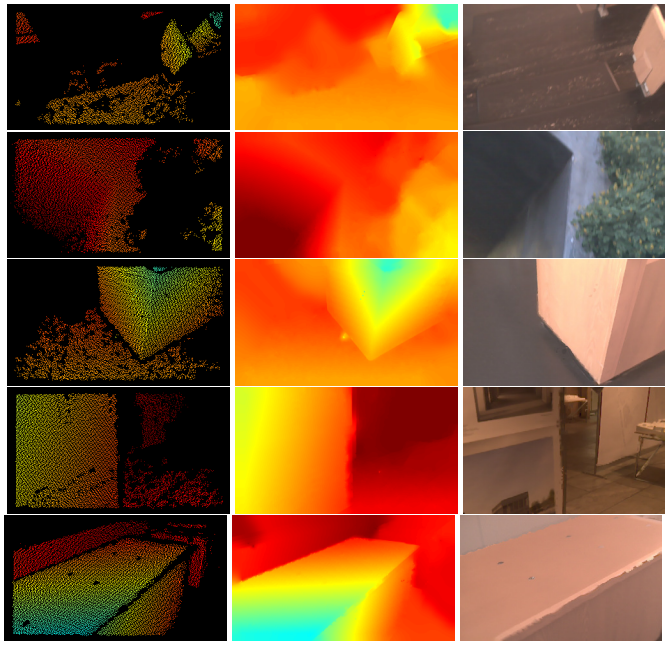

Fig. 5. Examples of inpainted depth maps (middle) and corrected views (right) with single depth maps (left) as input.

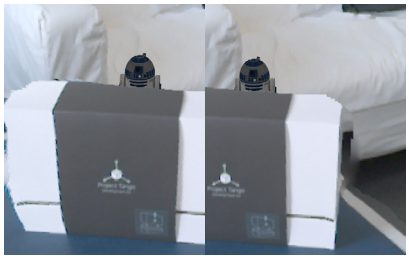

Fig. 6. Example stereo image pair generated by our method. For this figure, we ran the first part of the pipeline (steps (a) and (b) which operate in the source frame) only once during the computation of both output views. A virtual model of R2-D2 is rendered into the scene, which gets occluded by the real box.

hardware (like many other current mobile devices) does not support the concurrent use of the front- and back-facing cameras, which made it impossible to run an eye-tracking algorithm on images of the front camera. In order to simulate this scenario and for evaluation purposes, we therefore used a second mobile device which observes the first one. Both devices localize themselves against a pre-built map of the environment using Google Tango's odometry [15] and re-localization [25] capabilities. The observing device then wirelessly sends its camera position to the first device in real-time. This way, we can choose the observing camera position as the virtual target viewpoint for the first device that performs the view correction.

## 5 Results

We evaluated our approach on two hardware platforms: 1) a PC with an Intel Core i7-4770K processor and an Nvidia GeForce 780 GTX graphics card, and 2) a Google Tango Development Kit Tablet with an Nvidia Tegra K1 chip, which is equipped with a structured light depth sensor and a 4MP color camera. All datasets have been recorded with the Tango tablet, whose depth camera records depth maps with ca. 5 Hz (leading to regular fluctuations in computation time when moving the camera, as more or less pixels need to be inpainted depending on the availability of recent depth data).

We use the following image resolutions for all experiments:

Fig. 7. Examples of view correction results from our system, as seen by the observer. Top: Fixed relative observer pose. In the left image, an AR object is rendered into the view. Bottom: Dynamic observer pose with two tablets (*c.f.* Sec. 4.2.2). We show these experiments in action in the supplementary video.

- Input depth map size: $320 \times 180$ pixels.
- Input color image size: $1280 \times 720$ pixels.
- Target image size: $480 \times 300$ pixels (for monocular output).

Ideally, the target image size would be equal to the screen resolution. However, on the one hand the use of larger resolutions increases the runtimes. On the other hand, the resolution of the depth sensor used as input is very low, and upscaling the depth maps has its limits, leading to a limited gain in quality. In the choice of this resolution, we therefore prioritized interactive runtimes on the tablet device. If preserving the highest possible color detail was critical, we think that a fast possibility would be to perform the direct reprojection (pipeline step (d)) to a high resolution image, while keeping the inpainting at a low resolution and upsampling its result to the high resolution.

