Occlusion Reasoning for Temporal Interpolation using Optical Flow

Evan Herbst [*] Steve Seitz [*] Simon Baker [†]

September 12, 2009

Abstract

We present an optical-flow-based algorithm to smoothly interpolate between two images. We reason about the depth ordering of objects, and show how bidirectional flow can be used to reduce holes in the estimated flow at the interpolated time and perform occlusion reasoning. We develop a purely pixel-wise algorithm and then add spatial regularization. We evaluate our algorithm on the interpolation set of the Middlebury flow benchmark.

1 Introduction

We present an algorithm to use the optical flow between two images of a scene to interpolate between the two views. We use a recent algorithm as a base and add reasoning about the depth ordering of scene objects and about occlusions and disocclusions that occur during the interval between the times at which the two images were taken.

Occlusion reasoning when rerendering using two-frame stereo is straightforward because depth maps provide occlusion information, but is nontrivial when rerendering from only optical flow. We add occlusion reasoning and bidirectional flow to a flow-based algorithm for two-frame temporal video interpolation. Improvements in two-frame interpolation affect the quality of general view interpolation when we don't have 1-d motion or a perfectly still scene (i.e. a stereo problem). They are also useful in video retiming and slow-motion applications, and potentially in video editing for allowing artists to choose the ideal view in which to make edits that will then be propagated.

All of the algorithms we present have a structure (shown in fig. 1) that generalizes the algorithm given in the appendix of [1], which is based on *forward warping*, or *splatting*, the flow field. Given two frames and the flow fields in both directions between them, we first create the flow field at the intermediate time. We use this *interpolated flow field* to create *occlusion masks* specifying which pixels visible at the intermediate time are visible in each input frame. Then we blend the two input frames to create the interpolated frame.

This "naive algorithm" makes three major classes of error. Some pixels at the intermediate time aren't splatted to; we call these *holes*, and have to fill them somehow. The algorithm also doesn't consider the depth order of objects in the scene; the objects whose pixels are splatted from last will appear in front in the interpolated frame. Choosing an order in which to splat input pixels can't solve this problem. Thirdly, no occlusion reasoning is done, so that many pixels in the interpolated frame are a blend of two different objects from the two input frames. We use bidirectional splatting to greatly cut down on holes, color consistency between input frames to do *ordering reasoning*, and flow consistency checks between the input and interpolated frames to give us *occlusion reasoning*; the combination of these three techniques qualitatively improves the interpolated frames for many datasets. We further improve interpolation by doing spatial regularization over the estimated flow field and occlusion masks, and introduce a heuristic to improve ordering reasoning for most datasets.

We compare our results to those of the naive algorithm on the Middlebury interpolation evaluation sets using both a recent Black & Anandan implementation [12] and Seitz and Baker's filter flow [2]. Based

[*]University of Washington Computer Science Department
[†]Microsoft Research Redmond

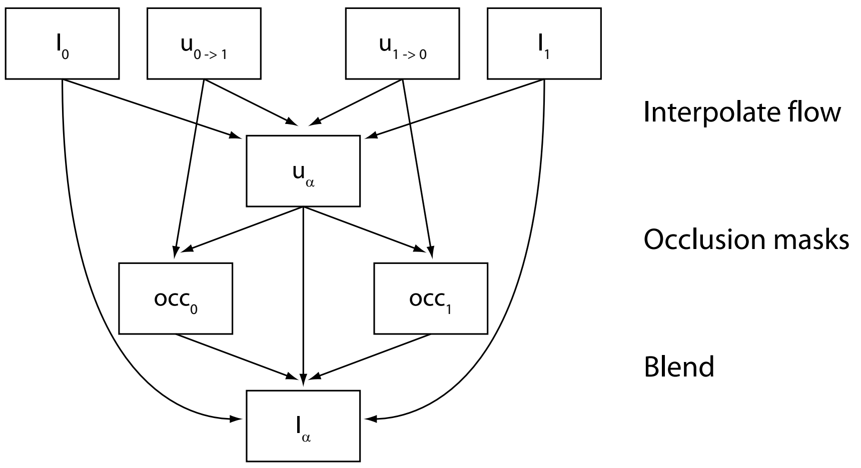


Figure 1: data flow through the three steps of our general approach. Input frames I_0 and I_1; forward and backward optical flows $\vec{u}_{0\to1}, \vec{u}_{1\to0}$; occlusion masks occ_0, occ_1; interpolated frame I_α.

on results on the Middlebury interpolation training sets, for which we have ground-truth flow in the forward direction although not the backward, we suspect our interpolation algorithm improves greatly with improving flow. In particular it works very well given ground-truth flow.

2 Related Work

Our algorithms use optical flow as the main source of information. The optical flow field between two images I, J of the same scene gives the 2-d motion of the elements of the scene as a function of their locations in I. That is, if the object seen at location (x, y) in I (and probably also at many other nearby pixel locations) is seen at $(x + 2, y - 3)$ in J, the optical flow $\vec{u}_I\ (x, y)$ is $(2, -3)$.

Many recent flow algorithms build on Horn and Schunck's influential paper [9] that iteratively finds approximate neighborhood-wide solutions to the now well-known *optical flow constraint equation* (OFCE) and regularizes between neighborhoods. Black and Anandan [4], one of the first of these, upgraded Horn and Schunck's approach with an outlier-robust model of flow vectors. Seitz and Baker [2] model the transformation between two images as a general linear filter that varies spatially; this approach models lighting, focus changes and other effects in addition to movement of scene elements. There have been a very large number of flow algorithms introduced recently; these two are the methods we use in this paper to provide input for our interpolation algorithms.

View interpolation is a very general problem and there has been work on a variety of specific instantiations of it. Chen and Williams [5] is one early approach to image-based view interpolation, the category of algorithms that warp regions of existing images to be shown in a synthesized view. Baker *et al.* [3] reconstruct a scene from two-view stereo by modeling it as a set of planar layers, as opposed to the most common model, a set of points in 3-d with topology determined in whatever way is most convenient. Zitnick *et al.* [13] model a scene captured by an array of eight cameras as a set of layers, and mattes where the edges of layers meet in the interpolated view. Fitzgibbon *et al.* [7] interpolate directly from the original (two or more) images, with little 3-d representation, and removes artifacts by constraining the local image color statistics of the interpolated view.

There has also been work on temporal video interpolation using flow. Ferreira *et al.* [6] solve, instead of the usual first-order OFCE, a more detailed second-order differential equation (derived in the same way as the OFCE) to produce a series of interpolated frames from two images. In the authors' words, this method works best for "sequences created through camera translation or translation of objects in an otherwise static scene". Baker *et al.* [1] present a simple two-frame algorithm based on *splatting*, following the flow from an original image to the intermediate time; we'll use this algorithm as our baseline. Bhat

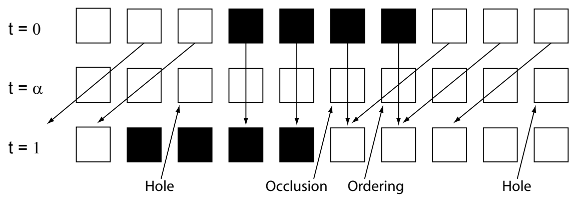


Figure 2: forward splatting demonstrated on a synthetic dataset. Top and bottom rows are I_0 and I_1. Arrows show the forward flow and which pixel locations it passes through at $t = \alpha$. Three types of problem: holes (intermediate-time pixels with no flows through them); ordering (intermediate-time pixels with multiple flows through them); occlusion (multiple flows to the same $t = 1$ location).

et al. [8] propagate detail from infrequent high-resolution frames to frequent low-resolution ones, adding resolution to existing images. Mahajan *et al.* [10] compute a coarse approximation to optical flow; the biggest difference between standard optical flow approaches and theirs is that "optical flow assumes that a point moves in a straight line[;] the path framework…instead assumes that all points/pixels passing through [a given spacetime location]…have the same flow". The approach of [10] is similar enough to ours that it would be informative to see quantitative results for their method.

3 Algorithms

We formulate the temporal interpolation problem as follows. Given two video frames I_0 and I_1 taken at times $t = 0$ and $t = 1$ respectively, produce an image I_α (the "interpolated frame") that could plausibly have been taken as part of the original video at $t = \alpha$, where $\alpha \in [0, 1]$. Each image I_k is composed of a set of pixel locations $\vec{p}_{kj}$. In addition to the input frames, we use the forward flow field $\vec{u}_0(\vec{x}), \vec{p} \in \{\vec{p}_{0j}\}$ and the backward flow field $\vec{u}_1(\vec{p}), \vec{p} \in \{\vec{p}_{1j}\}$, each of which is scaled so as to take pixels forward in time by one time unit. I.e., $\vec{u}_0(\cdot)$ gives the flow from $t = 0$ to $t = 1$ and $\vec{u}_1(\cdot)$ gives the negative of the flow from $t = 1$ to $t = 0$.

We assume perfect (non-noisy) input flow fields, although later we'll relax this requirement. We have to assume the motion between the two frames is linear so we can reasonably use the flow fields to interpolate point positions. This assumption is common to most two-frame temporal video interpolation methods.

We build on a very simple baseline algorithm using forward flow only and doing no occlusion reasoning, and add ordering reasoning and occlusion reasoning. We then incorporate spatial regularization over the interpolated flow and occlusion masks to further improve results.

3.1 A Pixelwise Algorithm

3.1.1 Baseline

We start with the simple algorithm given in the appendix of [1], which is based on *forward warping*, or *splatting*, the flow field. This "naive splatting" algorithm uses the *forward flow* field from I_0 to I_1, which is assumed to be correct, and produces first the flow field at the intermediate time, next the two occlusion masks for the interpolated frame with respect to the original frames, and then the interpolated frame. The occlusion mask for the interpolated frame with respect to original frame i gives the visibility of each interpolated-frame pixel in the original frame. I.e., the mask is 0 at $\vec{p}_{\alpha j}$ iff the scene element visible at $\vec{p}_{\alpha j}$ is occluded in I_i. All the algorithms we describe in this paper produce these same images in this order.

We demonstrate the naive algorithm on a simple synthetic example dataset shown in fig. 2. Here, and in all examples in this paper, $\alpha = 0.5$ for simplicity.

Each flow vector passes near one or more pixels at $t = \alpha$. At each intermediate-time pixel we choose one of these *candidate flows* to assign to the interpolated flow field. Intermediate-time pixels with no

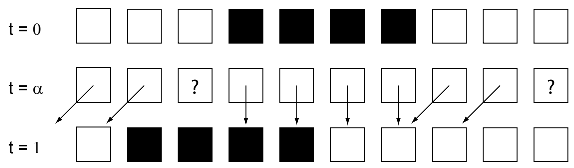


Figure 3: one interpolated flow field we might create for the example in fig. 2, shown pointing forward in time from $t = \alpha$ but meant to be valid for the entire interval $t \in [0, 1]$.

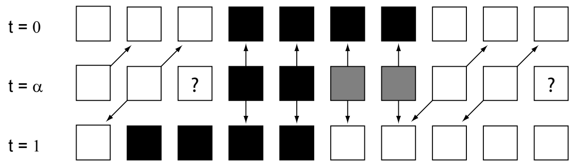


Figure 4: to blend, follow the interpolated flow backward and forward and average the resulting colors.

nearby flows are called *holes* and will be discussed later; for now we don't specify flows for them. One possible interpolated flow field is shown in fig. 3.

Now we want to create the interpolated frame. We use Boolean *occlusion masks* M_0, M_1 that tell us which input frames each intermediate-time pixel is visible in: each pixel at $t = \alpha$ is set to the average (weighted appropriately for α) of the corresponding pixels in the one or two input frames in which it's visible. One simple way to create occlusion masks is to assume the whole scene is visible in both input frames, i.e. set both masks to 1 everywhere. Fig. 4 shows the interpolated flows we follow during blending, assuming such *trivial occlusion masks*, and the resulting interpolated pixels.

3.1.2 Problems with the Baseline

Fig. 2 shows three classes of artifact that we'll fix with improvements to the algorithm. Firstly, the baseline algorithm does no reasoning about the depth ordering of regions: at intermediate-time pixels with multiple flow candidates, it picks one arbitrarily. Secondly, splatting leaves holes in the interpolated flow field. Even given non-noisy flow, holes occur at the intermediate time because optical flow isn't a bijection between pixels at its start and end times; occlusions, disocclusions, expansions and contractions of objects all cause holes. We can lessen the problem by splatting each source pixel to multiple target pixels, but unless we use an arbitrarily large *splatting radius* we might see holes anyway. The heuristic we use to remove them is to iteratively set the unsplatted-to target pixel with the most splatted-to neighbors to the average of its neighbors (this is the heuristic used in [1]). Thirdly, the naive algorithm doesn't reason about occlusion: blending averages both input frames regardless of which objects are visible in each. We'll introduce three algorithmic improvements the combination of which will solve these three problems.

3.1.3 Ordering Reasoning

Ordering reasoning infers the order of the depths of objects from the camera. Stereo algorithms assume a rigid scene and so can map flow directly to depth. When we're working with a general flow field, ordering isn't so straightforward.

Splatting generates a set of "candidate" flows for each $\vec{p}_{\alpha i}$, one from each source pixel that flows to or near $\vec{p}_{\alpha i}$. There can be any number of candidates at a target pixel; holes are simply pixels with no candidates. We choose to handle multiple candidate flows by copying one to the interpolated flow field and ignoring the others. During splatting, we follow each proposed flow at $\vec{p}_{\alpha i}$ to $t = 0$ and 1, and rank candidate flows by the similarity of the colors at the corresponding locations in I_0 and I_1, bilinearly interpolating intensities. In fig. 2 ("ordering"), this is how we know how the right edge of the black object

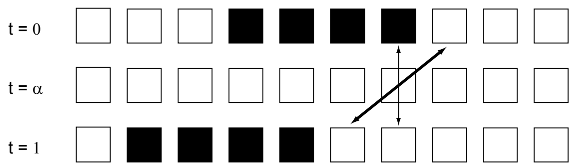


Figure 5: ordering reasoning on the example of fig. 2. At a given intermediate-time pixel, follow each candidate flow forward and backward. Here the diagonal flow will be chosen as the more color-consistent.

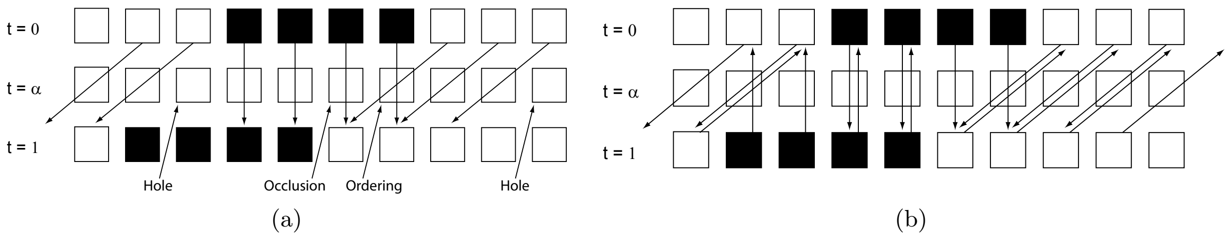


Figure 6: bidirectional splatting reduces holes. (a) forward splatting (fig. 2 repeated); (b) bidirectional splatting.

moves over time. Fig. 5 shows the candidate flows (from the flow field of fig. 2) for a particular pixel. At each $\vec{p}_{\alpha i}$ we choose the candidate that's the most color-consistent between I_0 and I_1. Given relatively small movement between the input images, this color consistency criterion should correctly handle all occlusions (and disocclusions, once we use bidirectional flow; thus we also correctly capture the disoccluding edge at the left of fig. 2).

Many pixels at the interpolated time flow out of image bounds at $t = 0$ or 1; for purposes of comparison with other color consistencies, we set the color consistency of these flows (i.e., the Euclidean color-vector distance between the hypothesized values at $t = 0$ and $t = 1$) to a constant ρ. For comparisons in RGB space we let $\rho \approx 5$. Since each value in the interpolated flow field is copied from one of the input flow fields, each pixel at $t = \alpha$ is in image bounds at at least one of $t = 0$ and $t = 1$, so this is the only special case to be handled.

