Lightfield Superpixel Segmentation and Segmentation-Based Editing

Author: Lucas Kasser

Brown CS Honors Thesis

Advisor: James Tompkin
Reader: Daniel Ritchie

Contents

1 Acknowledgements

The segmentation method presented in this paper was developed with the assistance of Henry Stone, Numair Khan, and James Tompkin. Numair was instrumental in designing and writing the cross-EPI matching and splitting code in section 3.3.3. Thank you especially to James for all the advice and moral support offered throughout this project.

2 Introduction

The ultimate goal of this thesis project is to be able to enable a user to easily edit a structured lightfield. A lightfield is a representation of all of the light in some scene; while a traditional RGB image captures the light that flows into a single point from a variety of angles, a lightfield image attempts to capture more information about the light traveling through a region. Instead of taking a picture from a single perspective, lightfields are captured with special camera rigs that are able to sample a scene in both the spacial and angular domain (see figure 1).

More intuitively, a structured lightfield can be represented as a set of images taken from cameras arranged in a grid. This in turn can be represented as a 4D function, $f(x, y, u, v)$, where (x, y) parameterize the pixel location in an image, and (u, v) parameterize the image itself, as the images are arranged in a grid. In this paper we largely deal with the simpler, 3D lightfields which consist of a linear array of images rather than a grid, and can thus be represented as $f(x, y, u)$.

Lightfields have many applications. Because we are sampling the scene from many points instead of just one, we have much more information about the scene. For instance, if we can find pixel correspondence across the images, we can in turn use the focal length of the cameras and their relative positions to compute the depth of the objects in the scene. This depth information can then be used for many post-processing effects, like dynamic Depth of Field and background removal. It is also possible to interpolate between existing camera views to construct views from novel camera positions. This in turn can be used to render a VR scene to a user without explicitly reconstructing the scene's geometry. Finally, lightfield displays such as the *Looking Glass* [1] are beginning to emerge on the consumer market. Because of these numerous applications, lightfield cameras and phones are starting to become readily available to users.

While the redundant information present in lightfields enables the useful operations discussed above, this redundancy can also make performing even simple edits difficult. When editing a single image, it is trivial to change the color of a pixel. In a lightfield image, however, we need to consistently propagate the edit to every corresponding pixel in every other view. We can use disparity estimation algorithms [21] [27] to determine which location the pixel projects to in the other views, but these algorithms only estimate disparity for the central view. If the pixel is occluded by another object in other views, we will mistakenly project the change over the occluding object. We therefore need some way to detect and accommodate occlusion.

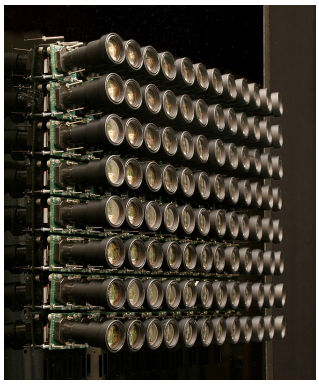

Figure 1: A lightfield can be taken with an array of cameras

Several algorithms have been introduced in an effort to edit lightfields in an occlusion aware manner. Chen et al. [4] extend 2D image patches to 4D lightfield patches. This approach enables them to use many of the standard patch-based editing methods on lightfields. They handle occlusion by estimating the depth for all views in the lightfield and thus avoid projecting lower depths over higher depths. Zhang et al. [26] also use patch-based editing methods, but they decompose the central view into multiple depth layers,

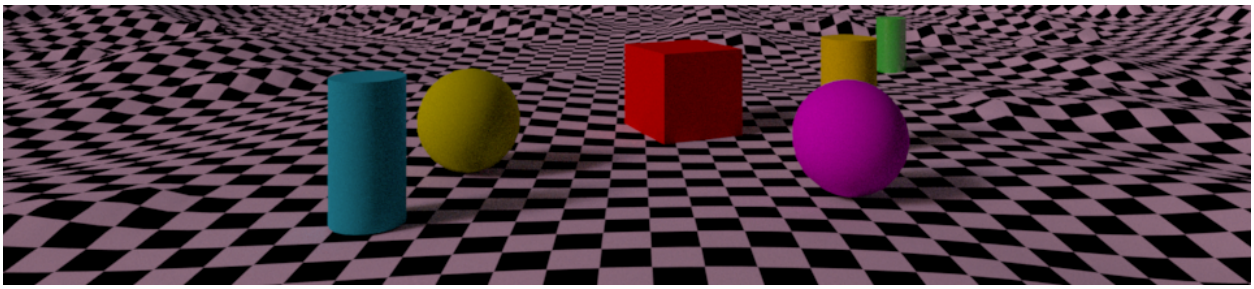

Figure 2: The central view from the Milky Way Dataset

where the decomposition supports semi-transparency in the segmentation. They use these depth layers as occlusion maps for projecting edits that the user makes to the central view. A disadvantage of this method is that the user can only make edits in the central view, as they only have disparity estimates for that view. Finally, Jarabo et al. [12] use affinity fields to determine pixel correspondence across views. They compress the entire lightfield using these affinities, perform the user's edit, and then decompress and render the result once the user is ready to export their edited results. This enables very performant edits.

In contrast, we use an occlusion aware method for editing lightfields using 4D, cross-view superpixels. First, we will introduce an algorithm for 3D lightfield segmentation and we will discuss how to perform basic edits using these superpixels. In the next section we will discuss how to use these superpixels as the basis for an object removal operation.

3 Superpixel Oversegmentation

3.1 Introduction

For the purposes of this problem, we will consider a lightfield to be a collection of images taken at the same time of the same scene. We are specifically interested in *structured* lightfields; therefore, we consider only lightfields composed of pictures taken from cameras that are colinear and have the same orientation.

We would like to develop an algorithm that takes in a structured lightfield in the form of a set of images and produces a set of superpixels for the lightfield. The edges of the superpixels should conform to the depth and color edges in the lightfield, and all pixels contained within a superpixel in one view should also have their corresponding pixels contained within the same superpixel in every other view. This way, the superpixels give us a sense of object occlusion.

We limit the scope of this segmentation to 3D lightfields, rather than 4D (the cameras are arranged in a 1D line rather than a 2D grid). We designed this method to work on a synthetic lightfield (figure 2) that was both 3D and had a large number of camera views.

Following the general idea presented in [5], rather than performing the segmentation in image space, we instead work in "Epipolar Image" space. "Epipolar Images," or EPIs, are cross-view slices of the lightfield. To generate the i^{th} EPI in the lightfield, we stack the i^{th} row of each image in order, forming a new image (see figs. 4 and 3). We can use the slopes of the edges in the EPI to compute the depths of the objects in the lightfield. Unlike

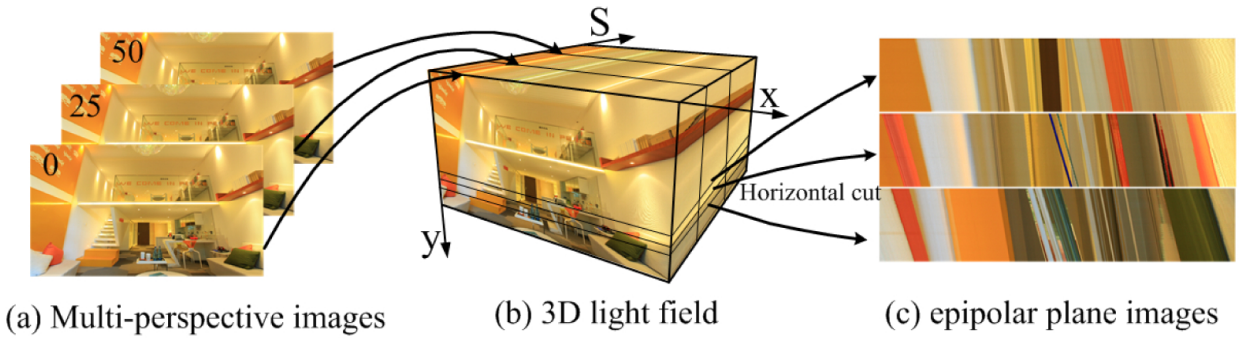

Figure 3: We construct an EPI by slicing the lightfield perpendicular to the image plane. Figure taken from [22]

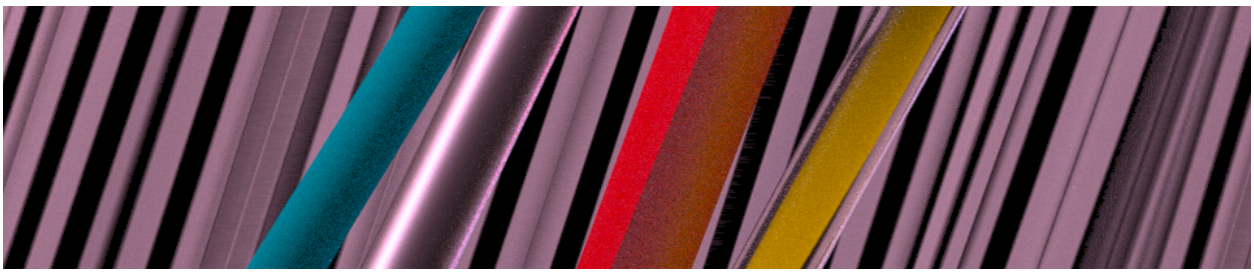

Figure 4: An example of an Epipolar Image (EPI) from the Milky Way dataset

working in image space, EPI objects have several nice properties: lambertian objects are always trapezoids and objects with more horizontal slopes always occlude object with more vertical slopes.

If we can consistently extract the trapezoidal segments from the EPIs and find the cross-EPI correspondence of those segments, we can extract 3D view-consistent object segmentations. In order to automatically segment an EPI, we need to extract each object present in the EPI. We are aiming for an oversegmentation; we need not necessarily extract each object as a single object as long as we extract all of its components. For an example of an over-segmentation produced by our algorithm, see fig. 5.

3.2 Related Works

There has been some work in supervised segmentation for lightfields. Wanner et al. [10] define an optimization function using intrinsic geometric information present in the lightfield as well as estimated disparity. Using this function, they train a Markov Random Field (MRF) to assign per-pixel labels to the central view in the lightfield given some user annotations. This approach only produces labels for the central view, which removes valuable information about the lightfield such as correspondance and occlusion.

Mihara et al. [18] extend Wanner's work by segmenting using an MRF-based graph-cut algorithm that produces labels for all views of the lightfield. Their method uses the appearance and 4D position of each ray in the light field as criteria for consistently segmenting a light field. They augment these neighborhood similarity constraints with depth/disparity

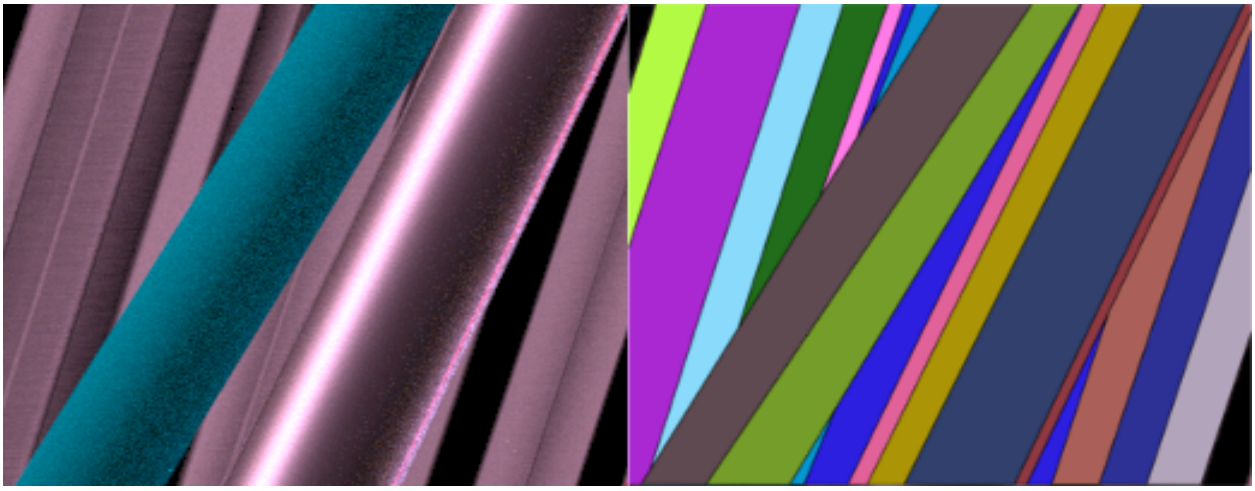

Figure 5: Left: Initial EPI; Right: Segmented EPI

measures which are determined using an SVM trained on user annotated data. However, their per-pixel graph cut is a computationally expensive operation, and causes their method to have a very long runtime.