All steps of our algorithm have been implemented with CUDA on the GPU. Our implementations are optimized: we make use of shared memory to run multiple convolution iterations for inpainting in a single CUDA kernel call without having to write data back to global memory, and try to avoid thread divergence.

In order to compare the inpainting quality and the processing speed of the TV inpainting approach with the weighted convolution approach, we show two inpainted example frames in Fig. 8 (left). The results are comparable; depending on the scene, sometimes one approach gives slightly better results than the other and vice versa. Most importantly, both approaches preserve dominant edges in the depth maps which is crucial to minimize artifacts when the depth map is rendered from the target view point.

### 5.1 Qualitative Evaluation

We show qualitative results for all steps in our pipeline (*c.f.* Fig. 2, 10) using the weighted convolution inpainting. In particular, in Fig. 5 we show some examples of using sparse depth maps as input to the algorithm. Given sufficient depth information nearby, our method is able to fill in missing regions and correctly handle them. Fig. 6 shows by example how virtual objects can get occluded by real objects in an AR use case for the scenario of a VR headset, using the inpainted target frame depth map for occlusion testing. Fig. 7 shows additional examples of view correction results, as seen by the observer. Furthermore, we strongly recommend viewing the supplementary video to this work since the method's performance is best viewed in motion. In particular, the supplementary video shows a comparison between using the temporal consistency step and inpainting every frame independently, which is very hard to show on static images.

### 5.2 Quantitative Evaluation

For a quantitative error analysis of our results, we recorded an RGB-D dataset and for each frame set the target camera frame for view correction to the camera pose of its previous frame, which we use as the

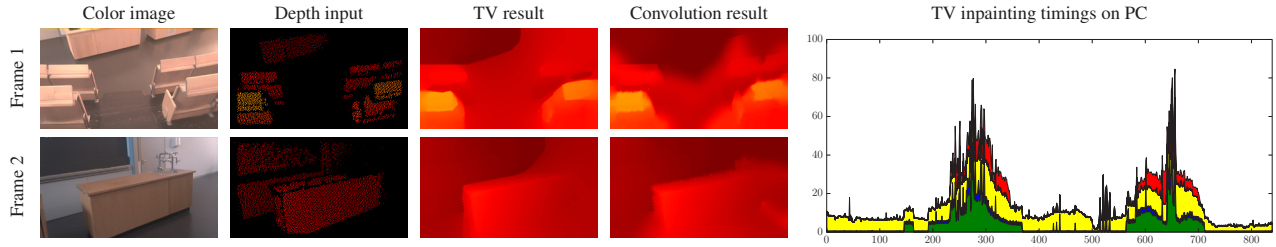


Fig. 8. Comparison between Huber-TV inpainting and our weighted convolution inpainting approach for two example frames. We used $\alpha = 1$ for computing the weights used by the TV inpainting (*c.f.* Eq. 5). The performance plot corresponds to the right plot in Fig. 9. We developed optimized implementations of both algorithms; convolution-based inpainting is roughly an order of magnitude faster.

