Photo Tours

Avanish Kushal[1], Ben Self[1], Yasutaka Furukawa[1,2], David Gallup[2], Carlos Hernandez[2], Brian Curless[1], and Steven M. Seitz[1,2]

[1] University of Washington, Seattle
{kushalav, selfb, furukawa, curless, steve}@cs.washington.edu

[2] Google Inc.
{dgallup, chernand}@google.com

Abstract

This paper describes an effort to automatically create "tours" of thousands *of the world's landmarks from geotagged user-contributed photos on the Internet. These* photo tours *take you through each site's most popular viewpoints on a tour that maximizes visual quality and traversal efficiency. This planning problem is framed as a form of the Traveling Salesman Problem on a graph with photos as nodes and transition costs on edges and pairs of edges, permitting efficient solution even for large graphs containing thousands of photos. Our approach is highly scalable and is the basis for the Photo Tours feature in Google Maps.*

1. Introduction

What does the Pantheon look like? Answering this type of question is a fundamental goal of computer graphics, yet we still lack truly effective visualizations. The problem is not a lack of good imagery—there's plenty available via Internet search—but rather displaying such imagery in a coherent, informative, and efficient way. Work on image-based rendering [6, 3, 11] has yielded significant progress towards addressing the first problem—coherence—through more realistic transitions between photos which allow the viewer to maintain context and physical relationships. However, for a visualization to be informative, it must communicate what that scene is *about*, i.e., the most interesting and relevant content. Finding good content and navigating efficiently through the scene remain significant challenges for current state-of-the-art IBR systems like Photosynth and Streetview. Photosynth, for example, has four different modes of viewing the scene (3D view, overhead, 2D view, point cloud), with a dozen different controls (buttons or other click behaviors) in the main 3D mode - which take time to master. Even for an expert, it can be difficult to find his way around a new scene.

Our goal is to produce visualizations of real scenes that are as coherent, informative, and efficient as possible by encoding these criteria as constraints and objectives and optimizing them directly. In order to achieve this goal, we take away user interaction, and instead pose the problem of generating the sequence of frames that best conveys the essence of that scene. We call such a sequence a *photo tour*. Watching a photo tour is like looking over the shoulder of an expert Photosynth user, who knows all the highlights and how to traverse them. Furthermore, the movie can communicate the most interesting aspects of the scene in a relatively short amount of time. And while our focus is on automation, we note that photo tours can be used to compliment an interactive system, e.g., as a quick scene overview or auto-play mode.

Snavely et al. [18] proposed a way of computing good paths (orbits and panaromas) through photo collections. They also addressed the problem of two-point planning, i.e., recovering the best path from one image to another—an important subcase which we generalize here to the general planning problem. An important distinction is that [18], like prior work in image-based rendering, sought to produce an interactive system, whereas our objective is a completely automatically generated *movie*. We produce photo tours of popular tourist sites by processing imagery from photo sharing sites like Flickr.com. The goal is to automatically plan a tour through the most interesting (i.e., frequently photographed) objects and viewpoints, with compelling 3D transitions that convey the spatial relations between views. In this paper, we show how to convert this problem into a path planning problem (solving a form of the of the Traveling Salesman Problem (TSP) [2]) through a graph with images as nodes, and edge weights encoding the quality of the rendered transitions between each pair of images. However, it differs from the traditional TSP, in that we are required to visit only a subset of nodes in the graph. The most related formulation is the Traveling Tourist Problem [13], which solves for the shortest tour that would allow a tourist

