Fast Depth Densification for Occlusion-aware Augmented Reality

ALEKSANDER HOLYNSKI, University of Washington*
JOHANNES KOPF, Facebook

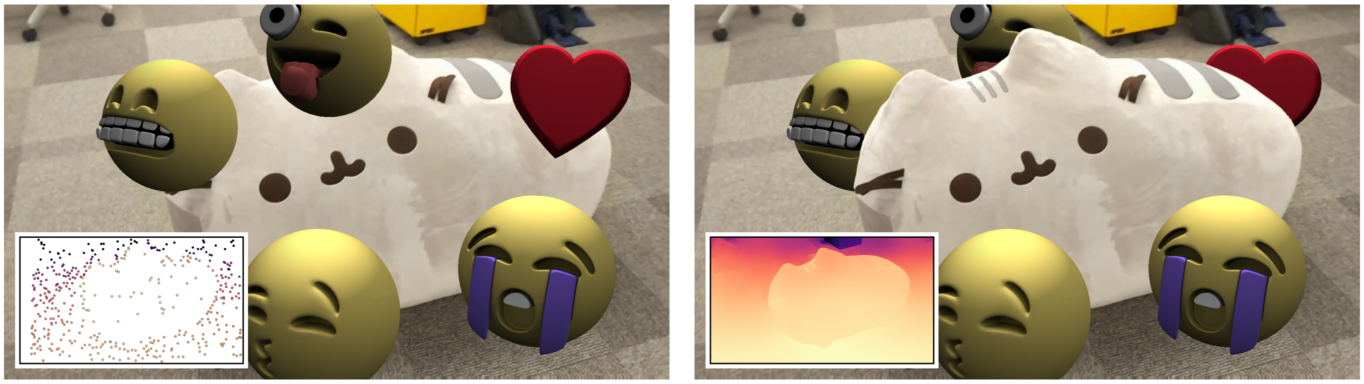

(a) AR Overlay using Sparse SLAM Reconstruction (b) Occlusion-aware AR with Our Densification

Fig. 1. Left: SLAM systems track only a few sparse point features. This limits AR effects to pure overlays because the scene geometry is not known for most pixels. Right: Our technique propagates the sparse depth to every pixel to produce dense depth maps. They exhibit sharp discontinuities at depth edges but are smooth everywhere else, which makes them particularly suitable for occlusion-aware AR video effects.

Current AR systems only track sparse geometric features but do not compute depth for all pixels. For this reason, most AR effects are pure overlays that can never be occluded by real objects. We present a novel algorithm that propagates sparse depth to every pixel in near realtime. The produced depth maps are spatio-temporally smooth but exhibit sharp discontinuities at depth edges. This enables AR effects that can fully interact with and be occluded by the real scene. Our algorithm uses a video and a sparse SLAM reconstruction as input. It starts by estimating soft depth edges from the gradient of optical flow fields. Because optical flow is unreliable near occlusions we compute forward and backward flow fields and fuse the resulting depth edges using a novel reliability measure. We then localize the depth edges by thinning and aligning them with image edges. Finally, we optimize the propagated depth smoothly but encourage discontinuities at the recovered depth edges. We present results for numerous real-world examples and demonstrate the effectiveness for several occlusion-aware AR video effects. To quantitatively evaluate our algorithm we characterize the properties that make depth maps desirable for AR applications, and present novel evaluation metrics that capture how well these are satisfied. Our results compare favorably to a set of competitive baseline algorithms in this context.

CCS Concepts: • **Computing methodologies → Reconstruction**; **Mixed / augmented reality**; *Computational photography*; Video segmentation;

*This work was done while Aleksander was working as an intern at Facebook.

Authors' addresses: Aleksander Holynski, University of Washington*, holynski@cs.washington.edu; Johannes Kopf, Facebook, jkopf@fb.com.

0730-0301/2018/11-ART194
https://doi.org/10.1145/3272127.3275083

Additional Key Words and Phrases: Augmented Reality, 3D Reconstruction, Video Analysis, Depth Estimation, Simultaneous Localization and Mapping

1 INTRODUCTION

Augmented reality (AR) is a transformative technology that blurs the line between reality and the virtual world by enhancing a live video stream with interactive computer generated 3D content. AR has many applications ranging from bringing virtual monsters into your bedroom (gaming), to previewing virtual furniture in your living room (shopping), or even breaking down the mechanics of your car's engine (learning), among many others.

Recent advancements in mobile hardware and tracking technology enable consumers to experience AR directly on their cell phone screens. This is typically powered by SLAM algorithms, such as Google's ARCore[1] and Apple's ARKit[2], which track a few dozen scene points in 3D and compute the 6-DOF trajectory of the device. Once the camera pose is known, we can overlay 3D content on the image, such as face masks or artificial animated characters.

However, since the tracked points are sparse, these effects cannot interact with the scene geometry, because it is not known for most pixels. This means that virtual objects can never be *occluded* by real objects (e.g., the Pusheen plush doll in Figure 1a). The absence of occlusion is often jarring and can break the illusion of reality.

In this work we propose a method that overcomes this limitation by *densifying* the sparse 3D points, so that depth is known at every pixel. This enables a much richer set of effects that fully interact with the scene geometry and make use of occlusions (Figure 1b).

Multi-view stereo (MVS) methods can be used to reconstruct dense geometry from multiple images, but they are often not the

[1]https://developers.google.com/ar/
[2]https://developer.apple.com/arkit/

best tool to achieve our goal, since they often (1) are not designed for video sequences and produce temporal artifacts such as flickering, (2) produce noisy results in untextured regions, (3) leave holes when omitting unconfident pixels, (4) frequently suffer from misaligned depth edges ("edge-fattening"), and (5) are prohibitively slow.

In addition, the requirements for dense, occlusion-aware AR applications are notably different from MVS. While the primary measure that these methods optimize is geometric accuracy of depth and normals, this property is less critical for AR. Instead, we are interested in a different set of objectives:

Sharp discontinuities: depth edges must be sharp and well-aligned with image edges to produce convincing occlusions.

Smoothness: away from discontinuities, and in particular across texture edges, the depth should be smooth to avoid spurious intersections with virtual objects.

Temporal coherence: the depth needs to be consistent across frames to avoid flickering.

Completeness: every pixel must have an assigned depth, so we can apply the effect everywhere.

Speed: for real-time AR effects we need to compute results at fast rates and with little delay.

We designed our method to satisfy all of these objectives. It takes the sparse points computed by a SLAM system as input and propagates their depths to the remaining pixels in a smooth, depth-edge aware, and temporally coherent fashion. The algorithm starts with computing a soft depth edge likelihood map by merging forward and backward optical flow fields using a novel reliability measure. Using a future frame causes a slight delay in the output, which depends on keyframe spacing, but is enforced to be at most 116ms in our experiments. A variant of our method can be run in a fully causal fashion, i.e., without any latency, at the expense of somewhat lower result quality. The depth edges are thinned and aligned with the image edges using an extension of the Canny edge detector that takes the soft depth edges into account. Finally, we densify the sparse points by optimizing the propagation of their depths smoothly (except at depth edges) and in a temporally coherent manner.

Our method runs at near-realtime rates on a desktop machine, and modern mobile processor benchmarks indicate that a fast mobile phone implementation should also be possible. We have tested our method on a variety of video sequences, and compared against a set of state-of-the-art baseline algorithms. We also designed evaluation metrics that capture the objectives stated above and perform extensive numerical evaluations. We demonstrate the effectiveness for AR applications with two example effects that make use of dense depth.