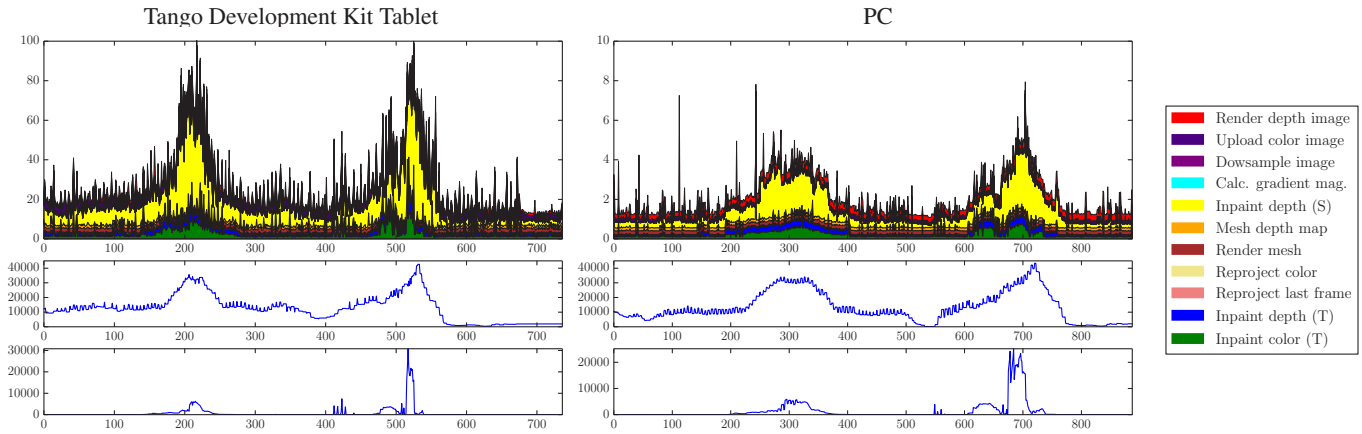


Fig. 9. First row: Example timings in milliseconds (y axis) for a dataset over a series of frames (x axis) for the different steps of the algorithm, on a Project Tango Development Kit Tablet (left) and on a PC (right). The tablet drops some frames and thus frame numbers do not correspond. Most steps are very fast compared to the inpainting and therefore hardly appear in the plot. Since the inpainting speed strongly depends on the size of the inpainting regions, we show the number of inpainted pixels in the source frame in the second row, and the number of inpainted pixels in the target frame in the bottom row. This plot is made for the case of using a mesh for depth input. The first step therefore refers to rendering the mesh. The second and third steps refer to transferring (uploading) $I_S$ to the GPU and scaling it to the depth map's resolution.

Table 1. Average time in milliseconds for individual algorithm steps, and for a complete frame, corresponding to the graphs shown in Fig. 9.

| Source frame steps | Tablet | PC | Target frame steps | Tablet | PC |
|---|---|---|---|---|---|
| Render depth image | 1.5 | 0.4 | Render mesh | 2.0 | 0.2 |
| Upload color image | 3.0 | 0.2 | Reproject color | 0.6 | <0.1 |
| Dowsample image | 1.0 | <0.1 | Reproject last frame | 0.4 | <0.1 |
| Calc. gradient mag. | 0.3 | <0.1 | Inpaint depth (T) | 2.1 | 0.2 |
| Inpaint depth (S) | 11.8 | 0.8 | Inpaint color (T) | 2.6 | 0.2 |
| Mesh depth map | 1.5 | 0.2 | | | |
| **Complete frame** | | | | **26.7** | **2.2** |

Table 2. Average and 99% quantile of frame time in milliseconds on the tablet depending on the target image size, for the dataset used in Fig. 9. This table shows averages out of 3 runs.

| Resol. | Avg | 99% | Resol. | Avg | 99% | Resol. | Avg | 99% |
|---|---|---|---|---|---|---|---|---|
| 480×300 | 27.9 | 100.3 | 960×600 | 37.5 | 179.0 | 1600×1000 | 56.2 | 374.2 |
| 640×400 | 30.7 | 117.8 | 1280×800 | 45.2 | 206.2 | 1920×1200 | 82.4 | 414.7 |

ground truth. Since we have the corresponding input image, we know how the warped target image should look like. Hence, we measure the average pixel-wise error between the view correction result and the ground truth image over an entire sequence. We did not enforce temporal consistency (step (e) in the pipeline) for this experiment. The evaluation has been limited to pixels which are farther away than 40 pixels from the image border. This is to avoid comparing border regions where there is very little or no adjacent color and depth information which can be used for inpainting, such that it would not be fair to evaluate these regions. Pixel values are in $[0, 255]$. The average per-channel absolute difference per pixel is 5.7 for using the depth camera directly, and 5.5 for using a reconstructed mesh. If performing all inpainting steps with uniform weights instead of using the proposed weighting, the error roughly stays the same for the case of using mesh input, while it slightly increases (by ca. 0.07) to 5.8 for the case of using depth maps as input, since more inpainting is necessary in this case. Note that the difference in error averaged over all pixels is small since many image regions are homogeneous.

Fig. 10 shows the difference images (last row) for two example frames (left and right double-columns) from this sequence. Further, rows 1-7 show the results of intermediate steps of our pipeline while comparing the use of a single depth map as input against 3D reconstruction of multiple depth maps (in different columns).