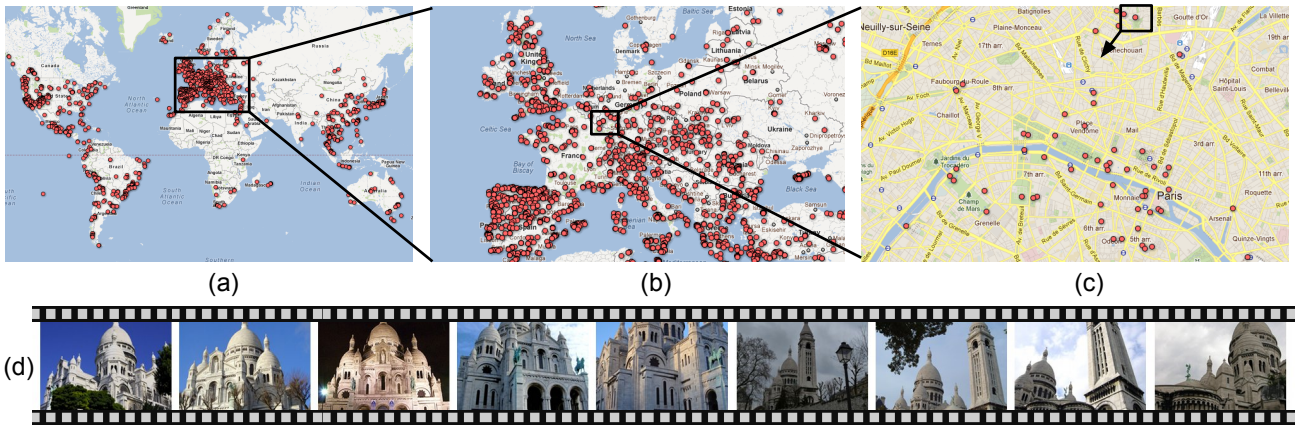

Figure 1. We have computed thousands of *photo tours* across the globe, shown as red dots (a). (b) shows a closeup of Europe, and (c) a zoom-in to one neighborhood in Paris. (d) shows the sequence of views in the photo tour for Sacre-Couer near Paris; the movie itself includes fluid 3D transitions between consecutive views (see video).

to *see* all the nodes in a graph, by visiting each node or one of its neighbors (it is assumed that every node is seen by its neighbors—the edges represent lines of sight). The formulation, however, is scene-based rather than photo based, doesn't account for partial visibility or rendering quality, and requires coverage of all the nodes (an overkill in our case). We draw heavily on prior work in IBR for producing effective image transitions, but by optimizing for fixed *tours* rather than interactive experiences, we seek to sidestep many of the challenges and pitfalls that have made prior systems hard to learn and navigate. Our technique use per image depth proxies to give a compelling sense of parallax, which is further enhanced by using a technique called the Ken-Burns effect from 2-D photography.

Our system represents the first attempt to automatically construct informative and visually pleasing tours *at world-scale* by harvesting the vast stores of community photo collections on the Internet. The photo tours feature in Google Maps implements our method to generate movies for thousands of sites. These sites are indicated as red dots on the maps in Figure 1, which shows the distribution of *photo tours* across the globe, in Europe and around Paris. Our approach resembles Photosynth in terms of scale and computer vision techniques, but differs in our use of community photo collections(CPC), rather than photos contributed by a single user. CPC's are useful in that they also capture the distribution of photos that people take of a scene, which can be mined to guide the user to the highlights.

2. Problem Definition

Consider a collection of geo-registered images $I = \{I_1, I_2 ... I_m\}$ with known camera pose and sparse 3D points, recovered through a Structure from Motion (SfM) procedure, which also specifies which 3D points are visible in which images (details deferred to Section 6). Suppose we also have a depth map for each photo, e.g., recovered through stereo, represented as a set of depth values over a regular pixel grid.

Construct an image graph G_I, consisting of a node for each image and an edge between a pair of nodes if they share common visible 3D points. A tour $P = \{P_1, P_2 ... P_{|P|}\}$ on this graph is a sequence of nodes in G_I such that each node P_j is connected to the next node P_{j+1} through an edge, call it e_j. Our goal is to compute a tour that is informative, coherent, and efficient. We address the first objective by computing a set of *canonical views* C [17] capturing the most frequently photographed scene content, and constraining the tour to include these nodes. We will achieve *efficiency*, by computing shortest paths through the graph. *Coherence* is achieved by ensuring the viewer maintains context as the tour transitions between photos, and encoded via two objectives defining *rendering* and *smoothness* costs, respectively, as follows.

