Technical Report CSE-09-03-02

A Consistent Segmentation Approach to Image-based Rendering

Ke Colin Zheng, University of Washington
Alex Colburn, University of Washington
Aseem Agarwala, Adobe Systems, Inc.
Maneesh Agrawala, University of California, Berkeley
David Salesin, Adobe Systems, Inc.
Brian Curless, University of Washington
Michael F. Cohen, Microsoft Research

Abstract

This paper presents an approach to render novel views from input photographs, a task which is commonly referred to as *image based rendering*. We first compute dense view dependent depthmaps using consistent segmentation. This method jointly computes multi-view stereo and segments input photographs while accounting for mixed pixels (matting). We take the images with depth as our input and then propose two rendering algorithms to render novel views using the segmentation, both realtime and off-line. We demonstrate the results of our approach on a wide variety of scenes.

1 Introduction

The goal of our work is to render high-quality novel viewpoints of a scene from a small number of handheld snapshots. Given our primary application to be novel view synthesis, we compute dense view dependent depthmap for each input photograph. We further propose two rendering algorithms with different speed and quality trade-offs.

Multi-view stereo [Seitz et al. 2006] and image-based rendering [Kang and Shum 2002] have been well-studied research problems. However, given our application, we extend previous approaches in a number of significant ways in order to produce higher quality results.

2 Related work

Image-based rendering (IBR) techniques use multiple captured images to support the rendering of novel viewpoints [Kang and Shum 2002]. Rendering novel viewpoints of a scene by re-sampling a set of captured images is a well-studied problem [Buehler et al. 2001]. IBR techniques vary in how much they rely on constructing a geometric proxy to allow a ray from one image to be projected into the new view. Since we are concerned primarily with a small region of the input viewpoints, we are able to construct a proxy by determining the depths for each of the input images using multi-view stereo, similar to Heigl et al. [1999]. Our technique merges a set of images with depth in a spirit similar to the Layered Depth Image (LDI) [Shade et al. 1998]. However, we compute depths for segments, and also perform the final merge at render time. Zitnick et al. [2004] also use multi-view stereo and real-time rendering in their system for multi-viewpoint video, though their constraint that cameras lie along a line allows some different choices. Most IBR systems are designed to operate across a much wider range of viewpoints than our system and typically use multiple capture devices and a more controlled environment [Taylor 1996; Levoy 2006].

Stereo is an active area of research in computer vision, for both two views [Scharstein and Szeliski 2002] and multiple views [Seitz et al. 2006]. Stereo can be seen as a more constrained version of the general optical flow problem; our stereo implementation is an extension of the optical flow work of Zitnick et al. [2005]. Their approach to motion estimation is particularly well-suited to image-based rendering because it explicitly models pixels that are mixtures of several scene points; that is, an alpha value per pixel is computed. Recent work [Zitnick et al. 2004] has shown that this matting information improves novel view interpolation near depth discontinuities. One difficulty in extending this optical flow algorithm, however, is that while their paper provides a good conceptual framework, it contains very little in the way of algorithmic details. Thus, a further contribution of our paper could be considered to be the first detailed treatment of an algorithm based on this approach, in a way that makes the ideas implementable.

A number of papers have used advanced graphics hardware to accelerate the rendering of imagery captured from a collection of viewpoints. The early work on light fields [Levoy and Hanrahan 1996; Gortler et al. 1996] rendered new images by interpolating the colors seen along rays. The lightfield was first resampled from the input images. The GPU was used to quickly index into a lightfield data structure. In one of the early works leveraging per-pixel depth, Pulli et al. [1997] created a textured triangle mesh from each depth image and rendered and blended these with constant weights. They also introduced the notion of a soft-z buffer to deal with slight inaccuracies in depth estimation. We take a similar approach but are able to deal with much more complex geometries, use a per-pixel weighting, and have encoded the first soft-z into the GPU acceleration. Buehler et al. [2001] also rendered per-pixel weighted textured triangle meshes. We use a similar per-pixel weighting, but are also able to deal with much more complex and accurate geometries. We also use a "reverse soft-z" buffer to fill holes caused by disocclusions during rendering.

3 System overview

Since the collection of input photographs is typically shot with a hand-held camera rather than a calibrated rig, we first recover the viewpoints and orientations of each camera using structure-from-motion (Section 4). This step also returns a sparse cloud of 3D scene points. We compute a triangulation to determine the neighboring viewpoints, which is useful both for determining scene depths and for choosing which nearby input photographs to use when rendering a novel view. Our multi-view stereo algorithm then breaks each photograph into small segments and determines the depth of each segment in each view (Section 5).

We have developed two rendering algorithms to display novel views from the textured segments with depth (Section 6. The first rendering algorithm is implemented as a real-time renderer leveraging the GPU. The second algorithm is implemented as an off-line renderer and produces higher quality results.

4 Structure from motion and triangulation

The first step of our system is to recover the relative location and orientation of each photograph, which we perform using the structure-from-motion system of Snavely et al. [2006]. Their system matches features between each pair of images and iteratively estimates the projection matrix of each photograph. We then project the cameras onto a 2D manifold in 3D space (in our case, a least squares plane fit), and we triangulate the projected points via a Delaunay triangulation [Shewchuk 1996] to form a *view mesh*. We

consider any pair of cameras connected by a triangle edge to be neighbors in the view mesh.