2 PREVIOUS WORK

In this section, we highlight some of the work in related areas.

SLAM. Simultaneous localization and mapping algorithms [Engel et al. 2018, 2014; R. Mur-Artal and Tardos 2015] compute the camera trajectory as well as a geometric scene representation from a video stream. This problem is related to Structure from Motion, but a fundamental difference is that SLAM techniques are optimized for realtime applications and specificially designed for video sequences. SLAM algorithms are typically used for tracking in AR applications.

Most methods, however, track only a select set of independent image features, which results in a sparse scene representation and limits AR effects to pure overlays.

Dense SLAM. Some SLAM methods attempt to reconstruct a dense depth map that covers all pixels in the video [M. Pizzoli 2014; Newcombe et al. 2011]. These methods are often slower, however, and may be less accurate (see discussion Engel et al.'s paper [2018]). Similar to MVS methods (discussed below), they are also not explicitly designed to have sharp depth discontinuities that are well-aligned with image edges, which might result in artifacts at occlusion contours.

There are also intermediate, semi-dense approaches that reconstruct a subset of pixels [Engel et al. 2014]. However, these methods have the same limitation w.r.t. virtual object occlusions as sparse methods.

MVS. Multi-view stereo methods [Furukawa and Hernández 2015; Seitz et al. 2006] compute dense geometry from overlapping images. However, as stated in the introduction, their depth maps are highly optimized for geometric accuracy, but might not work well for AR applications. For example, most algorithms drop uncertain pixels (e.g., in untextured areas) and edges in the estimated geometry are often not well aligned with image edges.

Video-based Depth Estimation. Most stereo algorithms are not designed to produce temporally coherent results for video sequences, however, there are some exceptions. Zhang et al. [2009] optimize multiple video frames jointly with an explicit geometric coherency term. However, similar to many MVS algorithms the runtime is prohibitively slow for our applications (several minutes per frame).

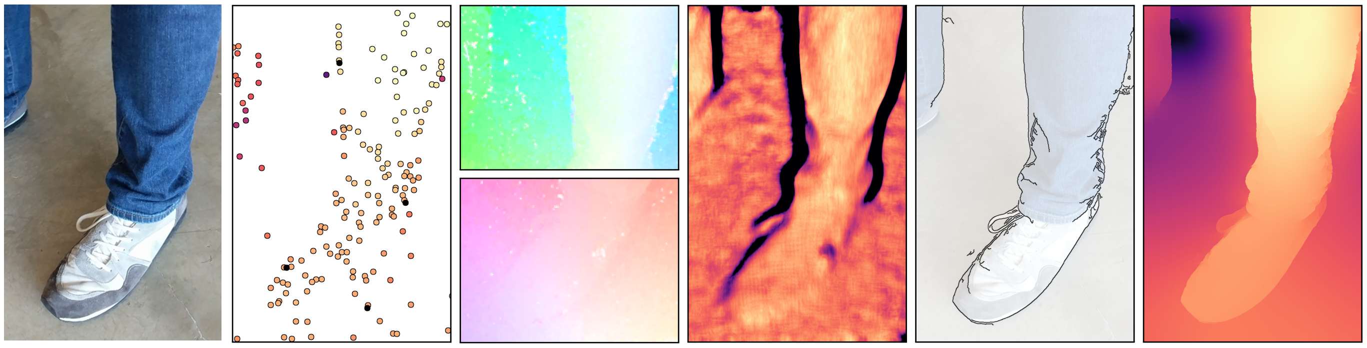
Hosni et al. [2011] use a weighted 3D box filter to spatio-temporally smooth cost volumes before extracting disparity values. Richardt et al. [2010] use instead a bilateral grid to spatio-temporally smooth a cost volume. Both methods require a desktop GPU to achieve interactive speeds. Stühmer et al [2010] estimate depth from multiple optical flow fields. However, optical flow estimation is unreliable in untextured regions, and it is slow at high quality settings.

Edge-aware Filtering. Sparse annotations, such as depth from SLAM points, can be densified using edge-aware filtering techniques. These techniques are typically very fast and differ in the kind of smoothness constraints they impose on the solution.

Levin et al. [2004] propose a quadratic optimization to propagate color scribbles on a grayscale image. A similar technique can be used to propagate depth from sparse points [Shan et al. 2014].

A joint bilateral filter [Petschnigg et al. 2004] can also be used to propagate sparse constraints while respecting intensity edges. The bilateral solver [Barron and Poole 2016] can be used in a similar way, e.g., in the Google Jump system [Anderson et al. 2016] it smoothes optical flow fields in an edge-aware manner.

Weerasekera et al. [2018] use the output of a single-view depth estimation network to propagate sparse depth values. Similarly, [Zhang and Funkhouser 2018] constrain the propagation of sparse depth values using the output of a neural network which predicts



(a) Input frame (Full HD, 2 megapixels) (b) SLAM points (DSO-SLAM, 13.8ms) (c) Flow to future / past (DIS-FLOW, 0.4ms) (d) Soft depth edges (Section 4.2, 0.2ms) (e) Localized depth edges (Section 4.3, 12.6ms) (f) Densification depth (Section 4.4, 21.3ms)

Fig. 2. Overview of the major algorithm stages. Given an input video (a) we use existing methods to compute sparse SLAM points (b) and optical flow to a nearby future and past frame (c). We fuse the most reliable responses from the forward and backward flow fields and compute "soft" depth edges that are not well-localized, yet. We thin, localize, and binarize the depth edges (e). Finally, we propagate the sparse SLAM point depths to every pixel in a depth-edge-aware manner (f).

surface normals and occlusion boundaries. Both methods are tested on desktop GPUs, and run at less than interactive framerates.

Park et al. [2014] describe techniques for upsampling and hole-filling depth maps produced by active sensors using constrained optimization. Bonneel et al. [2015] extend edge-aware filtering to video and add temporal constraints to reduce flickering. A survey of additional densification methods can be found in [Pan et al. 2018].

The drawback of many of these image-guided filtering techniques is that the propagation stops not only at depth edges, but also at texture edges. This can lead to false discontinuities in the depth maps. Additionally, many of these methods are prohibitively slow for real-time AR applications. We compare our method to three of the above filtering techniques in Section 5.4.

3 OVERVIEW

The inputs to our algorithm are (1) a sequence of video frames, typically captured with a cell phone camera, (2) camera parameters at every frame, and (3) sparse depth annotation (at least for some frames). We use an existing SLAM system [Engel et al. 2018] to compute (2) and (3). Our algorithm propagates the sparse depth annotation to every pixel in near realtime and with only a short delay of a few frames.

As mentioned before, the criteria that make depth maps desirable for AR applications are different from the goals that most MVS algorithms are optimized for, since we neither require correct *absolute* (or metric) depth, nor do we require perfect object normals. Rather, our algorithm is designed to produce *approximately* correct depth that satisfies the criteria stated in Section 1:

Discontinuities: we estimate the location of depth edges and produce sharp discontinuities across them.

Smoothness: our depth maps are smooth everywhere else, in particular across texture edges.

Temporal coherence: we optimize consistency of depth over time.

Completeness: by design we propagate depth to every pixel.

Speed: our algorithm takes on average 48.4ms per frame and the output is delayed by at most 116ms. A causal variant of the method has no delay.