- *Rendering*: We seek a tour P that leads to high quality visualizations. Thus, each edge on the tour should produce high quality renderings between the images.
- *Smoothness*: Geometrically, P should result in a smooth (not jerky) camera motion when traversed. Thus, for each pair of consecutive edges on P, the corresponding transitions should be similar.

The overall objective is then to find the tour $P : C \subset P$ that minimizes

$$\sum_j RenderingCost(e_j) + \alpha \sum_j SmoothnessCost(e_j, e_{j+1}) \quad (1)$$

where α trades off transition quality for the smoothness of the tour, and is set to 1.0 in all our experiments.

In the remainder of this section, we elaborate on the cost functions used to encode the objectives. Later (in Section 3), we describe our method for generating tours that approximately minimize Eq 1.

2.1. Cost Functions over G_I

We now define the $RenderingCost$ for each edge in G_I and the $SmoothnessCost$ for each pair of edges that share a node in G_I. We can think of these two costs as, respectively, a first order cost that encourages high quality transitions between images and a second order cost that regularizes the tour to avoid jerky camera motion.

We construct these costs from a set of terms which we now describe. Let e_1 denote the transition from I_A to I_B and e_2 denote the transition from I_B to I_C; these edges correspond to a sequence of two consecutive transitions. We define the following desirable properties for a *good* tour:

1. **Camera Motion.** Larger transitions – in terms of motion of the camera center, as well as changes in the rotation matrix – can be expected to render poorly and thus should be penalized with higher costs. Let O_A be the optical center, and R_A the rotation matrix for image I_A (similarly for B and C). Then

$$\begin{aligned} t(e_1) &= ||O_A - O_B||_2, & (2)\\ r(e_1) &= ||R_A^{-1} R_B||_\theta, & (3)\\ t(e_1, e_2) &= ||O_A - 2O_B + O_C||_2, & (4)\\ r(e_1, e_2) &= ||(R_A^{-1} R_B)^{-1} (R_B^{-1} R_C)||_\theta & (5) \end{aligned}$$

   where $||R||_\theta$ is the angle of rotation. Equations 2 and 4 penalize the first and second derivatives of the camera motion, while Equations 3 and 5 penalize the first and second derivatives of the camera orientation.

2. **Optical motion.** Transitions which have large apparent scene motion – optical flow – in the renderings can be expected to have more artifacts. We approximate the overall flow as the average projected motion of the SfM points that are common to pairs of images and denote it as M_{e_1}. Then,

$$\begin{aligned} m(e_1) &= ||M_{e_1}||_2 & (6)\\ m(e_1, e_2) &= ||M_{e_1} - M_{e_2}||_2 & (7) \end{aligned}$$

   penalize the first and second order optical motion.

3. **Scene Area Coverage.** Transitions that zoom-in/zoom-out of the scene a lot cause the viewer to lose their sense of location - thus we would like the transitions to zoom-in/ zoom -out steadily. We encode this objective by trying to keep the surface area covered by an image roughly the same. We measure the surface area covered by each image's depth map as follows. For each pixel with depth z in its corresponding depth map, its surface area is roughly proportional to $\frac{z^2}{f^2}$, where f is the focal length of the image. Then the surface area of the depth map is just the sum over all pixels. Let S_A denote the surface area for image I_A (similarly for B and C), then

$$\begin{aligned} s(e_1) &= |S_A - S_B| & (8)\\ s(e_1, e_2) &= |S_A - 2S_B + S_C| & (9) \end{aligned}$$

   penalize the first and second order change in surface area.

4. **Stretching Artifacts.** We would like to exclude transitions where the depth map corresponding to one image has regions that are severely stretched when rendered from the viewpoint of the next camera on the tour. Consider a depth map pixel in I_A (but not on its boundary), corresponding to a point in 3D. The projection of this point into I_B gives a 2D location p; similarly, the original 4-neighborhood projects to locations N_p. We define the stretching associated with p as

$$L_p = ||p - \frac{1}{4} \sum_{q \in N_p} q||_2, \quad (10)$$

   which measures how much the average of the position of the 4 points around p differs from p. [1] Then the stretching penalty $l(e_1)$ associated with the edge e_1, is set to the 99 percentile value of L_p over the two depth maps involved in the transition.