3.1.4 Bidirectional Splatting

Splatting using only the forward flow leaves holes in the interpolated flow field. Heuristic hole filling does remove the holes, but not well. The forward and backward flow fields carry different information, so the performance of naive splatting in one direction doesn't relate to its performance using the other flow field. For any dataset on which forward splatting substantially outperforms backward splatting, if the dataset were horizontally flipped the reverse would be true. Heuristic hole filling doesn't work well in most cases; a better solution is to use both flow fields and not ignore any source of information.

As can be seen in fig. 6, bidirectional splatting avoids most of the holes in the interpolated flow field that we see with forward splatting.

3.1.5 Occlusion Reasoning

All of our algorithms blend the input frames using occlusion masks that give the visibility of scene elements at $t = 0$ and 1. The naive algorithm sets the masks to 1 everywhere; we can do better by using bidirectional flow information.

We make the assumption that each pixel at $t = \alpha$ is visible in at least one input frame, which isn't always true. I.e., there are in general holes even after bidirectional splatting. The sources of information we use in this paper aren't enough to tell us anything about these pixels, so ignore them for now. Then at each pixel at $t = \alpha$, at least one of the two occlusion masks must be 1.

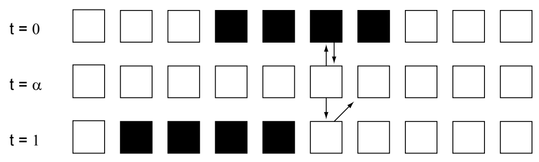


Figure 7: occlusion reasoning on the example of fig. 2. Follow the chosen flow forward and backward from $t = \alpha$ and compare to the flows at those locations at $t = 0$ and 1. We determine this pixel to be visible in I_0 and possibly in I_1.

At each intermediate-time pixel $\overrightarrow{p}_{\alpha i}$ we can follow the previously chosen flow $\overrightarrow{u}_\alpha(\overrightarrow{p}_{\alpha i})$ forward and backward in time to locations $\overrightarrow{p}_{0\leftarrow\alpha i}$ and $\overrightarrow{p}_{1\leftarrow\alpha i}$ in the two input frames respectively. We can then compare the flow at $\overrightarrow{p}_{\alpha i}$ to the flows we find at $\overrightarrow{p}_{0\leftarrow\alpha i}$ and $\overrightarrow{p}_{1\leftarrow\alpha i}$. Define

$$\begin{aligned} d_0(i) &\equiv -\alpha\, \overrightarrow{u}_\alpha(\overrightarrow{p}_{\alpha i}) + \alpha\, \overrightarrow{u}_0(\overrightarrow{p}_{0\leftarrow\alpha i}), \\ d_1(i) &\equiv (1-\alpha)\, \overrightarrow{u}_\alpha(\overrightarrow{p}_{\alpha i}) - (1-\alpha)\, \overrightarrow{u}_1(\overrightarrow{p}_{1\leftarrow\alpha i}) \end{aligned}$$

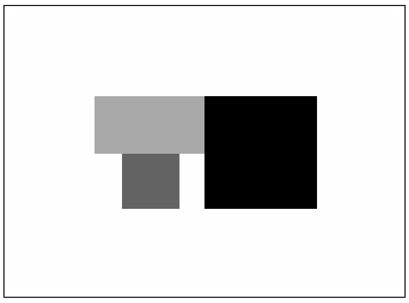
with the values of $\overrightarrow{u}_0(\cdot), \overrightarrow{u}_1(\cdot)$ interpolated bilinearly. In fig. 7, follow the arrows up and down from the intermediate-time location, then follow the input flows at the resulting locations back to $t = \alpha$; d_0, d_1 are the distances from our starting loction at which we end up. In this example $d_0 < d_1$. We denote $-d_0(i), -d_1(i)$ the *flow consistencies* of $\overrightarrow{p}_{\alpha i}$ wrt I_0 and I_1. We decide that $\overrightarrow{p}_{\alpha i}$ is always visible in the input frame with respect to which $\overrightarrow{p}_{\alpha i}$ has higher flow consistency. I.e., if $d_0(i) < d_1(i)$, we set the visibility flag for $\overrightarrow{p}_{\alpha i}$ wrt I_0 to 1. We set the other visibility flag at $\overrightarrow{p}_{\alpha i}$ to 1 if the larger of $d_0(i), d_1(i)$ is less than a constant threshold υ. The threshold is very dependent on the quality of the input flow fields. For good input flow fields taken from a manual annotation tool, we use $\upsilon = 1.0$; for Black & Anandan flows it needs to be larger.

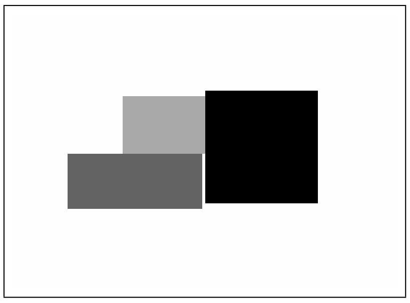
In fig. 9 we can see that occlusion reasoning removes a great deal of ghosting: each of the affected pixels is no longer being set to an average of either the background of I_0 and the foreground of I_1 or the foreground of I_0 and the background of I_1.

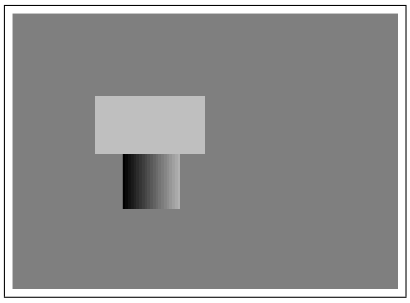
This is essentially the same occlusion reasoning done by [10], the difference being that they follow the flow linearly from $t = \alpha$ to $t = 0$ or 1 despite their flow model being nonlinear, whereas we explicitly assume all pixels flow linearly in $t \in [0, 1]$.

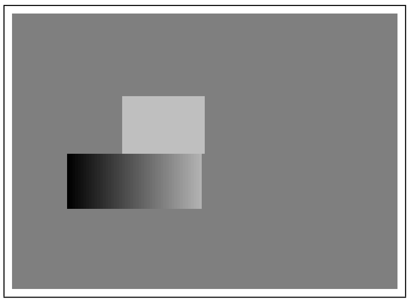
3.1.6 A Synthetic Example

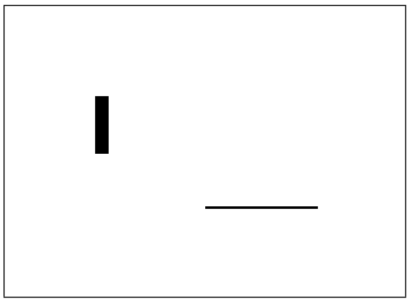
To illustrate our algorithms we introduce a simple synthetic dataset (fig. 8). The true motion is as follows: the light gray object moves right, the dark gray object moves left and expands in the x direction, and the black object moves up slightly. X and y dimensions of flows are shown in separate images. The lighter the color, the greater the flow; zero shows as medium gray. To simplify the presentation, all examples in this paper will use $\alpha = 0.5$. Fig. 9 gives various sub-results of a few algorithms on this dataset, showing that each algorithmic improvement we have introduced does visually improve rendering.

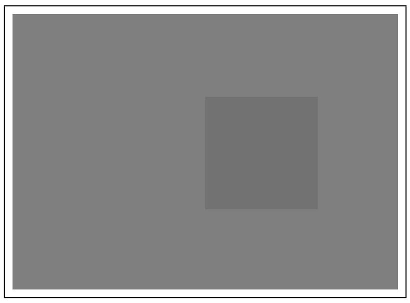

(a) I_0

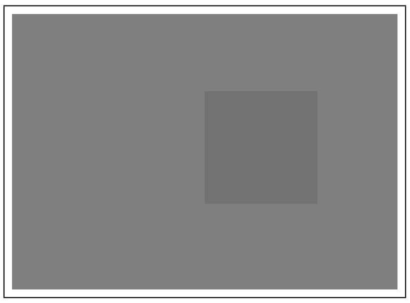

(b) I_1

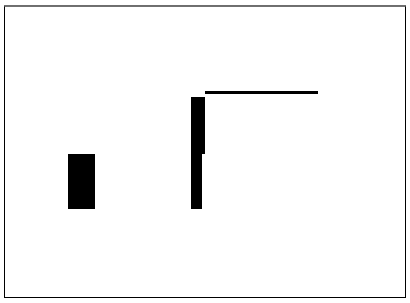

(c) forward flow $\overrightarrow{u}_0$, x direction

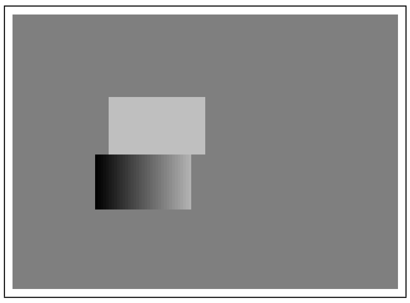

(d) backward flow $\overrightarrow{u}_1$, x direction

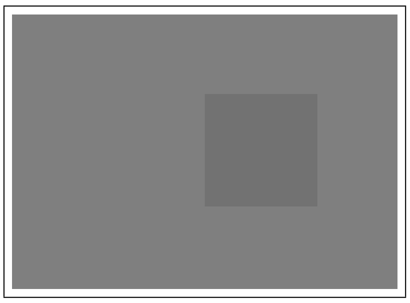

(e) occlusion mask M_0


(f) forward flow $\overrightarrow{u}_0$, y direction


(g) backward flow $\overrightarrow{u}_1$, y direction


(h) occlusion mask M_1


(i) interpolated flow $\overrightarrow{u}_\alpha$, x direction


(j) interpolated flow $\overrightarrow{u}_\alpha$, y direction

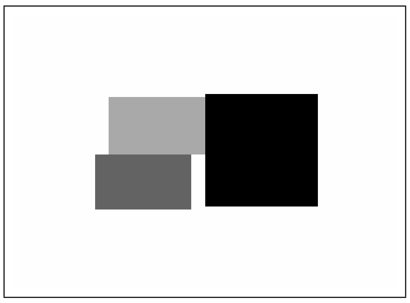

(k) interpolated frame I_α

Figure 8: a synthetic example dataset. True motion: the light gray object moves right; the black object moves up; the dark gray object moves left and expands horizontally.

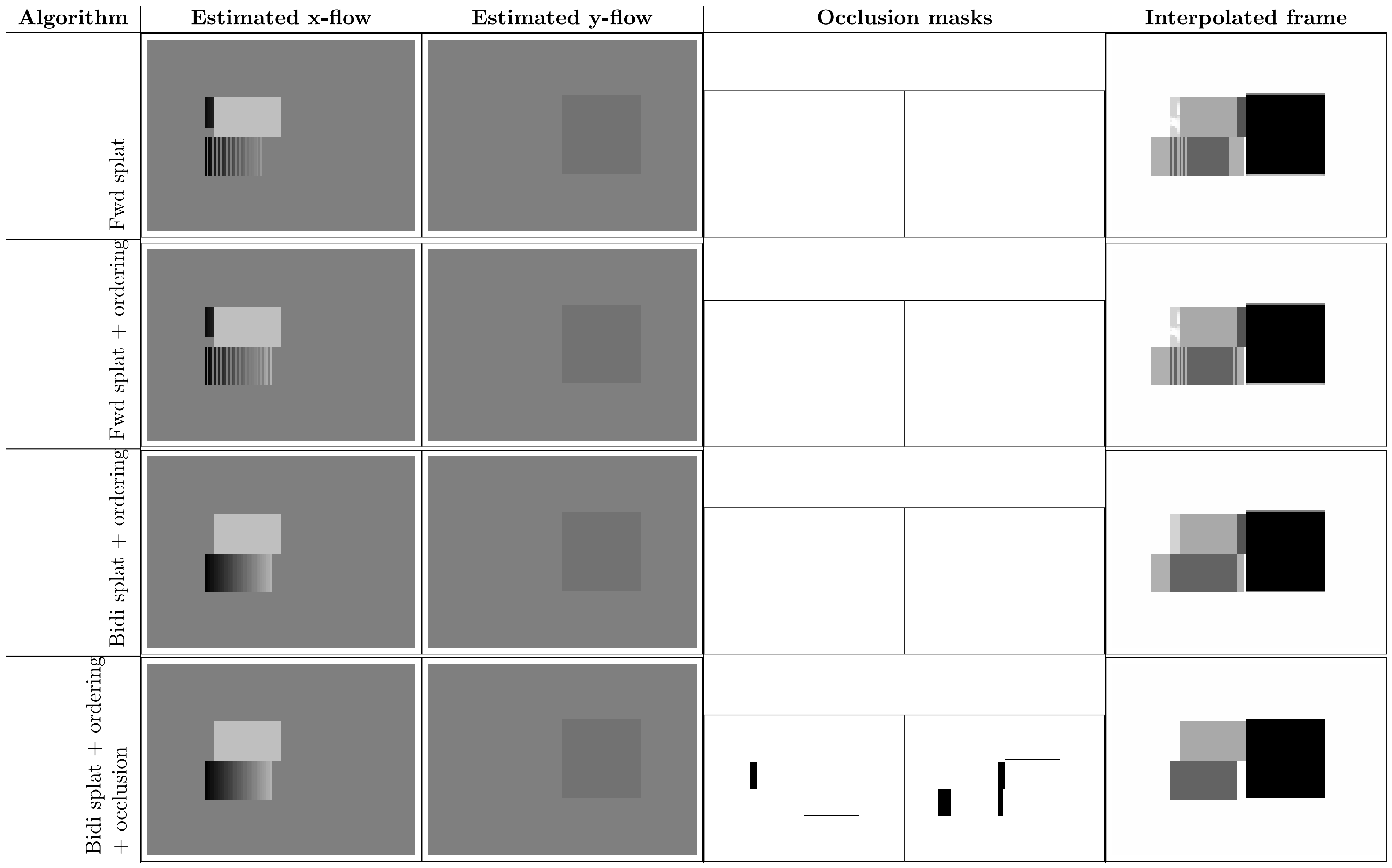


Figure 9: various algorithms applied to the dataset of fig. 8. The final non-regularized algorithm makes no errors on this example. Compare the occlusion masks and interpolated frames to the true ones in fig. 8.

3.2 Refinements

3.2.1 Spatial Regularization

So far our algorithms treat each pixel independently, but in the presence of noise in the input images and/or the flow, it's important to incorporate spatial regularization. We use it during the estimation of the interpolated flow field and of the occlusion masks.

Above we determined the interpolated flow field and occlusion masks one pixel at a time. In the case of the flow field, we choose one of the *candidate flows* for each pixel. Then we assign one of three combinations of visibility flags to each pixel. Both these procedures are specializations of the max-product algorithm over a 4-connected Markov random field whose nodes correspond to pixel locations at $t = \alpha$. We refer to candidate flows with the symbol $\mathcal{C}$.

The flow data term is a function of color consistency:

$$E_D^{flow}(\vec{p}_{\alpha i} \mapsto \mathcal{C}) = \left| I_0(\vec{p}_{\alpha i} - \alpha \vec{u}(\mathcal{C}, \vec{p}_{\alpha i})) - I_1(\vec{p}_{\alpha i} + (1-\alpha)\vec{u}(\mathcal{C}, \vec{p}_{\alpha i})) \right|^2$$

The data term for the occlusion-mask MRF includes a single term, measuring back-and-forth flow consistency. First define a "single-frame penalty" for each of I_0 and I_1 with respect to $\vec{p}_{\alpha i}$:

$$\begin{aligned}
\vec{p}_{0i} &\equiv \vec{p}_{\alpha i} - \alpha \vec{u}_\alpha(\vec{p}_{\alpha i}) \\
\vec{p}_{1i} &\equiv \vec{p}_{\alpha i} + (1-\alpha) \vec{u}_\alpha(\vec{p}_{\alpha i}) \\
penalty_0(\vec{p}_{\alpha i}) &= \left| \vec{u}_0(\vec{p}_{0i}) - \vec{u}_\alpha(\vec{p}_{\alpha i}) \right| \\
penalty_1(\vec{p}_{\alpha i}) &= \left| \vec{u}_1(\vec{p}_{1i}) - \vec{u}_\alpha(\vec{p}_{\alpha i}) \right|
\end{aligned}$$

As earlier, $\vec{u}_0(\cdot)$ and $\vec{u}_1(\cdot)$ are bilinearly interpolated. Now we can calculate the data energy. $o_k \in \{0, 1\}$ is the visibility flag for the intermediate-time pixel with respect to input frame k.

$$E_D^{occ}(\vec{p}_{\alpha i}, o_0, o_1) = \begin{cases} penalty_0(\vec{p}_{\alpha i}) - penalty_1(\vec{p}_{\alpha i}), & o_0 = 1 \wedge o_1 = 0 \\ penalty_1(\vec{p}_{\alpha i}) - penalty_0(\vec{p}_{\alpha i}), & o_0 = 0 \wedge o_1 = 1 \\ penalty_0(\vec{p}_{\alpha i}) + penalty_1(\vec{p}_{\alpha i}) - \eta, & o_0 = 1 \wedge o_1 = 1 \end{cases}$$

Here η parameterizes our prior preference for no occlusion ($o_0 = o_1 = 1$) at any pixel. This preference declines with increasing difference between $penalty_0(\vec{p}_\alpha)$ and $penalty_1(\vec{p}_\alpha)$.