Hog et al.'s work [10] out-performs the running time of the naive 3D MRF graph-cut in [18] by bundling rays according to depth. While they achieve impressive results without excessive computational time, they, like the other methods presented above, require the user to annotate the lightfield with "scribbles," denoting the number and type of objects in the scene. In contrast, we present a fully automatic segmentation method.

Criminisi et al. [5] use an EPI-based segmentation method to produce a segmentation for a 3D lightfield. They detect the edges in each EPI, perform a matching step on the edges, and then use the matches to reconstruct a segmentation for the original lightfield. We model our method on their general approach.

After our method was designed, [28] was published. They use an estimated center-view disparity map to project the lightfield pixels into 3D space, and then use a modified SLIC to perform clustering in that space. They then reproject the segmented pixels to 2D space to finalize the result. They produce good results for small baseline lightfields, and their method works on 4D lightfields. However, since their method does not explicitly enforce that the superpixels conform to edges and since they only use the disparity of the pixels in the central view, the superpixels do not always correctly classify pixels that were obscured in the central view.

3.3 Method

As shown in Figure 6, our method is divided into three parts. First, for the i^{th} EPI in the lightfield, e_i, we find the set S_i of linear parameterized edges for all i. Then we match the edges in S_i with other edges in S_i such that resulting set of matches M_i minimizes a custom loss function. Finally, we find cross-epi correspondence between the matches using a bipartite graph cut.

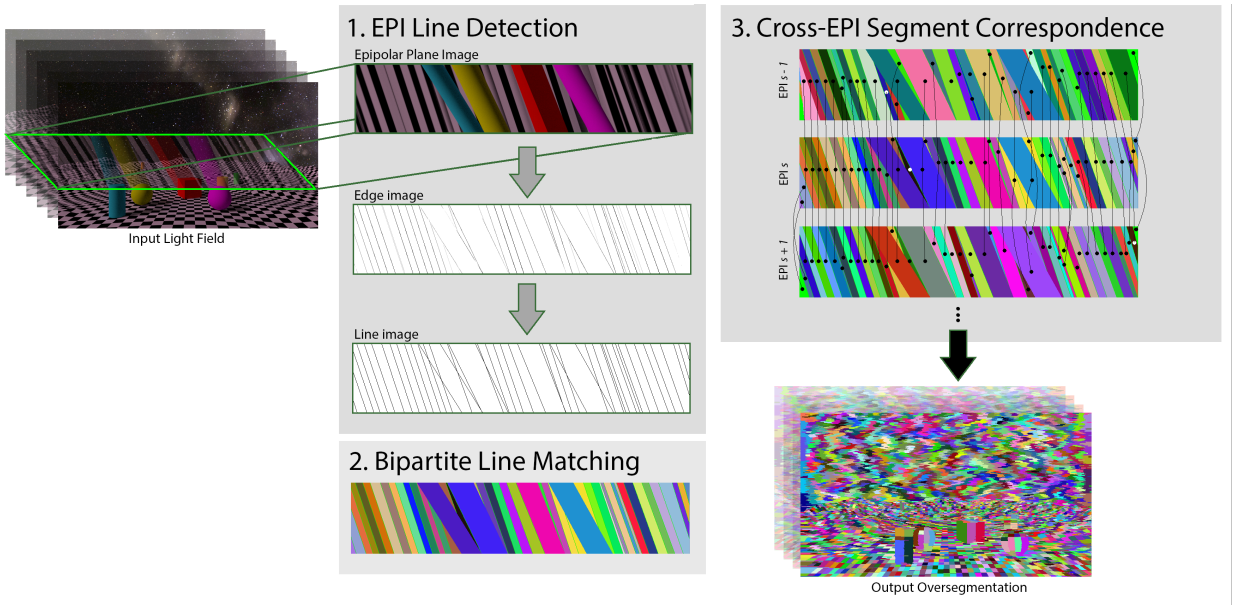

Figure 6: A high level overview of our pipeline. Figure taken from [13], but modified

3.3.1 Edge Detection

The edge detection step consists of two parts: detect the edge responses for each pixel and then fit parameterized lines to the edge responses. There are many methods to detect edge responses, perhaps the most well known of which is the Canny Edge Detector. Unfortunately, this method does not allow edges to bifurcate, which can produce incorrect edge responses in areas where edges intersect and noisy edge boundaries (see figure 7). Therefore, we instead use [17] to estimate edge responses for each pixel. First, we get the edge response value for each pixel, a value which becomes higher the more "edgy" a pixel is. We then run non-maximal suppression on these responses to produce an edge image.

Now that we have an edge image composed of the strongest local edge responses, we would like to extract the parameterized lines that form the edges. Due to the nature of EPI images, in lambertian scenes the edges will always be straight and non-horizontal. We exploit this by iteratively fitting lines to the edge image.

To fit the lines, we make a few assumptions. We assume (1) that every active pixel in the edge image lies on or near some true edge by at most some constant threshold α. We also assume (2) that every true edge in the image is represented by some pixels in the edge image. Finally, we assume (3) that each contiguous segment of pixels in the edge image represents part or all of only one true edge. That is, the segments representing two distinct true edges will be discontiguous. This is reasonable for non-intersecting edges, certainly, so only the clusters containing an intersection violate this assumption. Fortunately there tend to be several clusters that represent an edge, so the majority of clusters don't contain intersections even if their lines intersect.

Informed by assumptions (2) and (3), we segment the edge image by connectivity such that each cluster of connected pixels gets its own label. We then pick the cluster with the most pixels and fit a line to that cluster using least squares regression. Next, in accordance

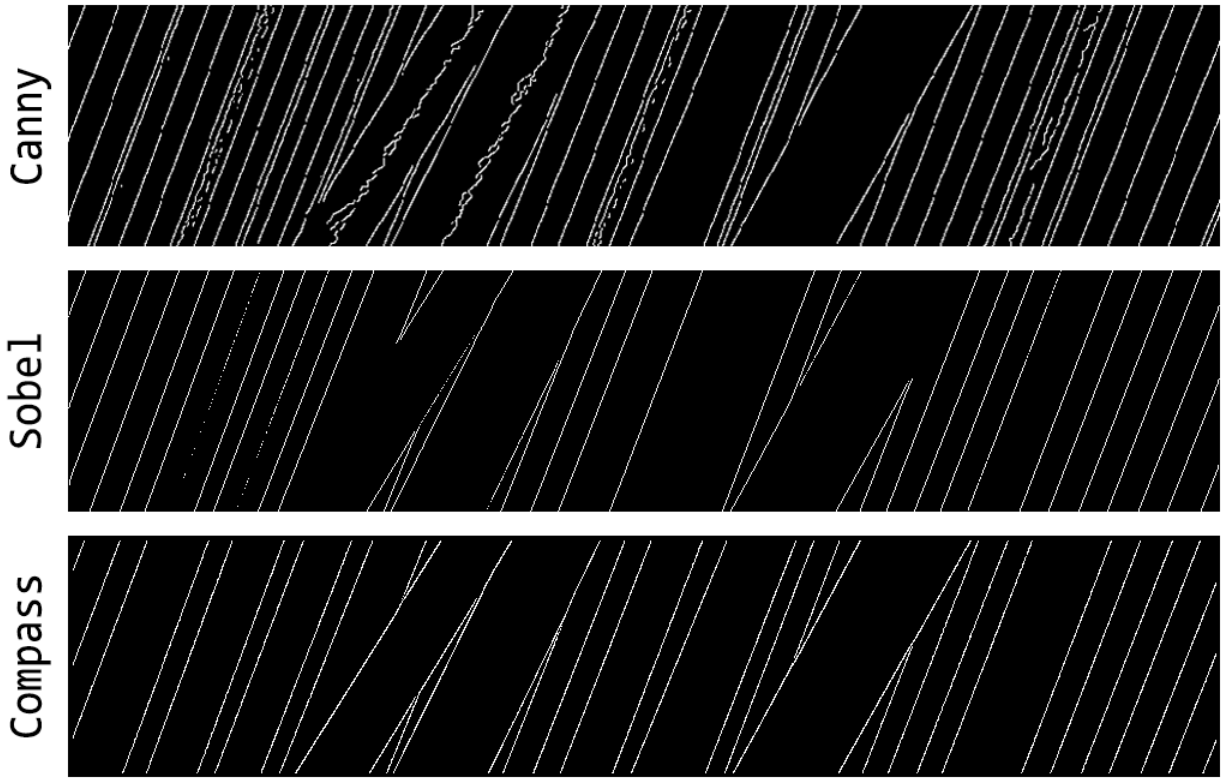

Figure 7: The compass edge operator performs better for this dataset than either canny or sobel edge detection methods

with (1), we create a set $P = \{P_1, \ldots, P_n\}$ of all active pixels within distance α of the parameterized line l, where P_i is a 2D location vector and P_i^l is the label of pixel i. We then refine our estimate by minimizing

$$\sum_{i=1}^{|P|} W_i \times log(distance\_to\_l(P_i)) \tag{1}$$

$$\text{where } W_i = \frac{\sum_{P_j \in P} \mathbb{1}_{\{P_j^l == P_i^l\}}}{|P|} \tag{2}$$

We use gradient descent to refine the line estimate, as the equation above is differentiable with respect to l. This tends to converge in a few hundred iterations and is therefore quite fast to compute.

Once we have refined the line estimate, we resample P using the new line parameters. We then repeat this process until the line converges to a stable location (4-5 times). We then record the final location of the line and delete all pixels within α distance of the line (see figure 8).

The motivation for the equation 1 requires some explanation. By assumption (3), we'd almost like to fit the line to just one cluster, as each cluster represents a true edge. However, some edges are represented by multiple clusters. Thus, we would really like to regress using all clusters representing that line. However, we don't know which clusters represent that line.

If multiple clusters represent a line, those clusters should contain colinear pixels. Conversely, if a nearby cluster represents a different line, that cluster's pixels should not be colinear with the other clusters. If we regress to all of the pixels in all nearby clusters, we will fit to neither true line, instead fitting to the average between them. This is undesirable. To avoid this, we weight clusters higher if they have more pixels within α of the current line estimate. This causes the line to move closer to the clusters that contain more pixels, in turn causing those clusters to be weighted higher in the next iteration. Within a few iterations, the line tends to converge to just one set of clusters.

Note also that we use log rather than L_2 distance. L_2 distance heavily penalizes outliers, but many of our points are outliers (edge pixels sampled from clusters representing different lines). The log distance metric ignores outliers and cares more about minimizing distance to the most points possible.

3.3.2 Edge Matching

Using the algorithm in the last section, we now have a set of n lines, $\{l_1, \ldots, l_n\}$ for our EPI. Each line is parameterized by $[x_0, x_1]$, where x_0 is the x coordinate of the intersection of the line with the top of the image, and x_1 is the x coordinate of the bottom intersection point. We assume that every true edge in the EPI is represented by a line, but some lines are extraneous and don't represent a boundary.

We would like to output a set of line matches, M, such that each match is in the form (l_i, l_j). We define a match to "cover" a set of pixels A if all elements in A are contained between l_i and l_j. A set of matches "covers" the union of the areas covered by its constituent

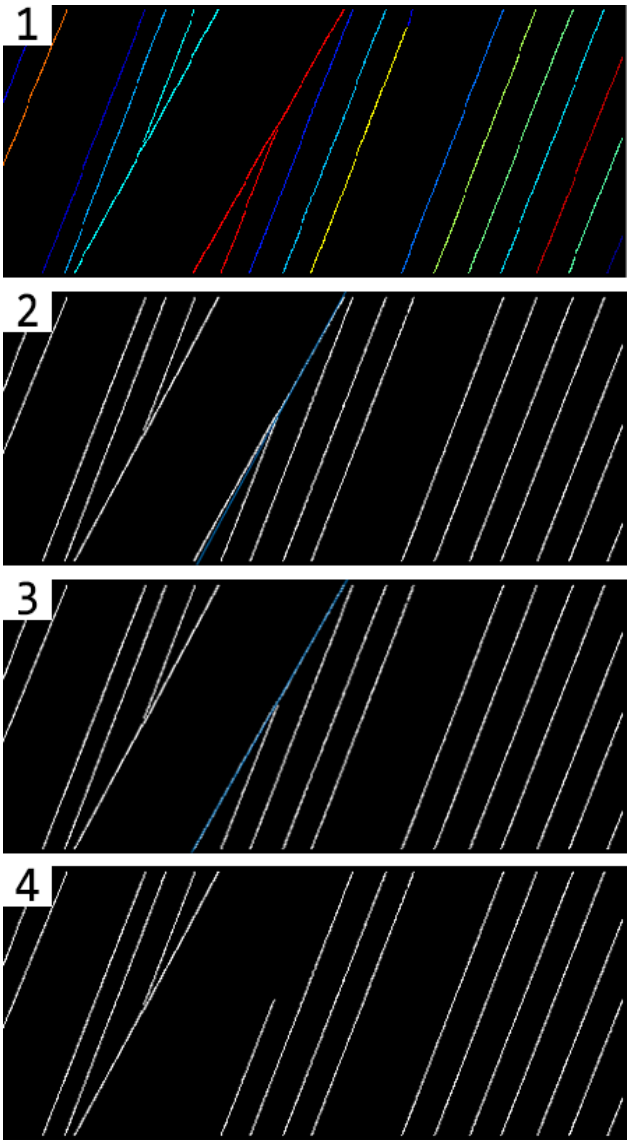

Figure 8: An overview of our line fitting algorithm. (1) We cluster the edge responses by connectivity; distinct colors represent distinct clusters. (2) We fit a line to the largest cluster using least squares regression. (3) We refine the estimate using equation 1. (4) We remove all pixels within α distance to the line.