### 5.3 Runtime

We determined the runtimes of our algorithm on the aforementioned hardware by running it on an example sequence. The sequence both contains parts for which dense depth information is available, and parts where larger regions need to be inpainted. On the PC, our method took a maximum of about 8 ms for one video frame, with significantly lower runtimes on average, mainly depending on how fast the inpainting steps converge. On the tablet, the method took a maximum of about 100 ms for one frame, however it often took less than 30 ms if sufficient depth data from the sensor or the previous frame was available. Fig. 9 shows a plot of the time taken by different algorithm components at each frame of the sequence, as well as the corresponding amount of pixels to be inpainted. Tab. 1 lists the average time taken by each algorithm step for this sequence and Tab. 2 compares the frame times on the tablet for different target image resolutions.

We compared the runtimes of the weighted convolution inpainting

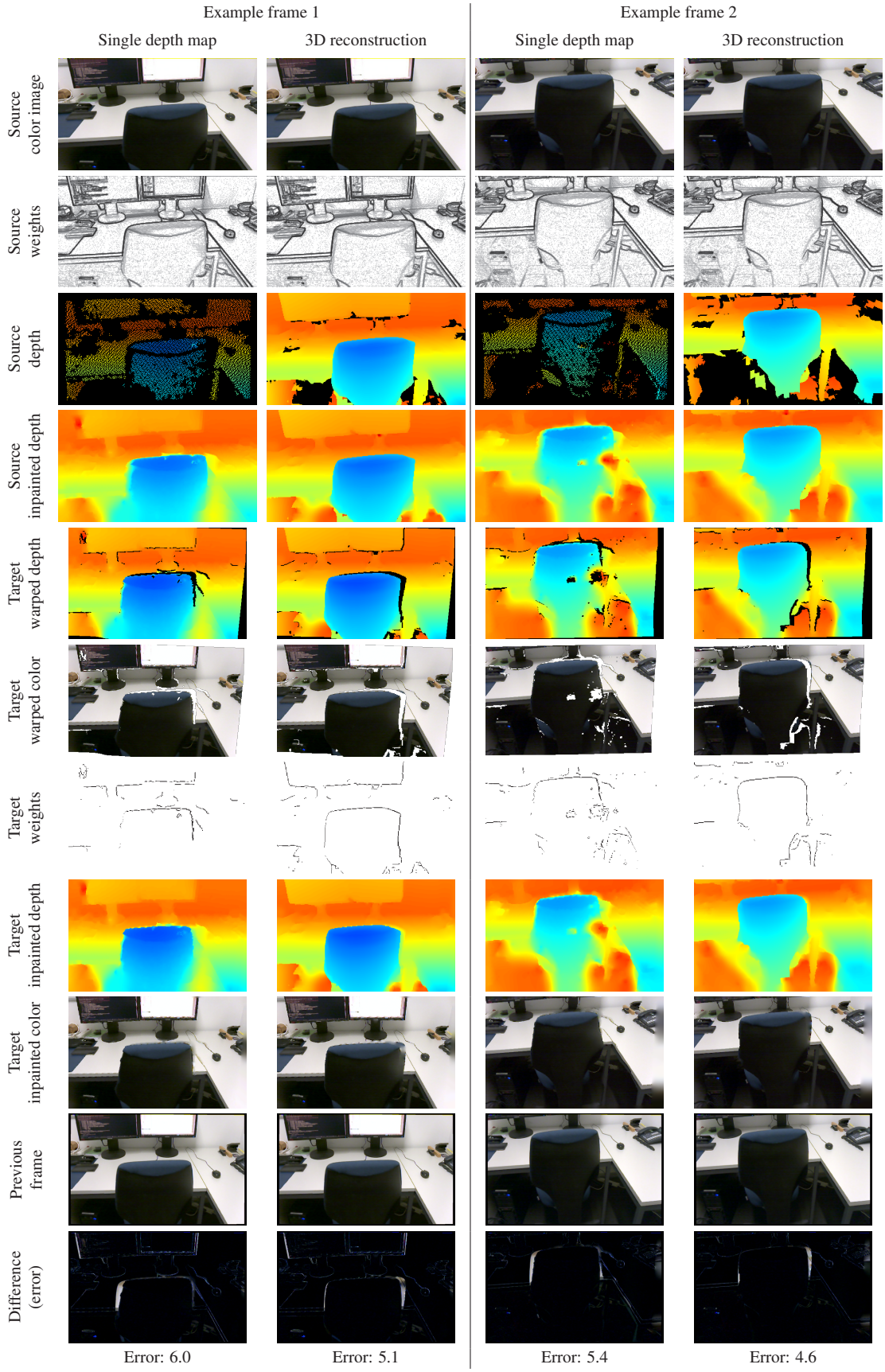


Fig. 10. Illustration of our pipeline for two frames of a sequence. For each frame, we evaluate both using a single depth map (*c.f.* Sec. 4.1.2), and using a reconstructed mesh (*c.f.* Sec. 4.1.3) as input. In addition, a quantitative evaluation has been performed for each frame in the sequence using its previous frame as target, by comparing the view-corrected result to it. The per-channel color error is given at the bottom.