The smoothness energy for the flow optimization penalizes dissimilarity of flow between the two intermediate-time pixels as well as dissimilarity of the colors of the corresponding input-frame pixels:

$$\begin{aligned}
rgb_0(\vec{p}_\alpha, \mathcal{C}) &\equiv I_0(\vec{p}_\alpha - \alpha \vec{u}(\mathcal{C}, \vec{p}_\alpha)) \\
rgb_1(\vec{p}_\alpha, \mathcal{C}) &\equiv I_1(\vec{p}_\alpha + (1-\alpha) \vec{u}(\mathcal{C}, \vec{p}_\alpha)) \\
E_S^{flow}(\vec{p}_{\alpha i} \mapsto \mathcal{C}_1, \vec{p}_{\alpha j} \mapsto \mathcal{C}_2) &= \phi_{fs}(\left| \vec{u}(\mathcal{C}_1, \vec{p}_{\alpha i}) - \vec{u}(\mathcal{C}_2, \vec{p}_{\alpha j}) \right|) \\
&\quad + \phi_{cs}(\left| rgb_0(\vec{p}_{\alpha i}, \mathcal{C}_1) - rgb_0(\vec{p}_{\alpha j}, \mathcal{C}_2) \right|) + \phi_{cs}(\left| rgb_1(\vec{p}_{\alpha i}, \mathcal{C}_1) - rgb_1(\vec{p}_{\alpha j}, \mathcal{C}_2) \right|),
\end{aligned}$$

where we define the flow smoothness penalty $\phi_{fs}(x) = \phi(x, 0.5)$ and the color smoothness penalty $\phi_{cs}(x) = \phi(x, 4)$ based on the robust penalty function $\phi(x, \sigma) = \frac{-1}{1+(\frac{x}{\sigma})^2}$ of [4].

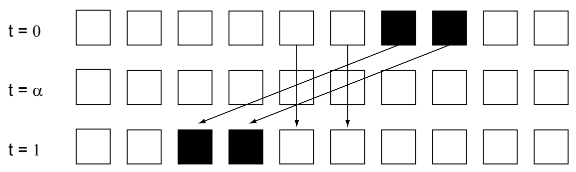


Figure 10: a small example dataset demonstrating temporal aliasing. Forward flow is shown with arrows. Both a black and a white region move through the two center pixels at $t = \alpha$. Colors and flows for other pixels are irrelevant and so not shown.

The occlusion-mask smoothness energy measures the similarity of the interpolated-frame pixels resulting from choosing particular masks:

$$rgb_\alpha(\vec{p}_\alpha, \mathcal{C}, o_0, o_1) \equiv o_0 rgb_0(\vec{p}_\alpha, \mathcal{C}) + o_1 rgb_1(\vec{p}_\alpha, \mathcal{C})$$

$$E_S^{occ}(\vec{p}_{\alpha i} \mapsto (\mathcal{C}_1, o_{0i}, o_{1i}), \vec{p}_{\alpha j} \mapsto (\mathcal{C}_2, o_{0j}, o_{1j})) = \phi_{cs}(\left| rgb_\alpha(\vec{p}_{\alpha i}, o_{0i}, o_{1i}) - rgb_\alpha(\vec{p}_{\alpha j}, o_{0j}, o_{1j}) \right|)$$

Recall that at this point $\mathcal{C}_1, \mathcal{C}_2$ are fixed.

We find empirically that replacing this with the Potts energy gains a bit of efficiency at the cost of a bit of accuracy.

As the MRF formulation for creating the intermediate flow explicitly chooses one of a set of candidates at each pixel, it makes sense to generate multiple candidates even at hole pixels. We do this by changing the operation performed at each hole pixel during hole filling from an average of neighboring values to a set union. This leads to very large numbers of candidates at some pixels in large hole areas, so after generating all candidates at a pixel we cluster them on the 2-d flow values. Parameter setting for clustering can be constrained by deciding that hole pixels shouldn't end up with more candidates than non-holes, which come by their candidates "naturally".

3.2.2 Temporal Aliasing

Fig. 10 demonstrates a problem for ordering reasoning that occurs on a great many datasets. The middle region in this example may be incorrectly rendered due to *temporal aliasing*, which occurs when two or more regions each of which is visible in each input frame pass through the same area at $t = \alpha$. From low-level cues alone, such as the cues we use in this paper, it isn't possible to decide which region is visible at the intermediate time. Temporal aliasing is common in our datasets. Since we're committed to using only two frames of video and no high-level scene understanding, we use a heuristic to alleviate inconsistent rendering decisions. We know that in stereo and most flow datasets, usually the faster-moving object of any two is closer to the camera. Therefore we add to the flow-field data energy a term to reward large flows:

$$E_D^{flow}(\vec{p}_{\alpha i}, \mathcal{C}) = \left| I_0(\vec{p}_{\alpha i} - \alpha \vec{u}(\mathcal{C}, \vec{p}_{\alpha i})) - I_1(\vec{p}_{\alpha i} + (1-\alpha)\vec{u}(\mathcal{C}, \vec{p}_{\alpha i})) \right|^2 + \nu \left| \vec{u}(\mathcal{C}, \vec{p}_{\alpha i}) \right|$$

3.3 Summary

The final algorithm resembles the following, starting with the two input frames and given α:

- Compute, with your choice of algorithm, forward and backward flow between I_0 and I_1.
- Splat the forward flow from $t = 0$ to $t = \alpha$ and the backward flow from $t = 1$ to $t = \alpha$ to get candidate flows at each pixel.

- Run inference over an MRF defined by $E_D^{flow}(\cdot)$ and $E_S^{flow}(\cdot)$ to choose which candidates become the interpolated flow field.
- Run inference over an MRF defined by $E_D^{occ}(\cdot)$ and $E_S^{occ}(\cdot)$ to get the occlusion masks occ_0 and occ_1.
- Using the interpolated flow and occlusion masks, create the final frame via the blending algorithm we discuss above, a more sophisticated Poisson blending algorithm, or your favorite algorithm.

4 Experiments

So far we've shown results with ground-truth flow on a synthetic dataset and with more or less ground-truth flow on a stereo dataset. Now we compare the baseline and multiple improved algorithms on the cones and teddy Middlebury stereo datasets and show some results with existing flow algorithms on real data (both stereo and general flow). Recall that our baseline algorithm is the one currently used by the Middlebury flow evaluation framework [11]; by comparing our results to it we're attempting to improve the quality of that evaluation.

The error measure we use is the same interpolation error used by the Middlebury flow evaluation. We take the root mean square Euclidean distance in some color space over all pixels in the interpolated frame:

$$Err(I_\alpha, I_\alpha^{true}) \equiv \sqrt{\frac{\left(\sum_{\vec{p}} \left| I_\alpha(\vec{p}) - I_\alpha^{true}(\vec{p}) \right|^2\right)}{3}}$$

The factor of 3 makes the measure interpretable in terms of a single color channel. The most obvious candidate color space is YIQ, which was designed so that Euclidean distance in YIQ is very similar to human-perceived color distance. RGB distance is more or less proportional to YIQ distance (although it isn't always even monotonic), so we use RGB distance here and at all other points in the algorithm when color distance is required.

4.1 Combating Temporal Aliasing

Fig. 11 shows that our heuristic (section 3.2.2) does improve consistency at least on stereo datasets. The input flow used here is the ground truth, with the (few) holes filled with the values at the corresponding pixels of the filter flow output.

4.2 Regularization on Real Data

Adding spatial regularization of the interpolated flow field and the occlusion masks improves results on the Middlebury cones and teddy stereo datasets, as shown in figs. 12 and 13. Fig. 11 showed that even regularization isn't able to deal with temporal aliasing without heuristics, but it does solve a number of other problems. Compare the second and fourth rows of figs. 12 and 13 for noticeable effects: in cones, regularization smooths the edges of the cones (they should be flat) and removes some of the ghosting of the tongue depressors, and in teddy it smooths the teddy's left arm. It also reduces interpolation error. The input flow used here is created the same way as in fig. 11: ground-truth flow with holes filled using the filter flow.

We do still see noticeable artifacts in both datasets. In cones, the tongue depressors are still doubled. In teddy, the birdhouse chimney and the right side of the teddy show ghosting, and there's a large gray spot between the birdhouse and the blue backdrop just above the stuffed animal. The diamond-shaped tongue-depressor ghost is unavoidable because in neither input frame do we see the patch of cloth that's behind that area. The gray spot is an area in which the ground-truth flow has a hole and the filter flow is incorrect. The ghosting in teddy occurs because the "ground-truth" flow tracks the outlines of the teddy

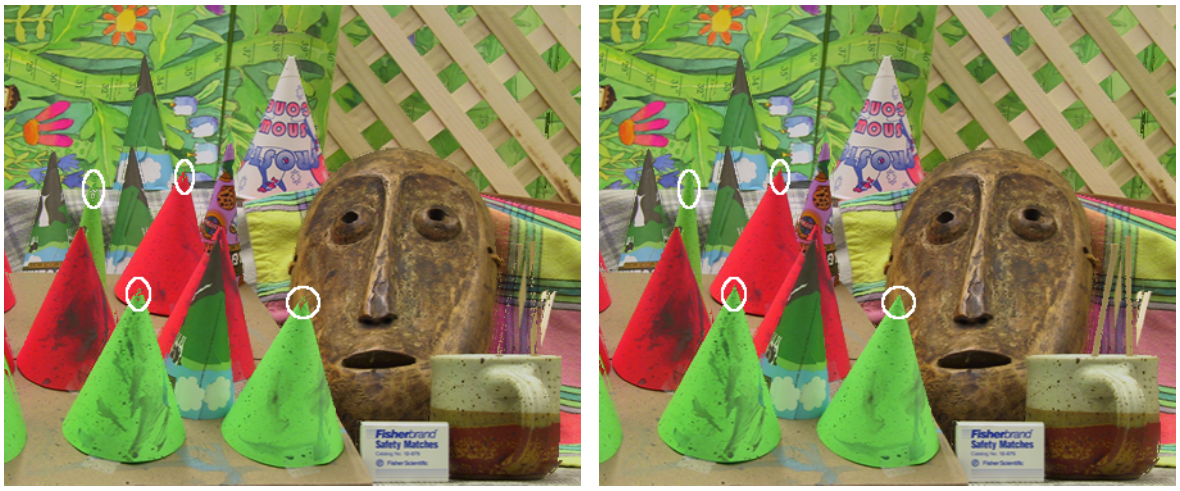


Figure 11: combating temporal aliasing in the Middlebury cones dataset. Interpolated frame without (a) and with (b) the prefer-large-flows heuristic. Interesting regions are circled.

and the birdhouse only to an accuracy of about a pixel; it's hard to define the outline of an object whose edges are blurred over a band multiple pixels wide.

4.3 Impact of Flow Algorithm

The quality of the input flow greatly affects the rendering quality. On the cones and teddy datasets (figs. 16 and 17), our algorithm improves on the naive algorithm much more with ground-truth flow than when given Black & Anandan or filter flow (table 1). In fact, interpolation error goes up when we apply our algorithm to Black & Anandan input flow. The less accurately the flow tells us how the boundaries of objects move, the less it makes sense to reason on a small scale about occlusions and disocclusions. The naive algorithm does well on low-quality flow because it doesn't try to extract too much information out of the flow.

Filter flow is usually more accurate than Black & Anandan, and is less smoothed. The lack of smoothness is sometimes helpful and sometimes unhelpful: we desire high-quality flow, but smooth flow helps prevent regularization from introducing unrealistically sharp boundaries.

The interpolation-error images in figs. 16 and 17 are normalized such that all pixels within the same table are comparable. Brightness is directly proportional to error. Fig. 14 gives the ground-truth interpolated frames for comparison.

Fig. 15 shows the forward-direction input flows used for the cones dataset, in order to compare the accuracy of the algorithms we use for flow. In particular Black & Anandan does very badly, so the interpolation error numbers for the datasets using that input flow probably aren't as meaningful as the others.

4.4 Videos

Since increasing video frame rate is a major application of our work, we include the results of generating views for multiple α values. Videos for the cones and teddy datasets, with $\alpha \in [0 : .1 : 1]$, for three interpolation methods, can be found at `http://grail.cs.washington.edu/projects/tvif/`. We have not incorporated temporal regularization because the naive way of connecting pixels (in a 6-neighbor grid) leads to blocky results and it's not obvious how to connect pixels along "the direction of the flow" when the flow is concurrently being optimized.

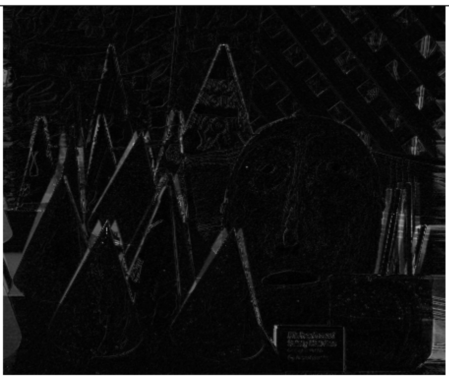
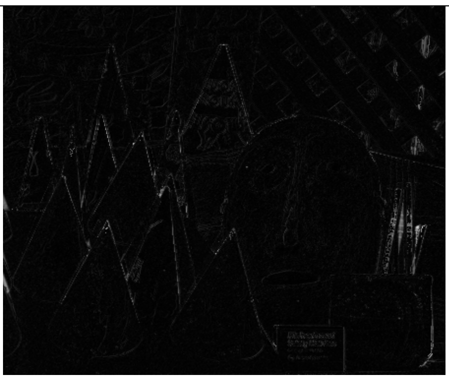
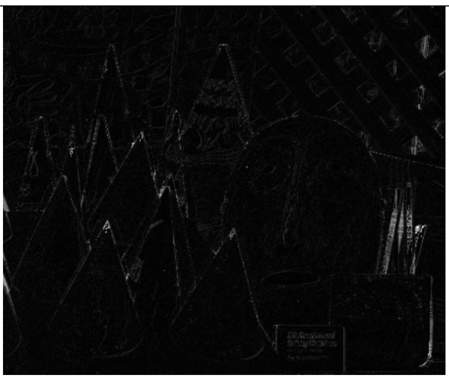
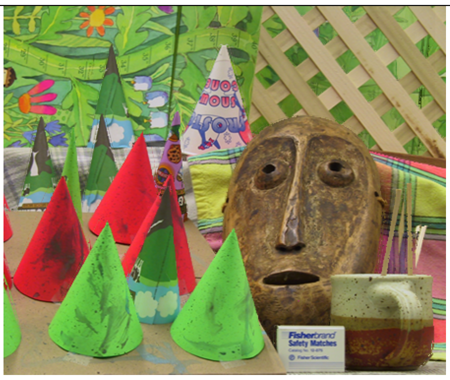
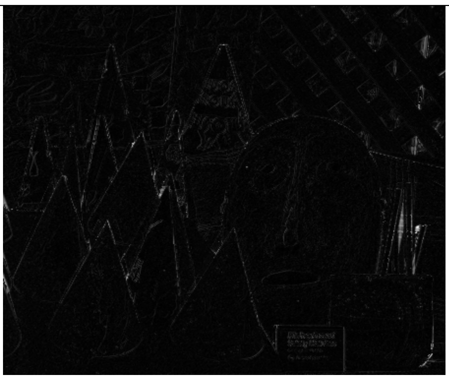
Interpolation	Interpolated frame	Error
Naive		 12.95
FNR		 10.76
Reg		 10.92
RegTA		 9.85

Figure 12: interpolated frames and interpolation error for various algorithms on the Middlebury Cones dataset with hole-filled ground-truth flow. FNR = final non-regularized; Reg = regularized without temporal-aliasing heuristic; RegTA = regularized with temporal-aliasing heuristic.