Our algorithm proceeds in three stages:

(1) *Estimate soft depth edges (Figure 2d, Section 4.2):* First, we examine the gradient of optical flow fields to estimate "soft" depth edges that are not well-localized, yet. Because optical flow is unreliable near occlusions we compute flow fields to a future and past frame (Figure 2c) and fuse the resulting depth edges using the observation that edges in the flow gradient are only reliable when the flow vectors are diverging.
(2) *Localize depth edges (Figure 2e, Section 4.3):* Next, we localize the depth edges using a modified version of the Canny edge detector [Canny 1986]. This procedure thins the edges and aligns them with the image gradient, so that they are precisely localized on the center of image edges. It uses hysteresis to avoid fluctuations in weak response regions.
(3) *Densification (Figure 2f, Section 4.4):* Finally, we propagate the sparse input depth to every pixel by solving a Poisson problem. The data term of the problem is designed to approximate the sparse input depth and to encourage temporal continuity, while the smoothness term encourages sharp discontinuities at the detected depth edges and a smooth result everywhere else.

4 METHOD

4.1 Camera Parameters and Sparse Depth

The first stage of our algorithm computes a "sparse reconstruction" using a SLAM system. This computes two entities that are required for the subsequent stages:

(1) Extrinsic camera parameters for every frame (i.e., rotation and translation); we assume the intrinsic parameters are known.

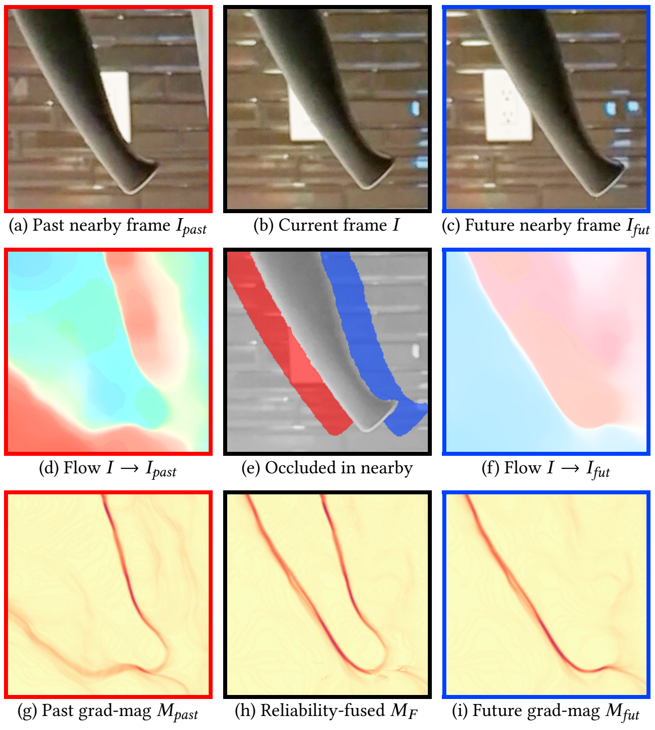

(a) Past nearby frame I_{past}
(b) Current frame I
(c) Future nearby frame I_{fut}
(d) Flow $I \longrightarrow I_{past}$
(e) Occluded in nearby
(f) Flow $I \longrightarrow I_{fut}$
(g) Past grad-mag M_{past}
(h) Reliability-fused M_F
(i) Future grad-mag M_{fut}

Fig. 3. For a current frame from the Kitchen sequence (b) we select two surrounding nearby frames (a,c) and compute optical flow (d,f). The flow fields are unreliable near pixels in the current frame that are occluded in the nearby frames (e, with manual annotation for illustration). The depth edges from the corresponding gradient magnitude images are unreliable as well (g,i). Using our reliability measure (see Figure 4) we can compute a fused result that contains only the most reliable pixels (h).

(2) Sparse 3D scene points (Figure 2b). Our algorithm will propagate their depth to the remaining pixels in a later stage to generate the dense result.

We experimented with various existing systems and settled on using DSO-SLAM [Engel et al. 2018], since it is fast and robust. Since DSO only provides 3D points at intermittent key frames, we reproject these to the surrounding non-key frames, so every frame has a sparse depth source.

4.2 Soft Depth Edges

The goal of this stage is to find "soft" depth edges, which means that they are defined by a continuous strength value and do not need to be accurately localized (Figure 2d). They will be thinned, binarized, and aligned with the image edges in the next section. For performance reasons we compute the soft edges on downscaled images (1/4 in each dimension) and upscale the results at the end of the stage.

We start by selecting a nearby frame with sufficiently large baseline to the current frame, and compute a dense optical flow field. In the flow field we can identify depth edges at places where the gradient magnitude is high, because sudden changes in depth imply corresponding changes in the flow, due to parallax (Figure 3, bottom row).

Unfortunately, optical flow is unreliable around pixels that are occluded in the nearby frame (compare Figures 3d and 3f with Figure 3e and note how only one of the two edges of the object are resolved in each flow image, respectively). We alleviate the situation by computing flow w.r.t. *two* nearby frames, one backward and one forward in time. These tend to have different sets of pixels that are unreliable (see manual annotations in Figure 3e). We fuse the two depth edge maps using a novel optical flow reliability measure, described below, to achieve a result that contains the correct edges from both (Figure 3h). In the supplementary material, we show a comparison of the reliability merging and the more naive approach of taking the per-element gradient maximum. As can be seen in the Bones sequence, the naive approach drastically increases the noise in the occlusion response, and thus the occlusion edges are less prominently defined. We also provide comparisons with off-the-shelf boundary estimation methods, such as [Dollár et al. 2006], showing that our use of flow gradients more frequently disambiguates occlusion boundaries from texture edges, such as in the soft edge response of the Georgian sequence.

Selecting Nearby Frames. The nearby frames need to provide sufficient translational motion so we achieve a strong flow gradient response and reduce noise. There are many sensible ways in which these could be selected; we use the following simple heuristic. We compute the spatial distance between the camera positions of the two DSO key frames that bracket the current frame. Then, we look forward and backward from the current frame and select the first frame in either direction whose camera position is at least half that distance away.

Optical Flow. We compute optical flow using DIS Flow [Kroeger et al. 2016], because it is one of the fastest available methods. We use the implementation in OpenCV and chose the `ultrafast` preset. Since the results exhibit a block pattern we smooth it using a 7×7 median filter.

Computing the Gradient Magnitude. Let F be one of the two flow images from the current frame to one of the nearby frames. We compute the gradient magnitude M, at every pixel p selecting the x or y component, whichever provides the higher value:

$$M(p) = \max\left(\left\|\nabla F_x(p)\right\|_1, \left\|\nabla F_y(p)\right\|_1\right). \tag{1}$$

Fusing Forward and Backward Results. As described above, different sets of pixels are reliable in the two flow fields. We observe that places where the edge-normal flow projections f are *diverging* are generally more reliable; this situation occurs at *disocclusions*, when both pixels are visible in the nearby image. The opposite is true for *converging* flow projections; this indicates an *occlusion*: one of the two pixels is not visible in the nearby image and therefore its flow vector cannot be accurately estimated.