While other cost measures such as those based on similarity in lighting, could be added, we found that in practice the diversity in terms of day/night images or in terms of changing seasons, provides for a much richer experience.

These costs are now incorporated into the (first order) $RenderingCost$ and (second order) $SmoothnessCost$:

$$\begin{aligned} &RenderingCost(e) & (11)\\ &\quad = t(e) + \alpha_r r(e) + \alpha_m m(e) + \alpha_s s(e) + \alpha_l l(e)\\ &SmoothnessCost(e_1, e_2) & (12)\\ &\quad = t(e_1, e_2) + \alpha_r r(e_1, e_2) + \alpha_m m(e_1, e_2) + \alpha_s s(e_1, e_2) \end{aligned}$$

We use $\alpha_r = 1, \alpha_m = 30, \alpha_s = .002$ and $\alpha_l = 100$ for all our experiments. Note we use the same constants to weight the first and second order weights. For the units, we scale translation so that the variance of the centers of canonical images is 1. We apply this same scale factor to the scene area measure. Rotation is measured in degrees, and optical motion is measured in terms of screen space so that the height of the screen is 1.

[1] This is equivalent to applying the discrete Laplacian and computing the magnitude.

3. Generating Tours

We now describe our algorithm for generating tours through photo collections. In section 2, we defined the problem as finding the tour P that passes through canonical views C while minimizing Eq. 1. Computing the optimal tour on a large graph G_I, however, would be prohibitively expensive.

To simplify the problem, we initially ignore the $SmoothnessCost$ when passing through the nodes corresponding to canonical views. A preliminary tour then can be seen as the concatenation of the shortest paths between pairs of canonical views in G_I as shown in Section 3.1. As a further simplification, we prune the graph before computing each of the shortest paths, which are then used to compute an ordering of the canonical views in Section 3.2. We then retain the ordering of canonical views in the preliminary tour and compute the final tour after re-introducing the $SmoothnessCost$ across canonical views in Section 3.3. Algorithm 1 summarizes our approach.

Algorithm 1 $P = Tour(I)$

(SfM, depth maps,C) $\leftarrow$ Pre-process(I)
Construct G_I and compute $RenderingCost$ per edge
for $C_j, C_k \in C$ **do**
  $\tilde{G}_I(j,k) \leftarrow Prune(G_I, C_j, C_k)$
  Compute $SmoothnessCost$ on $\tilde{G}_I(j,k)$
  Compute shortest paths in $\tilde{G}_I(j,k) : C_j \rightarrow C_k$
end for
Construct G_C
Store cost of $C_j \rightarrow C_k$ as edge weight in G_C
$\Pi \leftarrow TSP(G_C)$
$G_\Pi \leftarrow Splice(\tilde{G}(\Pi_1, \Pi_2), \cdots, \tilde{G}(\Pi_{|C|-1}, \Pi_{|C|}))$
Compute $SmoothnessCost$ per edge-pair $\forall C_i$
$P \leftarrow$ shortest path in G_Π from C_{Π_1} to $C_{\Pi_{|C|}}$
RETURN P

3.1. Approximate Shortest Paths

We can solve for the shortest path under the cost functions between pairs of canonical views in G_I, by using Dijkstra's algorithm [5] on G_H, the line graph of G_I. In particular, each edge in G_I becomes a node with weight $RenderingCost$ in G_H and each pair of edges with a node in common in G_I becomes an edge with weight $SmoothnessCost$ in G_H. Thus the costs are now encoded as node costs and edge costs respectively in G_H.

3.1.1 Pruning the Graph

While we can compute the shortest paths exactly under the given cost functions, in practice it is quite expensive. In particular, the line graph $G_H(V_H, E_H)$ has $|V_H| = |E_I|$ and $|E_H| = O(|V_I|^3)$, where V_I, E_I denote the vertices and edges of G_I, and thus running shortest paths is expensive [20].