5 Multi-view stereo

Our multi-view stereo algorithm can be viewed as an extension to the consistent-segmentation optical flow approach of Zitnick et al. [2005]. Their intuition is that optical flow is simpler to compute between two images that are consistently segmented (by consistent, we mean that any two pixels in the same segment in image I_i are also in the same segment in image I_j), since the problem reduces to finding a mapping between segments. Conversely, consistently segmenting two images of the same scene is easier if optical flow is known, since neighboring pixels with the same motion are likely to be in the same segment. Their algorithm iterates between refining segmentation, with motion treated as constant, and refining motion, with segmentation treated as constant.

We apply this basic approach to multi-view stereo, and extend it in four significant ways: (1) we compute stereo rather than optical flow by enforcing a soft epipolar constraint; (2) we incorporate prior, sparse knowledge of the 3D scene computed by structure from motion; (3) we consistently segment a view with respect to its neighboring n views rather than a single other view; and (4) we add an extra stage that merges the disparities computed from the n neighboring views into a single set of segments and their associated depths, resulting in a final set of segments for each view.

We should note that the work of Georgiev et al. [2006] also builds on the Zitnick et al. paper, imposing an epipolar constraint and computing disparities with respect to two neighbors. Their work, however, neither reconstructs nor uses any scene points derived through structure from motion, nor reconstructs depth maps — instead they perform view morphing on a specific camera manifold, customized for a special lens design.

Our multi-view stereo algorithm is applied to each reference view by comparing it to its n neighboring views. Let each view I_i contain k_i segments. Each pixel is treated as a mixture of two segments. Therefore, each pixel p of view I_i is assigned a primary and secondary segment, $s_1(p)$ and $s_2(p)$. A *coverage value* $\alpha(p)$ expresses the portion of the pixel that comes from the primary segment; thus, $0.5 < \alpha < 1$. Given the segmentation $(s_1(p), s_2(p))$, and the coverage values of each pixel, a mean color $\mu(k)$ for the k'th segment can be computed. Let C be the observed color of p in I, let $C_1 = \mu(s_1(p))$ be the mean color of the primary segment, and C_2 be the mean color of the secondary segment. Ultimately, we seek to compute a segmentation, such that the convex combination of the segment mean colors,

$$\alpha C_1 + (1-\alpha)C_2, \tag{1}$$

is as close to C as possible. Given a particular pair of mean colors C_1 and C_2, we can project the observed color C onto the line segment that connects them in color space to impute an α for that pixel, which amounts to computing

$$\alpha = \frac{(C-C_2)\cdot(C_1-C_2)}{\|C_1-C_2\|^2}. \tag{2}$$

where the numerator contains a dot product between two color difference vectors, and the result is clamped to α's valid range. In the end, the overlap between segments is usually fairly small; thus many pixels belong to exactly one segment. In such cases, we consider the pixel's primary and secondary segments to be the same, $s_1(p) = s_2(p)$, and set $\alpha = 1$.

To define a mapping between segments in two views, segment k in view I_i maps to a segment $\sigma_{ij}(k)$ in view I_j. Mappings are not required to be bijective or symmetric, which allows for occlusions and disocclusions. A mapping $\sigma_{ij}(k)$ implicitly defines a disparity for each pixel p that considers the k'th segment as primary, i.e., $s_1(p) = k$; the disparity $d_{ij}(k)$ is the displacement of the centroids of the segments. (Note that we use disparity and displacement interchangeably here, and that they correspond to 2D vectors.) In some cases, however, we are able to determine when a segment is partially occluded, making this disparity estimate invalid, and, as discussed in Section 5.3, we compute disparity by other means. For this reason, we separately keep track of a segment's disparity $d_{ij}(k)$, in addition to its mapping $\sigma_{ij}(k)$. Ultimately, we will combine the disparities $d_{ij}(k)$ to determine the depth of segment k.

The algorithm of Zitnick et al. iterates between updating the segmentation and disparities for two views. To handle n neighboring views when computing depths for a reference view I_i, our algorithm iterates between updating the segmentation of I_i and its n neighboring views I_j, and updating the mappings and disparities between I_i and each neighboring view (i.e., σ_{ij} and σ_{ji}). To compute segments and depths for all views, we loop over all the images, sequentially updating each image's segmentation and then disparities. We repeat this process 20 times, after which depths are merged. Note that this entire process is linear in the number of original views.

5.1 Initialization

We initialize the segmentation for each image by subdividing it into a quadtree structure. A quadtree node is subdivided if the standard deviation of the pixel colors within the node is larger than a certain threshold. We set the threshold to 90 (color channel values are in the range of $[0..255]$), and we do not subdivide regions to be smaller than 8×8 pixels, to avoid over-segmentation. We initialize the mapping between segments using the sparse cloud of 3D scene points computed by structure from motion. Each segment is initialized with the median disparity of the scene points that project to it, or is interpolated from several of the nearest projected scene points if no points project within its boundaries. Each segment k in image I_i is then mapped to neighboring image I_j according to its initial disparity, and the mapping $\sigma_{ij}(k)$ is set to the segment in I_j that covers the largest portion of the displaced segment k.