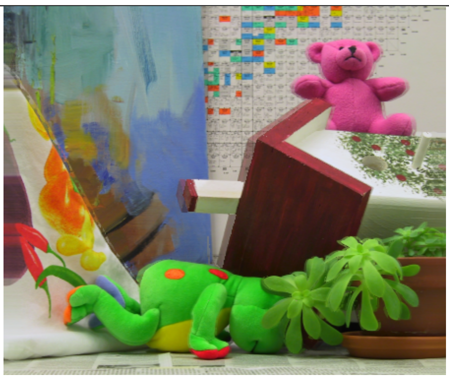
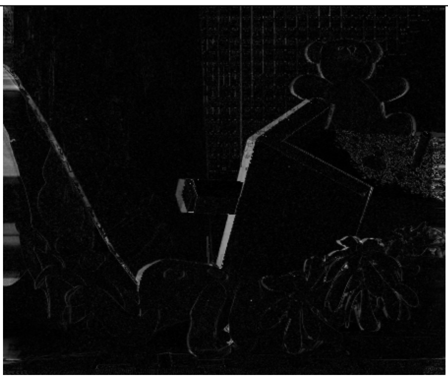
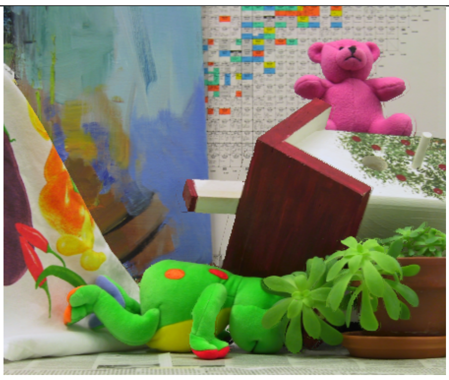
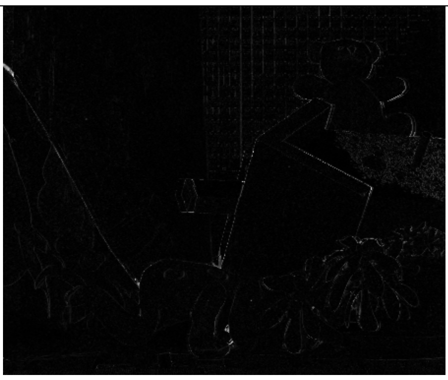
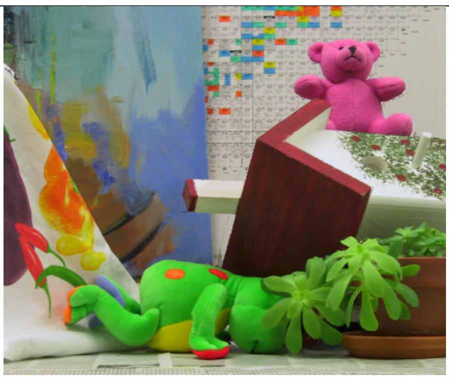
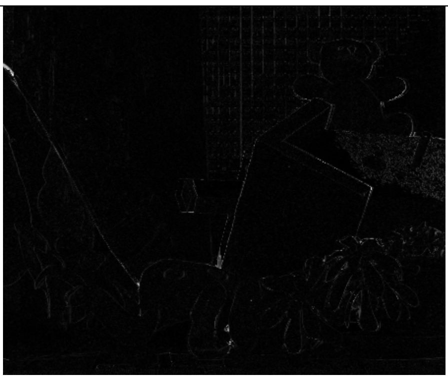
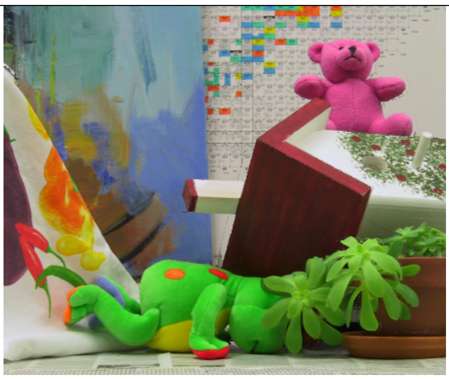
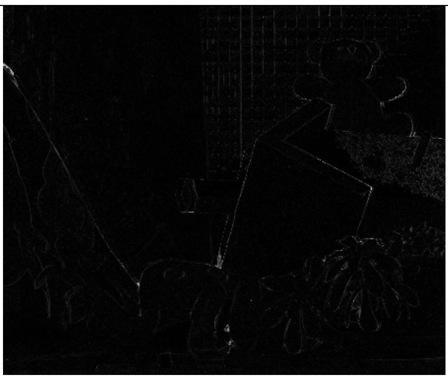
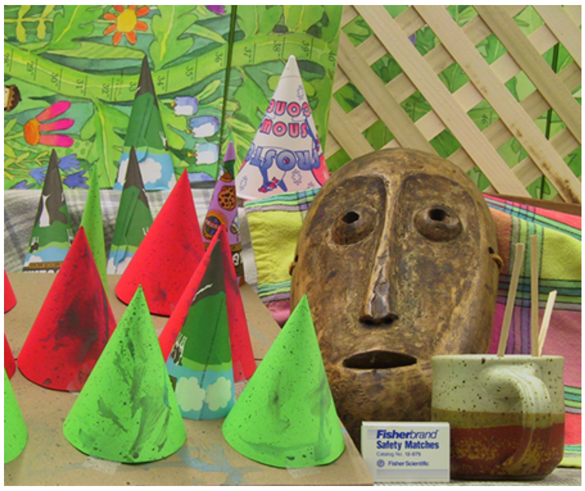
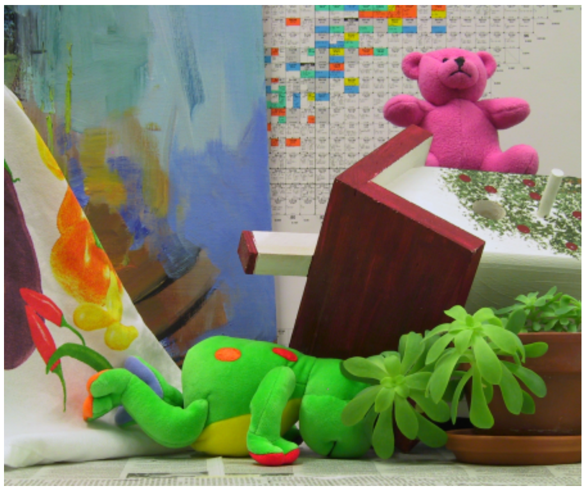
Interpolation	Interpolated frame	Error
Naive		 9.32
FNR		 6.66
Reg		 6.40
RegTA		 6.25

Figure 13: interpolated frames and interpolation error for various algorithms on the Middlebury Teddy dataset with hole-filled ground-truth flow. FNR = final non-regularized; Reg = regularized without temporal-aliasing heuristic; RegTA = regularized with temporal-aliasing heuristic.


(a)



(b)

Figure 14: true interpolated frames for the Middlebury cones and teddy datasets.

Table 1: average per-pixel interpolation error on the Middlebury cones and teddy datasets as a function of input flow.

Dataset	**Interpolation**	**Flow**		
		Ground truth	Filter flow	Black & Anandan
Cones	Naive	12.95	13.30	13.09
	Final non-regularized	10.76	12.71	13.73
	Regularized	9.85	12.31	13.71
Teddy	Naive	9.32	8.71	21.46
	Final non-regularized	6.66	7.53	25.21
	Regularized	6.25	7.34	24.15

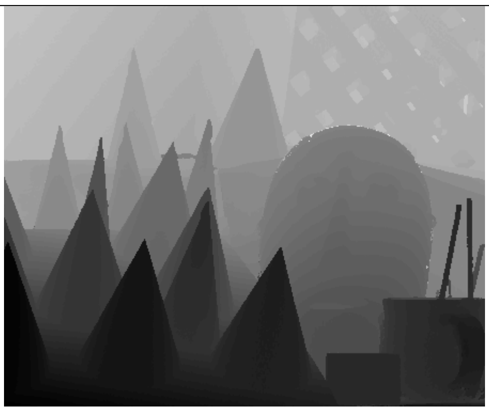
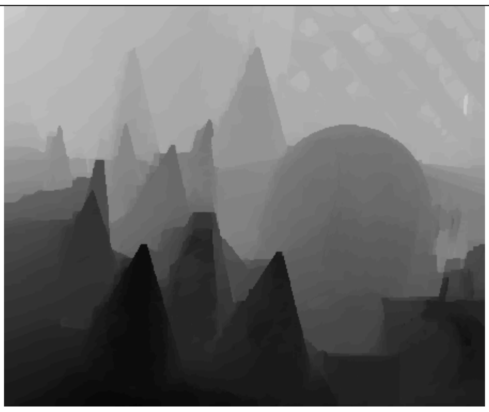
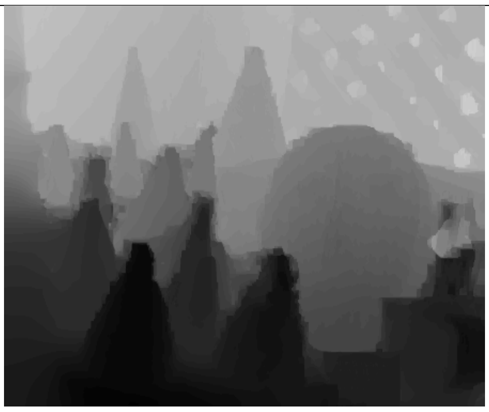
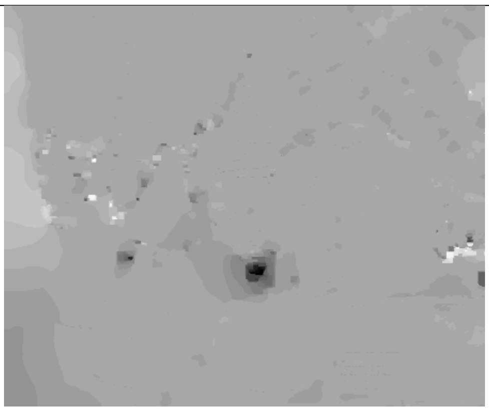
Flow algorithm	Forward x-flow	Forward y-flow
(Ground truth)		
Filter flow		
Black & Anandan		

Figure 15: forward flows used as input on the Middlebury cones dataset. These are stereo datasets: the true y-flow is zero.

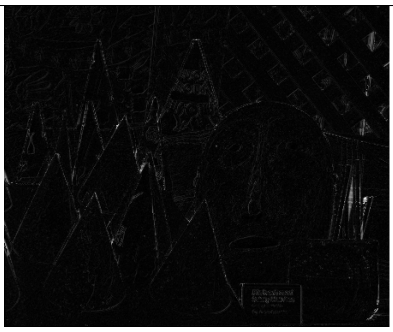
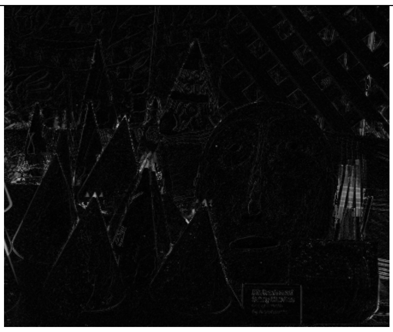
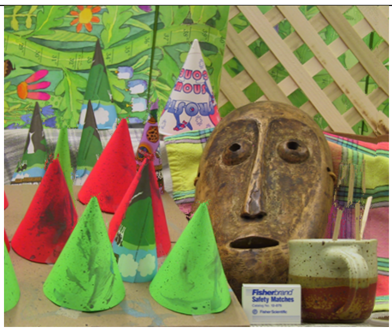
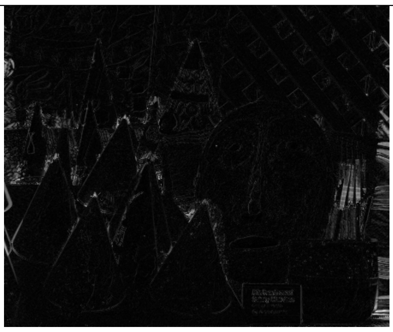
Input flow	Interpolated frame	Interpolation error
Ground truth		
Filter flow		
Black & Anandan		

Figure 16: interpolated frames and interpolation error of regularized algorithm on the Middlebury cones dataset with varying input flow.

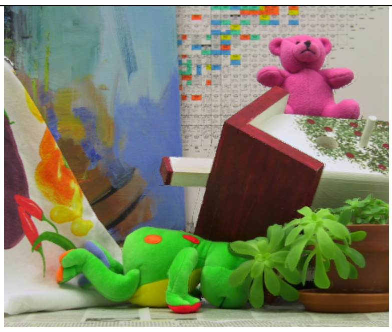
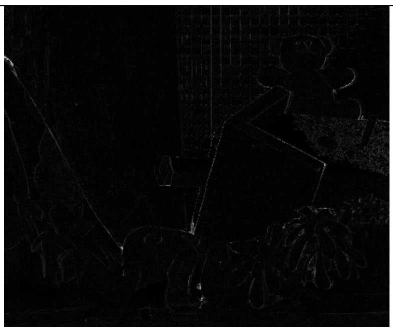
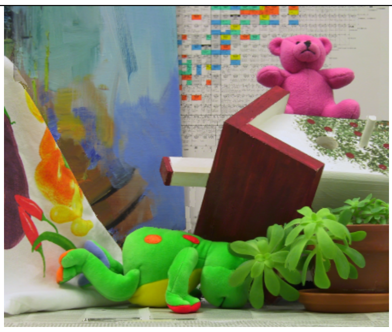
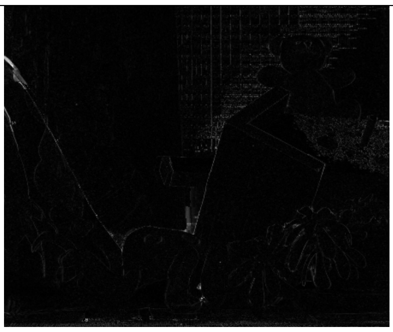
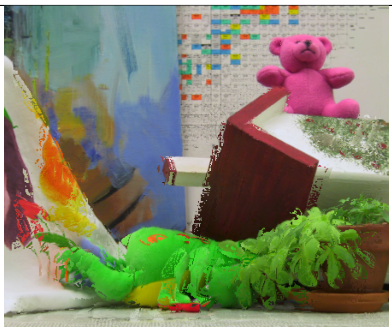
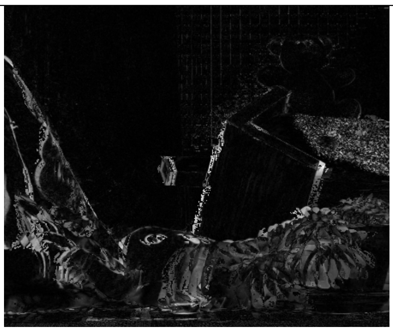
Input flow	Interpolated frame	Interpolation error
Ground truth		
Filter flow		
Black & Anandan		

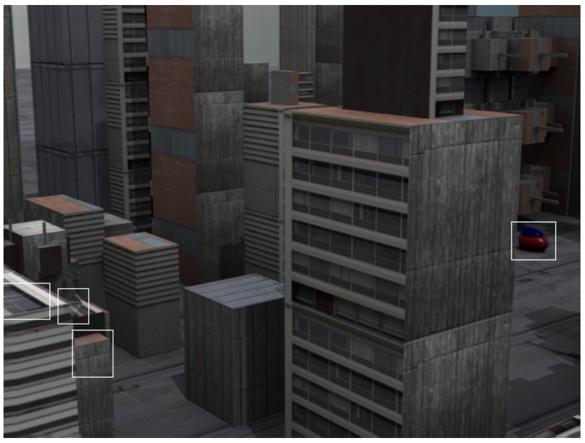
Figure 17: interpolated frames and interpolation error of regularized algorithm on the Middlebury teddy dataset with varying input flow.