To make the problem more tractable, we solve for approximate shortest paths on a pruned graph. For each pair of canonical views, we start from the original graph and remove vertices that are *unlikely* to be part of the shortest path. Consider a pair of canonical views C_j and C_k. We assign edge weights to G_I according to $RenderingCost$ and then run Dijkstra's shortest path algorithm on G_I from C_j(and C_k) to get the shortest distance $d(C_j, V)$ (and $d(C_k, V)$), to each other node V. Then we sort the vertices by $d(C_j, V) + d(C_k, V)$, the length of the shortest path under the $RenderingCost$ from C_j to C_k, constrained to pass through V, and keep the ones with the K (set to 50) smallest values. These vertices and the edges between them form a pruned graph $\tilde{G}_I(j,k)$.Figures 2a and 2b illustrate the reduction in graph size due to pruning for the Colosseum data set, while keeping the most promising edges.

After pruning, we include the $SmoothnessCost$ for the pairs of edges in $\tilde{G}_I(j,k)$ and compute shortest paths from C_j to C_k (after constructing the line graph as before). We perform the pruning and shortest path computations in parallel for each pair of canonical views.

3.2. Canonical Graph

After computing the shortest paths between pairs of canonical views, we solve for an ordering of the canonical views as though the tour were to take these shortest paths. To do so, we construct the canonical graph G_C, the complete graph over the canonical views, with the cost of the edge (C_j, C_k) between each pair of images in G_C set to the cost of the shortest path $(C_j \rightarrow C_k)$ between these images in $\tilde{G}_I(j,k)$, as shown in Figure 3. The problem now reduces to solving the traveling salesman problem(TSP) in G_C, to get an ordering Π. We now use an approximation algorithm Christofides [4] to solve our (metric)TSP.

3.3. Computing the Final Tour

At this stage, we could concatenate the shortest paths between canonical views according to the ordering Π, but the lack of smoothness across the canonical views can lead to jerky camera motions when transitioning through them. To avoid this, we first splice together the pruned sub-graphs $\tilde{G}(\Pi_1, \Pi_2), \tilde{G}(\Pi_2, \Pi_3), \cdots, \tilde{G}(\Pi_{|C|-1}, \Pi_{|C|})$ to form the graph G_Π. These graphs are spliced so that only the canonical views are in common; e.g., graphs $\tilde{G}(\Pi_1, \Pi_2)$ and $\tilde{G}(\Pi_2, \Pi_3)$ are spliced to meet exactly at C_{Π_2}. Any other nodes or edges that the sub-graphs have in common are duplicated rather than identified with each other across the sub-graphs when constructing G_Π. We can then construct a higher order graph from G_Π and use Dijkstra's algorithm to solve for the the shortest path from C_{Π_1} to $C_{\Pi_{|C|}}$, following

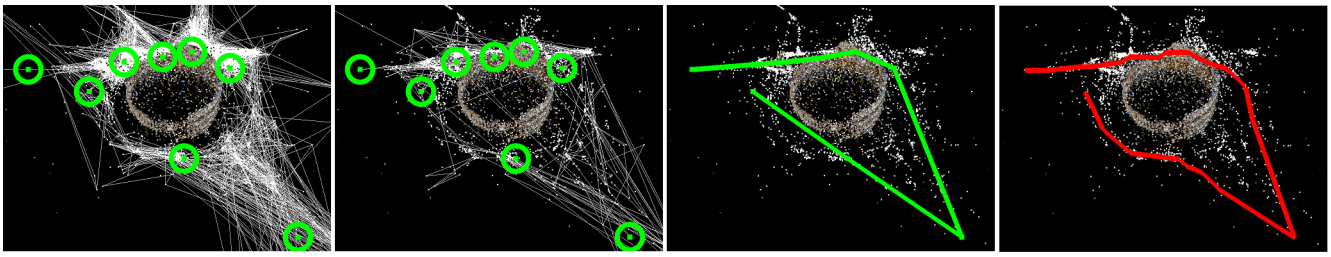

Figure 2. For the Colosseum (a) shows the point cloud overlayed with the unpruned G_I and canonical views highlighted as green circles. (b) shows the (overlaid) set of pruned graphs $\tilde{G}_I(j,k)$. (c) shows the ordered canonical views. (d) shows the final computed tour through the camera centers in red.