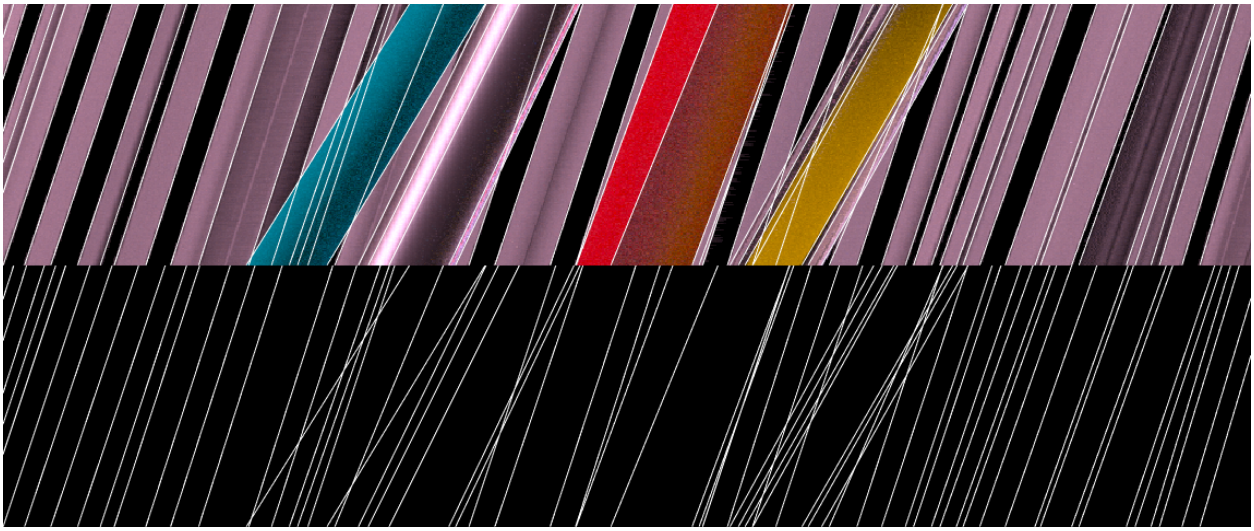

Figure 9: The result of our line fitting algorithm for an EPI. The top image is the original EPI with the parameterized lines superimposed, and the bottom image is the parameterized lines alone

matches. M should minimize the energy function $L(M)$ (defined below). Furthermore, M should meet these constraints:

- $\forall [l_i, l_j] \in M, i \neq j$, as a line should not match with itself.
- $\forall [l_i, l_j] \in M, l_i$ and l_j should not have a point of intersection located inside the EPI image, as the matching regions should be trapezoidal.
- M must cover the entire image.
- Each line must occur in at most two matches in M.
- WLOG $[l_i, l_j], [l_j, l_k] \in M \implies l_i(x_0) < l_j(x_0)$ and $l_j(x_0) < l_k(x_0)$ (a line can only be matched to once on the left and once on the right).

To compute the energy of a set of matches, M, we first create an ordered list of matches M'. To create M', we sort the matches in M by their depth (given by the angle of their boundary lines) such that $M'[0]$ is the region most in the foreground and $M'[n-1]$ is the region most in the background. Given M', we then define our loss function as:

$$L(M') = \sum_{i=0}^{|M'|-1} \alpha_1 * \mathit{EndsRatio}(M'[i]) + \alpha_2 * \mathit{ColorVariance}(M'[i], M'[1:i-1]) \quad (3)$$

Where

$$\mathit{EndsRatio}([l_i, l_j]) = \frac{\min(|l_i(x_0) - l_i(x_0)|, |l_i(x_1) - l_i(x_1)|)}{\max(|l_i(x_0) - l_i(x_0)|, |l_i(x_1) - l_i(x_1)|)}$$

The *EndsRatio* term encourages the top and bottom of the matched region to be of approximately equal length, resulting in regions that are parallelograms since EPI objects

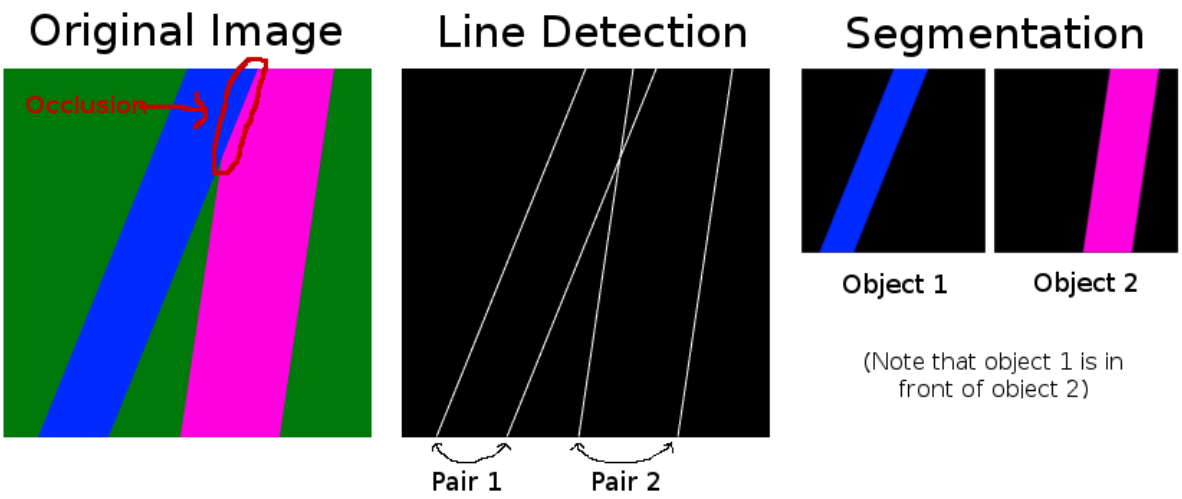

Figure 10: An example of an optimal edge matching solution

are usually parallelograms. The $ColorVariance([l_i, l_j], mask)$ term is the variance in color in the region between l_i and l_j after we remove from consideration the area covered by the matches in *mask* (in this case, all of the matches in front of match i). Finally, $\alpha_1 = 1$ and $\alpha_2 = 500$ are constant weights.

Minimizing this function is difficult. Originally we conducted a complete search of the space of valid matches, which yielded a globally optimal solution but took an extremely long time to run, as it was in $O(4^n)$. Due to this, we instead switched to a greedy algorithm. At each step, the algorithm considers all valid matches and picks the one that will increase the match energy the least. The algorithm terminates once the mask defined by its set of matches covers the entire EPI or it can no longer make any valid matches. In this way, it greedily minimizes the loss and maintains the constraints (as it never picks a match which would cause the constraints to be violated). An example matching output can be see in figure 11.

3.3.3 Cross-EPI Correspondence

Now that we have matches for each EPI, we need to find cross EPI correspondence between the matches. To find these "tracks" of matches, we first convert each match, $m = (l_0, l_1)$ into a 7D vector of $[l_0(x_0), l_0(x_1), l_1(x_0), l_1(x_1), r, g, b]$ where r, g, b are the average values of the match's red, green, and blue channels, respectively. Then, we construct a bipartite graph by connecting every vector from EPI_i to every vector in EPI_{i+1}. We then perform a bipartite graph cut on the matches, as in [19] to find the best matches in L1 space. Unfortunately, this forces the resulting tracks of matches to be continuous, even if the matches are no longer being pulled from the same underlying object. We therefore perform a post-process operation to split the tracks. Starting at the beginning of a track, we add the current match, m_i to a set S and compute the entropy of S. If the entropy of S is greater than a fixed parameter β, we split the track before m_i and start a new set, $S = \{m_i\}$. In this way, we find vertical correspondence across the image.

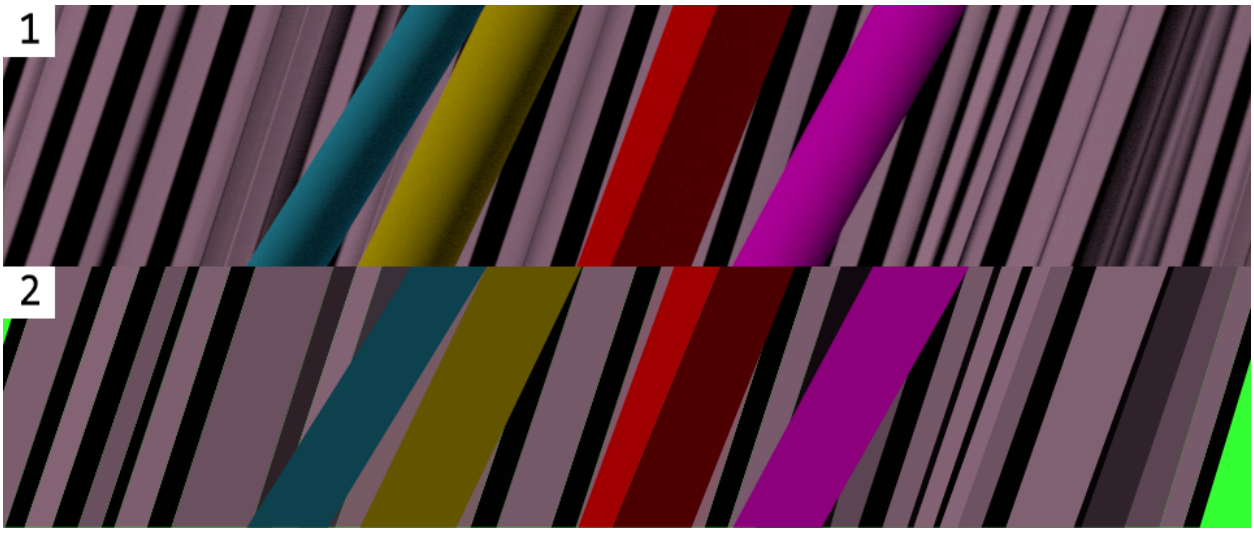

Figure 11: (1) The original EPI. (2) Our matched result. Note that we color each distinct match the average of the colors in the region that it covers. The bright green color represents regions not covered by any matches (there are no lines at the end of the EPI to match to, so these regions are not covered)

This step is not strictly necessary, as the previous two steps give us cross-epi superpixels. Since the lightfield is 3D, this is enough information to use to detect horizontal occlusions. However, it can be nice to have superpixels in all dimensions for purposes such as object selection.

3.4 Results

To visualize our results, we first run the segmentation algorithm on each EPI. We color each match the average color of the area covered by that match. We then reconstruct every view from these matches. The original and reconstructed central views are shown in figure 12. We also visualize the matched regions (resulting from step 3) for the central view in figure 13.

3.5 Conclusion

We present a method for segmenting a 3D large baseline dataset. This method produces reasonable segmentations for this dataset, but has its limitations. Since it uses a greedy algorithm for step 2, it can sometimes mismatch EPI segments, resulting in the object boundaries not being respected, as in figure 14. This could be improved by propagating matches from EPIs with lower losses from function (3) to EPIs with much higher losses. The algorithm also does not generalize well to small baseline lightfields. However, it inspired the work of [13]; their method generalizes this approach to 4D lightfields. I was also involved in the design of the method and writing the paper.

