Discontinuity Preserving Multiview Synthesis

Blake Foster & Rui Wang

Univ. of Massachusetts Amhest

Abstract

In this paper, we present a geometry-based method for multiview synthesis that is robust around depth discontinuities. Our goal is to synthesis new views from a sparse set of images taken with a handheld camera. Our method is based on explicit, view-dependent geometry reconstruction. We first apply structure from motion to recover camera parameters and a 3D point cloud representing the scene geometry. For each view, we then estimate a set of visible 3D points, and compute a visible surface using 2D Delaunay triangulation. The initial surface does not preserve depth discontinuities, leading to objectionable artifacts in rendering. To solve this problem, we ask the user to mark occlusion boundaries for a small subset of the input images. These marked boundaries are then propagated to the remaining images, and used to refine the visible surfaces by splitting triangles that cross occlusion boundaries. Using the results, we demonstrate realistic view synthesis running in real-time on modern GPUs.

Keywords: Image-based modeling and rendering, 3D scene analysis, occlusion boundaries, view synthesis, GPU

1. Introduction

In computer graphics, a fundamental challenge is to create images that look as close as possible to real photographs. Traditional approaches start from well-defined digital scenes and synthesize realistic images by simulating the physical process of illumination in the natural world. While such approaches have enjoyed great success due to tremendous progress in hardware and software developments, the complexity of the real world continues to overwhelm our abilities to model it successfully and render it interactively. For example, a synthesized image of a moderately complex kitchen scene often takes hours to compute, and even longer to model with fine details and physically accurate materials.

This challenge has motivated research in image-based modeling and rendering (IBR), the goal of which is to use a set of captured photographs to generate approximate 3D models, and then synthesize new views from the input data. Such approaches release the users from having to create full models with detailed geometry and materials. Moreover, as pixels in the final image come from the input photographs, the results often look more convincing than images synthesized with traditional rendering algorithms.

With the availability of large image data sets today, image-based modeling and rendering has gained significant popularity, and has started to emerge in 3D games and technology products such as Google's Street View and Microsoft's PhotoSynth. However, several key challenges remain in existing methods, limiting their practical applications. For example, light field rendering methods [LH96] require a great many input images to overcome rendering artifacts due to lack of explicit geometry. This incurs high costs in sampling and compression. Surface light fields [WAA*00] require a small number of input images but rely on the availability of an accurate 3D model. This incurs high costs in geometry acquisition. Layered-depth images [SGHS98] and 3D warping [MMB97] explore depth information associated with each input image to re-render the scene from nearby viewpoints. These methods focus on synthetic scenes where depth is readily available; for captured images, however, obtaining depth is significantly more challenging. A number of existing solutions recover scene depth using stereo rigs, laser range finders, or camera arrays. These devices are expensive and inconvenient to use to a standard user. There are also advanced computer vision algorithms that recover scene depth from a single image [HEH05], but these methods rely heavily on the training data and hence are not very robust for arbitrary scenes.

In this work, we are interested in designing an efficient image-based rendering method for smooth multiview synthesis, and the method should be easy to use for a standard user. Our method is based on explicit, view-dependent geometry reconstruction, for which we build upon previous work in scene analysis from uncalibrated images [PVGV*04]. The input to our algorithm is a set of photographs captured with a handheld camera. We start by applying structure from motion (SfM) [SSS06] to automatically recover camera parameters and a 3D point cloud representing the scene geometry. We then parameterize the locations of cameras onto 2D topology, and compute a camera graph, with which we can easily identify the neighboring views of each image.

Next, for each view, we estimate a set of visible 3D points, and compute a visible surface using 2D Delaunay triangulation. The surface computed in this way does not preserve depth discontinuities, leading to objectional artifacts in rendering. Similar problems also exist with standard stereo matching algorithms. Therefore, our main challenge is to develop a new method that can robustly preserve depth discontinuities. To solve this problem, we resort to user interactions. Specifically, we compute a small set of representative views ($6 \sim 12$) from the input using k-means clustering; we then ask the user to mark occlusion boundaries for each selected image. The marked boundaries are then propagated to the remaining images, and are used to refine the initially estimated visible surfaces by splitting triangles that cross occlusion boundaries. We show that this method can successfully preserve depth discontinuities while requiring only a small amount of user interactions.

To render a novel view, we first identify its closest three input views using the camera graph, then combine the depth information available at each input view to synthesize a new image. The output image may contain holes due to the lack of geometry in some areas. Therefore we also propose a hole-filling algorithm based on real-time stereo matching [YWB02] to cover the holes. All these steps are implemented on the GPU to achieve real-time rendering speed.

As we recover view-dependent geometry for each image, we avoid the need for any global geometry, which is expensive to recover and prone to multiview alignment errors. Using view-dependent geometry also allows the view synthesis error to be confined locally along the line of sight, causing fewer rendering artifacts. In addition, our discontinuity-preserving method robustly keeps sharp edges around object boundaries, leading to more convincing synthesis results.

2. Related Work

Structure from Motion The last two decades have seen a remarkable progress in 3D computer vision algorithms, especially in analysis of unstructured images. Structure from motion (SfM) [HZ04] is a set of techniques that aim to recover camera parameters and sparse 3D geometry from uncalibrated images. [TK92] first introduced factorization methods to compute multi-frame structure from motion; [SK94] proposed global optimization techniques and bundle adjustment for robust 3D reconstruction from correspondences; self-calibration techniques have also been studied extensively [PKG99], which can upgrade a projective reconstruction to a metric reconstruction.

Our method directly uses *Bundler* – an open-source SfM implementation introduced in [SSS06]. Bundler uses SIFT [Low99] features and image EXIF tags to reliably recover camera parameters and 3D points for a variety of data sets. The images may be taken with different cameras at varying focal lengths and resolutions, and even under different illumination.