to that of TV based inpainting. On average, the processing time of the TV inpainting is an order of magnitude slower. Fig. 8 (right) shows the processing times of our pipeline with TV-inpainting on the same sequence as used for the weighted convolution benchmarking in Fig. 9.

## 6 Discussion

Our test setup has several technical limitations. First, the front camera of the tablet device cannot be used simultaneously to the back camera, making it impossible to perform eye tracking on the same off-the-shelf device. Second, the depth camera on the tablet device uses a low resolution and operates at 5 Hz only. Therefore, we used a relatively low output resolution, and the hardware is not suitable for observing fast-moving objects. Our system inherits the general limitations of active depth cameras: *e.g.*, limited range, inability to work in sunlight, inability to measure certain materials. Some of these could be removed by using passive stereo instead as in [2, 3], or by using it in addition to the active depth camera. In particular, we think that passive stereo may be very useful to measure edges, which are often not measured well by active cameras. Moreover, the color camera of the tablet has a relatively small field of view. The user's viewpoint therefore must not be too close to the screen, otherwise the target view might extend beyond the input camera's field of view. This could be improved by using a fisheye camera.

There are also some conceptual limitations to the algorithm. Since the inpainting algorithm only diffuses colors which creates areas of relatively constant color, inpainted regions will be over-smooth and appear less plausible in strongly textured scenes. However, higher-quality inpainting methods would be computationally expensive. In general, view-dependent effects such as specular highlights are not accounted for by the algorithm. Moreover, for depth input via 3D reconstruction we observed slight misalignments of disocclusion boundaries due to small errors in the reconstruction or the odometry. Further work could tackle these issues to improve the quality of the results.

## 7 Conclusion

We presented an efficient approach for rendering novel synthesized views from potentially sparse depth and image data. The core algorithm of our processing pipeline is a diffusion-based inpainting algorithm that favors depth discontinuities at possible object edges, which are usually indicated by color or brightness changes in the input image. We ensure that the inpainting returns temporally consistent results, which is important to avoid flickering artifacts. We further discussed a method for the screen-camera calibration of the setup. Our experiments demonstrate that our method can yield pleasing results at real-time frame rates on current mobile devices.

## Acknowledgments

Thomas Schöps was supported by a Qualcomm PhD Fellowship until June 2016, and by a Google PhD Fellowship starting from July 2016. Pablo Speciale and Martin R. Oswald have received funding from the European Union Horizon 2020 research and innovation programme, under grant agreements No. 637221.

## References

[1] R. G. D. A. Azevedo, F. Ismério, A. B. Raposo, and L. F. G. Soares. Real-time depth-image-based rendering for 3DTV using OpenCL. In *ISVC*, pages 97–106, 2014.

[2] D. Baričević, T. Höllerer, P. Sen, and M. Turk. User-perspective augmented reality magic lens from gradients. In *VRST*, 2014.

[3] D. Baričević, T. Höllerer, P. Sen, and M. Turk. User-perspective AR magic lens from gradient-based IBR and semi-dense stereo. *TVCG*, 23(7):1838–1851, 2017.

[4] X. Bresson, S. Esedoḡlu, P. Vandergheynst, J.-P. Thiran, and S. Osher. Fast global minimization of the active contour/snake model. *JMIV*, 28(2):151–167, 2007.

[5] P. Buyssens, M. Daisy, D. Tschumperlé, and O. Lézoray. Superpixel-based depth map inpainting for RGB-D view synthesis. In *ICIP*, pages 4332–4336, 2015.

[6] P. Buyssens, O. Le Meur, M. Daisy, D. Tschumperlé, and O. Lézoray. Depth-guided disocclusion inpainting of synthesized RGB-D images. *TIP*, 26(2):525–538, 2017.

[7] A. Chambolle and T. Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *JMIV*, 40(1):120–145, May 2011.