We turn this observation into a per-pixel reliability score as follows. Given a pixel of interest p, we find two helper pixels p_0 and p_1 that are offset at unit distance in the gradient direction d and its opposite. We compute the projection of the flow vectors at p_0 and

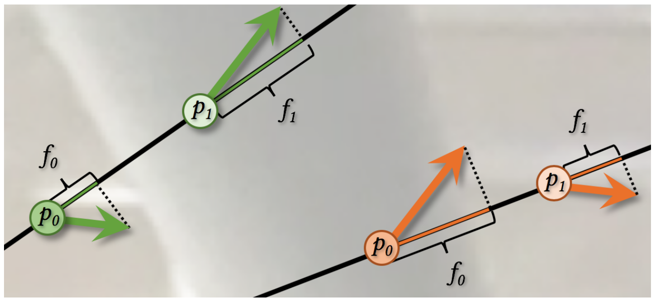
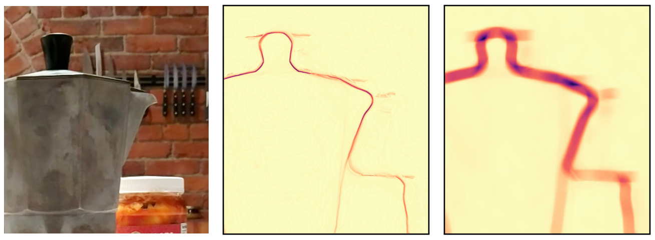


Fig. 4. Our flow gradient reliability measure explained on two examples on a crop from Figure 3. We measure the flow at two helper pixels p_0, p_1 on either side of the edge, and compute the projections f_0, f_1 onto the line perpendicular to the gradient. Left example: the flow projections are diverging ($f_1 - f_0 > 0$), which indicates a reliable edge, since both pixels are visible in the nearby image. Right example: the flow projections are converging ($f_1 - f_0 < 0$), which indicates an unreliable edge, since at least one of the pixels might not be visible in the nearby image.



(a) Color image (b) Initial soft depth edges M_F (c) Spatio-temporal filtered $\widetilde{M}_F$

Fig. 5. (b) Depth edges are well identified in the flow gradient image, but not well aligned with the image edges. (c) We apply spatio-temporal filtering to reduce noise and ensure the depth edges overlap with their corresponding image edges. Now they are ready for alignment with the image edges (see Figure 6).

p_1 on d:

$$f_0 = F(p_0) \cdot d, \qquad f_1 = F(p_1) \cdot d, \tag{2}$$

and obtain the reliability score as their difference,

$$r = f_1 - f_0. \tag{3}$$

r will be positive (reliable) for diverging flow projections, and negative (unreliable) for converging flow projections. Figure 4 illustrates this on two examples.

Now we can fuse the gradient magnitude from both nearby images by selecting at each pixel the more reliable quantity:

$$M_F(p) = \begin{cases} M_{prev}(p), & \text{if } r_{prev}(p) > r_{next}(p) \\ M_{next}(p), & \text{else} \end{cases} \tag{4}$$

Filtering. The fused gradient magnitude M_F identifies depth edges well without getting confused by texture edges (Figure 5a-b). However, they are not well aligned with color images. To make the alignment with image edges in the following stage easier, we blur M_F with a wide box filter of size $k_F = 31$, to ensure edges overlap with their corresponding image edges.

We also suppress noise by applying a temporal median filter that includes samples from $k_T = 7$ frames. We determine the temporal neighbors of pixels by estimating a per-frame homography warp from the SLAM points (using the OpenCV `findHomography()` function).

Finally, we normalize M_F by dividing it by the 90th percentile value. This makes the parameter settings more invariant to the video content.

The final spatio-temporally filtered soft depth edges $\widetilde{M}_F$ are shown in Figure 5c.

4.3 Localizing the Depth Edges

In this section, we accurately localize the depth edges by thinning and binarizing them, and aligning them with the color image edges. We achieve these goals by modifying the Canny edge detector [1986], which performs a similar operation just on intensity image edges.

Recall the basic operation of the Canny detector:

(1) *Intensity gradient magnitude:* Compute the (blurred) gradient magnitude of the intensity image $\widetilde{M}_I$ (we normalize it by dividing out the 90th percentile).
(2) *Non-maximum suppression:* all values of $\widetilde{M}_I$ except the local maxima are suppressed, to thin edges down to a width of a single pixel.
(3) *Double thresholding:* Using two thresholds τ_{high} and τ_{low} the edge pixels are classified into strong ($\widetilde{M}_I > \tau_{high}$), weak ($\tau_{high} \geq \widetilde{M}_I \geq \tau_{low}$), and suppressed edge pixels ($\tau_{low} > \widetilde{M}_I$).
(4) *Hysteresis:* Every strong edge pixel is selected, but weak edge pixels are only selected when they are connected to a strong edge. This effectively removes spurious weak edge pixels.

Figure 6c shows the result of applying the Canny detector on a color image. The detector identifies and localizes all intensity edges well. Notably, this set of edges includes also all major depth edges. This indicates that we can achieve our goal by selectively suppressing only the texture edges while keeping the depth edges.

We do so by injecting another threshold on the soft depth edge map $\widetilde{M}_F$ into the algorithm. More precisely, we change the definition of a strong edge pixel to require not just a high intensity gradient response ($\widetilde{M}_I > \tau_{high}$) but also a high response in the soft edge map ($\widetilde{M}_F > \tau_{flow}$). The definition of weak and suppressed edge pixels remain unchanged.

With this modification we start depth edges only where the soft edge map indicates a high confidence, but we allow continuing them into low soft edge response regions as long as there remains a sufficiently strong image edge. This helps bridging over gaps in the soft edge map due to noise in the optical flow image. Figure 6e shows the result of the modified detector. It effectively preserves most of the depth edges while suppressing most texture edges.

4.4 Densification

In the final stage we use the localized depth edges to control the propagation of sparse SLAM point depths to the remaining pixels. We set this up as a quadratic optimization problem, using the following constraint terms.

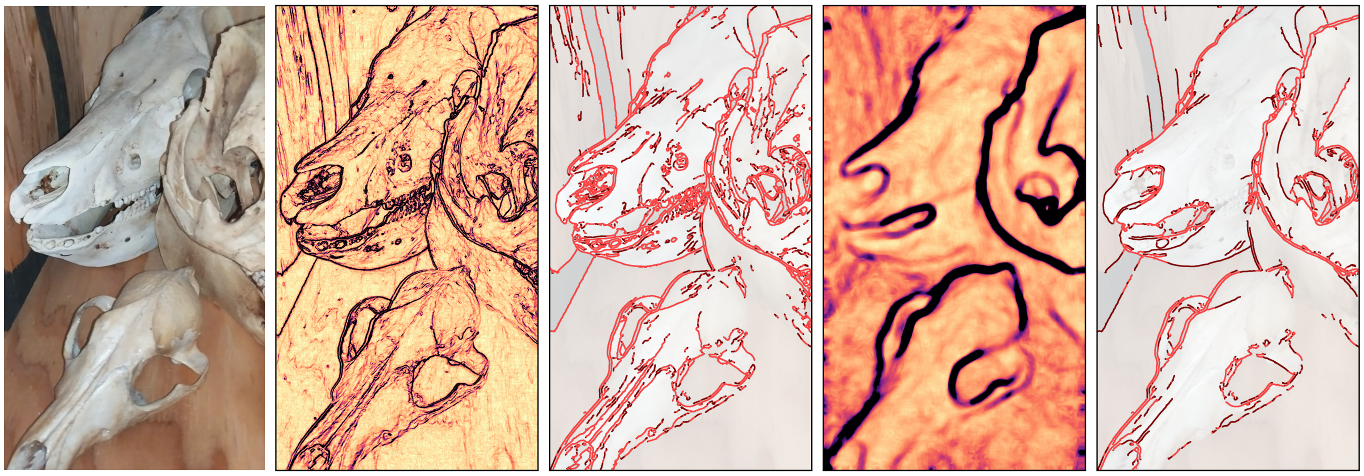

(a) Image I (b) Magnitude of intensity gradient $\widetilde{M}_I$ (c) Standard Canny detector using $\widetilde{M}_I$ (d) Magnitude of flow gradient $\widetilde{M}_D$ (e) Modified Canny detector using $\widetilde{M}_I$ and $\widetilde{M}_D$

Fig. 6. Localizing and aligning the soft depth edges. (a-b) Input image and corresponding magnitude of intensity gradient. (c) An edge detector that uses this image selects both texture and depth edges. Strong and weak edge pixels are drawn in bright and dark color, respectively. (d-e) We inject our soft edge map in the detector algorithm, which results in suppressed texture edges. The remaining edges are mostly depth edges and well aligned with the intensity edges (Note, that edges are drawn thick here for illustration purposes).