4.5 Middlebury Flow Evaluation

Appendix A shows interpolated frames and interpolation error for the eight datasets in the Middlebury interpolation evaluation set, with each of Black & Anandan and filter flow as input. Table 6 gives numerical results. All algorithmic parameters were optimized for the cones dataset. The final algorithm numerically improves on the naive algorithm mostly only on the synthetic dataset (Urban) and the stereo dataset (Teddy).

Here we present brief explanations of differences in the few most visually interesting comparisons of results for the naive and final regularized algorithms. Some of these comparisons favor the naive algorithm and some favor ours. Each section includes blowups of the same regions (outlined in white) in the two interpolated frames, and explains the differences with reference to the algorithmic differences.

4.5.1 Urban, Black & Anandan


(a) naive algorithm

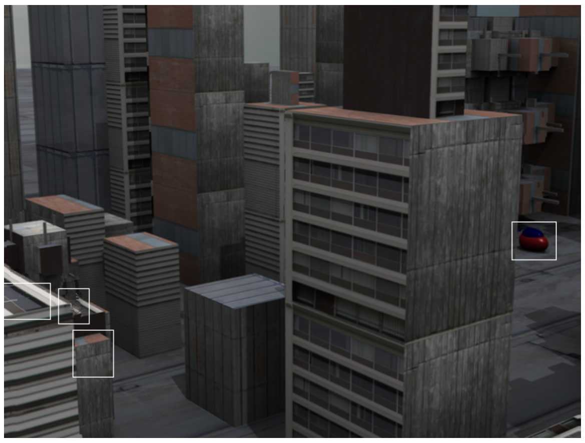
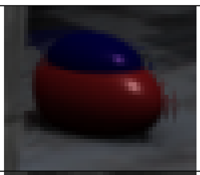
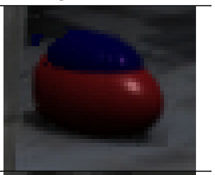
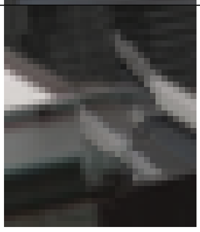
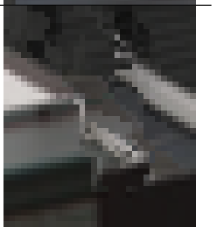
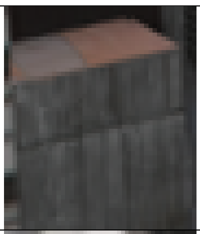
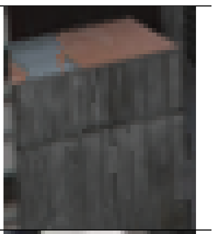
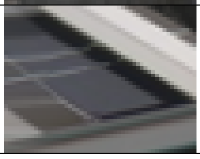
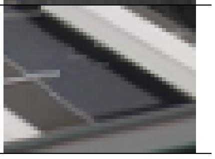

(b) final algorithm

Figure 18: interpolated frames for the urban dataset with Black & Anandan flow.

Table 2:

crop #	description	naive	regularized
1	Occlusion reasoning removes ghosting on the car, the fastest-moving part of the scene.		
2	No combination of our algorithmic changes is able to recover from the incorrect flow at the corner of the roof. Regularization makes it visually worse by reducing smoothing.		
3	Occlusion reasoning and regularization allow the blue piece to show up against the brown.		
4	Occlusion reasoning removes the incorrect second set of white lines on the roof.		

4.5.2 Backyard, Black & Anandan

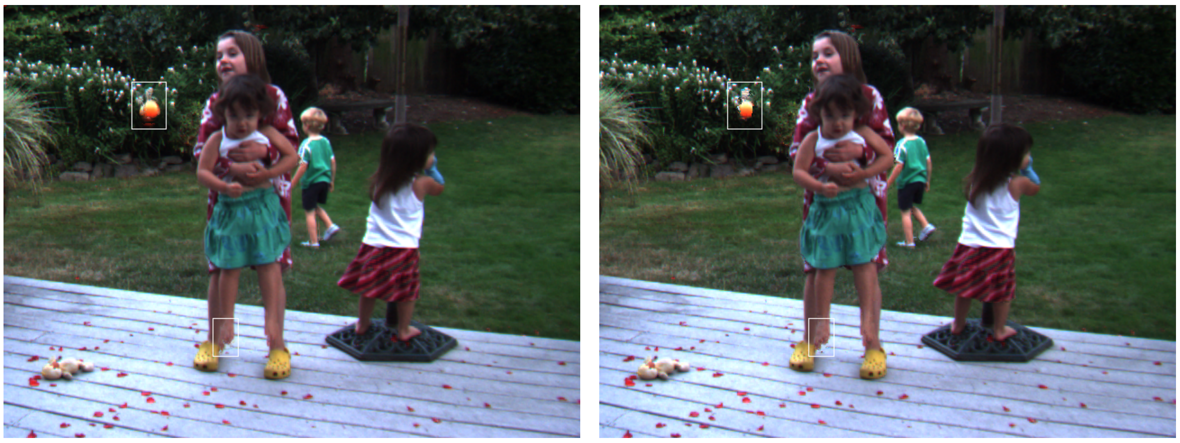
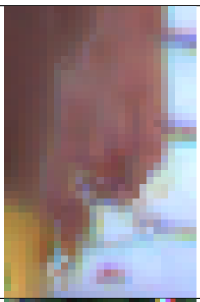
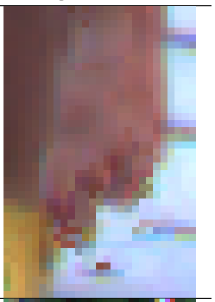
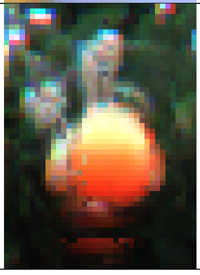
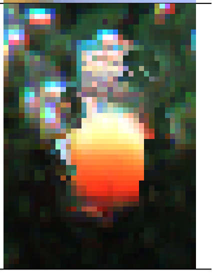

(a) naive algorithm (b) final algorithm

Figure 19: interpolated frames for the backyard dataset with Black & Anandan flow.

Table 3:

crop #	description	naive	regularized
1	Regularization makes the shape of the girl's foot more realistic.		
2	Regularization removes ghosting on the ball but also removes many pixels that are in fact part of the ball.		

4.5.3 Basketball, Black & Anandan

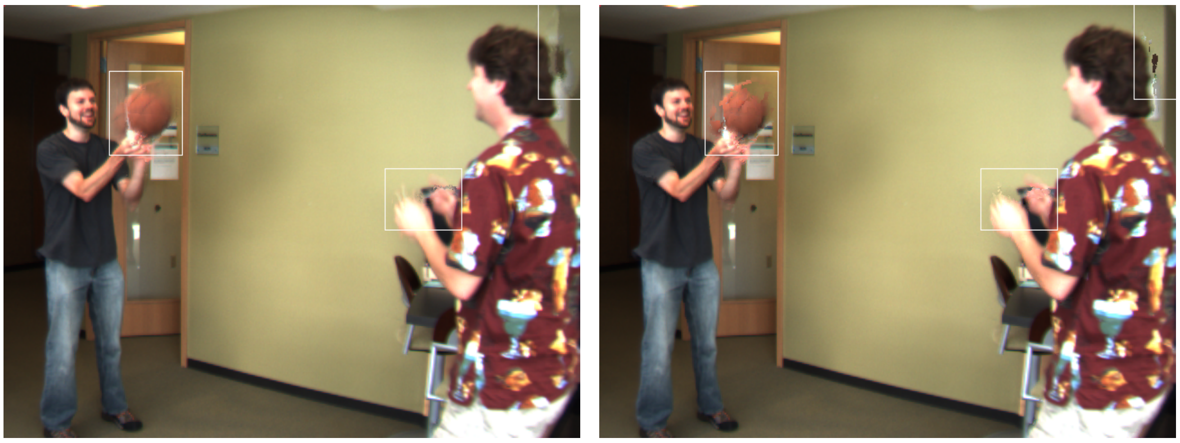
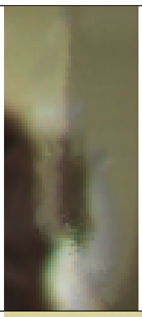
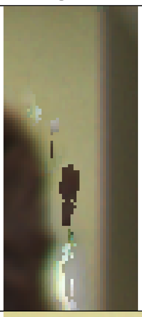
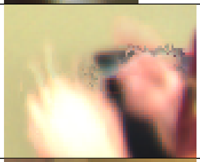
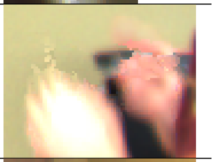
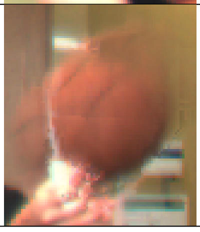
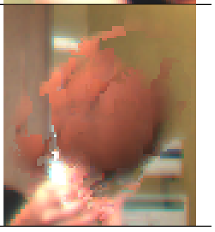

(a) naive algorithm (b) final algorithm

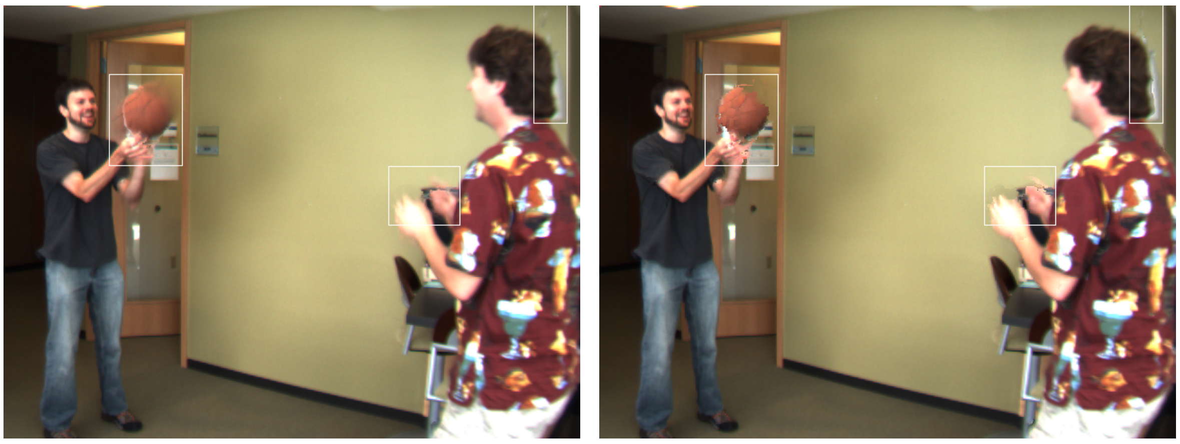
Figure 20: interpolated frames for the basketball dataset with Black & Anandan flow.

Table 4:

crop #	description	naive	regularized
1	Bidirectional flow avoids the phantom dark brown region but doesn't have a better suggestion for what to render in that area.		
2	Bidirectional flow and occlusion reasoning reduce ghosting of the hands.		
3	Again occlusion reasoning and regularization reduce the number of ghost pixels but actually visually worsen the result.		

4.5.4 Basketball, filter flow

The highlighted regions we examine here are exactly the same ones we saw for the Black & Anandan output.



(a) naive algorithm (b) final algorithm

Figure 21: interpolated frames for the basketball dataset with filter flow.

Table 5:

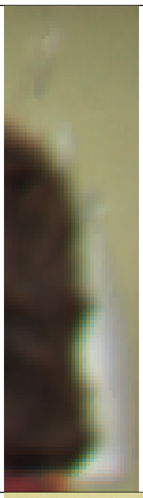
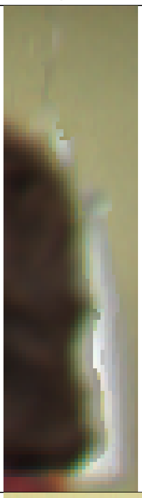
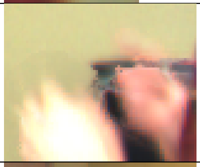
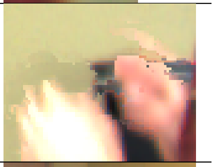
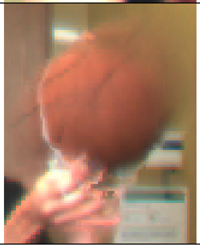
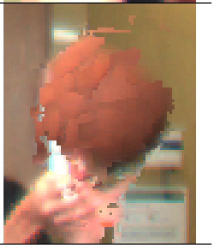
crop #	description	naive	regularized
1	The large hole behind the head that we saw for Black & Anandan is avoided.		
2	The hands are over-regularized, although there's even less ghosting than for Black & Anandan.		
3	Again, less ghosting than for Black & Anandan, and we see more of the actual ball than we did above, but the result is even more blocky.		

Table 6: average per-pixel interpolation error of various interpolation algorithms on Middlebury interpolation evaluation sets with each of two input flow algorithms.

Dataset	**Interpolation**	**Flow**	
		Black & Anandan	Filter flow
Army	Naive	1.91	1.84
	Final non-regularized	1.90	1.89
	Regularized	1.90	1.89
Mequon	Naive	2.84	2.84
	Final non-regularized	2.92	3.20
	Regularized	2.87	3.08
Urban	Naive	4.22	4.39
	Final non-regularized	4.15	4.15
	Regularized	4.09	4.10
Teddy	Naive	5.80	5.74
	Final non-regularized	5.73	5.65
	Regularized	5.60	5.40
Backyard	Naive	10.00	10.20
	Final non-regularized	10.20	10.50
	Regularized	10.20	10.30
Basketball	Naive	7.06	5.69
	Final non-regularized	8.18	6.19
	Regularized	8.02	6.07
Dumptruck	Naive	7.53	7.62
	Final non-regularized	7.66	8.09
	Regularized	7.49	7.83
Evergreen	Naive	7.13	7.13
	Final non-regularized	7.41	7.78
	Regularized	7.22	7.52

5 Conclusion

We have presented an optical-flow-based algorithm for two-frame image interpolation making use of reasoning about occlusions and disocclusions. We have shown that bidirectional flow can improve rendering results, as can spatial regularization. Our algorithm works well, at least for stereo datasets, when given very accurate flow fields. The improvement we see from our algorithm decreases as the quality of the input flow fields decreases. When optical flow algorithms improve, our algorithm might be useful in differentiating among flow algorithms whose performance is very good and very similar.

One problem we continually run into is that "ground-truth" flow fields don't line up exactly with any sort of outline in the input images, since outlines aren't very clear but we want to see relatively clear edges in the input flow. To combat this, and possibly allow for flow fields that are less exact near boundaries, we could incorporate matting into our model: allow each intermediate-time pixel to be in multiple layers each of which has a separate flow model. This might remove what ghosting is left in our results for the teddy dataset.

6 Acknowledgements

The authors wish to thank Rick Szeliski of Microsoft Research and Daniel Scharstein of Middlebury College for technical discussions, and Prof. Scharstein also for assistance in comparing our method to the current

Middlebury test framework.

References

[1] S. Baker, D. Scharstein, J. Lewis, S. Roth, M. Black, and R. Szeliski. A database and evaluation methodology for optical flow. In *IEEE International Conference on Computer Vision (ICCV)*, 2007.

[2] S. Baker and S. Seitz. Filter flow. In *IEEE International Conference on Computer Vision (ICCV)*, 2009.

[3] S. Baker, R. Szeliski, and P. Anandan. A layered approach to stereo reconstruction. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, 1998.

[4] M. Black and P. Anandan. Robust dynamic motion estimation over time. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, 1991.

[5] S. Chen and L. Williams. View interpolation for image synthesis. In *ACM SIGGRAPH*, 1993.

[6] P. Ferreira, J. T. ao, P. Carvalho, and L. Velho. Video interpolation through green's functions of matching equations. In *IEEE International Conference on Image Processing (ICIP)*, 2005.

[7] A. Fitzgibbon, Y. Wexler, and A. Zisserman. Image-based rendering using image-based priors. In *International Journal of Computer Vision (IJCV)*, 2005.

[8] A. Gupta, P. Bhat, M. Dontcheva, B. Curless, O. Deusen, and M. Cohen. Enhancing and experiencing space-time resolution with videos and stills. In *IEEE International Conference on Computational Photography (ICCP)*, 2009.

[9] B. Horn and B. Schunck. Determining optical flow. In *Artificial Intelligence*, 1981.