Image-based Modeling Image-based modeling techniques such as [DTM96,PVGV*04] attempt to create a 3D model of a scene of interest given a set of images. Compared to laser range scanning, image-based techniques are usually more flexible to work with and can easily produce high-resolution results. Given the camera parameters for each image in a multiview data set, we can apply stereo reconstruction to recover depth. This typically involves finding feature correspondences between image pairs, computing disparity, and then reconstructing a consensus polygonal mesh or depth map to represent the underlying geometry. [SCD*06] provides a detailed overview and evaluation of the state-of-the-art in multiview stereo reconstruction algorithms.

Recent multiview stereo algorithms focus on recovering a global consensus geometry, by using volumetric based methods [SD97], cost function minimization [KS00], or merging depth maps [GS05, NRK98]. Our method differs in that we do not require global geometry; instead, we rely on view-dependent geometry, in which we recover a different visible surface for each input image. This eliminates the need for aligning the multiview geometry, which is both time consuming and error-prone. Our method is closely related to the view-dependent geometry approach by [KS04], where each input image is associated with a separate depth map. The main difference in our work is that we exploit occlusion boundaries selected in image-space to help preserve depth-discontinuities in geometry-space. Therefore our method renders sharp edges between foreground and background depth layers during view synthesis.

For hole-filling at run-time, we employ GPU-based multiview stereo matching [YWB02]. For a given novel view, we are typically required to find the k nearest input views [ES04, GCS06]. Our method uses a 2D camera graph, computed by triangulating camera locations. This simplifies the selection and interpolation of nearest neighbors at run-time.

Image-based Rendering As described in [SCK07], image-based rendering techniques can be classified onto an image-geometry continuum. At one end, techniques such as plenoptic modeling [MB95], light fields [LH96], and lumigraph [GGSC96] require on little or no geometry but rely on a large number of input views to overcome aliasing artifacts in rendering. On the other end, techniques such as

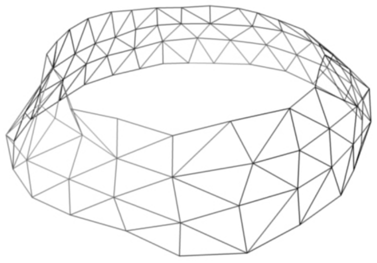

Figure 1: *An example of camera graph.*

surface light fields [WAA*00] and textured 3D rendering require only a small set of images, but rely on the availability of accurate, global 3D models for rendering. [CTCS00] provides mathematical analysis of the plenoptic sampling rate with respect to scene complexity, the number of image samples, and the output resolution. Our goal is to create a method that is convenient and easy to use for standard users. Therefore we would like to eliminate the need for either dense input images or accurate 3D models.

View-interpolation or morphing [CW93,SD96,MHM*09] attempts to synthesize new views by directly transforming pixels in image space using optical flow or feature correspondence. These methods do not recover explicit 3D geometry, but they require accurate image registration and work well only for views with small baselines.

Layered-depth images [SGHS98] and 3D warping [MMB97] explore depth information associated with each input image to re-render the scene from nearby viewpoints; these methods focus on synthetic scenes where depth is readily available. View-dependent texture mapping [DTM96] renders new views by warping and compositing input textures view-dependently. The 3D model is recovered using a photogrammetric modeling method.

[YPYW04] presented a unified approach for real-time depth estimation and view synthesis using modern graphics hardware. While achieving impressive performance, their method does not robustly preserve depth-discontinuities, which often causes blurring in the transition areas between foreground and background layers.

[SGSS08] introduced a method for visualizing large collections of photos downloaded from community photo Websites. They use planar proxies to represent images in order to create pleasing 3D viewing experience when browsing through the photos. However, their focus is on structured navigation of a large scene rather than edge-preserving view synthesis.

3. Reconstruction

Our algorithm begins with a set of photos from uncalibrated cameras. We assume only that the focal length is known.

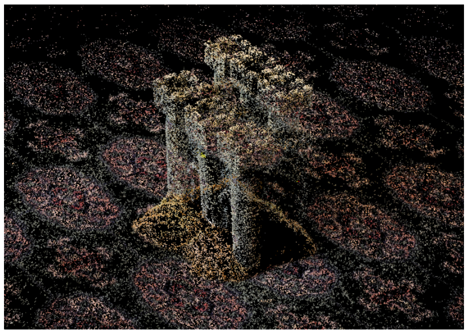

Figure 2: *Example 3D point cloud from Bundler*

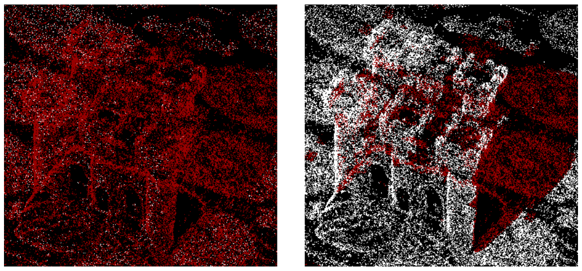

Figure 3: *The left image shows points (in white) that are marked visible for this view by Bundler; the right image shows marked visible points after our visibility testing step.*

Usually the focal length is available in an image's EXIF tags. We pass the images to Bundler [SSS06] to obtain (i) the cameras parameters for each view, (ii) a cloud of recovered 3D points, and (iii) the 2D location of each 3D point in each image where it is visible.

3.1. The Camera-Graph

Bundler provides a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{T}$ for each camera. The 3D location of a camera (the camera center) is then $\mathbf{C}_i = -\mathbf{R}_i^\top \cdot \mathbf{T}_i$. We make the simplifying assumption that all the camera centers lie roughly on a 2D topology such as the surface of a sphere. This is mainly for the efficiency of nearest neighbor search, and works well for most of our capturing scenarios. We only require an approximate 2D topology, so that we can compute a reasonable 1-1 mapping $M : \mathbb{R}^3 \to [-\pi,\pi) \times [-\pi,\pi)$ from the 3D camera positions to latitude-longitude coordinates for use in view-interpolation. To compute the mapping M, we first find best-fit sphere S for the camera locations. We then compute a rotation matrix $\mathbf{R}$ about the center of S that simultaneously minimizes the squared z-coordinates of all camera centers. The latitude and longitude of each camera, rotated by $\mathbf{R}$ about the center of the sphere, is then used to define the 2D location of each camera.