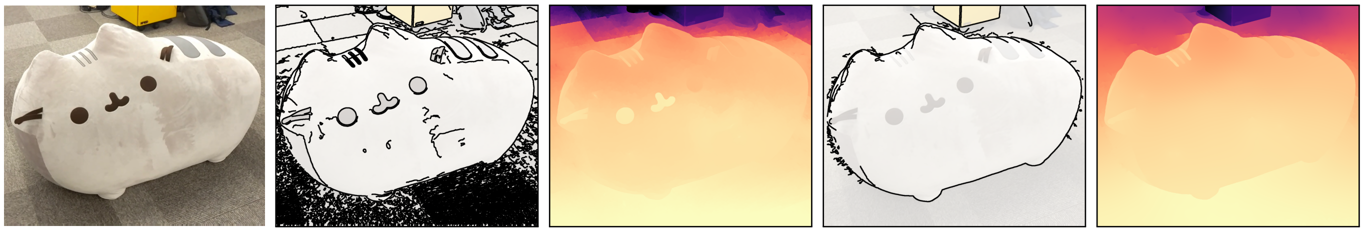

(a) Image I (b) Canny edges (c) Depth densified with Canny (d) Our depth edges (e) Our densification

Fig. 7. The standard Canny edge detector fires on texture edges (b), which causes incorrect discontinuities in the resulting depth map (c). Our algorithm suppresses most texture edges (d), which leads to a refined depth map (e).

A unary data term encourages approximating the depth of the SLAM points:

$$E_{data}(p) = w_{sparse}(p) \left\| D(p) - D_{sparse}(p) \right\|_2^2. \quad (5)$$

D_{sparse} is a depth map obtained by splatting the depths of SLAM points into single pixels, and w_{sparse} is 1 for all pixels that overlap a SLAM point and 0 everywhere else.

A second unary data term encourages the solution to be temporally coherent:

$$E_{temp}(p) = w_{temp}(p) \left\| D(p) - D_{temp}(p) \right\|_2^2. \quad (6)$$

D_{temp} is a depth map obtained by reprojecting the (dense) pixels of the previous frame using the projection matrices estimated by the SLAM system. w_{temp} is 1 for all pixels that overlap a reprojected point and 0 everywhere else (splatting gaps, near boundaries).

We use a spatially varying pairwise smoothness term:

$$E_{smooth}(p,q) = w_{pq} \left\| D(p) - D(q) \right\|_2^2, \quad (7)$$

with the weight

$$w_{pq} = \begin{cases} 0, & \text{if } B(p) + B(q) = 1, \\ \max\left(1 - \min(s_p, s_q), 0\right), & \text{else.} \end{cases} \quad (8)$$

B denotes the binarized depth edge map, computed in the previous section, and $s_p = \left(\widetilde{M}_F \cdot \widetilde{M}_I\right)(p)$, $s_q = \left(\widetilde{M}_F \cdot \widetilde{M}_I\right)(q)$. At depth edges we set the weight to zero to allow the values to drift apart without any penalty and form a crisp discontinuity. Everywhere else, we enforce high smoothness if either $\widetilde{M}_I$ is low (textureless regions) or $\widetilde{M}_F$ is low (possibly texture edge but not a depth edge).

We obtain the following combined continuous quadratic optimization problem:

$$\underset{D}{\operatorname{argmin}} \ \lambda_d \sum_p E_{data}(p) + \lambda_t \sum_p E_{temp}(p) + \lambda_s \sum_{(p,q)\in N} E_{smooth}(p,q), \quad (9)$$

where N is the set of horizontally and vertically neighboring pixels. We use the balancing coefficients $\lambda_d = 1, \lambda_t = 0.01, \lambda_s = 1$.

The solution to Equation 9 is a set of sparse linear equations. It is, in fact, a standard Poisson problem, for which we have specialized

Table 1. Breakdown of the average per-frame timings of the algorithm stages.

Algorithm Step	Duration
Sparse Reconstruction (DSO-SLAM)	13.8ms
Optical flow, for past and future frame (DIS-Flow)	0.4ms
Soft depth edges	0.2ms
Localized depth edges	12.6ms
Densification	21.3ms
Total	**48.4ms**

solvers that can optimize it rapidly. We use an implementation of the LAHBF solver [Szeliski 2006] to optimize it.

Figure 7 shows the impact of suppressing texture edges in the densification. When using standard Canny intensity edges, the resulting depth maps contains many false depth discontinuities (Figure 7c). These are mostly absent in our result (Figure 7e).

5 RESULTS & EVALUATION

While our method is ultimately intended to be run in a real-time setting, e.g., in the viewfinder of a smart phone, we implemented it in practice to operate on pre-captured video sequences, since it enables easier debugging and more reproducible results.

We captured and processed a number of video sequences with a Google Pixel 2 smart phone at full HD resolution (1920×1080 pixels). Screenshots from each sequence can be seen in Figure 8. The videos contain indoor and outdoor locations, of a variety of objects and scenes, often including objects that are hard to reconstruct with traditional multi-view stereo algorithms, such as moving objects, water, reflective surfaces, and thin structures.

In the supplementary material we provide the full set of input videos, final depth maps, as well as videos of the intermediate SLAM points, soft depth edges, and localized depth edges, in form of a web page for convenient inspection.

5.1 Effects

We implemented two AR effects (Figure 9) that make use of occlusions and the ability to interact with the dense scene geometry:

Object insertion: place virtual objects in the scene that can be occluded by real objects.

Lighting effect: insert a point light source that shades the scene with radial fall-off lighting.

In the supplementary material we demonstrate each effect on several videos.

5.2 Performance

All results were generated on a PC with 3.4 GHz 6-core Intel i7-6800K CPU. Our algorithm only uses the CPU. Our current implementation processes our 2-megapixel HD videos at an average of 48.3ms per frame. Table 1 breaks down the timings for various algorithm stages.

While our current implementation is on a desktop computer, a fast implementation on a phone seems possible because:

- Real-time SLAM has been demonstrated by ARKit and ARCore.
- Well optimized Canny runs faster on modern-generation mobile phones (e.g. Google Pixel 2XL and Apple iPhone X) than our implementation on a desktop[3].
- Poisson systems such as Eq. 9 can be solved with highly specialized solvers, and real-time speeds were achieved a decade ago by [McCann and Pollard 2008], using an evaluation system that is less powerful than today's phones.

Since these steps comprise over 98% of our runtime, we believe an optimized phone implementation of our method can achieve real-time speeds as well.

5.3 Evaluation Metrics

Most MVS and other depth reconstruction algorithms are optimized for geometric accuracy. However, as mentioned before, for our AR effects application we have different priorities; see the objectives stated in Sections 1 and 3.

In order to quantitatively assess our method and objectively compare our method to other baseline algorithms below, we propose three evaluation metrics that capture how well these objectives are achieved: (1) *Occlusion error* penalizes depth edges not being sharp, (2) *Texture error:* penalizes depth at texture edges not being smooth, and (3) *Temporal instability:* penalizes temporal jitter of static points.