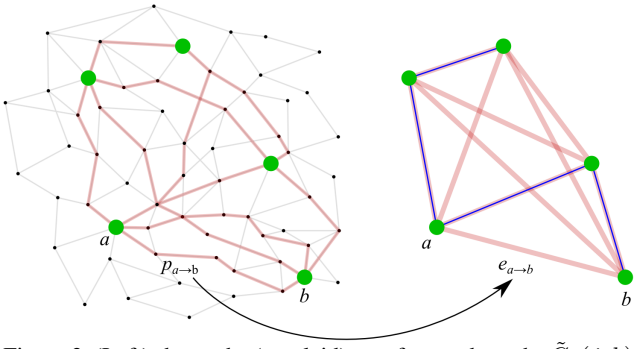

Figure 3. (Left) shows the (overlaid) set of pruned graphs $\tilde{G}_I(j,k)$ with the selected canonical view nodes in green and the shortest paths between each pair displayed in red. (Right) shows the construction of G_C with the blue line showing the optimal ordering of the nodes.

the approach described at the beginning of Section 3.1. By construction, this path – the final tour – will pass through all of the canonical views, and be smooth across the canonical nodes too. Figures 2c and 2d illustrate the path through the canonical graph and the tour for one data set (Colosseum).

4. Depth-map Reconstruction

We employ per-photo depth maps to serve as geometric proxies for image-based rendering. At one extreme, we could just use planar proxies, but this can lead to significant ghosting artifacts [19, 18]. At the other extreme, detailed depth maps reduce ghosting, but often significant outliers lead to other artifacts (e.g., stretching). Instead, we want to capture details where they are reliable, and fall back on (smooth) planar-proxy-like models elsewhere. We accomplish this by estimating partial depth maps which are then completed using a Markov Random Field (MRF) that down-weights low-confidence data. As these depth maps will be relatively smooth, we compute them at a fairly low resolution, $20,000$ pixels per depth map, while preserving the aspect ratio.

Our depth-map reconstruction algorithm, building on existing techniques, consists of three steps. First, given a reference image I, we follow Goesele et al. [12] and automatically select k neighboring images (k = 16) well-suited to recovering a depth map for I. Second, we run Furukawa's patch-based multi-view stereo software (PMVS) [10] on the k images to generate a semi-dense point cloud. We retain only the points X that were computed using I, as well as the confidence $\beta(x) \in [-1,1]$ assigned to each point x. Third, we formulate an MRF objective and solve for the depth d_p at each pixel p that minimizes

$$\sum_p w(p)\rho_d\left(\frac{|d_p - \overline{d_p}|}{\sigma_I}\right) + \lambda \sum_{p,q\in\mathbf{N}} \rho_s\left(\frac{|d_p - d_q|}{\sigma_I}\right). \quad (13)$$

The data cost (first term) penalizes the discrepancy between d_p and the expected depth $\overline{d_p}$, set to the average depth of the points in X that project within the extent of pixel p (call this subset $X(p)$). We use a robust norm – the Cauchy function $\rho_d(a) = \ln(1 + a^2)$ – to handle outliers. The penalty for each pixel is also scaled by a confidence $w(p)$, where

$$w(p) = \frac{1}{W}\sum_{x\in X(p)} \max(\beta(x) - \beta_{\min}, 0). \quad (14)$$

The numerator is the sum of $\beta(x)$, where $\beta_{\min}$ $(= 0.7)$ is subtracted from each score, so that only 3D points with very high confidence values can contribute. If no point projects to the pixel p, then its confidence $w(p) = 0$. We normalize $w(p)$ to lie in the range $[0,1]$ by dividing by $W = (1 - \beta_{\min})|X(p)|$, which is the upper bound on the value of the numerator, assuming a fully dense PMVS reconstruction with maximum per-point confidence.