Next, we compute a 2D Delaunay triangulation of the 2D camera locations. We call this the *camera graph*. The camera graph is used in the rendering phase to choose a set of nearby source views for any novel viewpoint. Figure 1 shows an ex-

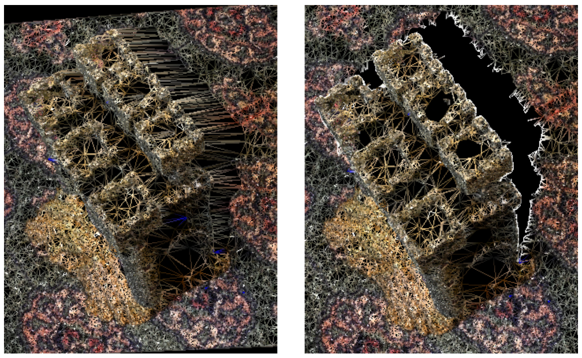

Figure 4: *A mesh before and after splitting the long edges that cross depth discontinuities.*

ample camera graph for a 360-degree dataset that we captured.

3.2. Initial Mesh-Construction

The point cloud provided by Bundler is not dense enough for direct point-based rendering. Therefore we need to recover mesh geometry. While we could use techniques that reconstruct a surface from an unstructured point cloud, such as [HDD*92], we found that such techniques often have difficulties providing smooth meshes, causing noticeable rendering artifacts. In practice a single global mesh does not give good results where there are depth-discontinuities that the camera cannot see around. We could also reconstruct a dense depth map using multi-view stereo matching [YPYW04]. However, we found that these approaches have difficulties handling sharp depth discontinuities.

Our method computes a set of meshes from the point cloud and uses the results for depth-interpolation at a novel view. Our method computes an explicit, view-dependent mesh for each source image, and then refines the mesh using edge information in image-space to preserve depth-discontinuities. Computing view-dependent meshes avoids the problem of reconstructing global geometry. While there will be error in each estimated mesh, the view-dependent property ensures that the error is restricted to the line of sight. Our rendering stage only uses meshes generated from source images that are close to the novel viewpoint, so this line of sight error remains largely hidden.

The mesh-generation step begins by identifying points that are visible in each source view. Bundler returns a list of images in which each point appears. However, we have found that this estimation is often very conservative – many points that are visible in a given view are not marked as visible by Bundler. Since a denser set of points will improve the quality of our results, we use a simple testing step to detect the visibility of every 3D point in every view.

The visibility testing can be done efficiently using OpenGL rasterization. The basic idea is to use the entire

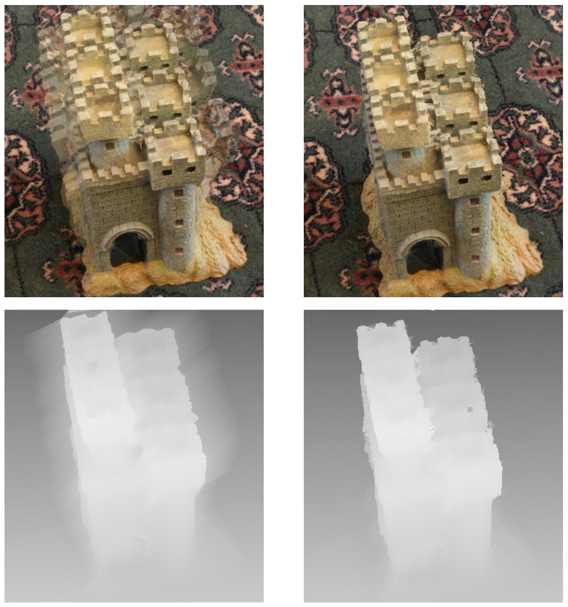

Figure 5: *Top: Synthesized view before and after edge-splitting. Bottom: The corresponding depth maps.*

3D point cloud to perform point-based rendering, and obtain a rough depth map for each source view. Specifically, from each of the original cameras, we render 3D point cloud using a sufficiently large point size and a shader that colors each point with the window z-coordinate. We then use this depth-map to find points in each view that are not already marked as visible. First, we compute the window z-coordinate of every 3D point in every view. Then any 3D point that lies within a range (typically about 0.01) of the value in the depth map is marked as visible. We perform the computation of this step on the GPU using NVIDIA's CUDA, and we are able to process $\sim 240,000$ points in 96 views in a few seconds. Figure 3 shows an example point cloud before and after the visibility-test in one of the source viewpoints.

Since the set of visible points at each view has a natural 2D parametrization, we can compute a mesh $\mathbf{M}_i$ for view i by triangulating the visible points in the 2D domain with a 2D Delaunay triangulation. We then replace the 2D points with their corresponding 3D points to obtain a 3D mesh. Since only visible points are considered, we automatically avoid the problem of including occluded points in the mesh.

Note that our visibility-testing step may incorrectly exclude some visible points around depth-discontinuities, because the point size used for rasterization may be large enough that some foreground points incorrectly occlude some visible background points. The direct consequence is that the triangulated mesh will contain long triangles that connect foreground and background points, and these triangles are not along the line of sight. However, this issue is not a serious concern because our mesh splitting step below

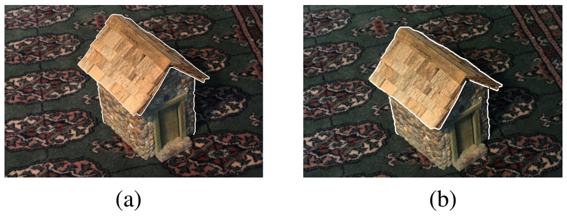

Figure 6: *(a) Manually-marked edges. (b) Automatically-marked edges in a nearby view.*

will split triangles that cross occlusion boundaries, thereby preserving the sharp depth-discontinuities.

3.3. Edge-Splitting

We could use the meshes computed above directly for view synthesis. While this will be good enough for rendering a large part of the scene, there are serious problems around depth-discontinuities, because each mesh will initially have a large number of long triangles connecting foreground and background objects. While these triangles are not difficult to detect, removing triangles from a mesh creates holes which must be filled. Our algorithm can both detect and remove these triangles, and with the help of some user input, fill in the missing geometry by splitting the long edges into segments that closely approximate the true geometry. Figure 4 shows a mesh before and after our edge-splitting algorithm. Figure 5 shows the rendered results and the corresponding depth maps before and after our edge-splitting algorithm.