These metrics correspond directly to the first three objectives stated in Sections 1 and 3. The other two objectives, *completeness* and *speed* are satisfied by design: our results are always 100% complete and our method operates at near real-time rates.

We describe the three metrics below, and use them in a comparative analysis in the next section.

Annotations. For evaluating the occlusion and texture error we need ground truth annotations of such occurrences, which we generated as follows. We selected five datasets (Bones, Cubes, Cutting Board, Pusheen, Shoes) and from each five random images, for a total of 25 images. For each image we computed a high quality offline MVS reconstruction [Schönberger et al. 2016]. We then computed Canny edges and classify each edge pixel as "occlusion", "texture" or "no edge" based on its depth profile. More precisely, we compute the median depth of 5 unit-spaced pixels on either side perpendicular to the edge. If that ratio of median depths is between 1 and 1.05 we consider it a texture edge pixel, if it is above 1.2 we consider it an occlusion edge pixel, and otherwise we ignore that pixel.

We then recruited five volunteers and asked them to clean up any errors in the automatic classification. They could only erase edges but not add new ones. Each volunteer processed five images, one from each dataset. The final annotations are included in the supplementary material.

Occlusion Error. This error measures for annotated occlusion pixels how crisp and well localized the edge in the depth map is. We extract a profile of 10 depth samples $\{d_i\}$ perpendicular to the edge, 5 on either side, and measure the deviation from an ideal step edge, after removing the mean and standard deviation:

$$E_{occ} = \frac{1}{N}\sum_i \left(\frac{d_i-\mu}{\sigma} - s_i\right)^2, \tag{10}$$

[3]https://browser.geekbench.com/v4/cpu/9747825

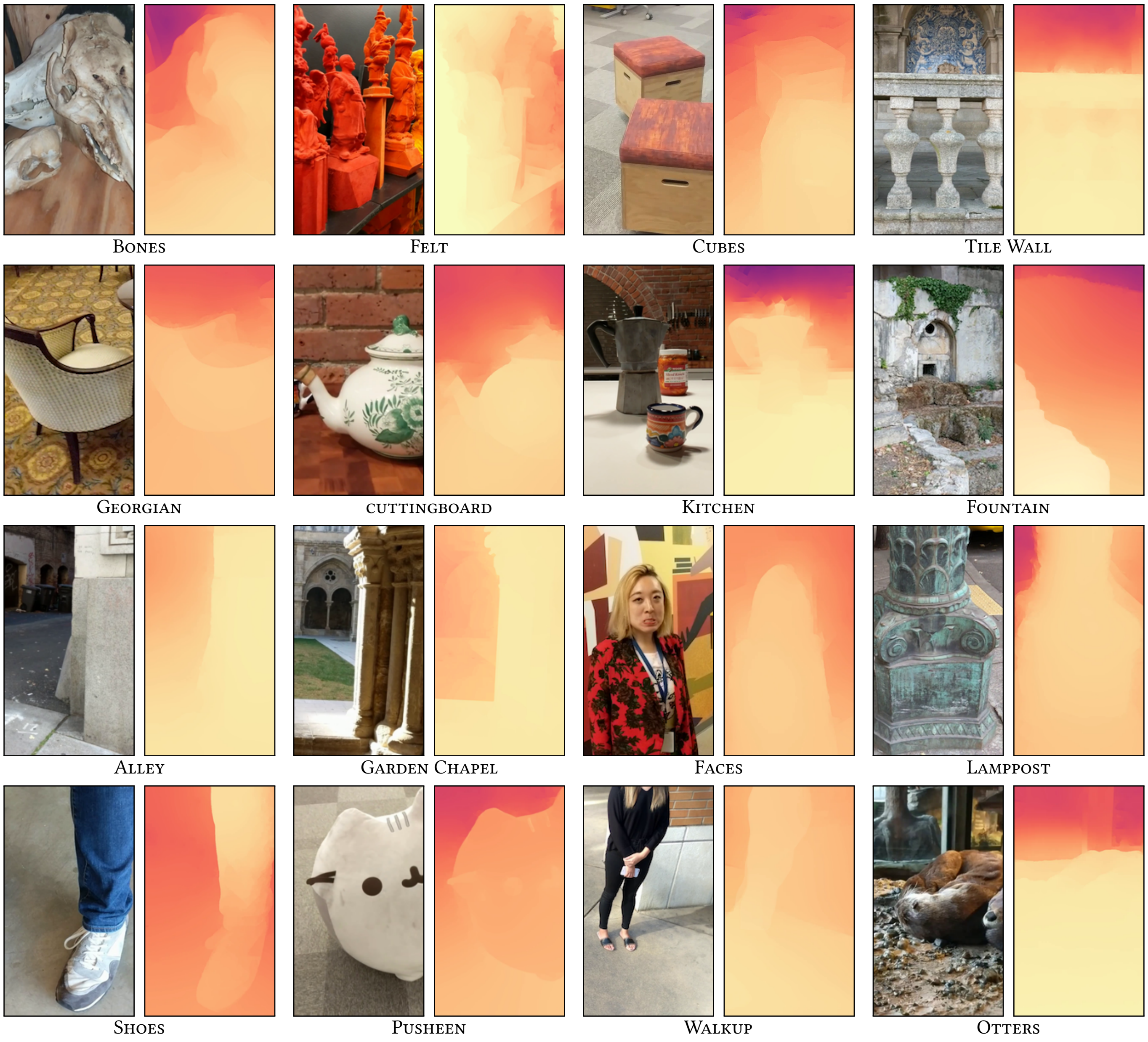


Fig. 8. Datasets we captured for this paper. Please refer to the supplementary material to see the full videos of input and intermediate and final results. Note that images have been cropped for visualization. Our datasets consist of both portrait and landscape videos.

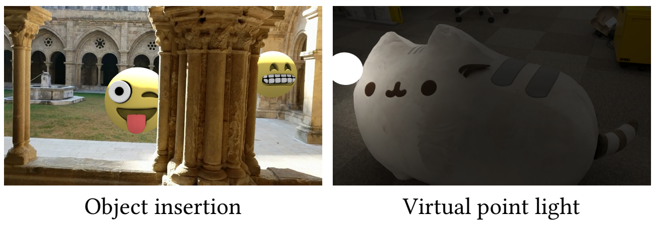


Fig. 9. Example occlusion-aware AR effects.

where μ and σ are the mean and standard deviation, respectively, and $s_i = \begin{cases} -1, & \text{if } i \leq N/2 \\ +1, & \text{else} \end{cases}$ is a step function.

Texture Error. Texture edges are color changes in regions that are not occlusion boundaries. We expect the depth map to be smooth here, because false depth discontinuities would cause self-occlusion artifacts or cracks in objects.