5.2 Segmentation update

We first describe how Zitnick et al. update the segmentation of view I_i with respect to neighboring view I_j, given a current segmentation and mapping. We then describe how we extend this update to handle n views.

For each pixel p in view I_i, we consider the segments that overlap a 5×5 window around p. For each of these segments, treating it as primary, we then pair it with every other segment (including itself), compute α according to equation (2) (or set it to 1, for self-pairings), and compute a score for every pairing for which $\alpha > 0.5$. We then choose the highest scoring pair as the segments for this pixel. We commit this segmentation choice after visiting every image pixel in the same fashion.

The score of a segment pairing at pixel p is computed as follows. Given the primary and secondary candidate segments $s_1(p)$ and $s_2(p)$, as well as α and the observed color C, an *inferred* primary color C_1' can be calculated such that it satisfies the blending equation:

$$C = \alpha C_1' + (1-\alpha)C_2. \tag{3}$$

Given the inferred primary color C_1' from a pair of candidate segments, we compute its score as follows:

$$N[C_1'; C_1, \Sigma(s_1(p))]\; v[p, s_1(p)]\; v[q, \sigma_{ij}(s_1(p))], \tag{4}$$

where $N[x;\mu,\Sigma]$ returns the probability of x given a normal distribution with mean μ and covariance matrix Σ, $v[p,k]$ measures the fraction of the 5×5 window centered at p covered by segment k, and where q is the pixel in I_j corresponding to p, i.e., $q = p + d_{ij}(s_1(p))$.

This scoring function encodes two objectives. The first is that the inferred primary color should have high probability, given the probability distribution of the colors of the primary segment. The second objective is that the primary segment should overlap significantly with a window around pixel p in I_i, and the corresponding segment should also overlap with a window around q in I_j.

Given this pairwise segmentation-update approach, the extension to n neighboring views is quite simple. When updating the segmentation for the reference view I_i, we multiply the product in equation (4) by the term $v[q, \sigma_{ij}(s_1(p))]$ for *each* neighboring view I_j, resulting in a product of $n+1$ overlap terms $v(\cdot)$.

5.3 Segmentation and disparity update

Given a segmentation for each view, in this step we update the mappings and disparities between segments in reference view I_i and each neighboring view I_j. We first describe the algorithm and objective function used by Zitnick et al. to choose this mapping, and then describe our extensions to incorporate epipolar constraints and a depth prior from the cloud of 3D scene points calculated by structure from motion.

For each segment k in I_i, we visit all segments within a large window around the centroid of the initial segment $\sigma_{ij}(k)$ in image I_j. We then score the compatibility of these candidate segments with k and set $\sigma_{ij}(k)$ to the candidate segment that yielded the highest score. (In our implementation, we repeat this process 20 times, starting with a 200×200 search window, steadily narrowing the search window with each iteration, down to 100×100.)

The scoring function for a segment k and a candidate mapping $\sigma_{ij}(k)$ is a product of three terms. The first term,

$$N[\mu(k);\mu(\sigma_{ij}(k)),\Sigma(\sigma_{ij}(k))]\; N[\mu(\sigma_{ij}(k));\mu(k),\Sigma(k)], \qquad (5)$$

measures how similar the colors are in the two corresponding segments, by measuring the probability of the mean color of segment k given the color distribution of segment $\sigma_{ij}(k)$, and vice versa. The second,

$$N[d_{ij}(k);\mu(d),\Sigma(d)], \qquad (6)$$

is a regularization term that measures the probability of the implied disparity $d_{ij}(k)$ given a normal distribution of disparities whose mean and covariance $\mu(d),\Sigma(d)$ are computed using the disparities of each pixel in a 100×100 window centered at the centroid of segment k. The third,

$$s_{ij}[k,\sigma_{ij}(k)] \qquad (7)$$

measures the similarity in shape between the two segments by comparing their sizes. The function s in this term is the ratio between the numbers of pixels in the smaller and larger segments.

For our purposes, we extend the disparity update algorithm in several ways. For one, a candidate mapping segment $\sigma_{ij}(k)$ is only considered only if its centroid falls near the epipolar line l_k in I_j of the centroid of segment k in I_i. We cull from consideration segments whose centroids are more than 25 pixels from the epipolar line. We also contribute two additional terms to the product to be maximized when choosing mappings. The first term penalizes displacements that are not parallel to corresponding epipolar lines:

$$\exp(-\hat{e}(k)\cdot\hat{d}_{ij}(k)) \qquad (8)$$

where $\exp(\cdot)$ is the exponential function, $\hat{e}(k)$ is the normalized direction of the epipolar line in image I_j associated with the centroid of segment k, and $\hat{d}_{ij}(k)$ is the normalized direction of displacement of that segment.

Finally, we (again) take advantage of the 3D scene points reconstructed in Section 4. In this case, if one or more of these points project into a segment k in image I_i, we compute the median displacement $m_{ij}(k)$ of their re-projections into image I_j, and multiply one more term into the scoring function:

$$\exp(-||m_{ij}(k)-d_{ij}(k)||) \qquad (9)$$

to encourage similarity in these displacements.