[10] D. Mahajan, F. Huang, W. Matusik, R. Ramamoorthi, and P. Belhumeur. Moving gradients: A path-based method for plausible image interpolation. In *ACM SIGGRAPH*, 2009.

[11] D. Scharstein. Middlebury optical flow evaluation. `http://vision.middlebury.edu/flow/eval/`.

[12] D. Sun. Black & anandan implementation. `http://www.cs.brown.edu/~dqsun/research/software.html`.

[13] L. Zitnick, S. Kang, M. Uyttendaele, S. Winder, and R. Szeliski. High-quality video view interpolation using a layered representation. In *ACM SIGGRAPH*, 2004.

A Detailed Middlebury Results

Interpolated frames and per-pixel interpolation error images for the Middlebury flow evaluation sets. Error images are normalized so that all pixels for a given dataset are comparable.

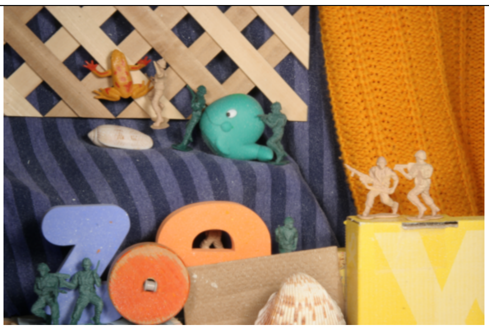
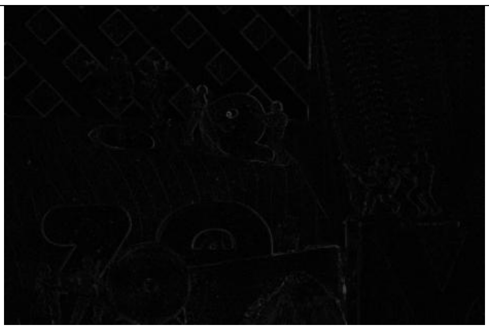
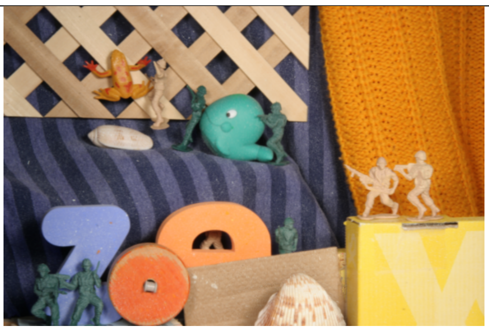
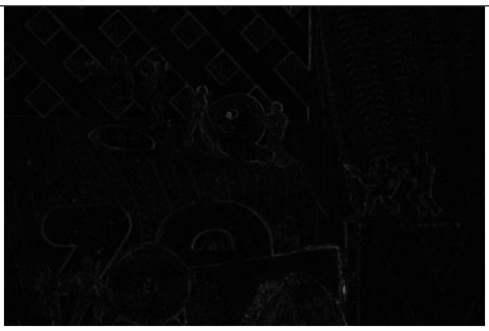
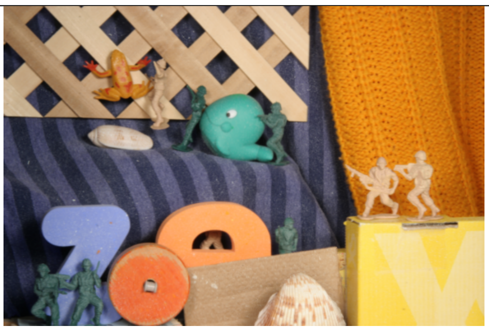
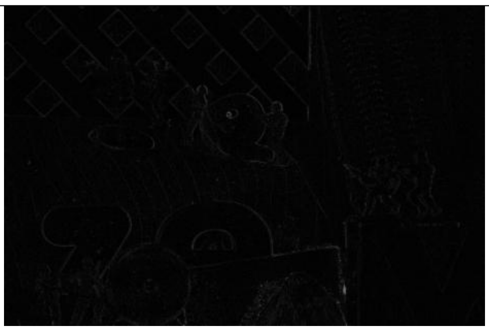
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 22: interpolated frames and interpolation error for the Middlebury Army dataset with filter flow [2] as input. Algorithm names from fig. 12.

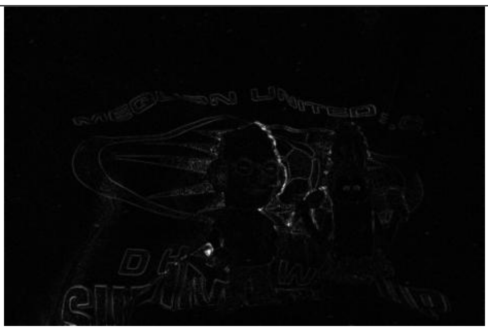
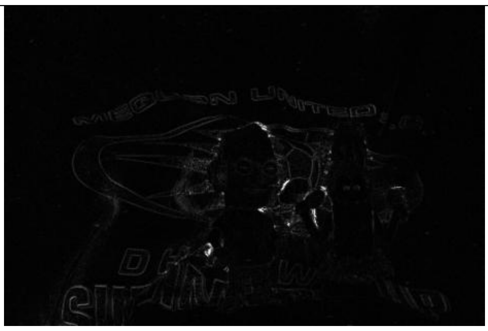
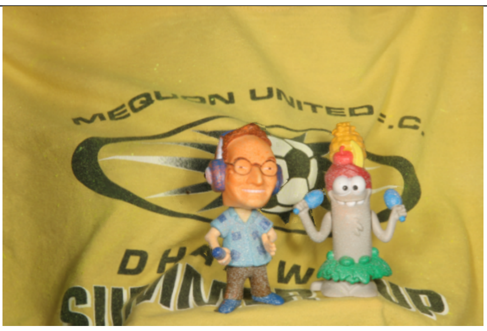
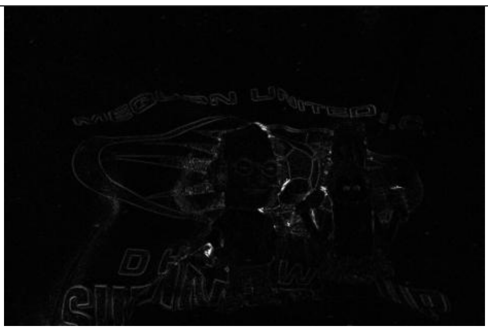
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 23: interpolated frames and interpolation error for the Middlebury Mequon dataset with filter flow [2] as input. Algorithm names from fig. 12.

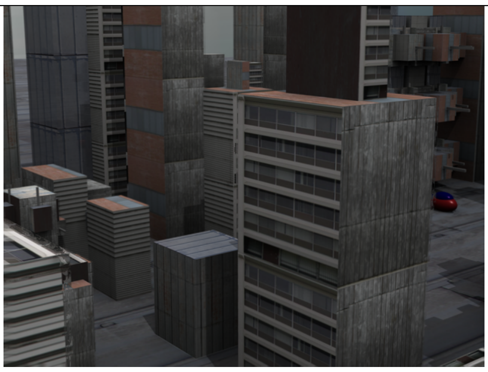
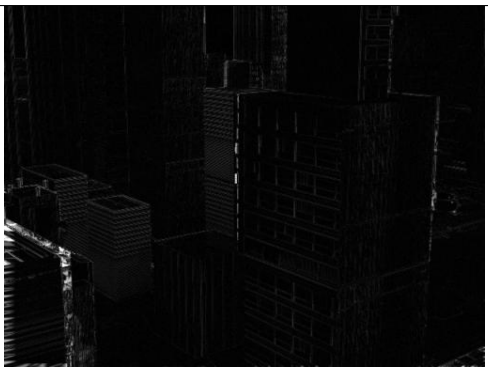
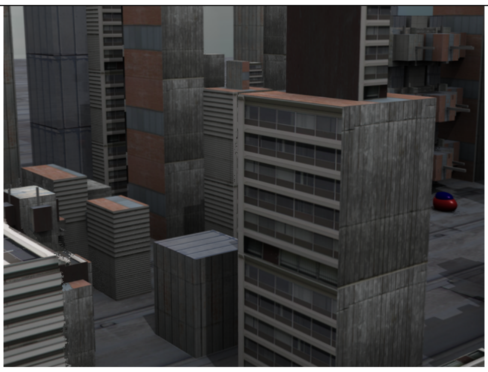
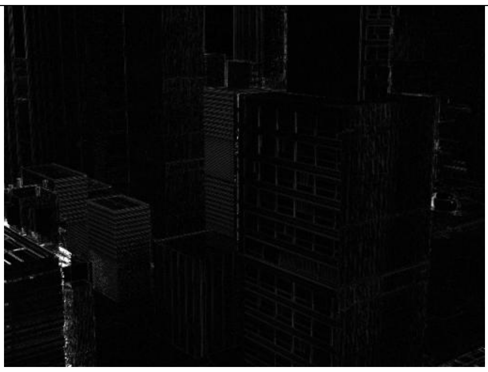
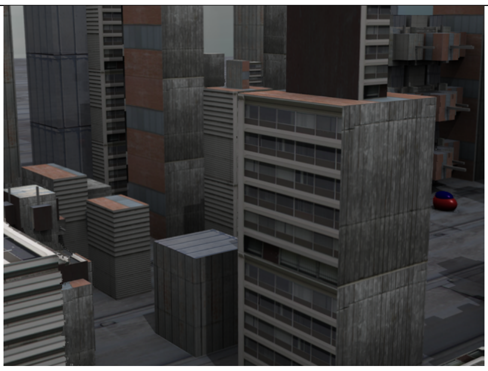
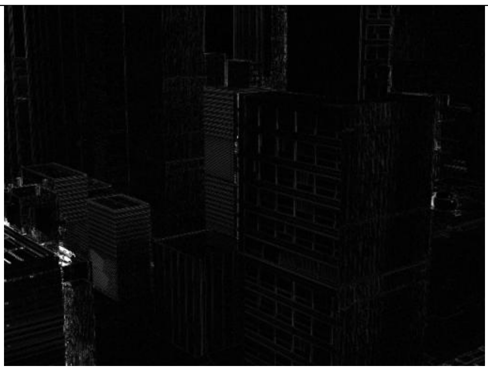
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 24: interpolated frames and interpolation error for the Middlebury Urban dataset with filter flow [2] as input. Algorithm names from fig. 12.

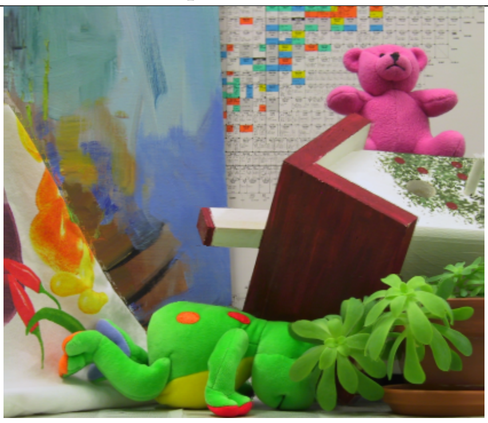
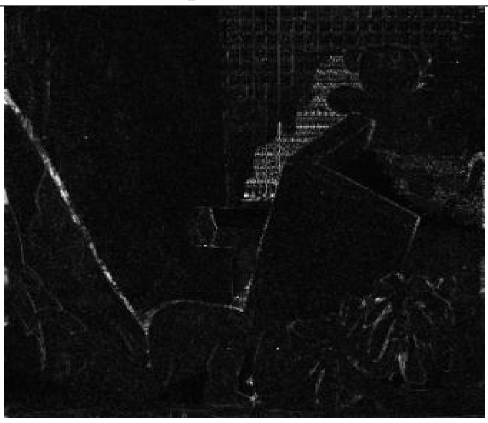
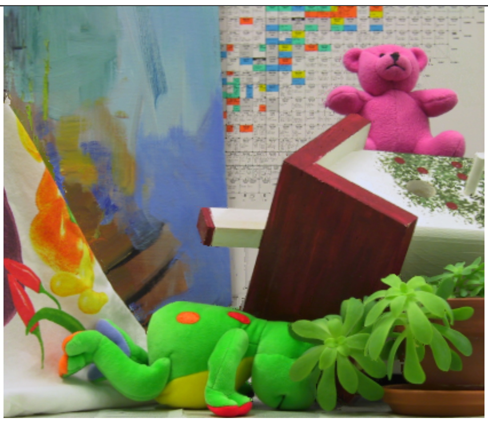
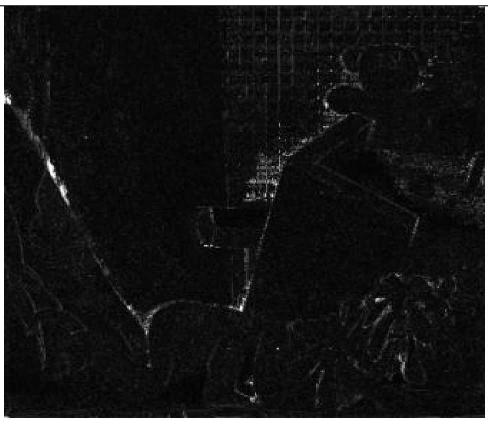
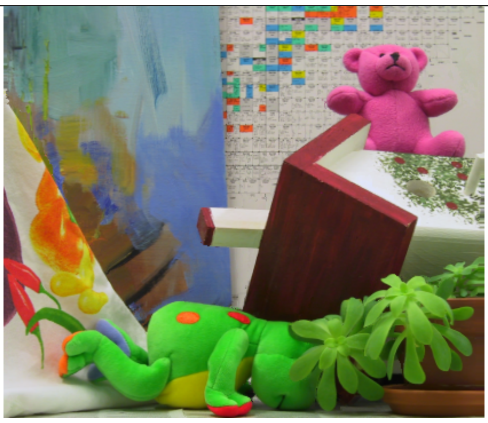
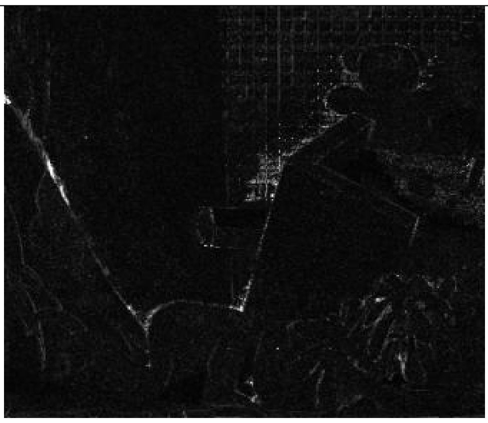
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 25: interpolated frames and interpolation error for the Middlebury Teddy dataset with filter flow [2] as input. Algorithm names from fig. 12.

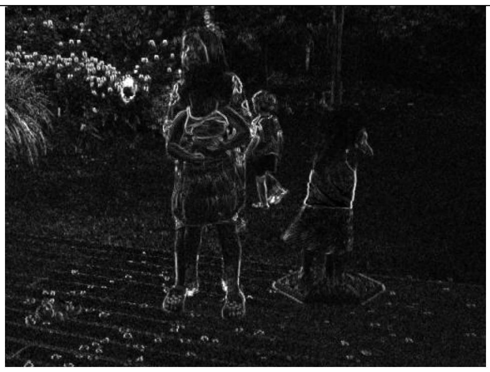
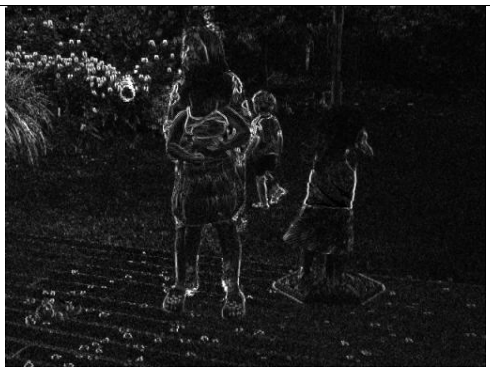
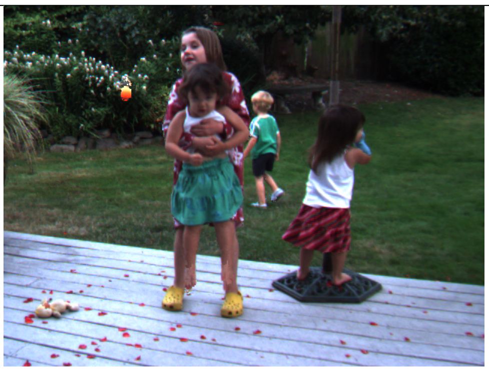
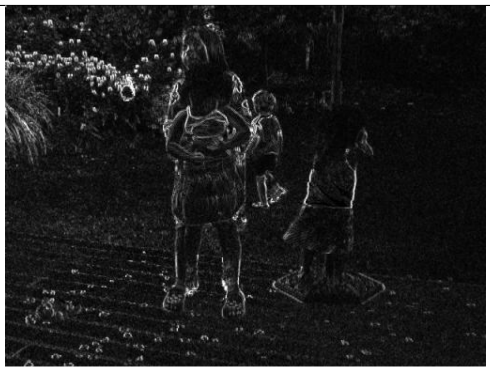
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 26: interpolated frames and interpolation error for the Middlebury Backyard dataset with filter flow [2] as input. Algorithm names from fig. 12.