The smoothness cost encourages neighboring pixels (defined by a four-connected neighborhood $\mathbf{N}$) to have similar depths. In this case, where outliers are not a concern (as they were in the data term), we apply the less robust Huber loss function ρ_s [14] to the absolute depth difference between neighboring pixels. λ is a relative scaling factor for the smoothness term, set to 2.0 in all of our the experiments.

The Cauchy and Huber norms, which we apply to depth differences, are designed to operate on normalized, unit-

less quantities. We compute a per-image normalization constant σ_I which depends on pixel spacing and observed scene depths. Specifically, let d' be the average depth of MVS points visible in I, then σ_I is calculated as the diameter of a sphere at depth d', whose projected diameter in I equals the depth-map pixel spacing.

5. Image-based Rendering

Given a sequence of photos along with the underlying 3D geometry, we use a standard view-dependent texture mapping technique to project the images onto their respective geometric proxies, render them to a virtual viewpoint, and blend the images to form a composite frame. As demonstrated by prior image-based rendering systems, this approach works fairly well and yields compelling viewing experiences [6, 9]. However, it is not free from rendering artifacts, particularly ghosting artifacts where the geometry is imprecise. While a more sophisticated IBR method (e.g., [11]) could be used to achieve further improvements; in this work, we focus on optimizing the camera path, rather than the IBR technique.

A key observation is that rendering artifacts are minimal when the camera is close to an input photo location. Hence, the visualization should focus most time near input camera positions, speeding up during transition between input camera viewpoints. To generate appealing sequences near an input viewpoint, we take inspiration from the work of Zheng et al. [21], which adds parallax effects to create smooth cinematic camera motion across images. This 3D version of the Ken-Burns effect, a popular technique in 2D film production in which the viewer gently pans and/or zooms across a single image, provides a visually appealing experience. While the authors in [21] treat each set in isolation to create a parallax effect for a single view, our goal is to create a coherent path through several views; i.e., the Ken-Burns effect must fit within the context of flying through a photo collection.

In the remainder of this section, we describe how the camera motion is interpolated and parameterized.

5.1. Camera Path Interpolation

We move a virtual camera that starts from one input viewpoint and interpolates smoothly towards the next viewpoint on our computed tour. At any point, the virtual viewpoint can be defined by interpolating each camera parameter independently with Monotonic Piecewise Cubic Interpolation [8], which guarantees monotonicity between adjacent sample points and prevents overshooting or oscillations. However, simply interpolating the camera center and rotation parameters independently guarantees monotonicity in the camera parameter space, but results in non-monotonic motions in screen space. To remedy this, we compute an *interest point* x_j for each image, as a point along the optical axis, at an a *interest distance* d_j equal to the average depth of the associated depth map. We then interpolate both the interest point and distance, to compute the optical center at any point on the curve, along with the other parameters camera yaw, pitch and field of view. Camera roll is set to 0 throughout.

Given a camera trajectory passing through a sequence of photos $\{P_1, P_2, \cdots P_{|P|}\}$, let S denote the parameterization of the trajectory curve, where $S = i$ when the camera is at P_i. The curve is divided into two types of segments: 1) Ken-Burns segments $\{[S_i^s, S_i^e]\}$ around each photo P_i, where rendering artifacts are minimal and the camera moves slowly and spends time D_i^{KB} in traversing it; and 2) transition segments $\{[S_i^e, S_{i+1}^s]\}$, where the camera moves from one photo to the next quickly in time D^{trans}. D^{trans} is set to a constant (0.7 seconds), while D_i^{KB} is different for each image, to allow the optical flow speed inside the rendering window to be roughly constant.

Minimizing Ken Burns Artifacts

The parameters S_i^s, S_i^e, and hence the Ken-Burns segments are computed to highlight the details of the photos. During these segments we want to avoid showing the boundary of the image and to minimize artifacts due to stretching.

As a pre-process, we narrow the field of view of every virtual camera by a factor of 0.8, to avoid seeing image boundaries during Ken-Burns panning. For each image P_i, we then set the Ken-Burns segment parameters $S_i^s \in [i - 0.3, i]$ and $S_i^e \in [i, i + 0.3]$, in discretizations of 0.01, to be the minimum and maximum parameter values, such that each parameter sample in $[S_i^s, S_i^e]$ passes a screen coverage and stretchiness tests. The screen coverage test simply requires that the resulting rendered image fills the screen. The stretchiness test requires that the majority of the depthmap edges (at least 99.7 percent) do not stretch more than a constant threshold (1.8) at the virtual viewpoint.