User-Marked Boundaries. Our edge-splitting algorithm requires the user to mark obvious depth discontinuities in a subset of the source images (typically around one image for every 15 to 30 degrees of camera rotation about the scene). The subset is chosen automatically with K-means clustering. If m images are to be manually marked, we use K-means to compute m clusters from the set of 3D camera locations, with the 3D Euclidean distance as our distance metric. Then in each cluster, the user manually marks the depth-discontinuities in the view that is closest to the cluster's centroid. Figure 6a shows an example of an image with manually-marked edges.

Edge Splitting. In each manually-marked view V_i, we use the user-marked boundaries to split the long foreground-to-background edges. We begin by reprojecting the corresponding mesh M_i onto the image plane. We must then find all triangles in M_i that intersect one or more manually-marked boundaries in 2D. We efficiently detect the intersections with a uniform grid spatial structure.

If a mesh-edge intersects at least one user-marked boundary, we have found a foreground-to-background triangle. We call these the *bad* triangles. Our algorithm will then attempt to subdivide each bad triangle into a set of new triangles that more closely approximate the scene geometry. We make the simplifying assumption that each edge of a bad triangle crosses at most one depth discontinuity, and thus at most one user-marked boundary. If this assumption is violated, we use the boundary that has the strongest image-space gradient. Thus we can guarantee that any edge is either split in two by a single depth discontinuity, or left intact.

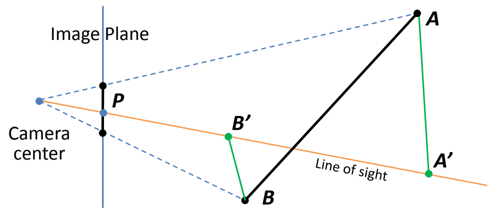

Figure 7: *An example of a long edge showing the original line segment $\overline{AB}$ and the new line segments $\overline{AA'}$ and $\overline{BB'}$.*

When an edge $\overline{AB}$ intersects a depth discontinuity at a 2D point P, we split the edge $\overline{AB}$ at P by creating two new points, A' and B', such that the projections of A' and B' coincide with P (that is, A' and B' both lie on the line of sight through P). The edge $\overline{AB}$ is then replaced with edges $\overline{AA'}$ and $\overline{BB'}$. Since A' and B' both project to P, the 2D geometry will appear the same with the addition of a new point along the segment $\overline{AB}$. However, we can shift A' and B' along the line of site through P so that the new edges $\overline{AA'}$ and $\overline{BB'}$ more closely approximate the local geometry around A and B respectively. Fig 7 shows an illustration.

The 3D locations of A' and B' are inferred from A and B and their neighbors, respectively. For A', we choose the depth to be the average of the depth of A and all its neighboring vertices that are connected by mesh-edges (excluding bad edges). We then position A' on the line of site through P at the desired depth. We do the same for B' using B and its neighbors. If A and B have accurate 3D locations, we can guarantee that A' and B' will be reasonable approximations. While we could possibly gain some accuracy by fitting planes to the local geometry around A and B, we have found that in practice, using the average depth gives better results. While we introduce some inaccuracy in this approach, we also avoid introducing errors due to inaccurate plane-fitting when the local geometry is imperfect. Since the error is along the line of sight and the mesh will only be rendered from nearby novel viewpoints, the error typically has little effect. Figure 7 shows an example of the two split points A' and B'.

Once the bad edges have been split, we need to retriangulate the affected triangles. A foreground-to-background triangle can have either one, two, or three bad edges, and we need to handle each case independently. Since the user-marked edges can be arbitrary, all three possible cases must be considered.

Retriangulation. If a triangle has only one bad edge (Fig 8a), there is only one way to re-triangulate it, namely by

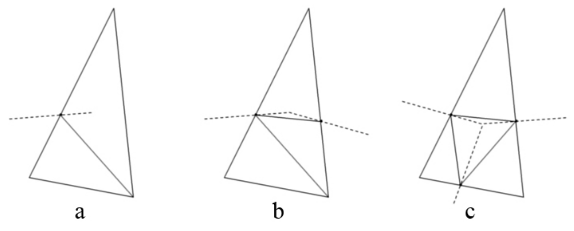

Figure 8: *The three possibilities for splitting a bad triangle. The dotted lines represent user-marked edges, and the solid lines represent the splits.*

connecting each of the new vertices with the original vertex opposite the single bad edge. This splits the original triangle into two new triangles.

If a triangle has two bad edges (Fig 8b), we add new edges connecting the split points, thus splitting the original triangle into a triangle and a quadrilateral. While the 2D projection of the mesh appears to have one new edge, there are in fact two new edges. Each split-point has a foreground copy (which is always closer along the line of sight) and a background copy (which is always farther along the line of sight). One of the new edges joins the foreground split-points and the other joins the background split-points.

If a triangle has three bad edges (Fig 8c), we create three new triangles, each one containing one of the original vertices and two of the new vertices. The central triangle is not filled in. If we were to fill in the central triangle, we would in fact have three new vertices at each split point. Each vertex connected to the central triangle would not be connected to any of the original vertices, and thus there would be no way to infer its depth. Instead, we make the assumption that every pixel in the central triangle is on the same depth plane as one of the neighboring triangles. The depth plane for each pixel is then selected by the rendering algorithm. At this stage, we simply maintain a list of these triangles, along with a list of depth estimates for each one. This list will be passed to the rendering algorithm.

Automatic Boundaries. We only require the user to mark a subset of the source images. In the remaining images, we attempt to 'propagate' the boundaries from the nearest user-marked view. This is done by finding 3D locations for the user-marked edges, and reprojecting them into unmarked views. The 3D locations are determined from the 3D locations of the new vertices in the foreground to background triangles. We began by finding all bad triangles that were split along exactly two edges. Each of these triangles will have two split points, and two new edges between the two copies of the split points. We can determine which new edge is attached to the foreground copies by checking the distances from the endpoints to the camera center. The foreground copy is then reprojected into the unmarked view, which creates a single line segment. This is repeated for every such triangle, creating a set of edges in the unmarked view. These edges are then used to split the mesh in the same manner as the user-marked edges in the marked views. Fig 6b shows an example of an image with automatically-marked edges.