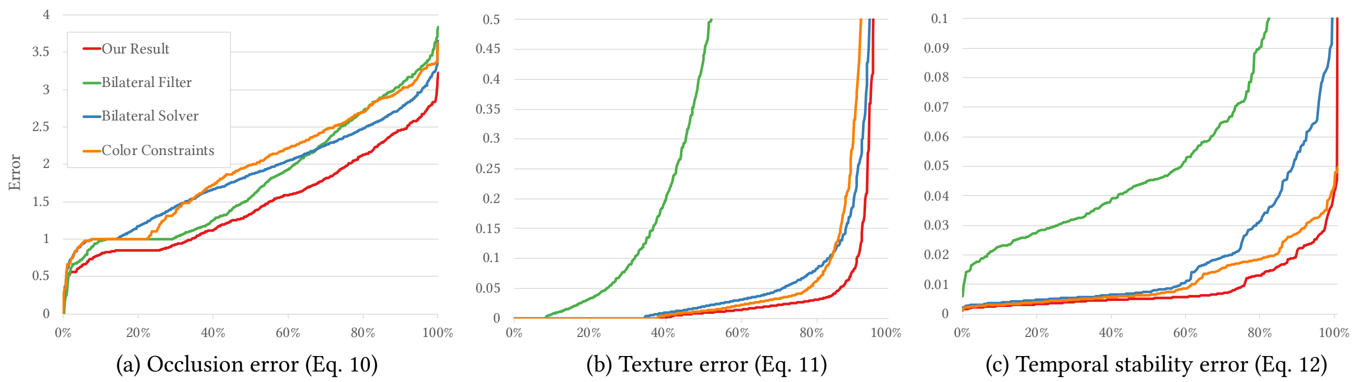


Fig. 10. Comparing our method against several baselines using our evaluation metrics. The graphs plot the cumulative error histogram for all annotated samples for five datasets (see text).

Table 2. A comparison of combined error values (lower is better).

Method	E_{comb}
Bilateral Filter	37.11
Bilateral Solver	4.05
Color Constraints	3.46
Our Result	**2.54**

The texture error measures for annotated texture edge pixels how much the depth profile deviates from a flat profile:

$$E_{tex} = \frac{1}{N} \sum_i \left(\frac{d_i - \mu}{\mu} \right)^2 . \tag{11}$$

Note, that, unlike in Eq. 10 we are not dividing by the standard deviation, since this would amplify the flat profiles. Instead, we divide by the mean depth to make the error invariant to the scene scale.

Temporal Stability Error. Abrupt depth changes in the video can cause flickering when rendering effects. The temporal stability error penalizes variation in the 3D position of static scene points. For the five evaluation datasets we track about 100 points on the middle 100 frames with a KLT tracker, and keep all tracks that span all frames. The error is defined as the variance of the 3D positions that are obtained when unprojecting the points using the depth map:

$$E_{ts} = \frac{1}{N} \sum_{f=1}^{100} \Big(U_f\big(p_f, D_f(p_f)\big) - \mu \Big)^2 . \tag{12}$$

p_f is the tracked point in frame f, D_f is the depth map for the frame, and U_f is the unprojection function, which takes a 2D image coordinate and depth and returns the corresponding 3D world position. $\mu = \frac{1}{N} \sum_f U_f\big(p_f, D_f(p_f)\big)$ is the mean 3D world position.

Combined Error. It is useful for parameter tuning to have a single combined scalar error that balances the various objectives. Since we consider all three metrics equally important, we determine coefficients that balances out their relative scales:

$$E_{comb} = 0.7\, \widetilde{E_{occ}} \; + \; 65\, \widetilde{E_{tex}} \; + \; 200\, \widetilde{E_{ts}}, \tag{13}$$

where $\widetilde{\cdot}$ indicates the median across all annotated samples. A comparison to the baseline methods can be seen in Figure 2. We obtained these coefficients by iteratively tuning our method for each metric separately, and then taking the value that maps the median of each metric to 1.

5.4 Comparative Evaluation

We compared our algorithm to various baselines using the metrics defined in the previous section.

Bilateral Solver. We compare against the fast bilateral solver [Barron and Poole 2016], using the publicly available implementation[4]. We use the color frames as reference image for the bilateral solver, and set the target image t and confidence image c as follows:

$$(t,c)(p) = \begin{cases} \big(D_{sparse}(p),\ w_{sparse}(p)\big), & \text{if } w_{sparse}(p) > 0, \\ \big(D_{temp}(p),\ \lambda_{temp}^{bs} w_{temp}(p)\big), & \text{else,} \end{cases} \tag{14}$$

i.e., for pixels that fall under a SLAM point we use that point's depth as target with a confidence of one, and for all other pixels we use the reprojected points from the previous frame with a lower confidence λ_{temp}^{bs}, to make the result more temporally stable.

We tune the bilateral solver parameters as well as the temporal parameter λ_{temp}^{bs} to minimize the combined error E_{comb} and obtain the following settings:

$$\lambda = 1, \quad \sigma_{xy} = 5, \quad \sigma_i = 15, \quad \sigma_{uv} = 10, \quad \lambda_{temp}^{bs} = 0.8. \tag{15}$$

Bilateral Filter. We compare against a joint bilateral median filter [Petschnigg et al. 2004] guided by the color frames. Because the SLAM points are very sparse we increase the kernel size in increments of 10 pixels until there are at least 8 depth inside. To make the filter temporally coherent we also include samples from D_{temp} in a 10x10 kernel, weighted by a temporal parameter λ_{temp}^{bf}. To better preserve hard edges we use a median.

We tune the bilateral filter parameters as well as the temporal parameter λ_{temp}^{bf} to minimize the combined error E_{comb} and obtain

[4]https://github.com/poolio/bilateral_solver

Table 3. Parameters of our algorithm. We used the default settings provided here for all results.

Parameter	Section	Description
$k_F = 31$	4.2	Flow gradient box filter size
$k_T = 7$	4.2	Temporal median window size
$k_I = 5$	4.3	Image box filter size
$\tau_{high} = 0.04$	4.3	Canny image high threshold
$\tau_{low} = 0.01$	4.3	Canny image low threshold
$\tau_{flow} = 0.3$	4.3	Canny flow threshold
$\lambda_d = 1$	4.4	Balancing coefficients
$\lambda_t = 0.01$	4.4	Balancing coefficients
$\lambda_s = 1$	4.4	Balancing coefficients

the following settings:

$$\sigma_{spatial} = 10, \quad \sigma_{color} = 10, \quad \lambda_{temp}^{bf} = 0.85. \tag{16}$$

Color-based optimization constraints. We also compare against a variant of our densification that uses only color-based constraints instead of our estimated depth edges. In Eq. 9 we replace the E_{smooth} with the pairwise term by Levin et al. [2004].

We fix $\lambda_s = 1$ and tune the remaining modified densification parameters to minimize the combined error E_{comb} and obtain the following settings:

$$\lambda_d = 0.1, \quad \lambda_t = 0.1, \quad \sigma_{\mathbf{r}} = 0.01. \tag{17}$$

Discussion. We tuned our method as well as the baselines to minimize the combined error (Eq. 13). Then, we evaluated the individual metrics on the five evaluation datasets, and plot all samples in Figure 10. Our method provides a substantial improvement over the baselines in all metrics. For a qualitative comparison, please refer to the supplementary material.

5.5 Parameters

In Table 3 we lists all the parameters of our algorithm and their default settings. We omit parameters of the DSO SLAM and DIS-Flow components, since we did not change them from their default settings. All results shown in the paper and the supplementary material were generated with the same settings.

We tuned our method by minimizing the combined error E_{comb}, iteratively fixing some groups of parameters while varying others until we reached a local minimum.

5.6 Limitations

Our method has a number of failure cases inherited from the underlying SLAM algorithm:

View-dependent appearance: Similar to most 3D reconstruction methods our method has problems dealing with reflective and specular materials. This is actually a limitation of the SLAM component, which produces points at incorrect depths in these cases.

Small translation: The SLAM algorithm requires sufficient amount of translational motion to produce 3D points.

Textureless surfaces: The SLAM algorithm requires textured surfaces in order to accurately localize and track 3D points.

