Final

Steven Schwarcz, Ryan Dorson, Andrew Pachulski

December 20, 2015

1 Introduction

Removing a fence blocking certain pixels each frame from a video is a difficult problem in computer vision. All the information regarding what's behind the fence presumably exists somewhere in the video, but in any given frame some of it will be obscured. This paper discusses some of the techniques we attempted to isolate the fence, remove it, and reconstruct what lay behind it.

2 Optical Flow

In order to find out what information exists behind the fence, we first needed to discover which pixels contribute to a given real world point. Each point in the real world is represented by a pixel in several different images. By calculating the optical flow, we were able to create a mapping of how a given pixel moves between images. By looking at the pixels in different frames that represent the same point in space, we were able to develop a basis for replacing fence pixels with pixels that represent what lies behind the fence.

We divided the video into separate frames and calculated the optical flow between each subsequent frame. Between each frame, we calculated the direction and magnitude that a given pixel moved. This gave us a set of ||frames|| -1 optical flows which we used in subsequent caluclations. We implemented optical flow as described in C. Lui's doctoral thesis: Beyond Pixels: Exploring New Representations and Applications for Motion Analysis.

3 Removing Fence Pixels

With optical flow calculated, removing the fence pixels was simply a matter of removing the pixels with the highest flow. To do this, first we created a map of the magnitude of optical flow, which can be seen in Figure 1. Interestingly, many frames had very different average flows, almost as if the amount the camera moved between frames was not consistent. To account for this, the flow was adjusted by its average in various places throughout the algorithm.

By cutting off pixels moving faster than a certain speed, it was possible to take a best guess at which pixels were "fence" pixels, and remove them accordingly. This was not a

perfect system, but it tended to produce more false negatives than false positives, yielding high recall but low precision, which is what we needed for this algorithm. An example of an image with the fence removed can be seen in Figure 2.

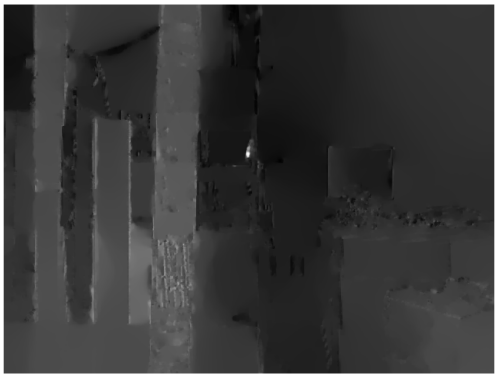

Figure 1: The magnitude of optical flow within the picture. the more white the pixels, the faster they are moving.

Figure 2: A couple of images of the fence with the fence pixels removed.

4 Reconstruction Attempt: Shifting Image

Before getting decent results, we tried a couple different techniques. The first of these was a technique in which we use the optical flow to guess where a pixel ought to be in the next frame. If it isn't there (based on the euclidean distance between the pixel we expect and the one we find), then we update the future image to include it. We essentially start at a specific image, and then work backwards to fill it in, using the pixels of the fence as a filter to make sure only the correct pixels are replaced. The code is included below. Results can be seen in Figure 3, but as can be seen they were not satisfactory. It's not clear if this is

because of some underlying glitches in the code, or simply because the technique as stated simply does not work.

After a lot of failed attempts with this technique, we ultimately abandoned it as unlikely to be fruitful within our tight timeline and moved on to try other techniques instead.

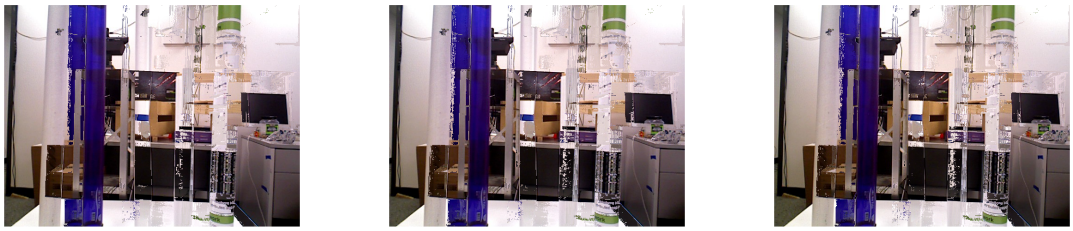

Figure 3: Several attempts at removing the fence using the shifting pixels technique. As can be seen, even when it worked correctly in some small patches, it still failed in too many places.