Outlier-Rejection. Due to error in the marked edges and 3D vertex positions, a small number of foreground-background edges remain in the meshes even in the user-marked views. The final step in our mesh generation algorithm is to detect and remove these edges. We do this with a histogram of 3D edge lengths. The histogram ranges from the length of the shortest edge to the length of the longest edge (before we split the bad triangles). Since most of the bad edges are eliminated at this stage, the histogram bins that would contain bad edges are relatively empty. We exploit this observation to choose a threshold for eliminating bad edges. We find the first bin in the histogram having fewer than a constant number of edges (typically around 500), and then delete any edge that falls into either this bin or a larger bin.

Each time we eliminate an edge, we create a hole in the mesh. Since we do not have any split points on the edges eliminated at this stage, we can't apply our previous approach for filling in the geometry. Instead, we treat the holes exactly like the central triangles in the 3-way splits, and fill in the geometry in the rendering stage.

4. Rendering

4.1. View Interpolation

Camera Interpolation. Our rendering algorithm begins with the choice of a novel viewpoint. The new viewpoint is selected by clicking on the desired location within a 2D view of the camera graph. This provides the 2D coordinates of the new camera, but not the position or rotation. Both the position and rotation are determined by interpolating from the three source cameras at the vertices of the triangle containing the novel viewpoint. The barycentric coordinates are used to obtain weights w_1, w_2, and w_3, which guarantees smooth transitions between frames across the entire camera graph. For rotation, we use spherical linear interpolation (SLerp). Given two quaternions q_1 and q_2, we compute the interpolated rotation by:

$$SLerp(q_1,q_2,t) = \frac{q_1 sin((1-t)\theta) + q_2 sin(t\theta)}{sin(\theta)} \qquad (1)$$

where θ is the angle between q_1 and q_2. This works well for interpolating two quaternions, but it is not immediately obvious how to apply SLerp to three quaternions. If we represent the rotations for our three source cameras using quaternions q_1, q_2, and q_3, we define three possible interpolators, q_{123}, q_{213}, and q_{312}, where

$$q_{ijk} = SLerp(q_i, SLerp(q_j,q_k,w_j/(w_j+w_k)),w_i) \qquad (2)$$

Each of these three quaternions will differ slightly, and so by choosing one, we could not gaurantee smooth transitions between frames. Our solution is to combine them by computing a weighted average of q_{123}, q_{132}, and q_{312}. They are close enough that simple linear interpolation will suffice. Our new camera rotation is thus given by:

$$q = \frac{w_1 \cdot q_{123} + w_2 \cdot q_{213} + w_3 \cdot q_{312}}{||w_1 \cdot q_{123} + w_2 \cdot q_{213} + w_3 \cdot q_{312}||} \qquad (3)$$

Free-Navigation. Our viewer also supports free navigation in the 3D view without clicking on the camera graph. In free navigation mode, the novel viewpoint is chosen by mouse navigation within the 3D view window. We choose our source views by projecting the novel camera into the camera graph. If the novel viewpoint falls inside a triangle, we process the interpolation as described above. Otherwise, we use the two nearest views, weighted by their relative distance to the novel view.

Depth Blending. Given a novel camera, we can easily obtain depth estimates for most pixels of the new image. This is done by rasterizing the meshes associated with the three source cameras into the new viewpoint, and then blending the results using the barycentric weights w_1, w_2, and w_3 described above. For now, we assume that at least one source mesh has projected geometry covering each pixel. The next section describes how we handle cases of holes, where no geometry exists. If one or two meshes do not have projected geometry covering a pixel, we exclude them from the computation for that pixel.

To obtain the color for a pixel, we reproject the blended 3D point to the three source views in order to obtain three colors C_1, C_2, and C_3. Using barycentric weights again, we compute a weighted sum of the three colors, and assign that as the synthesized color.

Vertex Weights. We improve the accuracy of our system by observing that the 3D location an original vertex, which was computed from Bundler, is more trustworthy than a split vertex, which was computed on a bad triangle and inferred from the geometry of its surrounding points. We incorporate this observation into our algorithm by assigning a confidence value to each vertex. We define the confidence to be 1 for each of the original vertices, and 0 for the new vertices that were added to split occlusion-boundaries. We place the confidence value for each vertex in the alpha channel when rasterizing the source meshes, thus providing an interpolated confidence value for each pixel covered by a source mesh. This gives us three additional weights u_1, u_2 and u_3 which represent the confidence for pixel p in the three source meshes. Putting everything together, we compute the blended 3D location as:

$$P = \frac{\sum_{i=1}^{3} u_i \cdot g_i \cdot w_i \cdot P_i}{\sum_{i=1}^{3} u_i \cdot g_i \cdot w_i} \qquad (4)$$

where w_i's are the barycentric weights, g_i is a binary value indicating whether the projected geometry of a source mesh covers the pixel, and P_i is the 3D location obtained from the ith source view. We then blend the color as described above.

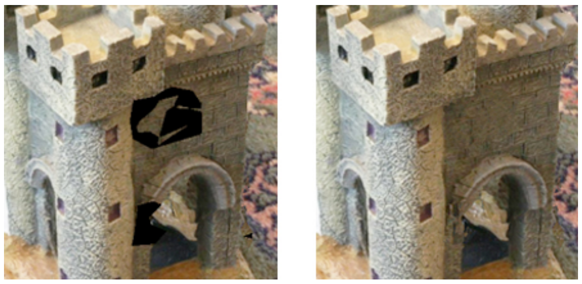
Figure 9: *A closeup of a synthesized view before and after our hole-filling algorithm.*

4.2. Hole Filling

If none of the three source meshes covers a pixel p, there will be a hole in the final rendered image. This happened in 1 to 3 percent of the pixels in our experiments. As described in section 3.3, in the case where a triangle is split on all three sides, the resulting mesh will have no geometry in the center subdivided triangle. In addition, the elimination of outlier triangles creates holes in the source meshes.