[8] T. F. Chan and J. Shen. Mathematical models for local nontexture inpaintings. *SIAM J. Appl. Math*, 62:1019–1043, 2002.

[9] S. E. Chen and L. Williams. View interpolation for image synthesis. In *SIGGRAPH*, pages 279–288. ACM, 1993.

[10] A. Criminisi, P. Perez, and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *Trans. Img. Proc.*, 2004.

[11] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH*, 1996.

[12] A. Delaunoy, J. Li, B. Jacquet, and M. Pollefeys. Two cameras and a screen: How to calibrate mobile devices? In *3DV*, 2014.

[13] L. C. Evans. *Partial differential equations*. Graduate studies in mathematics. American Mathematical Society, Providence (R.I.), 1998. Rimpr. avec corrections : 1999, 2002.

[14] J. Heikkila and O. Silven. A four-step camera calibration procedure with implicit image correction. In *CVPR*, pages 1106–1112, Jun 1997.

[15] J. A. Hesch, D. G. Kottas, S. L. Bowman, and S. I. Roumeliotis. Camera-IMU-based localization: Observability analysis and consistency improvement. *IJRR*, 2013.

[16] O. Kahler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. H. S. Torr, and D. W. Murray. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices. In *ISMAR*, 2015.

[17] N. Kawai, T. Sato, and N. Yokoya. Diminished reality based on image inpainting considering background geometry. *TVCG*, 22(3):1236–1247, 2016.

[18] M. Klingensmith, I. Dryanovski, S. Srinivasa, and J. Xiao. Chisel: Real Time Large Scale 3D Reconstruction Onboard a Mobile Device. In *RSS*, 2015.

[19] M. Kulkarni, A. N. Rajagopalan, and G. Rigoll. Depth inpainting with tensor voting using local geometry. In G. Csurka and J. Braz, editors, *VISAPP*, pages 22–30. SciTePress, 2012.

[20] S. Li, K. N. Ngan, and L. Sheng. Screen-camera calibration using a thread. In *ICIP*, pages 3435–3439, Oct 2014.

[21] C. Linz, C. Lipski, and M. A. Magnor. Multi-image interpolation based on graph-cuts and symmetric optical flow. In *SIGGRAPH*. ACM, 2010.

[22] C. Lipski, C. Linz, K. Berger, and M. A. Magnor. Virtual video camera: image-based viewpoint navigation through space and time. In *SIGGRAPH Posters*. ACM, 2009.

[23] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH*, 1987.

[24] G. Luo, Y. Zhu, Z. Li, and L. Zhang. A hole filling approach based on background reconstruction for view synthesis in 3D video. In *CVPR*, 2016.

[25] S. Lynen, T. Sattler, M. Bosse, J. A. Hesch, M. Pollefeys, and R. Siegwart. Get out of my lab: Large-scale, real-time visual-inertial localization. In *RSS*, 2015.

[26] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D reconstruction at scale using voxel hashing. In *SIGGRAPH Asia*, 2013.

[27] M. M. Oliveira, B. Bowen, R. McKenna, and Y. sung Chang. Fast digital image inpainting. In *VIIP*, pages 261–266. ACTA Press, 2001.

[28] T. Pock and A. Chambolle. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *ICCV*, pages 1762–1769, Washington, DC, USA, 2011.

[29] S. M. Seitz and C. R. Dyer. View morphing. In *SIGGRAPH*, 1996.

[30] M. Tomioka, S. Ikeda, and K. Sato. Approximated user-perspective rendering in tablet-based augmented reality. In *ISMAR*, 2013.

[31] K. Wu, A. Ahmed, G. Georgiou, and S. Roumeliotis. A Square Root Inverse Filter for Efficient Vision-aided Inertial Navigation on Mobile Devices. In *RSS*, 2015.

[32] S. S. Yoon, H. Sohn, Y. J. Jung, and Y. M. Ro. Inter-view consistent hole filling in view extrapolation for multi-view image generation. In *ICIP*, October 2014.

[33] Z. Zhang. A flexible new technique for camera calibration. *PAMI*, 22(11):1330–1334, Nov 2000.

[34] C. L. Zitnick, S. B. Kang, M. Uyttendaele, S. A. J. Winder, and R. Szeliski. High-quality video view interpolation using a layered representation. *TOG*, 23(3):600–608, 2004.