After iteratively optimizing all the disparities in the image according to the scoring function, a final pass is performed to account for segments that may have become partially occluded in moving from image I_i to I_j. (Here, we return to the original algorithm of Zitnick et al.) In this pass, we visit each segment k in I_i and determine if the size of its corresponding segment is substantially different from the size of segment $\sigma_{ij}(k)$ in I_j. We also determine if the mapping for a segment k is not symmetric, i.e., if $k \neq \sigma_{ji}(\sigma_{ij}(k))$. If neither of these conditions is true, then we simply set the disparity $d_{ij}(k)$ to the difference of the centroids of k and $\sigma_{ij}(k)$. However, if either of these conditions is true, then we suspect a disocclusion. In that case, we attempt to "borrow" the disparity of k's neighbors. In particular, for each segment that overlaps k in image I_i (i.e., each distinct segment that is either primary or secondary to k at one or more pixels), we apply its disparity to k and compare the mapped segment against each segment it overlaps in I_j by computing the average square color difference within their overlap. After considering all possibilities, the disparity and segment mapping with minimum color difference are stored with segment k.

5.4 Depth merging

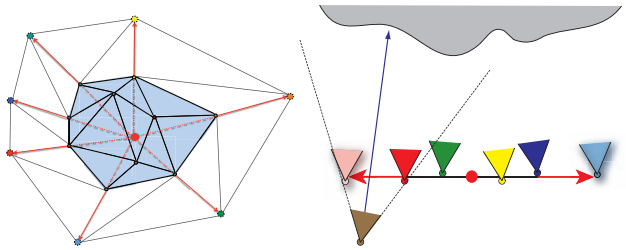
After the above two update steps are iterated to completion (20 iterations over all the images), the result is a segmentation and a set of n disparities for each segment, one disparity for each neighboring view. Since we need only one depth per segment, these disparity estimates must be combined. We do so in a weighted least squares fashion. To compute the depth of a segment k in image I_i, we consider the corresponding segment $\sigma_{ij}(k)$ in each neighboring view I_j. Each corresponding segment defines a 3D ray from the optical center of view I_j through the centroid of that segment on the view's image plane. Such a ray also exists for segment k in reference view I_i. We thus compute a 3D point that minimizes the Euclidean distance to these $n+1$ rays in a weighted least squares sense. Corresponding segments that we suspect are occluded in view I_i are given less weight. A mapping segment $\sigma_{ij}(k)$ is considered occluded if the mapping is not symmetric, i.e., $\sigma_{ji}(\sigma_{ij}(k)) \neq k$. We set the weights of these rays to 0.25, and the rest to 1.0.

5.5 Interactive correction

Unfortunately, multi-view stereo algorithms are not perfect, and they will sometimes assign incorrect depths; these errors can sometimes be seen as errant segments that coast across the field of view (often near depth boundaries). To handle them, we allow users to click on these errant segments in our interactive viewer; this input is used to improve the depth estimates. The extent of this interaction is quantified in Section 7.

6 Rendering algorithms

We have developed two rendering algorithms to display novel views from the textured segments with depth. The first rendering algorithm is implemented as a real-time renderer leveraging the GPU. This renderer allows a user to explore the scene and design camera



(a) (b)

Figure 1 **(a)** The view mesh is extended by creating copies of boundary vertices and placing them radially outward (red arrow) from the centroid (red dot) of all of the mesh vertices. These new vertices are then re-triangulated and added to the view mesh to form an extended view mesh. **(b)** The dotted blue arrow shows a ray projected through the image plane. The *blending weights* at this pixel correspond to the barycentric coordinates of the intersected triangle of viewpoints in the view mesh.

paths that best depict the scene. It is also used by the automatic camera path planner to evaluate whether novel viewpoints are ”valid” (i.e., wheter they can be rendered with minimal holes), and to estimate the amount of parallax to select paths that provide a strong 3D effect. The second algorithm is implemented as an off-line renderer and produces higher quality results; it is used to render the final result animation. Both renderers take the same basic approach; we therefore first describe the general rendering algorithm, and then we describe the specifics of the interactive renderer implementation including GPU acceleration, and finally we describe the differences in the off-line rendering algorithm.

To render the scene from a novel view, we project a ray from the origin of the novel view through each pixel in the image plane Figure 1b). If the ray intersects any segments, we combine their color and depth values and assign the combined color to the pixel. We calculate each segment’s contribution in three steps. First we choose which segments should contribute to the the pixel value. Second, we compute a *blending weight* for each contributing segment color value. Finally we employ a *soft z-buffer* to resolve depth inconsistencies and normalize the weights to combine the remaining color values.

When choosing which segments should contribute to a pixel value, we only consider those segments belonging to three original viewpoints with rays most closely aligned with that corresponding to the pixel to be rendered. To select the viewpoints for each pixel of the novel view, we first construct a *view mesh* by projecting the viewpoint camera positions onto a 2D manifold in 3D space, in this case, a plane fit to the camera locations using least squares. The original vertices are then triangulated via a Delaunay triangulation [Shewchuk 1996]. We also extend the view mesh by duplicating the vertices on the mesh boundary (Figure 1a). These duplicate vertices are positioned radially outward from the mesh center at a distance four times the distance from the center to the vertices on the boundary. The original and duplicate vertices are then re-triangulated to form the final view mesh. The triangles on the view mesh boundary will contain two vertices with the same camera ID, while interior triangles will have three distinct camera IDs.