Instead of trying to fix the holes offline, we propose to fill the holes on the fly using a modified version of [YWB02], which performs real-time multi-view stereo matching on the GPU. The original algorithm in [YWB02] tried to guess the pixel depth in a novel image by taking discrete steps in a global depth range. The depth assignment is chosen as the one that results in the best color consistency in the source views.

In our case, we can compute the depth more efficiently and robustly in most holes because our reconstruction step provides reliable depth estimates for the missing triangles. Specifically, in both cases of holes above, our reconstruction step determined three guesses for the depth of any pixel that is part of one of these triangles. Thus for each source mesh, if a pixel p lies in a hole, we can intersect the ray through p with each of the three possible planes to get an estimated 3D location, and then use the 3D location that minimizes the variance across the three source images. If possible, we repeat this process for each of the source meshes, which will provide up to three depth-estimates. The final depth-estimate is obtained by blending the depth-estimates with our three weights, w_1, w_2, and w_3.

Any remaining hole pixels p that cannot be filled with the above method will occur in locations where we have neither geometry from any of the meshes nor a set of possible depth planes. In these cases, we test a fixed number of depth values (typically around 20) spaced evenly along the line of site through p, ranging from the near clipping plane to the far clipping plane. We take the 3D location to be the point on the

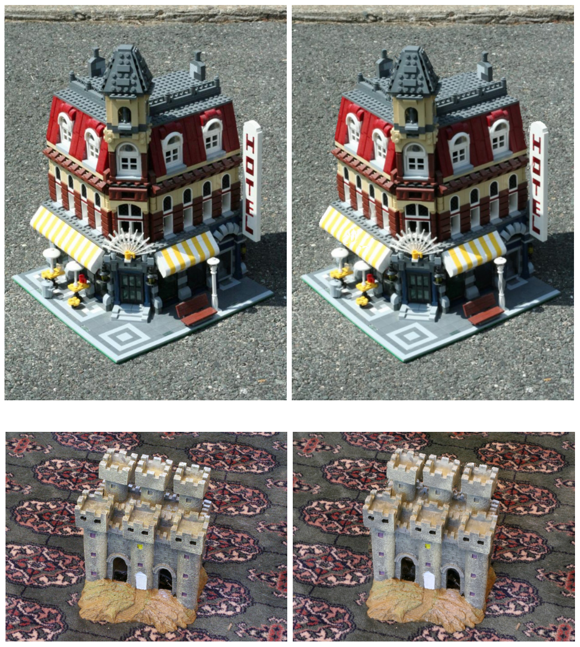

Figure 10: *Left column: original images; right column: synthesized images from the same viewpoint. Note that the synthesized images look qualitatively the same to the originals, but slight differences may be observed in tone and details geometry on the boundary.*

line of site that minimizes the color variance across the three source images. We found this approach to be quite efficient as the number of hole pixels that need this computation is very small.

Figure 9 shows a comparison of rendered images with hole filling and without. Clearly the hole filling results in a much more pleasing image, while adding only a small amount of computation overhead on the fly.

5. Results

We have tested our algorithm on a number of datasets, ranging from 64 to 135 images. In each case, while there are some artifacts, the majority of the scene is realistically rendered. Figure 11 summarizes some of our datasets. In figures 12, 13, 14, and 15, we show the results from each dataset including (a) the camera triangulation, showing the location of the novel camera, (b) the synthesized image, and (c) the depthmap obtained from blending the three source images.

We tested our system at 640×480 on a Dell Studio XPS with an Intel Core i7 920 CPU (2.67 GHz), 6 GB of RAM, and a GeForce GTX 280 GPU. Since the computation for each view is mostly independent, the algorithm is easily parallelized to take advantage of multicore CPU's. We ran 8 threads in each of our test cases. Each thread processes a subset of the views. The computation times are reported in figure 11 for the mesh-generation algorithm only. In all cases, the computation time is dominated by Bundler, which runs for 30 to 45 minutes on our datasets.

The framerates are reported both with hole-filling enabled (20 depth steps) and with hole-filling disabled. Column f lists the approximate percentages of pixels for which hole-filling was required in each dataset. Since hole-filling is only used on a small percentage of pixels, errors in the hole-filling algorithm are typically difficult to spot. As a result, we can get good results with a relatively small number of depth planes, even if there is a large depth range in the scene. Using a larger number of depth planes does improve the quality, but at the cost of a lower frame rate. We have found that in practice, the improvement is barely noticeable above about 20 depth steps.

To demonstrate the accuracy of our system, we ran our algorithm on two of the datasets with one image excluded. We then resynthesized the missing image in each dataset. Figure 10 shows the results. The resynthesized and original views are difficult to distinguish. There is a slight difference in color in the second example, but this is only due to slight differences in lighting between the original view and the source views for the resynthesized image.

6. Limitations and Future Work

In this paper, we have presented a practical method for multiview synthesis that is robust around depth discontinuities. Our method computes a visible surface for each input view by triangulating visible points detected via structure from motion. We propose a method that exploits user-marked occlusion boundaries on a small set of representative images to robustly preserve depth discontinuities. We also employ a GPU-based real-time stereo matching algorithm for hole filling during view synthesis.

Our method currently assumes that camera locations exist on a 2D topology, such as a planar surface or a hemisphere. This makes it easy to locate neighboring views for a novel viewpoint. We observe that this assumption is true in many capturing scenarios. In cases where the cameras are not obviously aligned on a plane or hemisphere, we can apply nonlinear dimensionality reduction to analyze the camera parametrization. It is also possible to relax this assumption and use a kd-tree for efficient nearest neighbor search in case of a general camera topology.

The quality of our rendering will degrade if the input images are very sparse or non-uniformly distributed. In this case, the visible surfaces of adjacent views do not have sufficient overlaps, causing the results to contain many holes. Although our real-time hole filling algorithm can partially account for this problem, we won't be able to robustly preserve depth discontinuities.