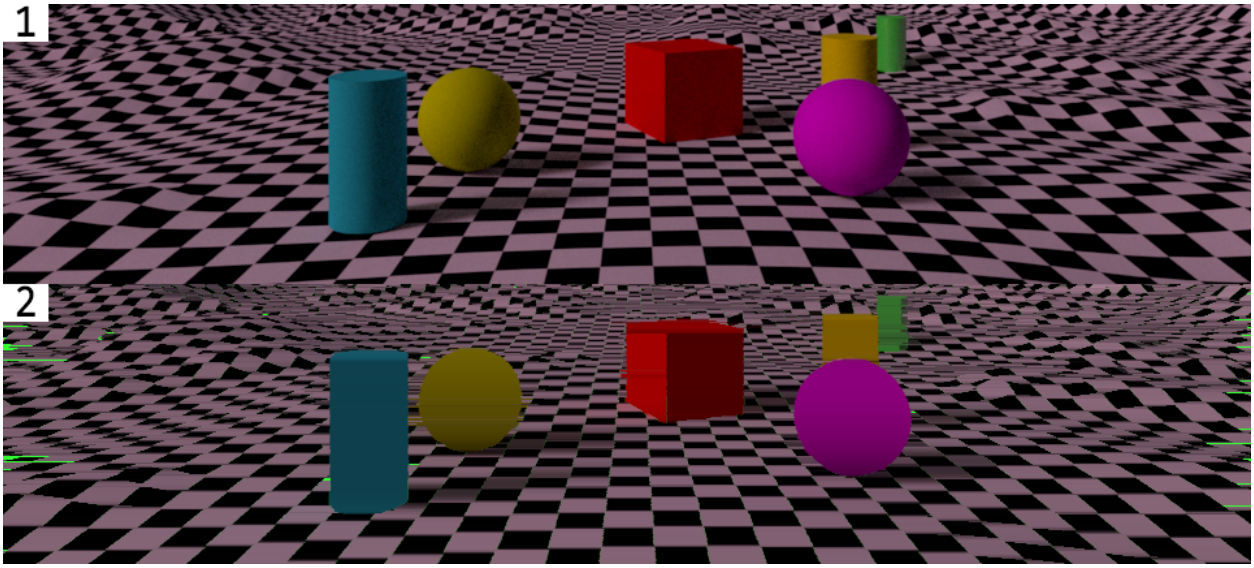

Figure 12: The segmentation of the central view produced by step 2 of our algorithm. To generate this image, we run the algorithm shown in Figure 11 on every EPI. We then stack the results and slice across each EPI (in the same way that we generated the EPIs initially, only we now generate views). We show here the central view. Note that this does not incorporate step 3; it only provides horizontal cross-image segmentations.

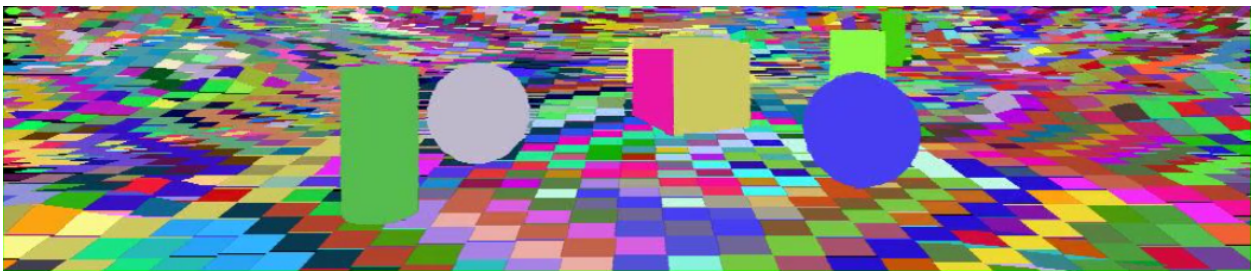

Figure 13: The final 3D segmentation for the central view. Note that this each segment is randomly colored; distinct colors represent distinct superpixels.

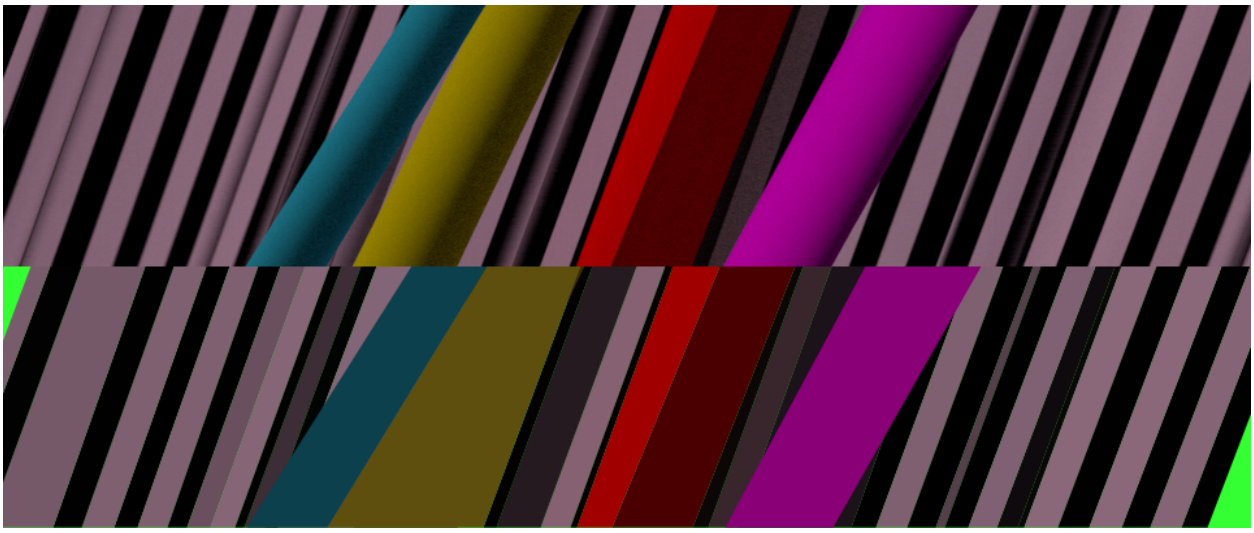

Figure 14: An example of a failure case of this algorithm. The yellow sphere matches to the boundary of the blue cylinder because increasing the total area decreases the per-pixel variance, even though the area being included is purple rather than yellow. This error tends to occur when the objects have noisy texture, as is happening here with the sphere's shadow.

4 Basic Editing Operations

Using lightfield superpixels we can perform basic editing operations.

4.1 Pixel edit propagation

Suppose that a user would like to change a pixel's color in the central view. We need to consistently propagate that change to all other views in an occlusion aware manner; if we just project the color change using the pixel's estimated disparity, we run the risk of projecting it over an occluding object. We will use our light-field segmentation to project in an occlusion-aware manner using the following method:

```
1 for each pixel, p, in user's edit:
2   set p to the new color in the central view
3   get the label, l, of p's segmentation in the central view
4   for each view, v:
5     project p to pixel p' in view v using the central view disparity estimates
6     get the label of the projected pixel, l'
7     if l == l'
8       set p' to the new color
9     else continue
```

Since the labels of objects in the segmentation are theoretically consistent across views, if p and p' have different labels, they must be representing different objects. Furthermore, since foreground always keep their labels during occlusions, it must be that p is occluded in view v. We implemented this algorithm and projected a simple edit using the segmentations from the previous section, shown in figure 15. For this example we use ground truth disparity,

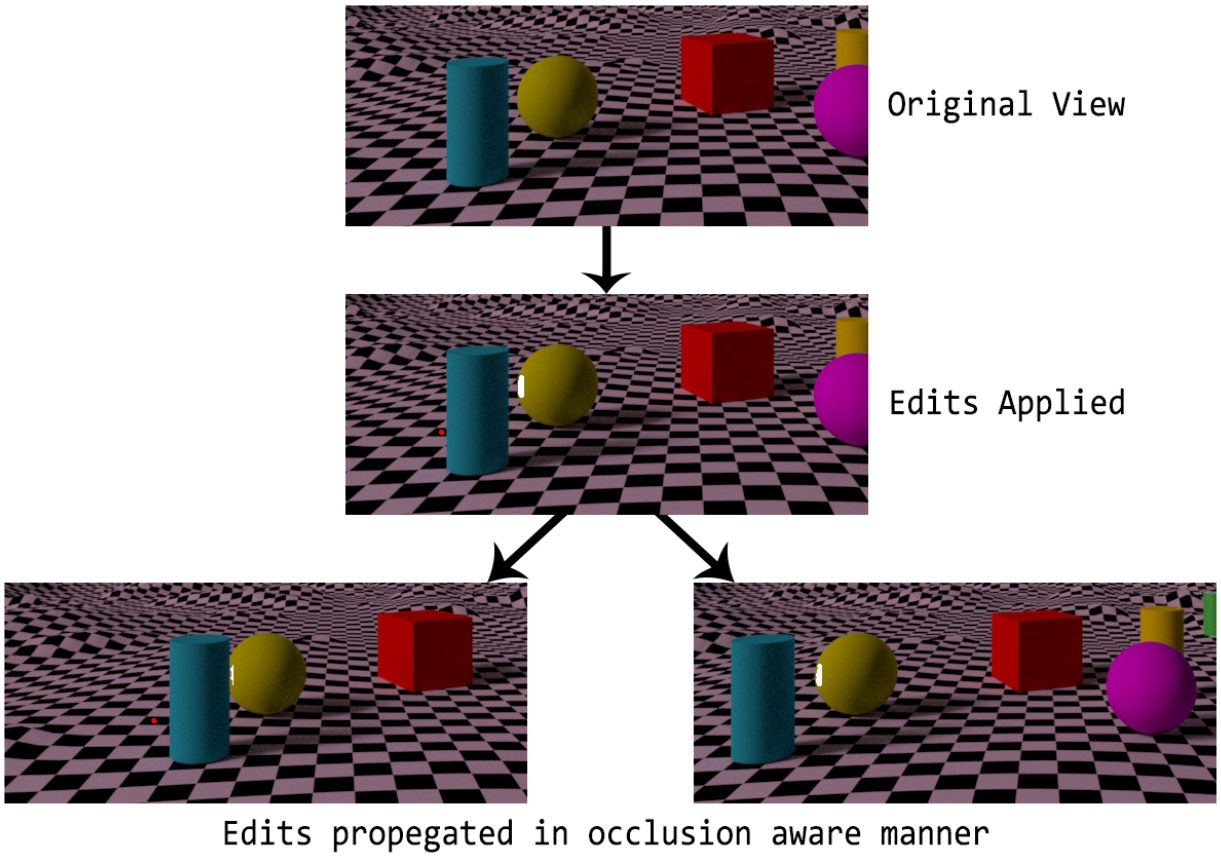

Figure 15: An overview of our pixel edit propagation system. A user performs an edit in some view (in this case the central view). We then use the estimated disparity of the pixels in that view to project the edited pixels to all other views. We can use the labels produced by our segmentation to detect if the pixels will be projected behind an occluding object.

but in an actual application we'd estimate the disparity of the central view using one of the several available methods.

4.2 Disparity estimate propagation

While the above method works well for editing the central view, it could potentially fail if a user wants to edit another view. As we only have disparity estimates for the pixels visible in the central view, if a user edits an occluded pixel, we won't be able to project their edit to the other views. We can solve this problem by using our superpixels to propagate disparity estimates to every pixel in the following manner:

```
for each pixel, p, in the central view:
  for each view, v:
    project p to pixel p' in view v using the central view disparity estimates
    get the label of the projected pixel, l'
    if l == l'
      set the disparity of p' to the disparity of p
    else continue
```

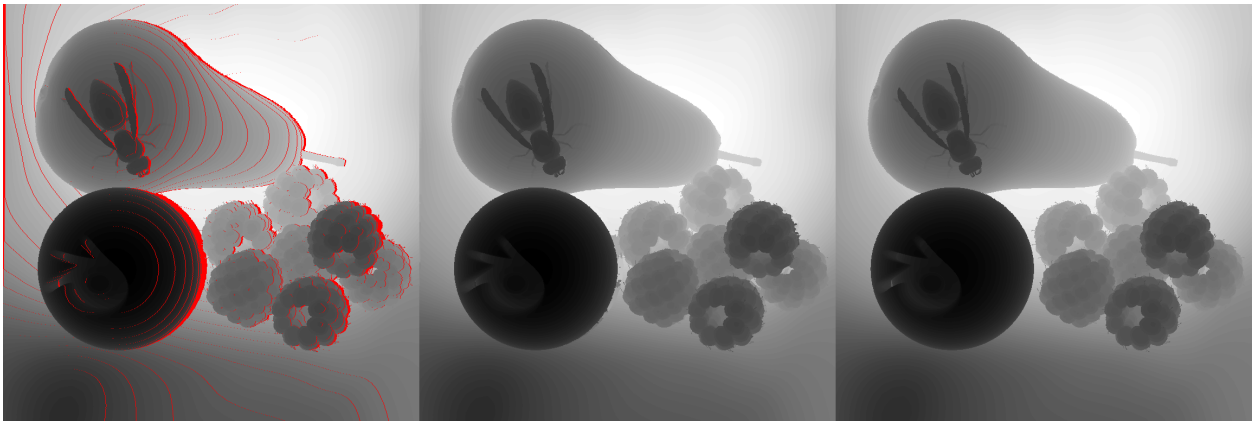

Figure 16: Left to right: the depth projection with missing depths (marked in red); the depth projection with our estimated depths; the ground truth depth

```
for each pixel, p, in the lightfield without disparity:
  get the label, l, of p
  d' = average disparity of the pixels with label l
  set the disparity of p to d'
```

This assumes that superpixels represent regions of objects' surfaces with basically uniform disparity and uses that disparity to infer the missing disparities. A more sophisticated method might fit a plane to the superpixel's disparity map and interpolate on that basis, but we found that this method works for small baseline lightfields, at least. While this method fails for objects that are totally obscured in the central view, it produces reasonable results, shown in figure 16. For this example we use the superpixels produced by [13] and the Still Life lightfield from the old HCI dataset, as we were unable to obtain dense disparity estimates for the Milky Way dataset. Note that we did use ground truth rather than estimated disparity

The incorrect depth estimates occur largely as a result of incorrect superpixel boundaries. We post-process the depth estimates with a 3x3 median filter to alleviate some of the floating pixels, but this does not solve large clusters of errors. A more robust method might additionally use the color of the nearby regions to detect and correct erroneous depths, but our simple method works well enough for demonstration purposes.