Given the novel view and the view mesh, we are now ready to determine which viewpoints will contribute to each pixel in the novel view and with what weights. Each pixel in the novel view defines a ray from the novel viewpoint through the pixel on the image plane. This ray is intersected with the view mesh (looking backwards if necessary). The viewpoints corresponding to vertices of the intersected triangle on the view mesh are *closest* for that pixel in the novel view, and thus the ones whose segments will contribute to that novel view pixel. We assign a *blending weight* to each contributing segment equal to the barycentric coordinates for the corresponding viewpoint in the intersected triangle. This is similar to the weights given in [Buehler et al. 2001].

The contributing segments also lie in some depth order from the novel view. Often, segments from different viewpoints represent the same piece of geometry in the real world and thus will lie at approximately the same depth. Slight differences in depth are due to noise and errors in the capture, viewpoint positioning and depth estimation. As the novel viewpoint changes, the exact ordering of these segments may change. Rendering only the closest segment may thus lead to popping artifacts as the z ordering flips. To avoid these temporal incoherencies, we implement a *soft z-buffer* [Pulli et al. 1997]. A soft z-buffer allows us to consistently resolve conflicting depth information by combining all of the segments that may contribute to a pixel, and estimating the most likely RGBA and z values for the pixel. The soft z-buffer assigns a z-weight for each contributing segment beginning with a weight of 1.0 for the closest segment (at a distance z_0) dropping off smoothly to 0.0 as the distance increases beyond z_0. The z-weights are multiplied by the *blending weight*, and the results are normalized. The final pixel value is the normalized weighted sum of the textures from the contributing segments.

When the novel view diverges from the original viewpoints, the parallax at depth discontinuities may cause segments to separate enough so that a ray hits no segments. We are then left with a hole-filling problem. We address this later in the context of the interactive and offline renderers.

6.1 Interactive renderer

To render the scene from a novel view at interactive frame rates (at least 30 fps), we need to pose the rendering problem in terms of GPU operations. We now describe the rendering steps in terms of polygons, texture maps and GPU pixel shaders. We render the scene in four steps. First, we choose which segments should contribute to the pixel value and calculate the *blending weight* for each contributing segment color value. Second, we render all of the segments to three offscreen buffers. Third, we employ a *soft z-buffer* to resolve depth inconsistencies between the three offscreen buffers and combine their color values. Finally, we fill holes using a *reverse soft z-buffer* and local area sampling.

Rendering the extended view mesh To choose which segments should contribute to the pixel value and to calculate the *blending weights* we render the extended view mesh from the novel view to an offscreen buffer. Setting the three triangle vertex colors to red, green, and black encodes two of the barycentric coordinates in the Red and Green channels; the third coordinate is implicit. The Blue and Alpha channels are used to store an ordered 3-element list storing the ID’s of the three viewpoints (we use 5 bits to encode a viewpoint ID, so 3 IDs can be stored in 16 bits, allowing for a total of 32 input viewpoints)[1].

When rendering the extended view mesh, there are two special cases that should be highlighted. First, if the novel view lies in front of the view mesh, the projection step requires a backwards projection (i.e., projecting geometry that is behind the viewer through the center of projection). Second, the projection of the view mesh nears a singularity as the novel view moves close to the view mesh itself. Therefore, if the novel view is determined to lie within some

[1] It is not strictly necessary to encode both viewpoint IDs and the barycentric weights into one off-screen buffer, but doing so saves a rendering pass and reduces texture memory usage.

small distance from the view mesh, the view mesh is not rendered at all. Rather, the nearest point on the mesh is found. The blending weights and viewpoint IDs are then constant across all pixels in the novel view, set to the barycentric coordinates of the point on the view mesh and the vertex IDs of the triangle the point lies in.

Rendering segments Each segment is rendered as a texture mapped rectangle. The rectangle's vertex locations and texture coordinates are defined by the bounding box of the segment. A segment ID and associated viewpoint ID is recorded with each rectangle. Rather than create a separate texture map for each segment rectangle, we create two RGBA textures for each viewpoint, plus one *Segment ID* texture. As described in Section **??** pixels near the boundaries of segments are split into two overlapping layers to account for mixed pixels along the boundaries. Thus, the first RGBA texture contains the segment interiors and one of the two layers. The other RGBA texture contains the second layer along segment boundaries. Finally we create a third 2-channel 16-bit integer *Segment ID* texture map containing segment ID values indicating to which segment a pixel belongs.

For each segment, a pixel shader combines the texture maps and discards fragments within the segment bounding rectangles but laying outside the segment itself. To render a rectangle, we encode the segment ID as a vertex attribute (e.g. color or normal data). The shader uses this value in conjunction with the *Segment ID* texture and the two RGBA textures to compose the segment color values on the fly.

```
void main(){
    if (SegmentID == SegIDMap[0] or
        SegmentID == SegIDMap[1])
        ExtractAndDraw();
    else
        discard;
}
```

Using this GPU based texture representation has two benefits, both of which increase rendering speed. First, it reduces the number of texture swaps from thousands per viewpoint to only three. Second, by removing the texture swap from the inner loop we are able to take advantage of vertex arrays or vertex buffer objects and utilize the GPU more efficiently.