Missing SLAM points: We cannot recover the depth for objects that stick out from their surrounding but do not have any SLAM points on them. This problem is very present in the Tile Wall dataset, which misses points on the foreground stone railing.

Dynamic scenes: Our method tolerates slight scene motion, such as in the Faces dataset, but does not produce good results in the presence for highly dynamic or transient objects. Nevertheless, our method is quick to recover once the dynamic objects have stabilized. An example of this can be seen in the Walkup dataset.

Additionally, our algorithm has its own limitations that lead to interesting avenues for future work:

Delay: Because we are computing optical flow to a future frame our method is not fully causal but outputs results with a slight delay (depending on the spacing of key frames). In practice, we find that under regular camera motion, optimal quality can be achieved while still enforcing the lookahead to never exceed 7 frames (or 116ms at 60Hz). We have also experimented with a causal variant of our method, which uses only previous frames. Examples can be seen in the Alley, Georgian, and Lamppost datasets, which all include a portion of the video where the camera stops moving, i.e. points at which there is no future keyframe to use as a flow reference. In these cases, the resulting depth video does not suffer greatly in quality, as a result of the temporal constraints, filtering, and initialization.

Low geometric accuracy: Our method is not suitable for applications that require high geometric accuracy. It is, for example, not suitable for lighting effects that rely on accurate scene normals.

Floating layers: In cases where an object has an occlusion edge on only one side, but has strong texture edges on all sides, the Canny algorithm may trace around the entire object. This usually does not affect the final depth map, except in cases where there are very few SLAM points on the occluding object, resulting in a "floating layer" effect. An example can be found in the Shoes sequence, where the bottom of the feet seem to be floating in front of the floor.

6 CONCLUSION

In this paper we have presented a fast algorithm for propagating a sparse depth source, such as SLAM points, to all remaining pixels in a video sequence. The resulting dense depth video is spatio-temporally smooth except at depth edges where it exhibits sharp discontinuities.

These properties make our algorithm particularly useful for AR video effects. Due to the absence of holes in the depth maps, the effects can fully interact with the scene geometry, and, for example, be occluded by real objects.

REFERENCES

Robert Anderson, David Gallup, Jonathan T. Barron, Janne Kontkanen, Noah Snavely, Carlos Hernandez Esteban, Sameer Agarwal, and Steven M. Seitz. 2016. Jump:

Virtual Reality Video. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 35, 6 (2016), article no. 198.

Jonathan T Barron and Ben Poole. 2016. The Fast Bilateral Solver. *European Conference on Computer Vision (ECCV)* (2016), 617–632.

Nicolas Bonneel, James Tompkin, Kalyan Sunkavalli, Deqing Sun, Sylvain Paris, and Hanspeter Pfister. 2015. Blind Video Temporal Consistency. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2015)* 34, 6 (2015).

John Canny. 1986. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 8, 6 (1986), 679–698.

P. Dollár, Z. Tu, and S. Belongie. 2006. Supervised Learning of Edges and Object Boundaries. In *CVPR*.

Jakob Engel, Vladlen Koltun, and Daniel Cremers. 2018. Direct Sparse Odometry. *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2018).

Jakob Engel, Thomas Schöps, and Daniel Cremers. 2014. LSD-SLAM: Large-Scale Direct Monocular SLAM. *European Conference on Computer Vision (ECCV)* (2014), 834–849.

Yasutaka Furukawa and Carlos Hernández. 2015. Multi-View Stereo: A Tutorial. *Foundations and Trends. in Computer Graphics and Vision* 9, 1-2 (2015), 1–148.

Asmaa Hosni, Christoph Rhemann, Michael Bleyer, and Margrit Gelautz. 2011. Temporally consistent disparity and optical flow via efficient spatio-temporal filtering. In *Pacific-Rim Symposium on Image and Video Technology*. Springer, 165–177.

Till Kroeger, Radu Timofte, Dengxin Dai, and Luc Van Gool. 2016. Fast Optical Flow using Dense Inverse Search. *Proceedings of the European Conference on Computer Vision (ECCV)* (2016).

Anat Levin, Dani Lischinski, and Yair Weiss. 2004. Colorization Using Optimization. *ACM Trans. Graph.* 23, 3 (2004), 689–694.

D. Scaramuzza M. Pizzoli, C. Forster. 2014. REMODE: Probabilistic, monocular dense reconstruction in real time. *International Conference on Robotics and Automation (ICRA)* (2014), 2609–2616.

James McCann and Nancy S Pollard. 2008. Real-time gradient-domain painting. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 93.

Richard A. Newcombe, Steven J. Lovegrove, and Andrew J. Davison. 2011. DTAM: Dense Tracking and Mapping in Real-time. *International Conference on Computer Vision (ICCV)* (2011), 2320–2327.

Liyuan Pan, Yuchao Dai, Miaomiao Liu, and Fatih Porikli. 2018. Depth Map Completion by Jointly Exploiting Blurry Color Images and Sparse Depth Maps. In *Applications of Computer Vision (WACV), 2018 IEEE Winter Conference on*. IEEE, 1377–1386.

Jaesik Park, Hyeongwoo Kim, Yu-Wing Tai, Michael S Brown, and In So Kweon. 2014. High-quality depth map upsampling and completion for RGB-D cameras. *IEEE Transactions on Image Processing* 23, 12 (2014), 5559–5572.

Georg Petschnigg, Richard Szeliski, Maneesh Agrawala, Michael Cohen, Hugues Hoppe, and Kentaro Toyama. 2004. Digital Photography with Flash and No-flash Image Pairs. *ACM Trans. Graph.* 23, 3 (2004), 664–672.

J.M.M. Montiel R. Mur-Artal and Juan D. Tardos. 2015. ORB-SLAM: a Versatile and Accurate Monocular SLAM System. *IEEE Transactions on Robotics* 31, 5 (2015), 1147–1163.

Christian Richardt, Douglas Orr, Ian Davies, Antonio Criminisi, and Neil A Dodgson. 2010. Real-time spatiotemporal stereo matching using the dual-cross-bilateral grid. In *European conference on Computer vision*. Springer, 510–523.

Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. 2016. Pixelwise View Selection for Unstructured Multi-View Stereo. (2016).

Steven M Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. 2006. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *null*. IEEE, 519–528.

Qi Shan, Brian Curless, Yasutaka Furukawa, Carlos Hernández, and Steven M. Seitz. 2014. Occluding Contours for Multi-view Stereo. *Conference on Computer Vision and Pattern Recognition* (2014), 4002–4009.

Jan Stühmer, Stefan Gumhold, and Daniel Cremers. 2010. Real-time Dense Geometry from a Handheld Camera. *Proceedings of the 32Nd DAGM Conference on Pattern Recognition* (2010), 11–20.

Richard Szeliski. 2006. Locally Adapted Hierarchical Basis Preconditioning. *ACM Trans. Graph.* 25, 3 (2006), 1135–1143.

Chamara Saroj Weerasekera, Thanuja Dharmasiri, Ravi Garg, Tom Drummond, and Ian Reid. 2018. Just-in-Time Reconstruction: Inpainting Sparse Maps using Single View Depth Predictors as Priors. *arXiv preprint arXiv:1805.04239* (2018).

Guofeng Zhang, Jiaya Jia, Tien-Tsin Wong, and Hujun Bao. 2009. Consistent Depth Maps Recovery from a Video Sequence. *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* 31, 6 (2009), 974–988.

Yinda Zhang and Thomas Funkhouser. 2018. Deep Depth Completion of a Single RGB-D Image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 175–185.