4.3 Object selection

In order to facilitate more advanced editing operations, a user may wish to perform a cross-view object selection. This is trivial when we have superpixels. The user merely selects which superpixels they would like the selection to contain by drawing "squiggles" on any view. Since the superpixels theoretically respect object boundaries, there will be some set of superpixels which contains the object they'd like to modify. We create a set, L of the labels of the pixels contained within the squiggles. We then perform the cross view selection by selecting all pixels whose label is a member of L. This approach is demonstrated in figure 17.

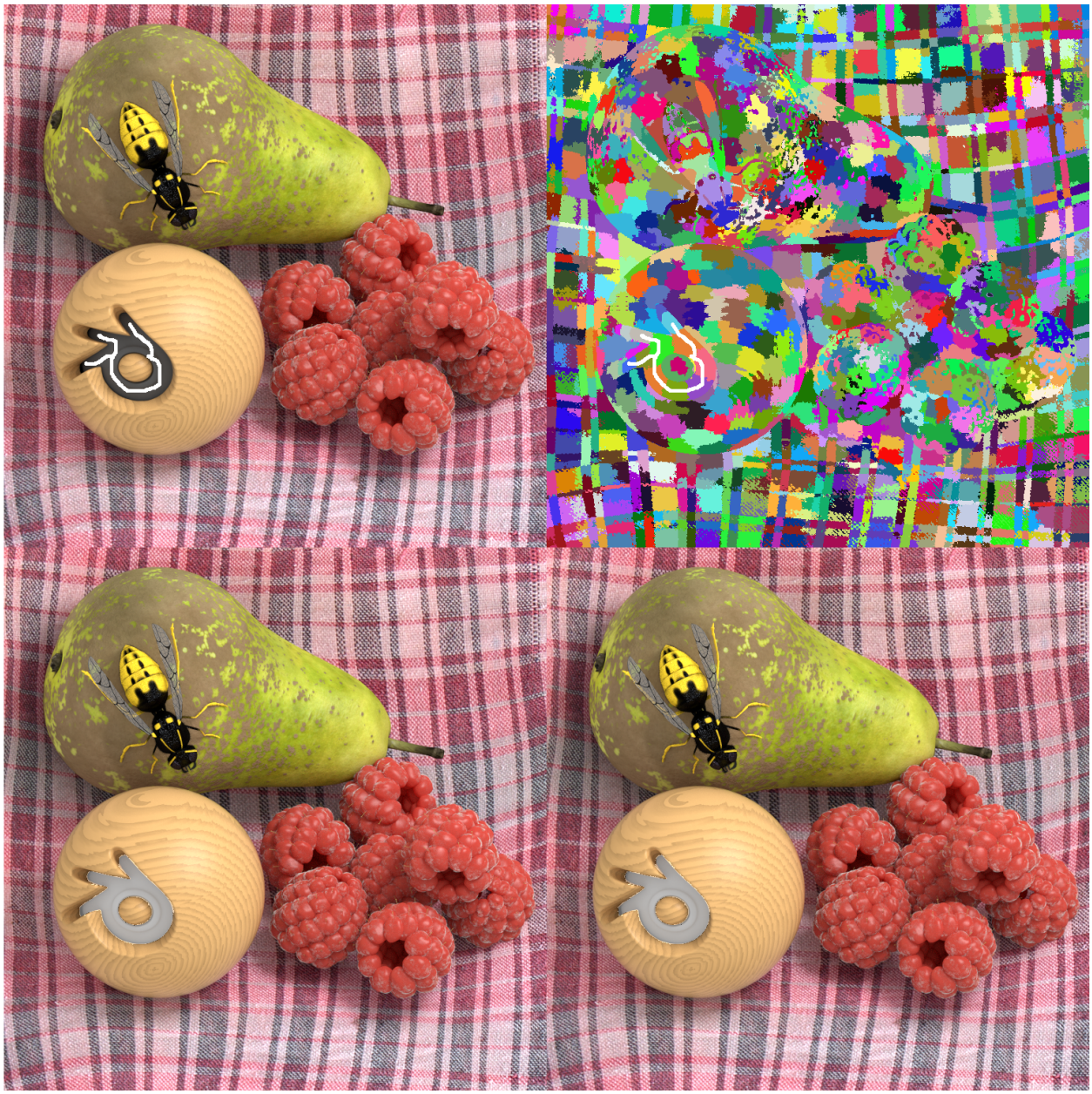

Figure 17: An example output of our selection method. From top left to bottom right: the central view with the user selection; the superpixels in the central view with the user selection; the resulting selection in the central view; the resulting selection in the leftmost view

5 Consistent Object Removal

5.1 Introduction

A more sophisticated editing operation that we may wish to perform on lightfields is object deletion. That is, given an object in our scene, we would like to remove the object by "inpainting" or "hallucinating" what is probably obscured by the object. While there has been considerable research on object inpainting in single view images, the problem becomes more complicated when applied to lightfields. In addition to the problem of hallucinating background information in a single view, we also need to be able to propagate that information in a view consistent manner.

Although it is generally more difficult to inpaint objects in lightfields, they do have the advantage of having more information. While in a single view image there is little to no explicit information about the scene behind an object, we can use the multiple views present in lightfield images to "see around" an object. Specifically, we can propagate the background information from views in which the background is visible to those in which it is not. While this is useful for large baseline lightfields, the utility of this technique decreases with the baseline, as we can't see as far around the object.

Previous inpainting methods on structured lightfields work best with very small baselines. In general, the user annotates an object in the central view, the algorithm uses some single-view infilling method and projects the results to all other views (see figure 18). These methods work well for removing a foreground object from a small baseline lightfield. However, they do not work well for more complex scenes where multiple objects can occlude one another and they do not take advantage of the background information present in other views. In part, this is because these algorithms generally only have access to disparity estimates for the central view, and thus cannot project pixels which do not appear in that view. Therefore, the algorithms are unable to "look around" objects.

We present a novel approach for lightfield inpainting that exploits the information present in other views using lightfield superpixels. This enables us to both infill more complicated scenes and to propagate background information from other views.

5.2 Related Works

5.2.1 Single Image Inpainting

Many papers exist for single-view inpainting. Classical algorithms propagate color information from exemplar regions into the masked region [20] [3] [6]. They can use user-defined boundaries to preserve object structure in the infilled regions, or they can automatically attempt to preserve structure by maintaining strong edges. These methods tend to produce good local results, but they lack semantic understanding of scenes. Thus, they do not perform well when applied to global inpainting tasks; for instance, they tend to produce unrealistic results when the masked region is large. Since they have no semantic understanding, they also tend to fail when objects need to maintain a particular global pattern in order to be recognizable, such as in the case of faces.

Recently neural networks have emerged and have surpassed many classical algorithms in both speed and accuracy. Modern learning based approaches use convolutional neural

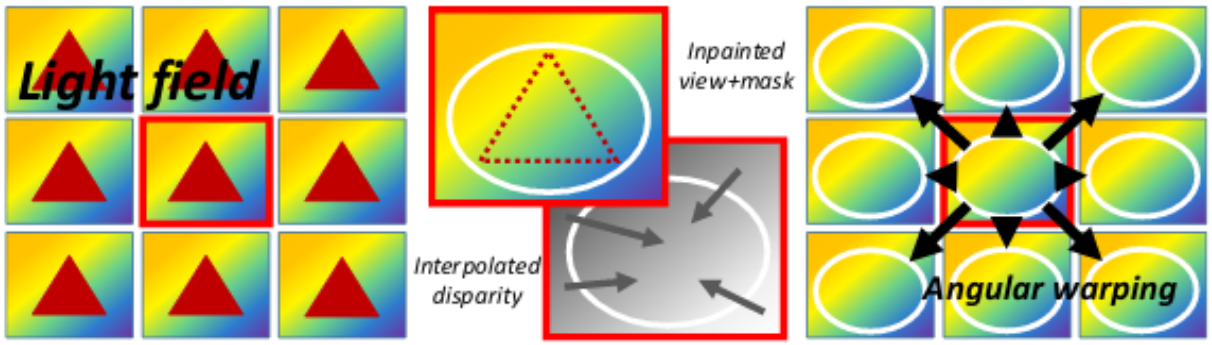

Figure 18: The general pipeline of non-segmentation based lightfield infilling methods (figure taken from [2])

networks to infill masked regions. They have achieved impressive results, and are able to have a better semantic understanding of scenes [11] [24]. However, while standard convolutional networks are good at globally consistent infilling, they also tend to produce textureless and distorted local results.

In an attempt to remedy this problem, [16] introduced the Partial Convolutional (PConv) layer. We will discuss this layer in detail soon, but essentially each layer inpaints slightly more than the previous, propagating information from the edge of the unknown region further into its center. This, they claim, helps prevent the network to being sensitive to whatever value the masked region was initialized to. It also helps propagate detail rather than producing uniform textures in the masked region. Using this novel approach, they achieved state-of-the-art accuracy on infilling tasks with arbitrarily shaped masks.

5.2.2 Structured Lightfield Inpainting

As previously discussed, lightfield inpainting is generally a more difficult task than single-view inpainting. Nevertheless, researchers have created several methods for performing this operation. Yatziv et al. [23] first align all corresponding pixels in the lightfield by warping each view. They then propagate pixels that are obscured from views in which they are visible. Finally, they use a high-dimensional patch-based texture synthesis method to infill any regions that are still missing information.

Le Pendu et al. [15] treat the lightfield as a low rank matrix. This enables them to take an arbitrary infilled region from the central view and propagate it to all other views using a low rank matrix completion algorithm. Their method is agnostic as to how the color information in the central view is infilled, focusing rather on propagating the infilled data once it's been created.

Allain et al. [2] introduce a warping method that very quickly performs infilling on small baseline lightfields. First, they estimate the disparity of the pixels in the central view using a fast, noisy approximation. They then use a single view inpainting algorithm to fill a user-defined mask in the central view and project that region to all other views. Their method works well with small baselines and clearly delimited foreground objects, but fails in more complex scenes with multiple depth layers.

5.3 Method

We introduce a segmentation-based method for consistently inpainting a lightfield. First, we use the method in 4.2 to propagate disparity to every pixel in the lightfield. Then, we use the method in 4.3 to select some set of superpixels, S, representing the object to be removed. Using S, we can consistently create a mask of the object for every view. Let M_v represent the mask for view v.

Our ultimate objective is to infill all masked pixels in all views. Consider the central view c with mask M_c. It may be that some pixels occluded in M_c are visible in other views. We would like to project those pixels to view c so that we can use them to infill M_c. If we project a pixel p_v from view v such that $p_v \notin M_v$, we can use that pixel to fill in M_c. Due to occlusion, it may be that several pixels from different views project to the same location in c. In this case, we keep the pixel with the highest disparity, as it represents the closest object, which should occlude the background objects. See figure 19 for our propagation result.

If the disparity is wide enough and the object is small enough, we may fill all of M_c in this step. If, however, the object is wide enough to obscure some background from all of the views, it may be that we need to hallucinate background for the remaining masked region, M_c'.

As discussed above, there are many methods for inpainting RGB images. Any method, automatic or manual, can be used for this step. For the example above, we use Photoshop to inpaint the region in the central view.

Now that we have inpainted color information we need to project it to the other views. Technically there is no disparity for this region, as the pixels exist only in one view. Therefore, we must also hallucinate the disparity. Other methods use low-rank matrix completion algorithms [15] or superpixel depth propegation techniques [2] (like the one we used for the central view occluded pixels). We introduce a novel depth infilling technique using a convolutional neural network, which is discussed in detail below.