We still need to choose which segments contribute to each pixel and by how much. A pixel is ultimately a sum of segments originating from at most three different viewpoints as encoded in the viewpointIDs in the offscreen rendered view mesh. We create three buffers to accumulate RGBA values, corresponding to the three viewpointIDs stored at each pixel in the rendered view mesh. When rendering a segment, we encode the segment's viewpoint ID as a vertex attribute. The pixel shader chooses to which of the three buffers a segment should contribute, if any, by matching the the segment's viewpoint ID with the ones encoded in the offscreen rendered view mesh at that pixel location. For example, if the segment's viewpoint ID matches the first of the view mesh's encoded viewpoint IDs (i.e., the one corresponding to the "red" barycentric coordinate), the pixel is accumulated in the first buffer using the first (red) barycentric coordinate as a weight. The same is done if there is a match with the second (green) or third of the view mesh's encoded viewpoint IDs, except the third barycentric weight is inferred from the other two (1 - red - green).

```
void main(){
  if (ViewID == ViewIDMap[0])
    Target = 0;
  else if (ViewID == ViewIDMap[1])
    Target = 1;
  else if (ViewID == ViewIDMap[2])
    Target = 2;
  else discard;

  if (SegmentID == SegIDMap[0] or
      SegmentID == SegIDMap[1])
        ExtractAndDraw(Target);
  else
     discard;
}
```

Before rendering any segments, the segments for each viewpoint are sorted to be processed in front-to-back order. The three rendering buffers are initialized to black background and zero alpha. To maintain proper color blending and z-buffering, the blending mode is set to GL_SRC_ALPHA_SATURATE, depth testing is enabled, and set to GL_ALWAYS. The pixel shader calculates the premultiplied pixel values and alphas to render.

Soft z-buffer The z-buffering is performed traditionally within a single viewpoint for each of the three buffers; however, we employ a *soft z-buffer* across the viewpoints to blend the three results. For each corresponding pixel in the three buffers, we compute a soft weight w_z by comparing each pixel's z-value with the z-value of pixel closest to the origin of the novel view. This distance Δz, where $\Delta z = z - z_{\text{closest}}$, is used to compute w_z in the following equation:

$$w_z(\Delta z) = \begin{cases} 1 & \text{if } \Delta z \leq \gamma \\ \frac{1}{2}(1+\cos\left(\frac{\pi(\Delta z-\gamma)}{\rho-2\gamma}\right)) & \text{if } \gamma < \Delta z \leq \rho-\gamma \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

where ρ is the depth range (max$-$min) of the entire scene, and γ is set to $\rho/10$.

The set of w_z's are normalized to sum to one. The depth, z for that pixel is then given the sum of the z-values weighted by the w_zs. Each *blending weight* stored in the view mesh texture is multiplied by its corresponding w_z. These new blending weights are normalized. The final pixel value is computed by scaling each pixel by the normalized *blending weight*, and composited based on their alpha values.

Hole filling Holes occur when, due to parallax, a nearby segment separates from a more distant segment. A pixel with a z-value of 1 indicates a hole. we fill small holes of less than 6 pixels in diameter during the final *soft z-buffer* pass. We assume that any hole should be filled with information from neighboring pixels. Since holes occur due to disocclusion, given two neighbors, we prefer to use the more distant one to fill the gap. To do so, we combine the pixel colors and z-values of the pixels in a 7×7 neighborhood. They are combined using the *soft z-buffer* calculation described above except in reverse. In other words, more distant z-values are given higher weights by inverting the ordering, by setting the z-values to $1-z$.

In summary When iterating over the segments to be rendered, only three textures are (re-)loaded per viewpoint: the two RGBA texture maps; and a texture for an ID image to map pixels to segments. A fourth texture, the barycentric weight map from the view mesh, is computed once and used throughout.

As a result of this GPU approach, we can render scenes at 30-45 frames-per-second on an NVIDIA 8800 series graphics card, whereas an implementation using one texture per segment achieved only 7.5 frames-per-second and did not calculate the blending weights on a per pixel basis, use a *soft z-buffer*, or fill holes.

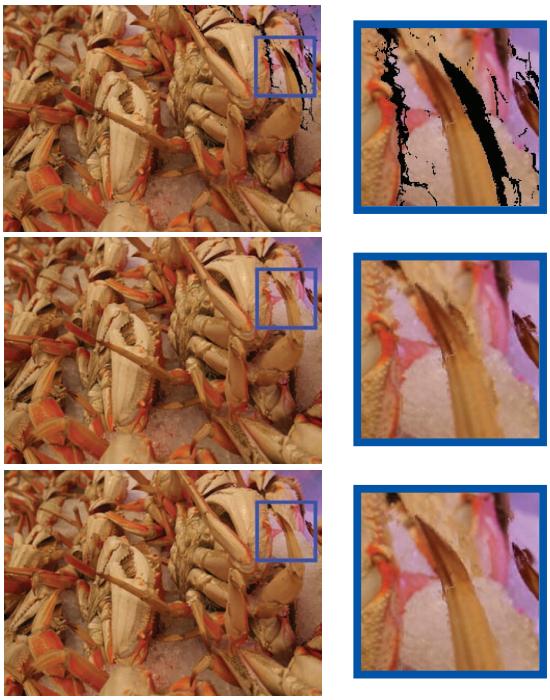


Figure 2 Three renderings of crabs at the market. The first row shows a novel viewpoint rendered from the segments of all the input photographs by the interactive renderer; many holes are visible. An inset, highlighted in blue, is shown on the right. The second row shows the result after inpainting without depth guidance; no holes remain but the result is incorrect. The final row shows the result after depth-guided inpainting in offline-rendering; no holes remain and the inferred background is correct.