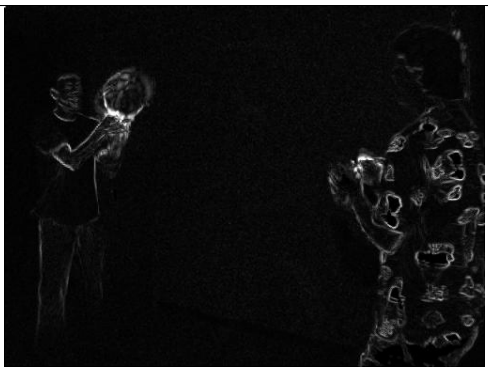
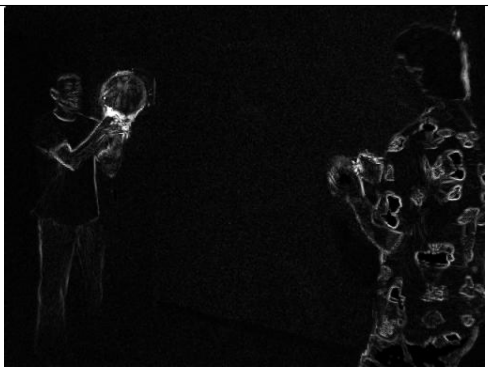
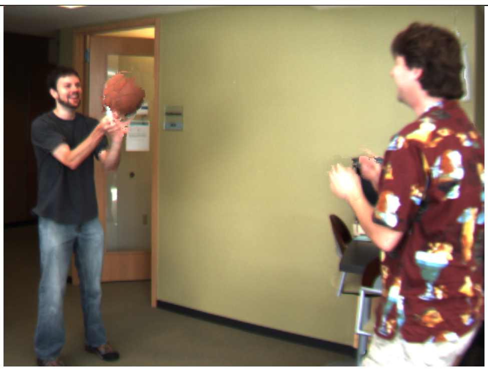
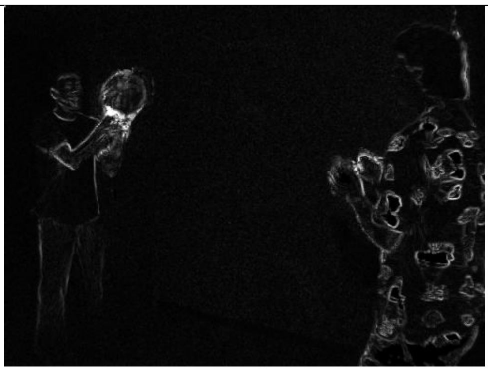
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 27: interpolated frames and interpolation error for the Middlebury Basketball dataset with filter flow [2] as input. Algorithm names from fig. 12.

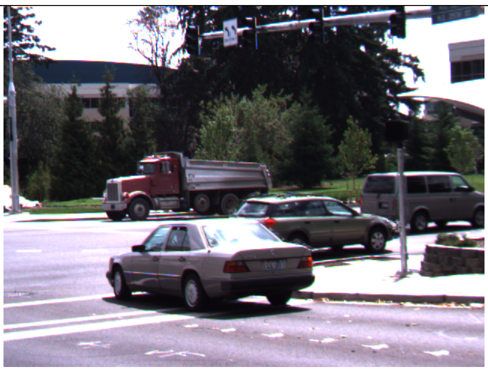
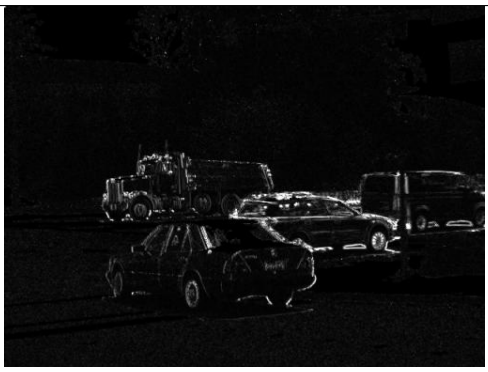
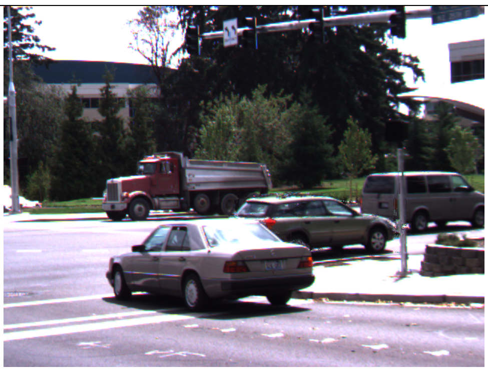
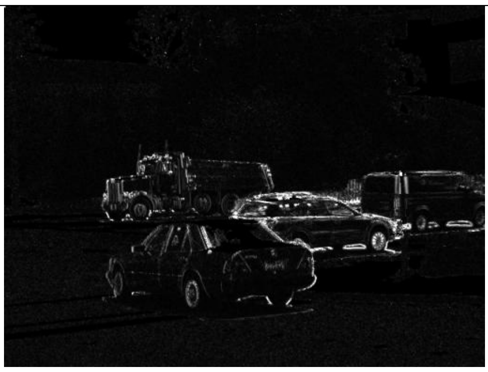
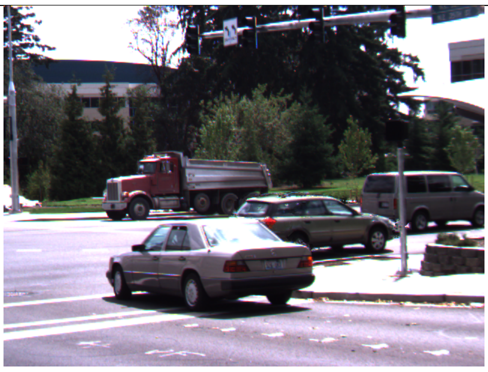
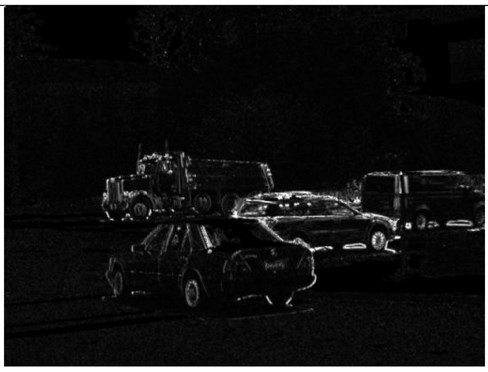
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 28: interpolated frames and interpolation error for the Middlebury Dumptruck dataset with filter flow [2] as input. Algorithm names from fig. 12.

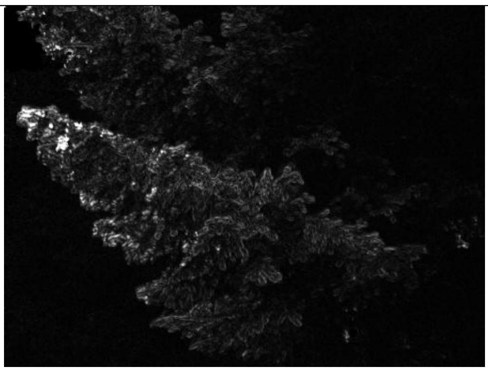
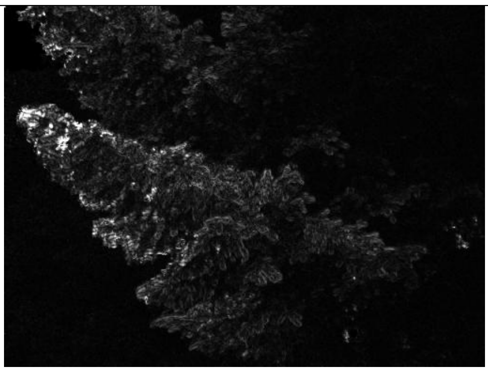
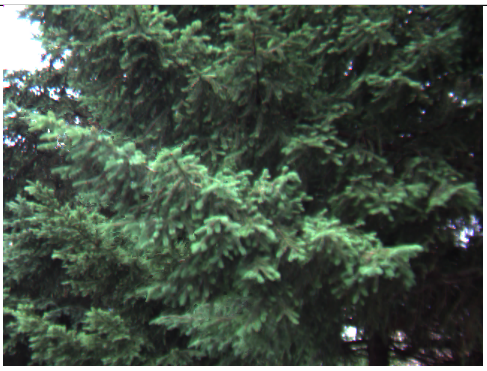
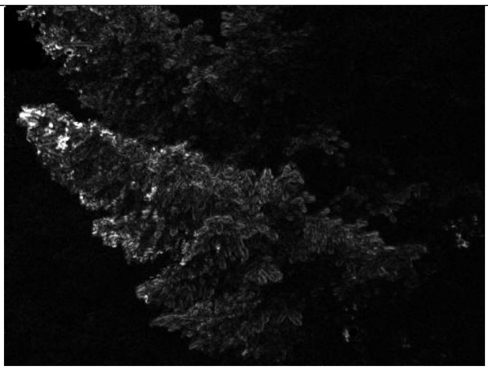
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 29: interpolated frames and interpolation error for the Middlebury Evergreen dataset with filter flow [2] as input. Algorithm names from fig. 12.

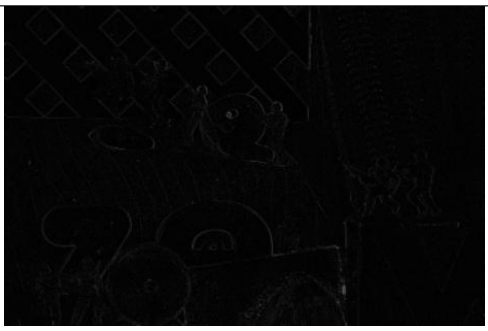
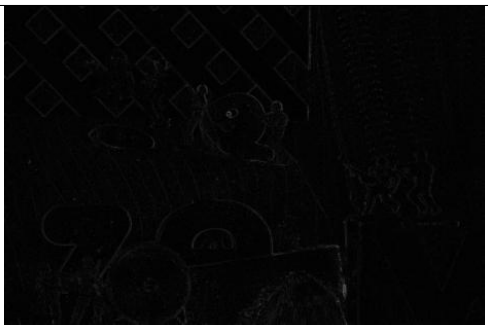
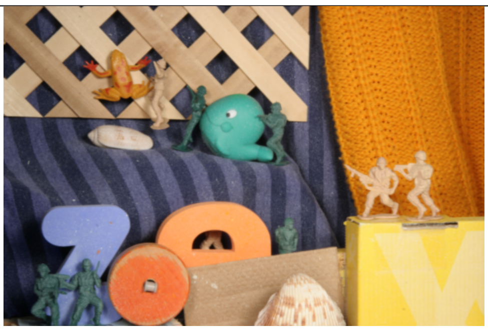
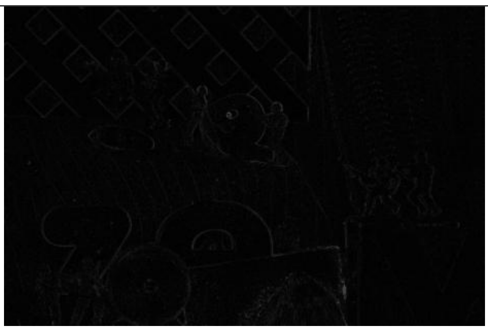
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 30: interpolated frames and interpolation error for the Middlebury Army dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

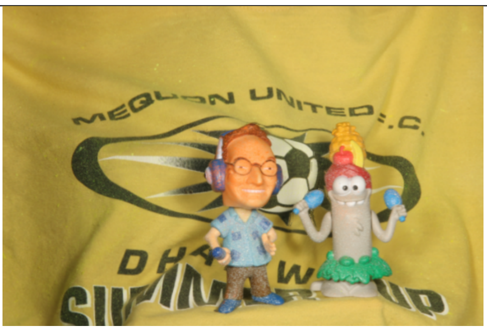
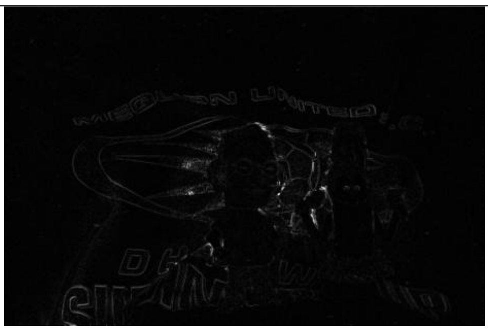
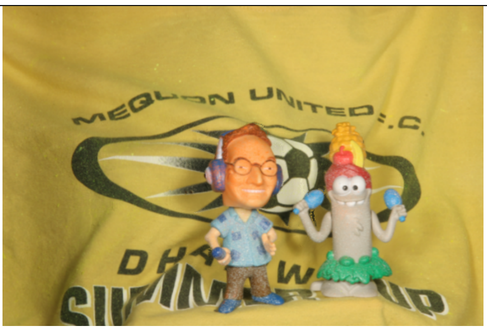
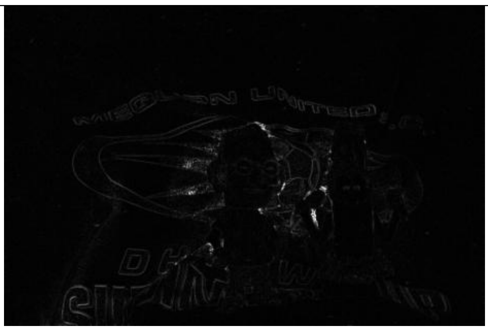
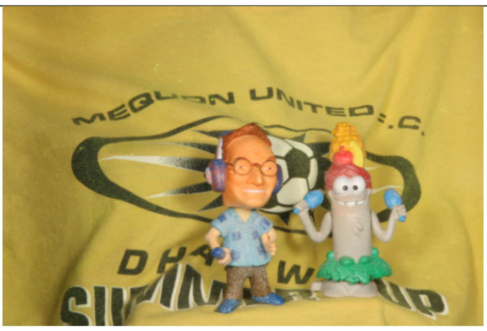
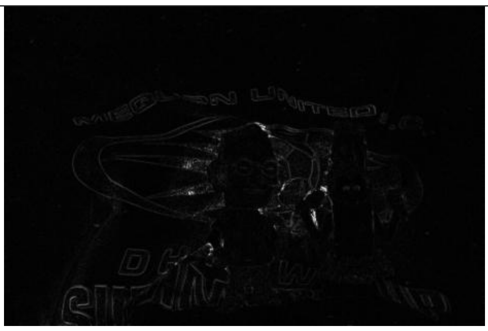
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 31: interpolated frames and interpolation error for the Middlebury Mequon dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