Our network gives us a disparity estimate for all of the infilled pixels, so we project these pixels to all views. Finally, we repeat this process of infilling M_v for every other view v in the lightfield, if we didn't already infill all of M_v by projecting our infilled regions from other views. Our results for this method are shown in figure 20.

5.4 Depth Infilling Network

There are many existing networks that perform monoscopic depth estimation from a single RGB image. Godard et al. [9] use an unsupervised network using pairs of stero images. They use the disparity estimates to project one view to the other, and use the difference in the resulting images as their loss function. Kim et al. [14] simultaneously train two adversarial networks, one of which detects global inconsistencies in the output, and the other of which detects local anomalies. They then merge the outputs of these networks using a variational framework and use it as their loss function. Kumar et al. [7] use a GAN to learn a loss function for predicting depth outputs for monocular images.

Our task is different than monocular depth estimation, however. In addition to having an RGB image, we also have a partially completed depth image. We would like to somehow exploit the existing depth information. There are non-learning approaches to this problem [12],

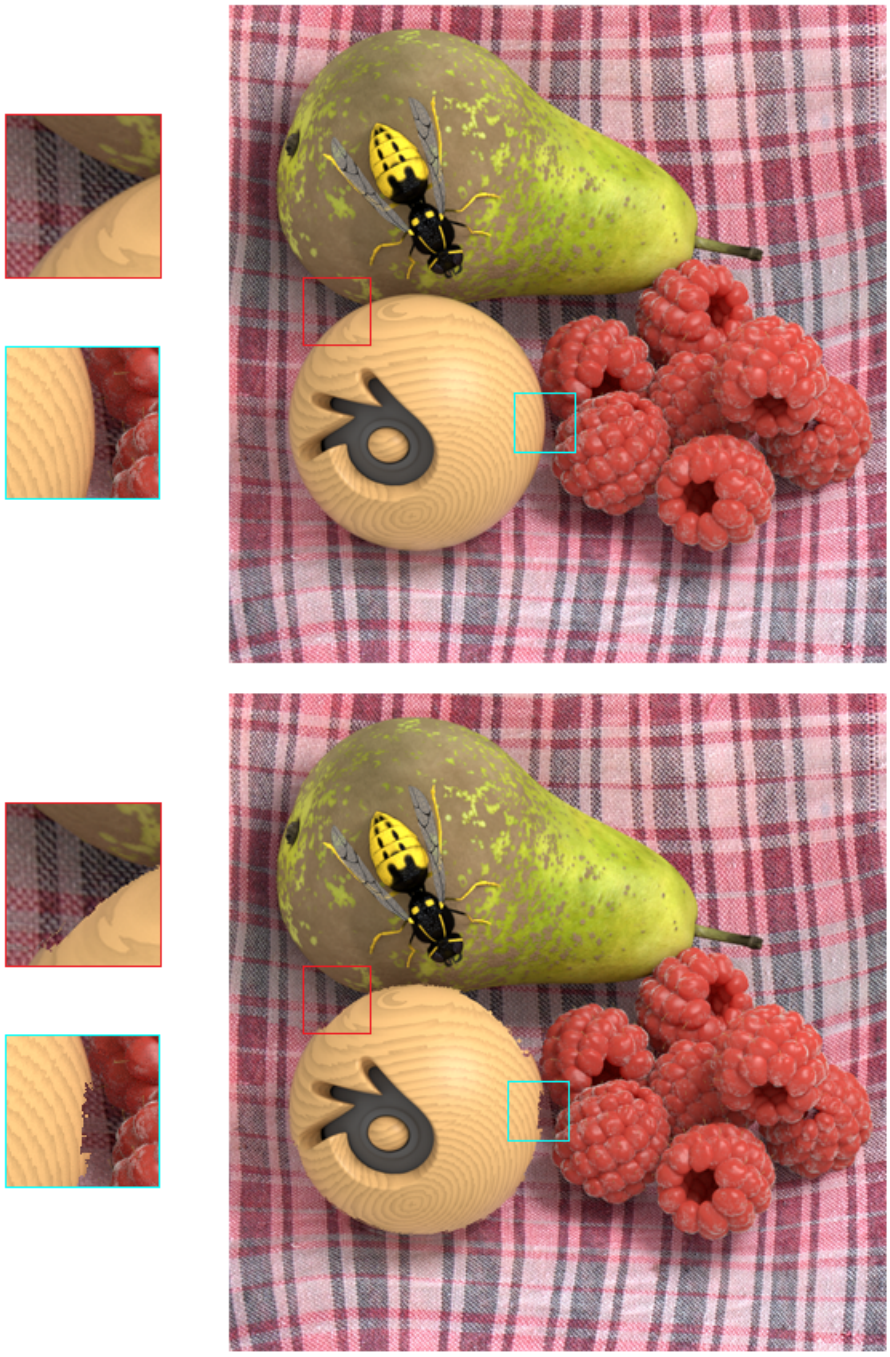

Figure 19: Results of propagating the pixels from other images into the central view using our depth infilling algorithm. The baseline is small, so not very much information gets propagated.

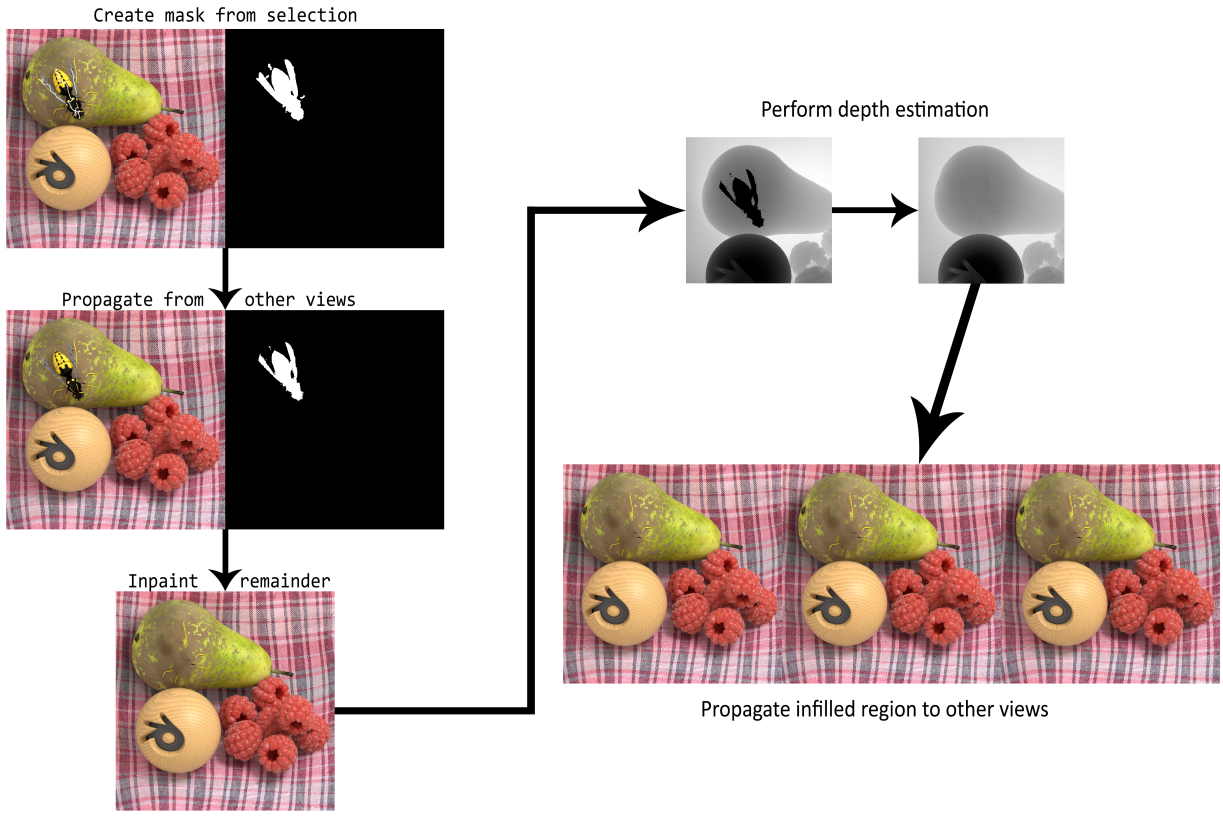

Figure 20: Our infilling pipeline. Our CNN can only operate on 256x256 images, so we have to crop and downsample the image in order to predict the depth. The masked depth input is on the left and our depth prediction is on the right. We show our final results for the leftmost, center, and rightmost views.

but given the efficacy of neural networks for similar tasks, we decided to try this approach. Note that we were not trying to achieve state-of-the-art performance in monocular depth estimation: that is beyond the scope of this research project. Instead, we are attempting to find a good way to use the existing depth information to inform the depths of the infilled region.

5.4.1 Datasets

One disadvantage of neural networks is the large amount of data that we need to train them. In this case especially, there are a dearth of existing RGBD datasets. There is the *NYU Depth v2*, which contains 500,000 RGB+D images taken with the Kinect v1. Unfortunately, the Kinect v1 tends to produce rather noisy depth images. The *Make3D* dataset contains very accurate depth maps for their images, but there are only 500 images; not enough to train a network on. *KITTI* is well-known, but it has a very limited domain (street scenes) and sparse depth maps. Therefore, we decided to use the *DIML Kinect v2* dataset which contains 220,000 RGB+D images taken with the Kinect v2. The Kinect v2 produces much better depth maps than the Kinect v1.

5.4.2 Network Loss

Another problem that we had to address was the network loss. Recent RGB infilling papers [11] [25] train local and global discriminators using adversarial architectures to ensure that the network preserves both high-level structure and low-level consistency. However, the current state-of-the-art, Liu et al. [16] does not use a GAN. Instead, they define their loss function using a combination of perceptual loss [8], style loss using a Gram matrix, TV loss, and L_1 loss. The perceptual loss takes the L_1 loss of the projection of the ground truth and output images into a different feature space using the pretrained Image-Net's VGG16. The idea is that this projection will represent high-level information about the image, and will therefore help the output to look realistic.

Unfortunately, as we are training our network to generate depth images rather than color images, we cannot use the perceptual loss term. ImageNet is trained on RGB images, and therefore does not work for depth images. In a future work, we could attempt to retrain ImageNet to encode depth images, but that was beyond the scope of this project. Therefore, as the major part of their loss function was unavailable, we instead used a simple L_1 loss. Once again, we were not attempting to achieve state of the art results; just design a method to exploit the existing depth information. Still, we found that L_1 loss worked well enough for our purposes.

5.4.3 Network Architecture

Our network architecture is inspired by [16]. They use a U-net with skip layers to encode the input and reconstruct the infilled image. Instead of using convolutional layers, as in a standard U-net, they instead use "Partial Convolutional" layers. Partial Convolutional, or PConv, layers are like standard convolutional layers that also take a binary mask as input. Specifically, let $\mathbf{X}$ be the pixels in the convolution, $\mathbf{M}$ be the pixels from the binary mask

in the convolution, $\mathbf{W}$ be the convolutional weights, and b be the bias. In this case, the resulting pixel after the convolutional operation is performed is defined as:

$$x' = \begin{cases} \mathbf{W}^T(\mathbf{X} \odot \mathbf{M}) \frac{\text{sum}(\mathbf{1})}{\text{sum}(\mathbf{M})} + b & \text{if sum}(\mathbf{M}) > 0 \\ 0 & \text{else} \end{cases}$$

We also update the input mask as follows for each layer:

$$x' = \begin{cases} 1 & \text{if sum}(\mathbf{M}) > 0 \\ 0 & \text{else} \end{cases}$$

In this way, each layer propagates some information from the edge of its masked region and erodes the mask. According to [16], this allows the network to ignore the initialization value of the masked region and better propagate information into the mask. [16] claims that these PConv layers are largely what allows them to achieve state-of-the-art performance on their image infilling task.

5.4.4 Network Results

Using their architecture, we implemented a U-net with PConv layers (figure 24), fed it the RGB and masked depth channels, and trained it on the L_1 difference between its output and ground truth. Unfortunately, this produced very bad depth maps: the depth boundaries did not seem to conform at all to the edges present in the RGB image. We found that this was because PConv layers mask *all* output channels, so the RGB information was being lost after the first layer.

Our subsequent network design fixed this issue by separating the depth and RGB channels into parallel networks. The RGB information was processed with standard convolutional layers, but the depth was still processed with PConv layers. In order to allow the depth estimates to respect the color edges, we feed each depth layer the concatenated output of the previous depth channel and the corresponding RGB convolution (figure 25). The images produced by this network, while better than the first, were very blurry. Blurry images are common with L_1 and L_2 losses, however, so we tried several other losses including MS-SSIM, and L_1 Charbonnier, but none of these yielded better results.