Depth of field and color effects Efficient, approximate depth-of-field rendering is accomplished using a variation on existing methods [Demers 2004; Kass et al. 2006; **?**]. For each pixel, we calculate a circle of confusion based on a user defined aperture size, and blur the result of our rendered scene accordingly. The blurring operation is performed efficiently by loading the scene into a MIPMAP and indexing into it based on the blur kernel radius. To improve visual quality, we index into a higher resolution level of the MIPMAP than strictly needed, and then filter with a Gaussian filter of suitable size to achieve the correct amount of blur. Note that when focusing on the background in a scene, this approach will not result in blurred foreground pixels that partially cover background pixels as they should, i.e., the blurry foreground will have a sharp silhouette.

To avoid such sharp silhouettes, when processing a pixel at or behind the focus plane, the pixel shader blends the pixel with a blurred version of the image at that pixel. The blur kernel size is based on the average z-value of nearby foreground pixels. The blending weight given to this blurred version of the image is given by the fraction of the neighboring pixels determined to be foreground. The size of the neighborhood is determined by the circle of confusion computed from the user specified aperture and focal depth.

6.2 Off-line rendering

The higher quality off-line rendering algorithm differs from the interactive renderer in three main ways. First, we extend the *soft z-buffer* described above to increase the accuracy of our pixel value estimate. Second, the renderer uses a texture synthesis approach to fill any holes and cracks that might appear in a novel view due to sparse data generated from the input photographs. Finally, depth of field effects are rendered with increased quality by simulating a camera aperture.

Soft z-buffer The soft z-buffer calculation is very similar to the process described in the real-time renderer. However, rather than using a traditional hard z-buffering within each viewpoint followed by a soft z-buffer across viewpoints, all segments from all contributing viewpoints are combined in a uniform manner. We assemble a depth ordered list of elements at each pixel location as the segments are projected onto the scene. Each element contains the sampling viewpoint ID, the RGBA color value, z-value, the *blending weight*, and the soft weight w_z as computed above. The soft z-buffer weights, w_z are computed when the list is complete.

Hole filling To fill holes the offline renderer uses a more principled approach, in particular the in-painting algorithm of Criminisi et al. [2003] — based on example-based texture synthesis — with two modifications. First, to accelerate computation, we search for matching (5×5) neighborhoods within a restricted window (100×100) around the target pixel, rather than over the entire image. The second, more significant, modification is based on the observation that nearly all large holes occur along depth discontinuities, because some region of background geometry was always occluded in the input photographs. In this case, the hole should be filled from background (far) regions rather than foreground (near) regions. We thus separate the depths of the pixels along the boundary into two clusters, and use these two clusters to classify pixels, as needed, as foreground or background. We then fill the hole with Criminisi's propagation order, using modified neighborhoods and neighborhood distance metrics. In particular, for a given target pixel to fill in, its neighborhood is formed only from pixels labeled background. If no such pixels exist in this neighborhood yet, then this pixel is placed at the bottom of the processing queue. Otherwise, the neighborhood is compared against other candidate source neighborhoods, measuring the L_2 differences between valid target pixels and all corresponding source pixels from a candidate neighborhood. For source pixels that are invalid (foreground or unknown), we set their colors to 0, which penalizes their matching to generally non-zero, valid target neighborhood pixels. Whenever a pixel is filled in, it is automatically classified as background. Thus pixels with invalid neighborhoods (e.g., those centered on the foreground occluder) will eventually be processed as the hole is filled from the background side. When copying in pixel color, we also inpaint its z by weighted blending from known neighbouring pixels, again favoring the back layer. The inpainted z assists in region selection for color manipulation effects. The third row of Figure 2 shows the results of our inpainting algorithm for a novel viewpoint rendering of one of our datasets. Note that Moreno-Noguer et al. [2007] also explored depth-sensitive inpainting, though their application has lower quality requirements since they use the inpainted regions for rendering defocused regions rather than novel viewpoints.

Depth of field Our rendering algorithm now provides a way to reconstruct any view within the viewing volume. In addition to changing viewpoint, we can synthetically focus the image to simulate camera depth of field. To do so, we apply an approach similar to what has been done in *synthetic-aperture photography* [Levoy and Hanrahan 1996; Isaksen et al. 2000]. We jitter viewpoints around the center of a synthetic aperture and reconstruct an image for each viewpoint. We then project all the images onto a given in-focus plane and average the result.

Figure 3 Crab example: the top row shows one of the input images, the middel row is the visulization of its segmentation, and the bottom row is the corresponding depth map.

7 Results

We demonstrate our results on a variety of scenes. The depth estimates are not as geometrically accurate as the top performers in this category, but are sufficient to synthesize novel views that are visualy appealing. The number of input images ranges from 8 to 15. As shown in Figure 4, only four of the results out of over 100 datasets required the user to click segments with errant depth, which typically took 3-5 minutes of user time.