5 Reconstruction Attempt: Mosaic Stitching

In this approach, we first began by considering a temporal window around each frame with the fences removed, from Section 2. We attempted to find the homographies between the frame in question and the frames surrounding it by matching SIFT features between the images and performing RANSAC. It is important to note that the frame in question had the fence removed, but the surrounding images in the window were the original images. This way, SIFT feature matching did not attempt to match up features found on the fence; rather it had to match features in the background, so that we could stitch together the scene in the background better. We then stitched each of the surrounding images to the frame in question, keeping the image the same size as the frame. When we did this, we attempted only to fill in pixels that were removed due to the fence. This caused some issues however, because there were obviously lines between each of the images and RANSAC did not consistently give good homographies, as seen in Figure 4.

I realized that about 60 frames away from any frame in question, we could see just about all of the pixels occluded in the frame in question. Therefore, we hard-coded the results of shifting this corresponding frame by a certain number of pixels. This produced the image in Figure 5, which shows desirable behavior; however, the RANSAC routine never provided an affine transformation that looked quite this good.

In search of an algorithm that provided the desired behavior, we stumbled upon [1], which describes a robust method for solving this problem. In the paper, the authors determine the probability of each fence being a pixel, using motion cues, appearance cues, and bundle adjustment. We were able to replicate the code for motion cues, which uses the principle component of the horizontal and vertical optical flow to project each pixel to a number between 0 and 1, corresponding to the probability of it being a fence given motion cues. Using this probability, we can compute the homographies between the frame in question and its surrounding frames within a 15 frame temporal window using a weighted least-squares

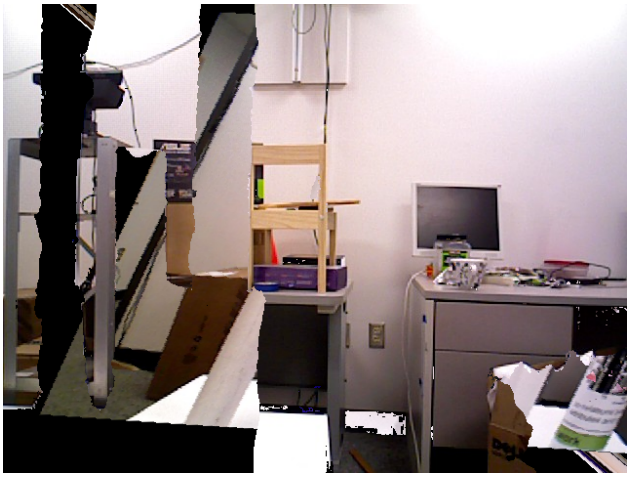

Figure 4: RANSAC provides a bad affine transformation matrix, so the missing pixels are filled in with nonsense.

Figure 5: Desirable behavior.

scheme. The idea is that we give the pixels with low probability of being fence a high weight (indicating that they are probably in the background scene) and then compute least-squares between SIFT feature matches with these weights. This seemed like a promising course of action, but it also provided less-than-average results. Had we been able fully replicate the code for recognizing appearance cues, we could have used a weighted probability depending on motion and appearance cues and bundle adjustment to solve this problem better. In addition, [1] talks about what they call a "naive solution," involving using the weighted least-squares affine transformation technique described above, and using temporal median filtering to restore occluded pixels. We partially replicated this technique, but applying the temporal median filter in a temporal window of frames created an image that appeared blurred. Had we been able to adjust this median filtering technique only to fill in only the removed fence pixels, then we may have ended up with a more desirable result. In short, we tried a large number of techniques that did not provide satisfactory results immediately. Given our tight time constraints, we were unable to explore these techniques more fully to

provide a desirable solution. However, we are happy with the progress we were able to make, as well as our exploration of interesting techniques.