Another direction for future work is to parallelize the triangle splitting computation on the GPU. Currently this step

		Images (a)	Clusters (b)	3D Points (c)	Time (d)	FPS (e)	Hole-Filling (f)
Castle	(fig 12)	96	12	243,384 ($\sim 100,000$)	5 m 11 s	8 / 10	1 - 3%
Lego house	(fig 13)	64	6	87,485 ($\sim 44,000$)	1 m 8 s	13 / 14	< 1%
Stone House	(fig 14)	76	6	44,059 ($\sim 7,000$)	15 s	11 / 19	1 - 2%
Birdhouse	(fig 15)	136	12	72,839 ($\sim 31,000$)	1 m 50 s	22 / 22	< 1%

Figure 11: *Summary of test datasets. The columns show (from left to right) the number of images in each dataset, the number of camera clusters, the total number of 3D points found by Bundler, the average number 3D points visible in each view, the time required to generate the meshes from the 3D point cloud, the average fps for rendering (with and without hole-filling), and the average percentage of pixels for which hole-filling was applied.*

takes 1 to 5 minutes to compute on the CPU. By improving its speed toward interactive rates, we may allow the user to obtain real-time feedbacks of edge marking, therefore shortening the processing time and modeling cycles.

References

[CTCS00] Chai J.-X., Tong X., Chan S.-C., Shum H.-Y.: Plenoptic sampling. In *Proc. SIGGRAPH '00* (2000), pp. 307–318. 3

[CW93] Chen S. E., Williams L.: View interpolation for image synthesis. In *Proc. SIGGRAPH '93* (1993), pp. 279–288. 3

[DTM96] Debevec P. E., Taylor C. J., Malik J.: Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *Proc. SIGGRAPH '96* (1996), pp. 11–20. 2, 3

[ES04] Esteban C. H., Schmitt F.: Silhouette and stereo fusion for 3d object modeling. *CVIU 96*, 3 (2004), 367–392. 2

[GCS06] Goesele M., Curless B., Seitz S. M.: Multi-view stereo revisited. In *Proc. CVPR '06* (2006), pp. 2402–2409. 2

[GGSC96] Gortler S. J., Grzeszczuk R., Szeliski R., Cohen M. F.: The lumigraph. In *Proc. SIGGRAPH '96* (1996), pp. 43–54. 2

[GS05] Gargallo P., Sturm P.: Bayesian 3d modeling from images using multiple depth maps. In *Proc. CVPR '05* (2005), pp. 885–891. 2

[HDD*92] Hoppe H., DeRose T., Duchamp T., McDonald J., Stuetzle W.: Surface reconstruction from unorganized points. In *Proceedings of SIGGRAPH '92* (1992), pp. 71–78. 4

[HEH05] Hoiem D., Efros A. A., Hebert M.: Automatic photo pop-up. *ACM Trans. Graph. 24*, 3 (2005), 577–584. 1

[HZ04] Hartley R., Zisserman A.: *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004. 2

[KS00] Kutulakos K. N., Seitz S. M.: A theory of shape by space carving. *Int. J. Comput. Vision 38*, 3 (2000), 199–218. 2

[KS04] Kang S. B., Szeliski R.: Extracting view-dependent depth maps from a collection of images. *Int. J. Comput. Vision 58*, 2 (2004), 139–163. 2

[LH96] Levoy M., Hanrahan P.: Light field rendering. In *Proc. SIGGRAPH '96* (1996), pp. 31–42. 1, 2

[Low99] Lowe D. G.: Object recognition from local scale-invariant features. In *Proc. ICCV '99* (1999), p. 1150. 2

[MB95] McMillan L., Bishop G.: Plenoptic modeling: an image-based rendering system. In *Proc. SIGGRAPH '95* (1995), pp. 39–46. 2

[MHM*09] Mahajan D., Huang F.-C., Matusik W., Ramamoorthi R., Belhumeur P.: Moving gradients: a path-based method for plausible image interpolation. *ACM Trans. Graph. 28*, 3 (2009), 1–11. 3

[MMB97] Mark W. R., McMillan L., Bishop G.: Post-rendering 3d warping. In *Proc. SI3D '97* (1997), pp. 7–16. 1, 3

[NRK98] Narayanan P. J., Rander P. W., Kanade T.: Constructing virtual worlds using dense stereo. In *Proc. ICCV '98* (1998), p. 3. 2

[PKG99] Pollefeys M., Koch R., Gool L. V.: Self-calibration and metric reconstruction inspite of varying and unknown intrinsic camera parameters. *Int. J. Comput. Vision 32*, 1 (1999), 7–25. 2

[PVGV*04] Pollefeys M., Van Gool L., Vergauwen M., Verbiest F., Cornelis K., Tops J., Koch R.: Visual modeling with a hand-held camera. *Int. J. Comput. Vision 59*, 3 (2004), 207–232. 2

[SCD*06] Seitz S. M., Curless B., Diebel J., Scharstein D., Szeliski R.: A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proc. CVPR '06* (2006), pp. 519–528. 2

[SCK07] Shum H.-Y., Chan S.-C., Kang S. B.: *Image-Based Rendering*. Springer, 2007. 2

[SD96] Seitz S. M., Dyer C. R.: View morphing. In *Proc. SIGGRAPH '96* (1996), pp. 21–30. 3

[SD97] Seitz S. M., Dyer C. R.: Photorealistic scene reconstruction by voxel coloring. In *Proc. CVPR '97* (1997), p. 1067. 2

[SGHS98] Shade J., Gortler S., He L.-w., Szeliski R.: Layered depth images. In *Proc. SIGGRAPH '98* (1998), pp. 231–242. 1, 3

[SGSS08] Snavely N., Garg R., Seitz S. M., Szeliski R.: Finding paths through the world's photos. *ACM Trans. Graph. 27*, 3 (2008), 1–11. 3

[SK94] Szeliski R., Kang S.: Recovering 3d shape and motion from image streams using non-linear least squares. *Journal of Visual Communication and Image Representation 5*, 1 (1994), 10–28. 2