We have tested our overall approach on 208 datasets capturing a variety of scenes. About half of the datasets (108 out of 208) produced visually appealing results. The other half were less successful. Here is a breakdown of what failed:

- Error due to data (35/208 or 17%): The data contained too much motion, resulting either in motion blur or poor correspondence.
- Error caused by structure-from-motion (SfM) (9/208 or 4%): too little parallax for SfM to recover the camera parameters.
- Error in stereo matching (56/208 or 27%): color shifts, too large a baseline, textureless regions, etc.

The majority of failure is due to error in stereo matching. For one, our stereo algorithm is tuned for small baselines (typically about four inches apart). Larger baselines will produce poor depth reconstruction. Also, our approach is subject to the same limitations as most multi-view stereo algorithms. For example, large, fairly untextured regions, such as a gray sky, may not work well. Our results certainly contain examples of non-Lambertian surfaces; however, widespread violations of the Lambertian assumption, such as large areas of reflective or translucent surfaces, will also cause problems. Finally, very thin structures, such as wires or trees in winter, tend to cause poor depth reconstruction, unless they are far enough away that little parallax exists with their background.

8 Conclusion

We described an approach to render novel views based on a dense geometry representation from a few snapshots of a scene, which are easy and cheap to capture with a digital camera.

References

BUEHLER, C., BOSSE, M., MCMILLAN, L., GORTLER, S. J., AND COHEN, M. F. 2001. Unstructured lumigraph rendering. In *Proceedings of ACM SIGGRAPH 2001*, 425–432.

CRIMINISI, A., PEREZ, P., AND TOYAMA, K. 2003. Object removal by exemplar-based inpainting. In *Proc. IEEE Computer Vision and Pattern Recognition (CVPR)*.

DEMERS, J. 2004. Depth of field a survey of techniques. In *GPU Gems 1*, R. Fernando, Ed. Addison Wesley, Mar., ch. 23, 375–390.

GEORGIEV, T., ZHENG, C., NAYAR, S., SALESIN, D., CURLESS, B., AND INTWALA, C. 2006. Spatio-angular resolution tradeoffs in integral photography. *Proceedings of Eurographics Symposium on Rendering*, 263–272.

GORTLER, S. J., GRZESZCZUK, R., SZELISKI, R., AND COHEN, M. 1996. The lumigraph. *ACM Trans. Graph.*, 43–54.

HEIGL, B., KOCH, R., POLLEFEYS, M., DENZLER, J., AND GOOL, L. J. V. 1999. Plenoptic modeling and rendering from image sequences taken by hand-held camera. In *DAGM-Symposium*, 94–101.

ISAKSEN, A., MCMILLAN, L., AND GORTLER, S. J. 2000. Dynamically reparameterized light fields. *ACM Trans. Graph.*, 297–306.

KANG, S. B., AND SHUM, H.-Y. 2002. A review of image-based rendering techniques. In *IEEE/SPIE Visual Communications and Image Processing 2000*, 2–13.

KASS, M., LEFOHN, A., AND OWENS, J. D. 2006. Interactive depth of field using simulated diffusion. Tech. Rep. 06-01, Pixar Animation Studios, Jan.

LEVOY, M., AND HANRAHAN, P. 1996. Light field rendering. *ACM Trans. Graph.*, 31–42.

LEVOY, M. 2006. Light fields and computational imaging. *IEEE Computer 39*, 8, 46–55.

MORENO-NOGUER, F., BELHUMEUR, P. N., AND NAYAR, S. K. 2007. Active refocusing of images and videos. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, ACM, New York, NY, USA, 67.

PULLI, K., COHEN, M. F., DUCHAMP, T., HOPPE, H., SHAPIRO, L., AND STUETZLE, W. 1997. View-based rendering: Visualizing real objects from scanned range and color data. In *Eurographics Rendering Workshop 1997*, 23–34.

SCHARSTEIN, D., AND SZELISKI, R. 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision 47*, 1-3 (apr/jun), 7–42.

SEITZ, S. M., CURLESS, B., DIEBEL, J., SCHARSTEIN, D., AND SZELISKI, R. 2006. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *2006 Conference on Computer Vision and Pattern Recognition (CVPR 2006)*, 519–528.

Figure 4 A variety of results produced by our solution. (First row) One rendered view of each scene. (Second row) The number of input photographs followed by the number of clicks required to fix errant segments (in red). Note that only 4 datasets required any user intervention of this kind.

SHADE, J., GORTLER, S. J., WEI HE, L., AND SZELISKI, R. 1998. Layered depth images. In *Proceedings of SIGGRAPH 98*, 231–242.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May, 203–222.

SNAVELY, N., SEITZ, S. M., AND SZELISKI, R. 2006. Photo tourism: exploring photo collections in 3d. *ACM Transactions on Graphics 25*, 3 (July), 835–846.

TAYLOR, D. 1996. Virtual camera movement: The way of the future? *American Cinematographer 77*, 9 (Sept.), 93–100.

ZITNICK, C. L., KANG, S. B., UYTTENDAELE, M., WINDER, S., AND SZELISKI, R. 2004. High-quality video view interpolation using a layered representation. *ACM Transactions on Graphics 23*, 3 (Aug.), 600–608.

ZITNICK, C. L., JOJIC, N., AND KANG, S. 2005. Consistent segmentation for optical flow estimation. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*.