Finally, we tried replacing the PConv layers with standard convolutional layers. Surprisingly, this produced far better results than the partial convolutional network, which seems to conflict with the results of [16]. The results for all three networks are given in figure 21.

5.4.5 PConv vs Convolutional Layers

We found the fact that straight convolutional layers work better than partial convolutional layers in this task surprising. We developed a few theories to explain the discrepancy:

1. Some feature of the second network's architecture (figure 25) is causing the network to not learn well. Perhaps the idea of the parallel RGB network is fundamentally flawed somehow.

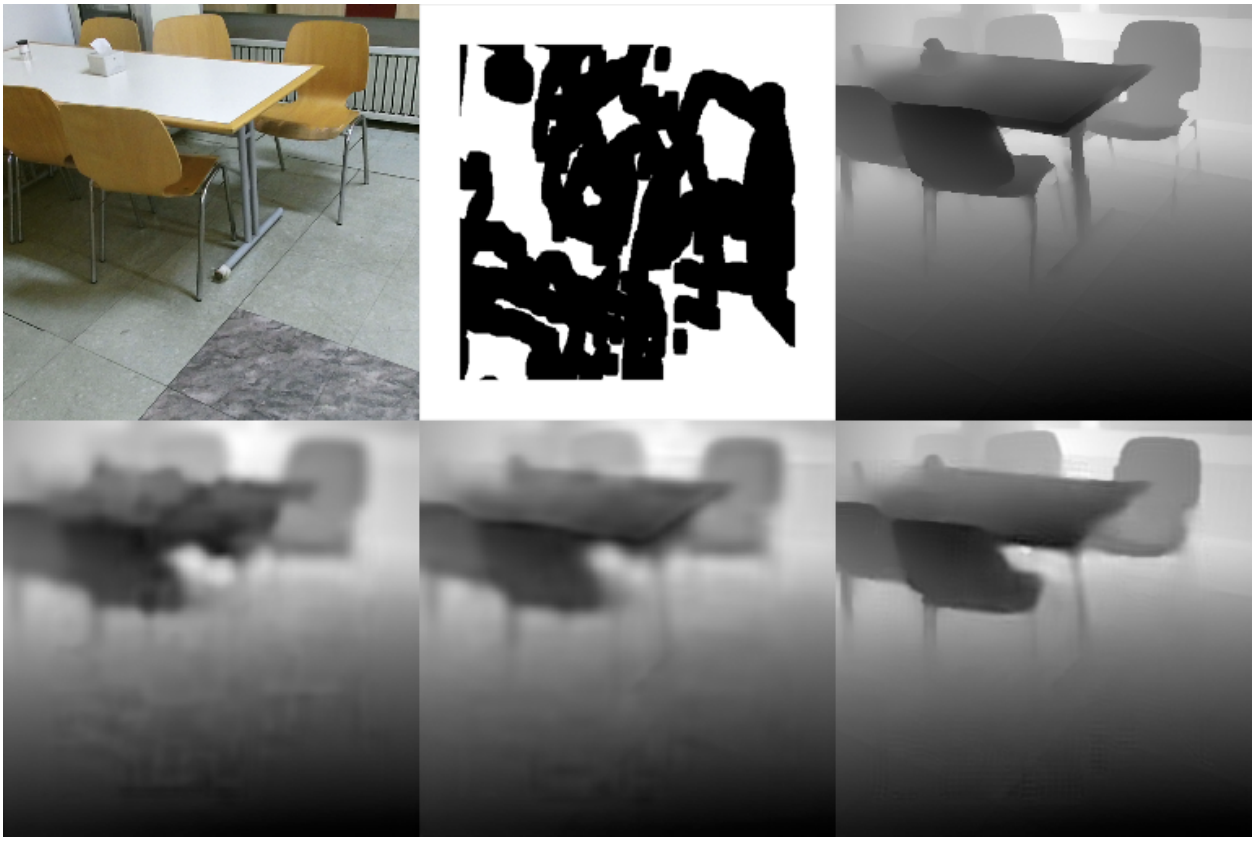

Figure 21: *Top row:* Input color image; mask for depth (depth information is set to 0 where mask is black); ground truth depth. *Bottom row:* original PConv network output; second PConv network output; CNN output

2. PConv layers do not work well with simple loss functions like L_1. Perhaps this is because PConv layers mask out large portions of their output, and thus have a difficult time encoding/learning global features of images. [16] found that PConv layers outperform convolutional layers in inpainting tasks largely because their loss function used a term based on ImageNet, a network which is already trained to encode high-level features. When we used L_1 loss, we removed that term, and thus crippled PConv layers.

3. Perhaps PConv layers have an especially hard time learning relative geometric relations between scene objects, and thus cannot predict depth well. This seems unlikely given that they can infill RGB images in a geometrically consistent manner, but it's possible that they don't do well with relative scales.

To differentiate between theories (1) and (2), we trained a network using the architecture in figure 24 and an identical network using convolutional layers instead of PConv layers. We used L_1 loss and asked each network to infill RGB images (to avoid the problem presented in (3)). If theory (1) is correct, we would expect to see the PConv network perform at or above the convolutional network, and if theory (2) is correct, we would expect the convolutional network to do better.

Our experimental results support theory (2), as seen in figure 22. The PConv network could not even learn to reconstruct the given ground truth inputs, and seemed to favor a sepia output. The convolutional network, while it did not do well, did learn to produce better images in half of the time (we likely could have gotten even better images if we had tuned the hyperparameters, but these results served to answer our experimental question). The PConv network was given many iterations to learn, but it did not improve past a loss of 0.4 (see figure 23). From this we conclude that its failure to learn was a result of the network itself and not the hyperparameters.

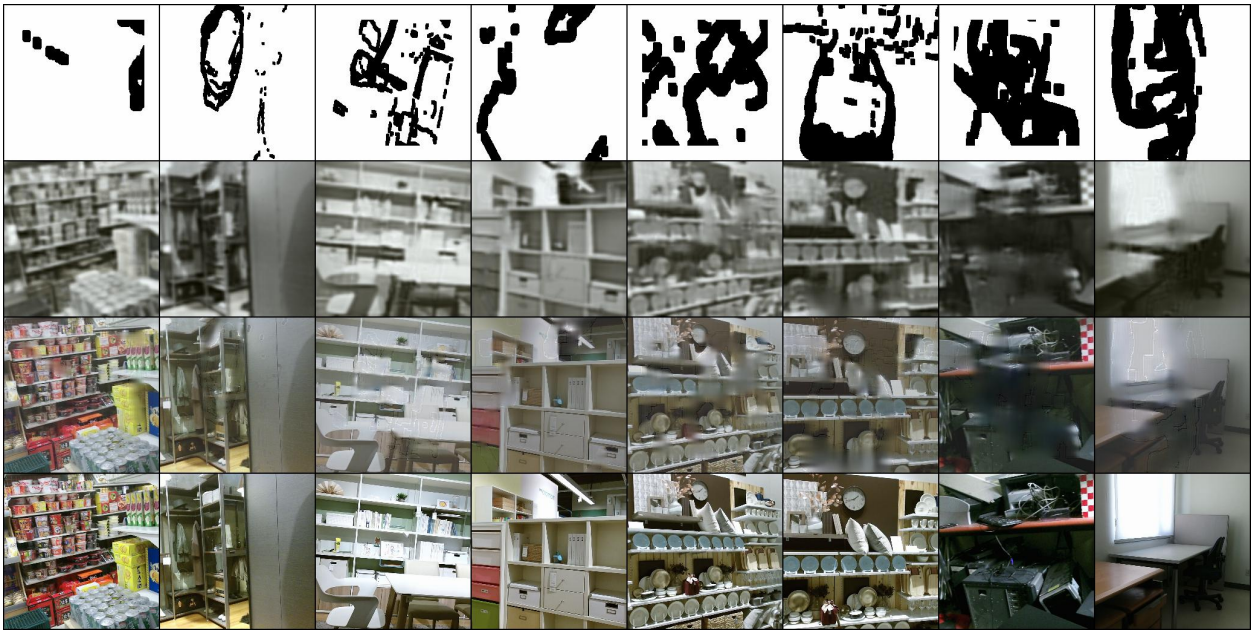

Figure 22: Top to bottom: The mask (black is masked out); the PConv network output; the convolutional network output; the ground truth image. Note that the input to the network is the ground truth image time the mask.

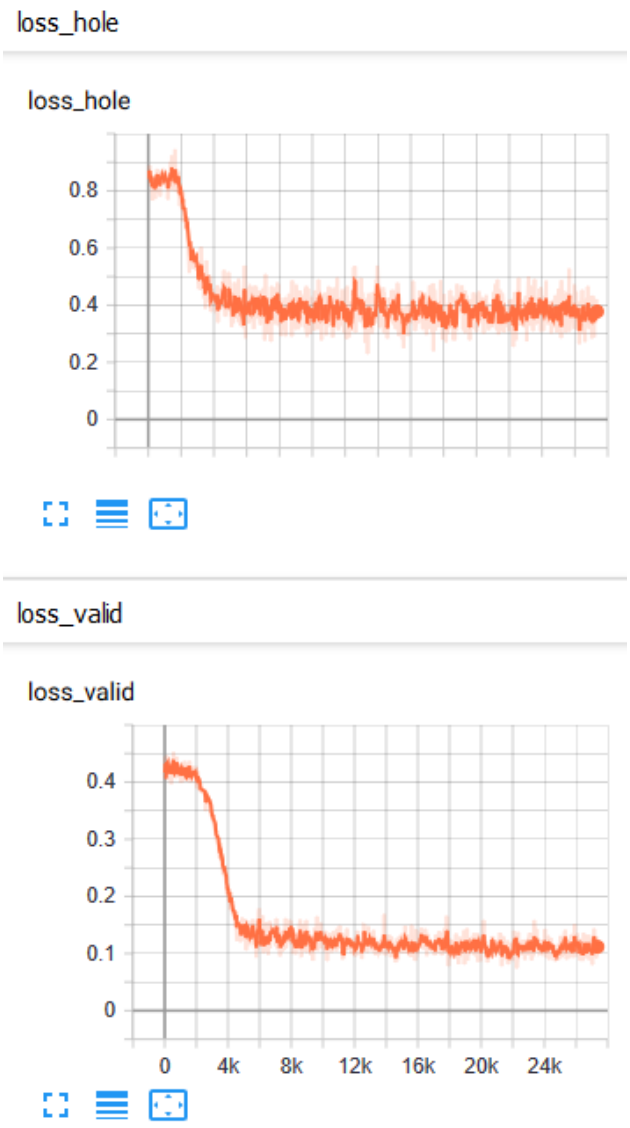

Figure 23: The loss of the PConv network for rgb reconstruction. `loss_hole` is the L_1 loss of the masked region and the ground truth while `loss_valid` is the L_1 difference between the non-masked region and the ground truth.