6 Code

```
%% Remove pixels of the fence by removing the fastest pixels in the image

imgs_rm = imgs;

threshold = 1.11;

for k = 1:266
    im = flows{k};
    im_rm = imgs_rm{k};

    % Get the magnitude of the flow
    magflow = sqrt(im(:, :, 1).^2 + im(:, :, 2).^2);
%     magflow = im(:, :, 1);

    avg = mean2(magflow);
    T = avg * threshold;

    for i = 1:size(im, 1)
        for j = 1:size(im, 2)
            % Use the magnitude of the flow to determine if the pixels
            % should remain
            if magflow(i, j) < T
                magflow(i, j) = 0;
            else
                im_rm(i, j, 1) = 0;
                im_rm(i, j, 2) = 0;
                im_rm(i, j, 3) = 0;
            end
        end
    end

    if k == 1
        maxmag = max(max(magflow));
    end

    % Save image
    imshow(im_rm);
    imwrite(im_rm, sprintf('fence_removal/imgs/rm_frame_%d_rgb.png', k + 100))
    imgs_rm{k} = im_rm;
end
```

```
imgs_rm = imgs;
```

```
im_to_change_idx = 60;

% Difference between pixels within which they are considered "the same"
Tdiff = 2;

iter = 22;

all_mask = zeros(size(imgs{1}, 1), size(imgs{1}, 2), iter);

% Keep track of the average flows
avg = zeros(iter, 1);

for k = iter:-1:1
    % Check if this is the image we're working with
    is_result = k == iter;

    next_im = imgs{im_to_change_idx - iter - 1 + k};
    cur_im = imgs{im_to_change_idx - iter + k};

    % Some images have an extra pixel for some reason
    next_im = next_im(1:480, 1:640, :);
    cur_im = cur_im(1:480, 1:640, :);

    if is_result
        main = cur_im;
    end

    mask = ones(size(cur_im, 1), size(cur_im, 2));



%       magflow = sqrt(im(:, :, 1).^2 + im(:, :, 2).^2);
    magflow = cur_im(:, :, 1);

    avg(k) = mean2(magflow);

    T = avg(k) * 1.11;

    for i = 1:size(cur_im, 1)
        for j = 1:size(cur_im, 2)
            if magflow(i, j) < T
                mask(i, j) = 0;
            end
        end
    end

    all_mask(:, :, k) = mask(:, :);



    % Go through all non-masked pixels and look where they should be next
    % frame based on their flow
    if ~is_result
        pixels = find(mask * -1 + 1);
```

```
        for i = 1:size(pixels, 1);
            [x, y] = ind2sub(size(mask), pixels(i));

            flow = magflow(x, y);

            % Adjust flow for different average flow in the next frame
            flow = (flow / avg(k)) * avg(k + 1);

            if (y + uint32(flow) <= size(next_im, 2))

                expected_color = cur_im(x, y, :);
                seen_color = next_im(x, y + uint32(flow), :);

                ec = reshape(expected_color, 3, 1);
                sc = reshape(seen_color, 3, 1);

                ydist = y + uint32((flow / avg(k + 1)) * sum(avg));

                if norm(double(ec - sc)) < Tdiff && ...
                        ydist <= size(main, 2) && ...
                        all_mask(x, ydist, iter)
                    % If colors are different, assume the fence got in the way
                     main(x, ydist, :) = seen_color;
                end
            end
        end
    else
        maxmag = max(max(magflow));
    end

    main = main(:, 1:640, :);

    imshow(main)
    imgs_rm{k} = cur_im;
end
```

References

[1] Y. Mu, W. Liu, S. Yan *Video De-Fencing*, (2012), available at `http://arxiv.org/pdf/1210.2388.pdf`.