[SSS06] Snavely N., Seitz S. M., Szeliski R.: Photo tourism: exploring photo collections in 3d. *ACM Trans. Graph. 25*, 3 (2006), 835–846. 2, 3

[TK92] Tomasi C., Kanade T.: Shape and motion from image streams under orthography: a factorization method. *Int. J. Comput. Vision 9*, 2 (1992), 137–154. 2

[WAA*00] Wood D. N., Azuma D. I., Aldinger K., Curless B., Duchamp T., Salesin D. H., Stuetzle W.: Surface light fields for 3d photography. In *Proc. SIGGRAPH '00* (2000), pp. 287–296. 1, 3

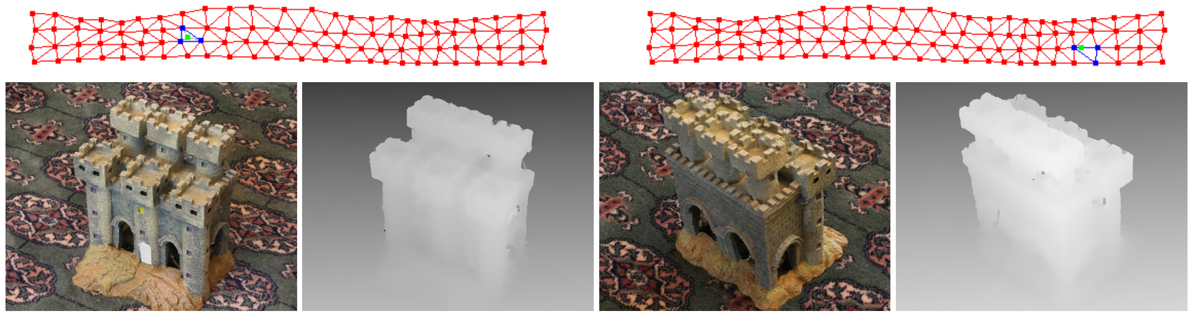

Figure 12: *Castle dataset*

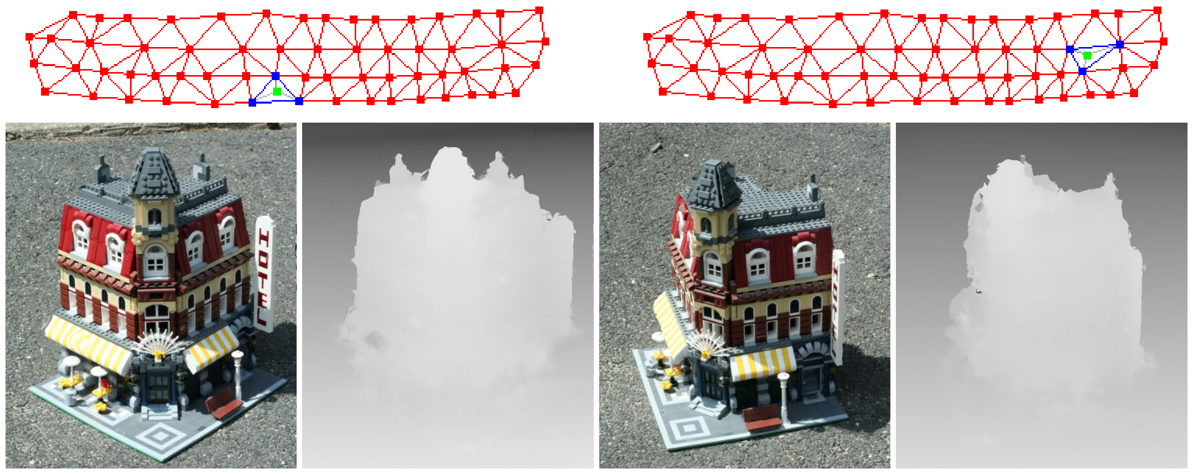

Figure 13: *Lego house dataset*

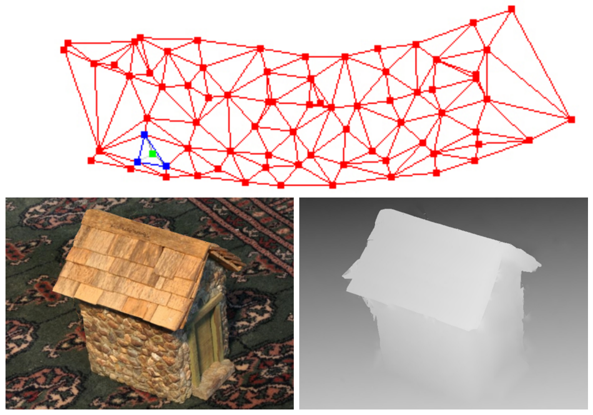

Figure 14: *Stone house dataset*

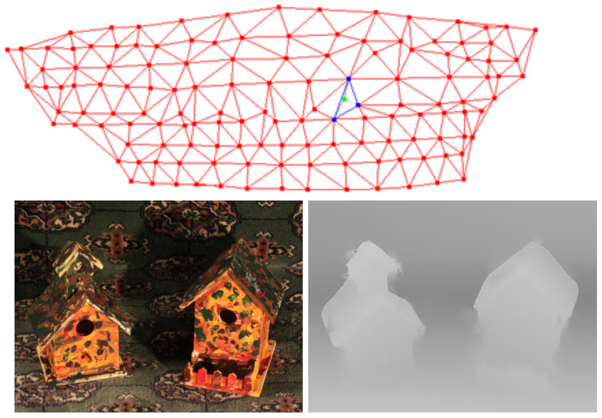

Figure 15: *Birdhouse dataset*

[YPYW04] YANG R., POLLEFEYS M., YANG H., WELCH G.: A unified approach to real-time, multi-resolution, multi-baseline 2d view synthesis and 3d depth estimation using commodity graphics hardware. *International Journal of Image and Graphics 4* (2004), 2004. 3, 4

[YWB02] YANG R., WELCH G., BISHOP G.: Real-time consensus-based scene reconstruction using commodity graphics hardware. In *Proc. of Pacific Graphics* (2002), p. 225. 2, 7