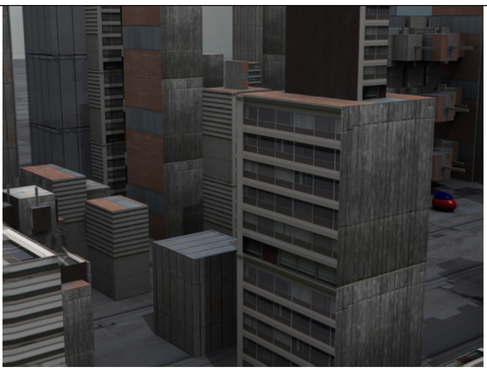
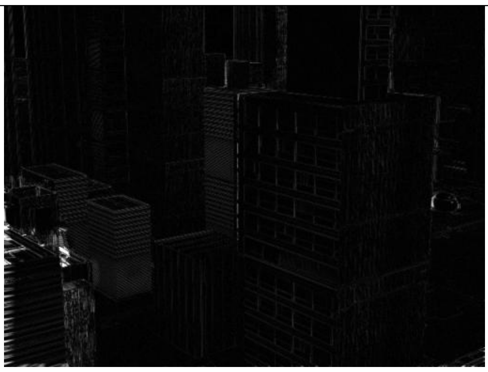
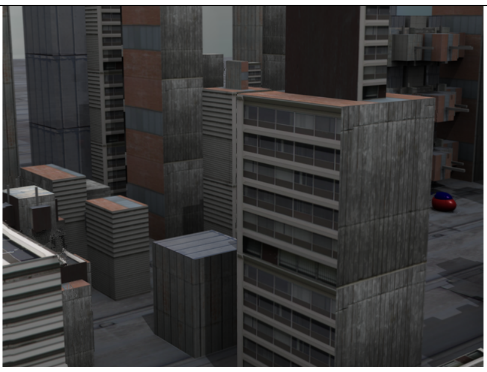
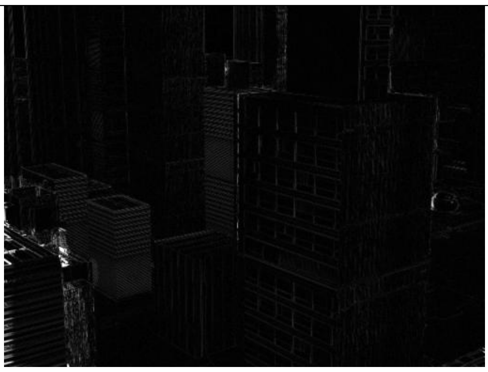
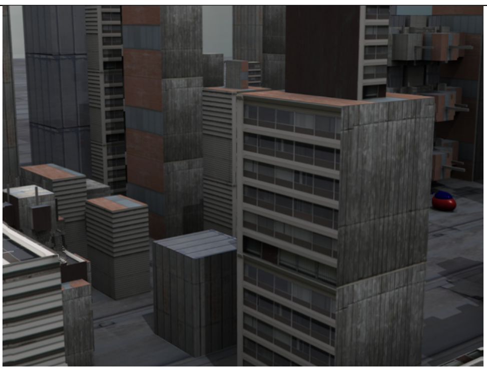
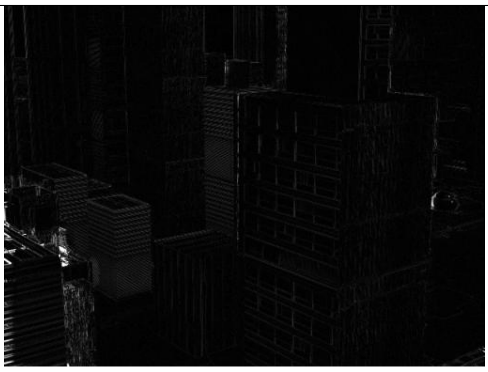
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 32: interpolated frames and interpolation error for the Middlebury Urban dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

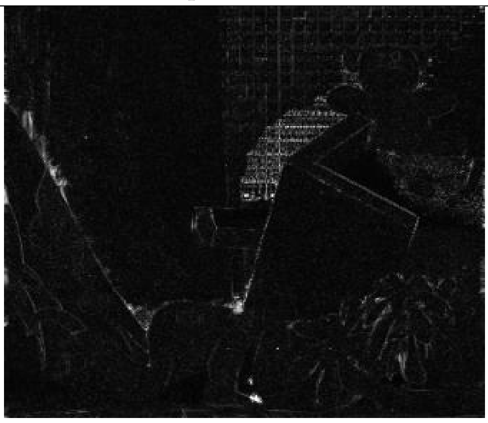
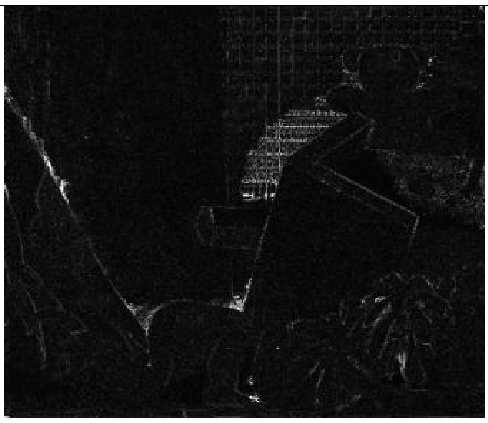
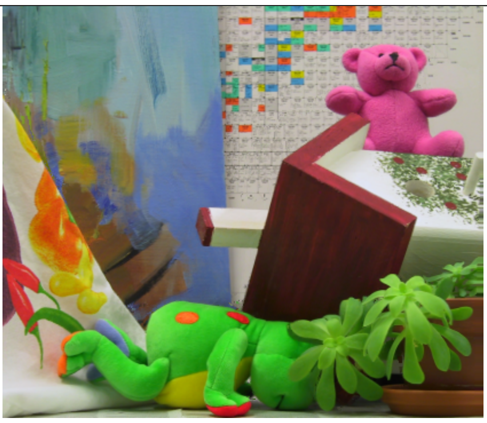
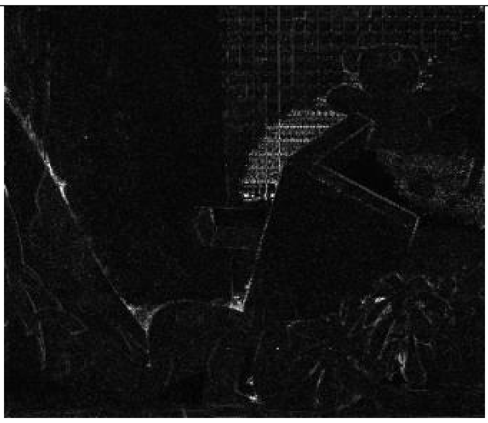
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 33: interpolated frames and interpolation error for the Middlebury Teddy dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

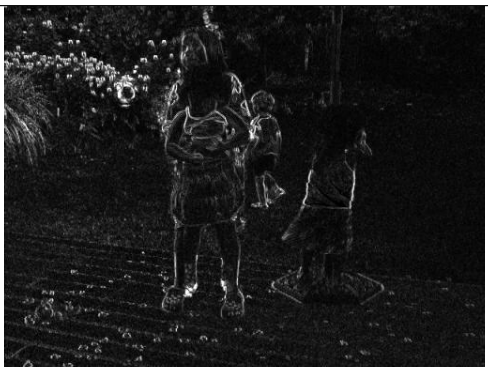
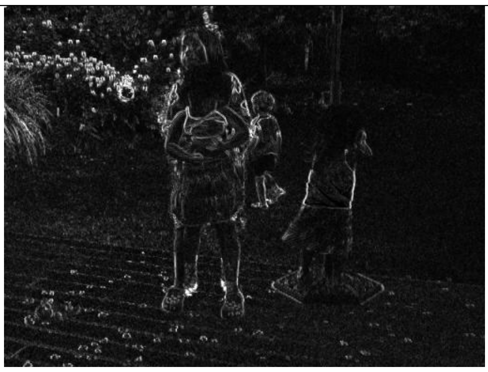
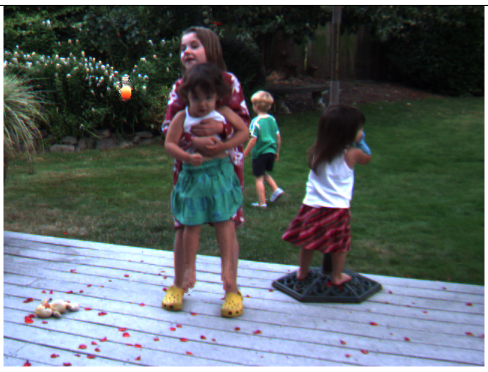
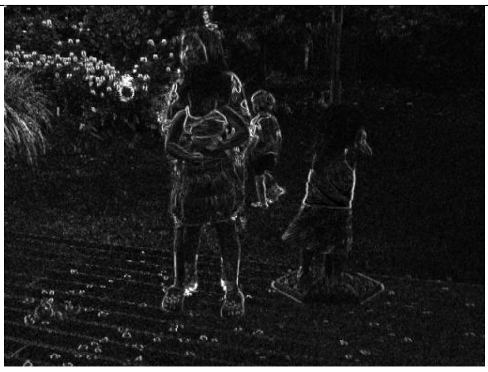
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 34: interpolated frames and interpolation error for the Middlebury Backyard dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

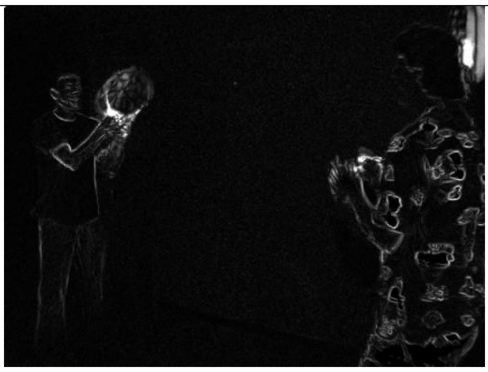
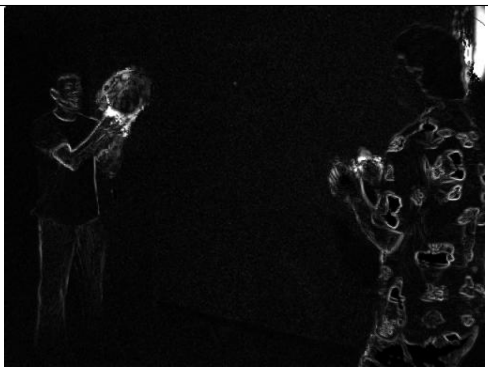
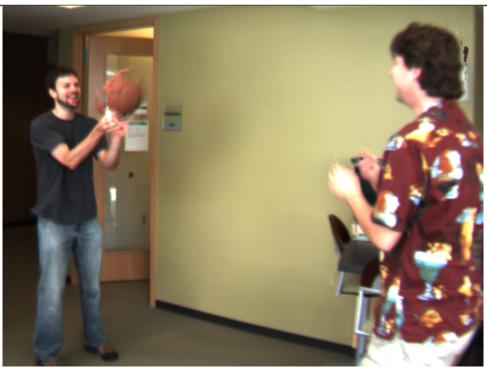
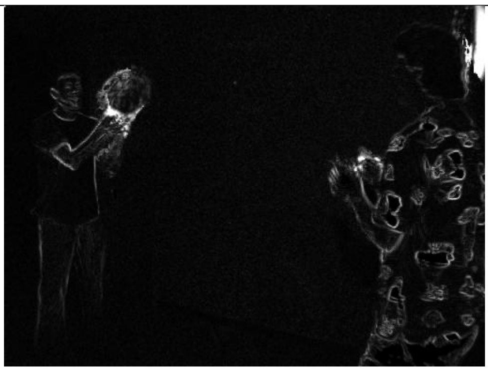
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 35: interpolated frames and interpolation error for the Middlebury Basketball dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

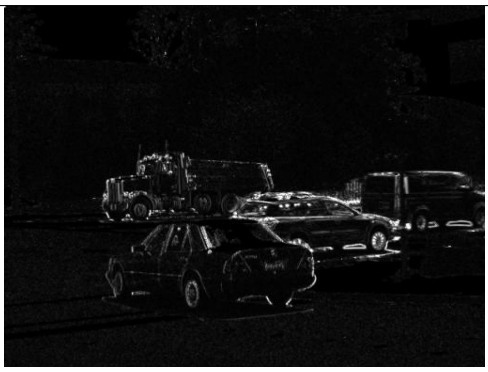
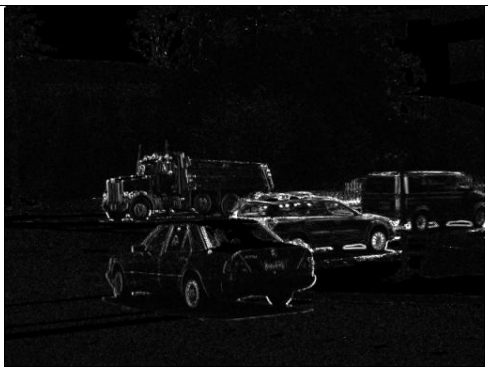
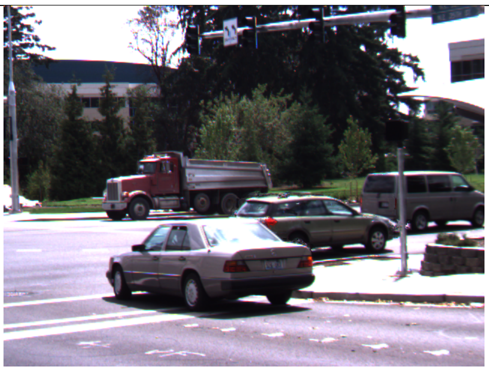
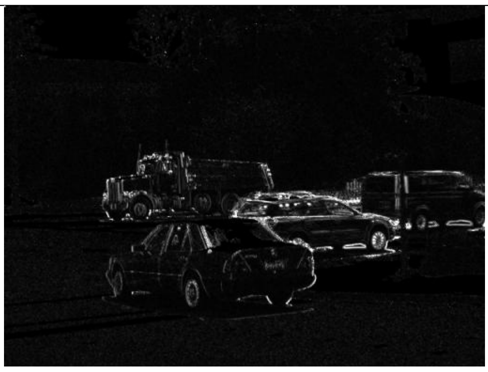
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 36: interpolated frames and interpolation error for the Middlebury Dumptruck dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.

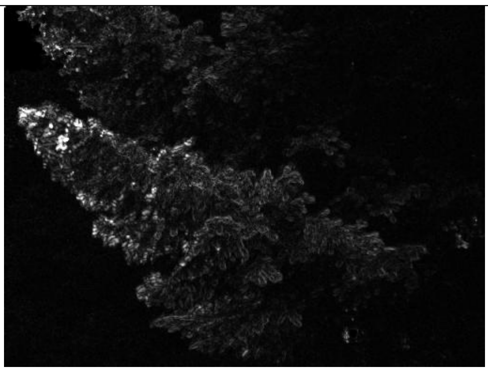
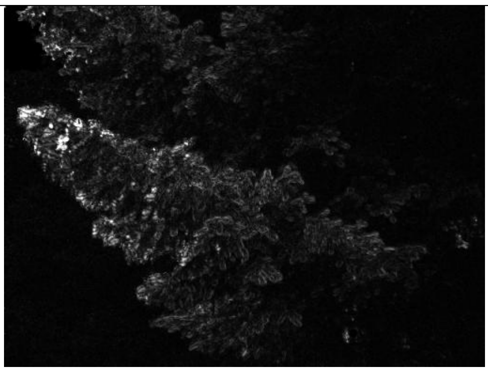
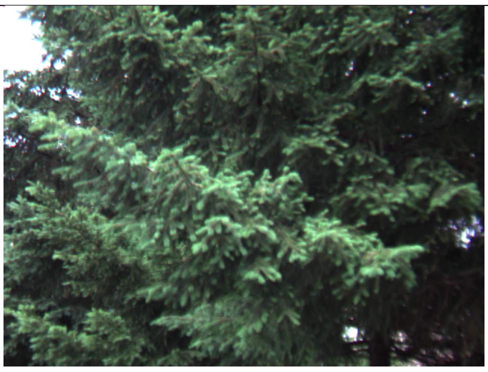
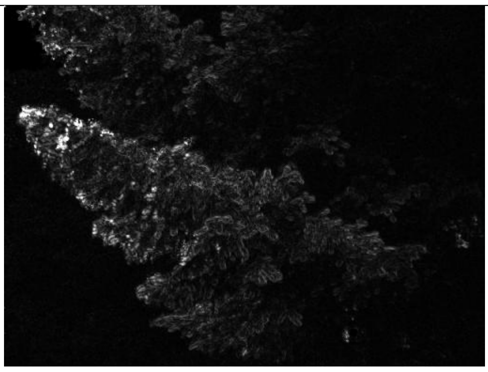
Interpolation	Interpolated frame	Interpolation error
Naive		
FNR		
RegTA		

Figure 37: interpolated frames and interpolation error for the Middlebury Evergreen dataset with Black & Anandan flow [12] as input. Algorithm names from fig. 12.