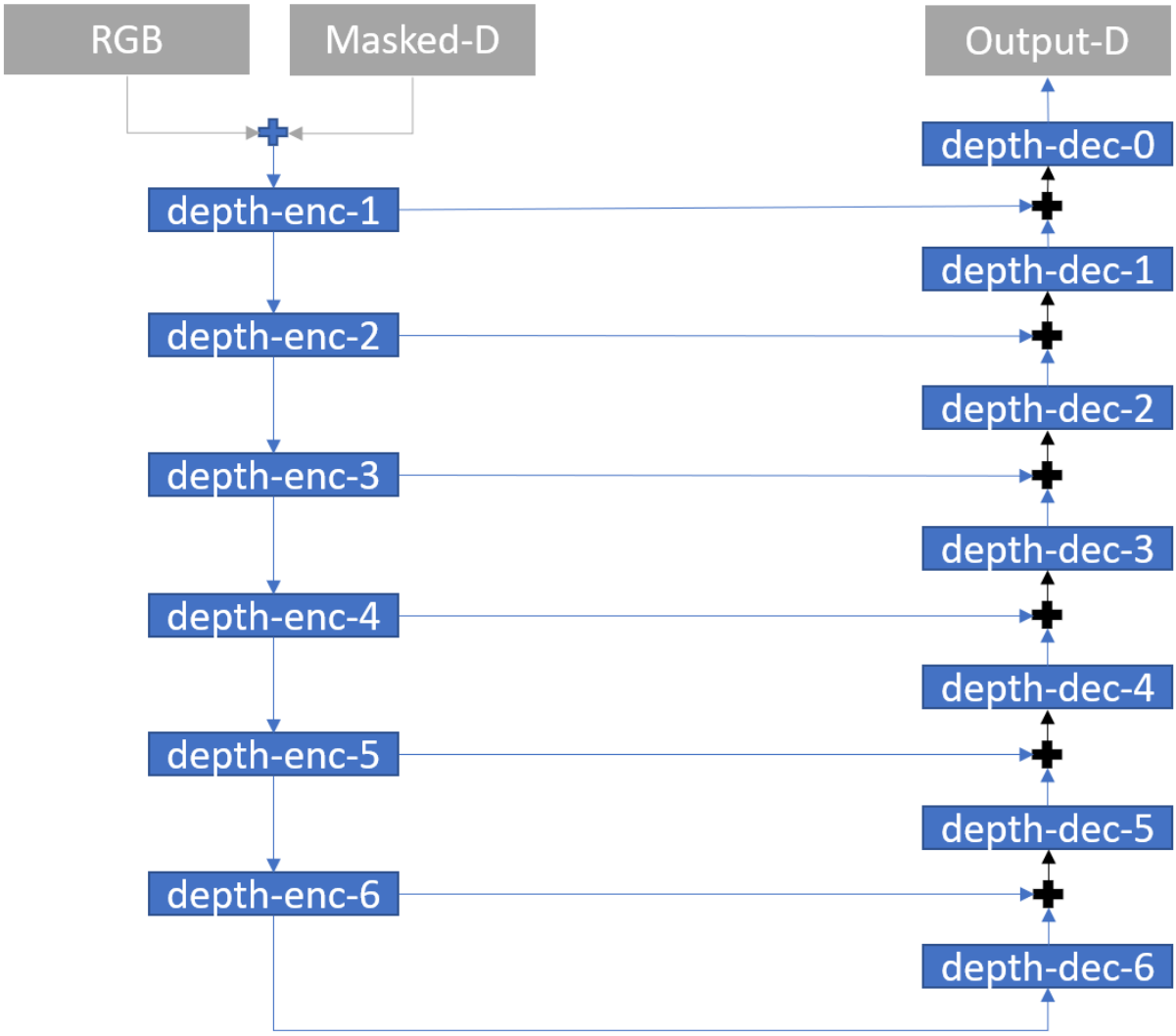

Figure 24: The original PConv network. All layers are PConv layers. The `enc` layers downsample the image to half size and the `dec` layers upsample it.

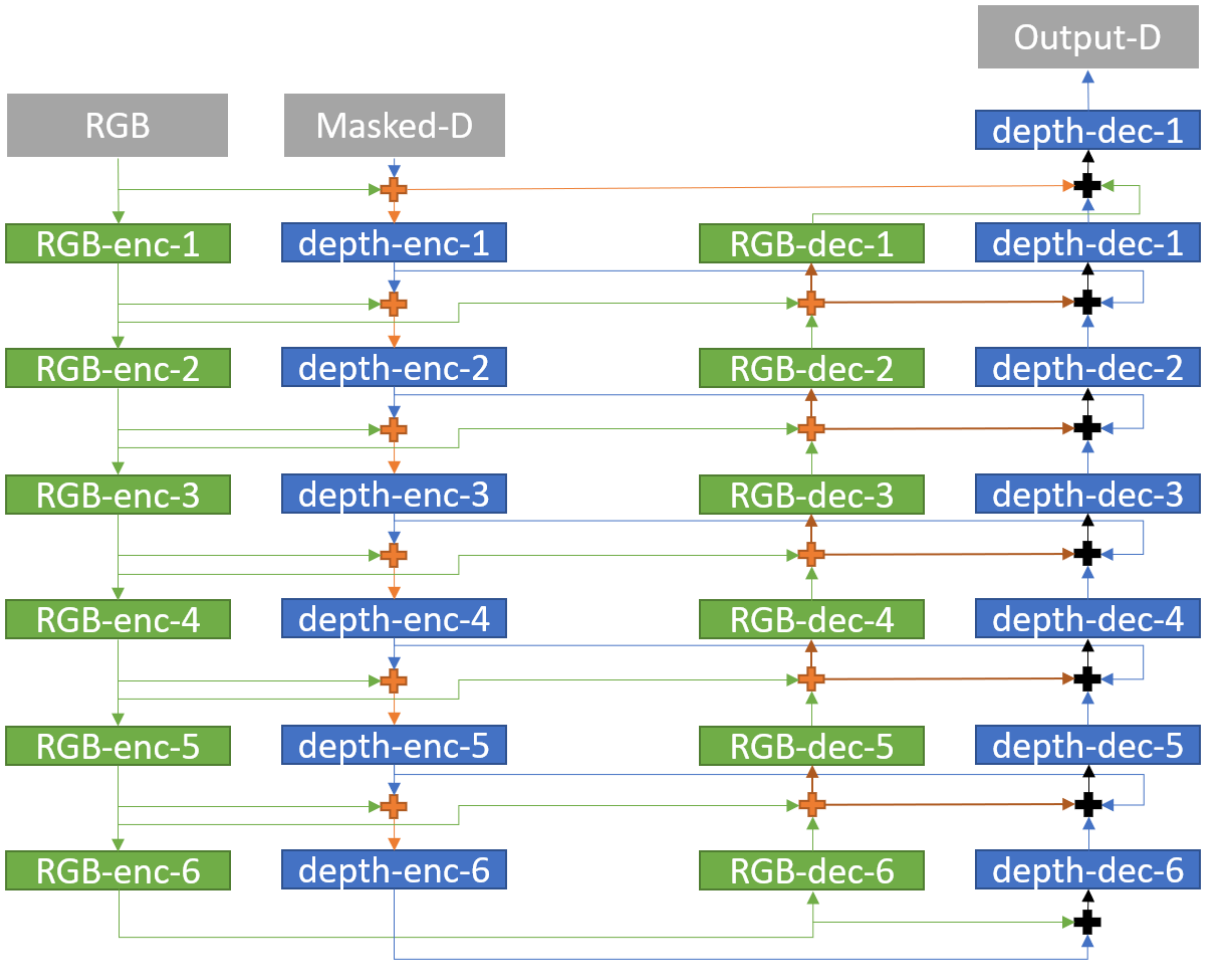

Figure 25: The improved network. The green layers are standard convolutional layers, and the blue are PConv. The + symbol indicates concatenation. The upsampling/downsampling is the same as in figure 24

6 Conclusion

Lightfields are quickly emerging in the consumer market, and as such the demand for editing techniques is growing. Throughout this paper, we have argued that segmentation-based methods are a viable way of editing lightfields. We present one of the first automatic segmentation techniques, intended for 3D large baseline lightfields. Unfortunately, while it produces reasonable results for the dataset that it was designed for, it does not generalize well to small baseline lightfields with fewer views. As such, its practical applications are limited; most plenoptic cameras have very small baselines.

Our segmentation algorithm did show the power of EPI based segmentations, however, and inspired the work of [13], a more general segmentation method that works with small baseline, 4D lightfields. Using their segmentations, we then discussed several occlusion aware edit propagation techniques. While these techniques work well with ground truth depth and accurate segmentations, they are vulnerable to incorrect estimates. Future work should focus on increasing segmentation accuracy and making the edit methods more robust in the face of erroneous results. For our examples, we used only the center and end views for edit propagation, but a more sophisticated method might exploit all views' estimates using a voting system. We assumed that the segmentations were entirely correct, but another method might use color consistency in addition to segmentation labels to ensure that it is projecting the edit to the correct pixel.

We also use our superpixels to propagate the central disparity estimates to every pixel in the lightfield, and we use that depth information to let the user edit any view and to "look around" objects when performing deletions. Once again, the quality of the results of this propagation step are largely dependent on the accuracy of the superpixels. A more robust method might also use the color and depth information of the surrounding pixels when propagating disparity, rather than just using the average depth of the superpixel.

"Looking around" object works well as long as the mask is correct, which it often is not. Incorrect masks (masks which do not quite reach the object boundary) can cause there to be floating pixels from the original object, as in figure 19. A post processing step that searches for floating pixels with vastly different depths from their surroundings might fix this issue. In actuality, however, the practical application of propagating information from other views is likely limited, as most lightfield cameras are extremely small baseline. Therefore, it is easier to just infill the central view, as in [12].

Finally, we present a CNN which can predict the depth of an infilled region. This works remarkably well given the simple loss function that it uses. Future work could try using a GAN to get even better results, or train an autoencoder like ImageNet on depth images for the loss function. We found that PConv layers did not work well with our network, but it would be interesting to see if they performed better with an autoencoder-based loss function, as we theorize in section 5.4.5.

References

[1] The looking glass. `https://lookingglassfactory.com`. Accessed: 2019-04-30.

[2] P. Allain, L. Guillo, and C. Guillemot. Fast light field inpainting propagation using angular warping and color-guided disparity interpolation. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 559–570. Springer, 2018.

[3] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), Aug. 2009.

[4] K. Chen, M. Chang, and Y. Chuang. Light field image editing by 4d patch synthesis. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015.

[5] A. Criminisi, S. B. Kang, R. Swaminathan, R. Szeliski, and P. Anandan. Extracting layers and analyzing their specular properties using epipolar-plane-image analysis. *Computer vision and image understanding*, 97(1):51–85, 2005.

[6] A. Criminisi, P. Pérez, and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on image processing*, 13(9):1200–1212, 2004.

[7] A. CS Kumar, S. M. Bhandarkar, and M. Prasad. Monocular depth prediction using generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 300–308, 2018.

[8] L. A. Gatys, A. S. Ecker, and M. Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

[9] C. Godard, O. Mac Aodha, and G. J. Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 270–279, 2017.

[10] M. Hog, N. Sabater, and C. Guillemot. Light field segmentation using a ray-based graph structure. In *ECCV*, 2016.

[11] S. Iizuka, E. Simo-Serra, and H. Ishikawa. Globally and locally consistent image completion. *ACM Transactions on Graphics (ToG)*, 36(4):107, 2017.

[12] A. Jarabo, B. Masia, and D. Gutierrez. Efficient propagation of light field edits. In *In Proc. of SIACG'11*, pages 75–80, 2011.

[13] N. Khan, M. H. Kim, L. Kasser, Q. Zhang, H. Stone, and J. Tompkin. View consistent 4d light field superpixel segmentation. *ICCV*, 2019 - In submission.

[14] Y. Kim, H. Jung, D. Min, and K. Sohn. Deep monocular depth estimation via integration of global and local predictions. *IEEE Transactions on Image Processing*, 27(8):4131–4144, Aug 2018.

[15] M. Le Pendu, X. Jiang, and C. Guillemot. Light field inpainting propagation via low rank matrix completion. *IEEE Transactions on Image Processing*, 27(4):1981–1993, April 2018.

[16] G. Liu, F. A. Reda, K. J. Shih, T. Wang, A. Tao, and B. Catanzaro. Image inpainting for irregular holes using partial convolutions. *CoRR*, abs/1804.07723, 2018.

[17] C. T. M. Ruzon. Color edge detection with the compass operator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 160–166, 1999.

[18] H. Mihara, T. Funatomi, K. Tanaka, H. Kubo, Y. Mukaigawa, and H. Nagahara. 4d light field segmentation with spatial and angular consistencies. In *ICCP*, 2016.

[19] K. Shafique and M. Shah. A noniterative greedy algorithm for multiframe point correspondence. *IEEE transactions on pattern analysis and machine intelligence*, 27(1):51–65, 2005.

[20] J. Sun, L. Yuan, J. Jia, and H.-Y. Shum. Image completion with structure propagation. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 861–868, New York, NY, USA, 2005. ACM.

[21] T.-C. Wang, A. A. Efros, and R. Ramamoorthi. Occlusion-aware depth estimation using light-field cameras. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[22] S. Xie, P. Wang, X. Sang, and C. Li. Augmented reality three-dimensional display with light field fusion. *Opt. Express*, 24(11):11483–11494, May 2016.

[23] L. Yatziv, G. Sapiro, and M. Levoy. Lightfield completion. In *2004 International Conference on Image Processing, 2004. ICIP '04.*, volume 3, pages 1787–1790 Vol. 3, Oct 2004.

[24] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. S. Huang. Generative image inpainting with contextual attention. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5505–5514, 2018.

[25] J. Yu, Z. Lin, J. Yang, X. Shen, X. Lu, and T. S. Huang. Generative image inpainting with contextual attention. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[26] F. Zhang, J. Wang, E. Shechtman, Z. Zhou, J. Shi, and S. Hu. Plenopatch: Patch-based plenoptic image manipulation. *IEEE Transactions on Visualization and Computer Graphics*, 23(5):1561–1573, May 2017.

[27] S. Zhang, H. Sheng, C. Li, J. Zhang, and Z. Xiong. Robust depth estimation for light field via spinning parallelogram operator. *Comput. Vis. Image Underst.*, 145(C):148–159, Apr. 2016.

[28] H. Zhu, Q. Zhang, and Q. Wang. 4d light field superpixel and segmentation. In *IEEE CVPR*, 2017.