Estimating Durations

We want the optical flow in each Ken-Burns segment to be roughly the same and compute an initial estimate of the duration $D_i^{KB} = D \cdot m_i$. Here, m_i is the average optical motion between the virtual camera at S_i^s and S_i^e based on the motion of the depthmap grid points associated with P_i, where D is chosen so that the average value of D_i^{KB} becomes 2.0 seconds. Lastly, we clamp each D_i^{KB} to be the range [0.7, 2.7] (in seconds) to ensure no segment is too short or too long.

Timing

Given the duration of every segment, i.e., $\{D_i^{KB}\}$ and D^{trans}, it is straightforward to compute times T_i^s and T_i^e

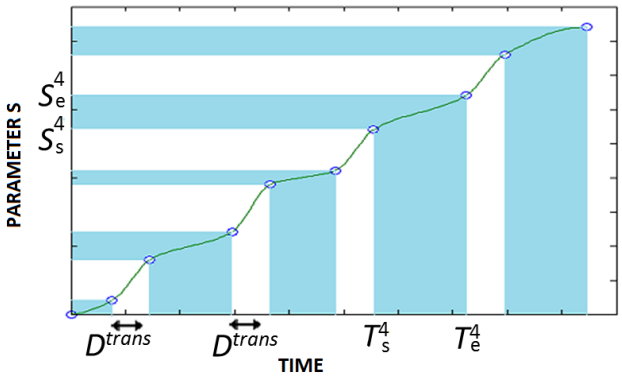

Figure 4. A tour is partitioned into a series of Ken-Burns segments (blue intervals) and transition segments (white intervals). The curve shows the mapping between the timing (x-axis) and the parameterization (y-axis) of a tour. Transition segments are fixed-length D^{trans} in time, while the lengths of Ken-Burns segments vary both in time and parameterization.

for each segment boundary S_i^s and S_i^e, respectively. We can then interpolate these times to give a complete mapping between S and time, again using Monotonic Piecewise Cubic Interpolation to avoid abrupt changes in speed as illustrated in Figure 4.

5.2. Rendering

For the rendering in a transition segment $[S_i^e, S_{i+1}^s]$, we simply render P_i and P_{i+1} into the virtual view and blend with a uniform weight per pixel. The blend weights are set to the fractional distances between images, where distance is measured in parameter space S.

During the Ken-Burns segment $[S_i^s, S_i^e]$ we warp a single image P_i, to render the photo tour.

6. Deployment within Google Maps

We obtained 3.2 million geo-tagged photos from Panaromio/Flickr. Based on the geo-tags, the photos were clustered into individual tourist sites. Each site was reconstructed using a structure-from-motion technique similar to [16], and placed on the map using a RANSAC method similar to [15]. These reconstructions were performed in parallel, over a cluster with 1000 CPUs, and took about 24 hours, consistent with the Rome-in-a-day projects [1, 7] Our depth-map pipeline required 6 hours and 10 minutes on the same cluster.

About 2.5 million images were successfully reconstructed by our structure-from-motion and stereo algorithms. These images were then processed by our tour generation algorithm to produce 20,857 photo tours of popular landmarks all over the world. (See Figure 1.) Generating these photo tours took 3.5 hours on the same cluster. Figure 6 shows a sampling of frames from two of our photo tours.

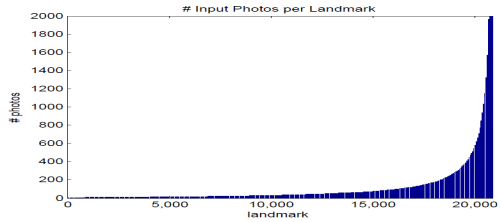

Figure 5. Size of each landmark SfM reconstruction, sorted by number of photos. Note the long-tail distribution: while most landmarks have less than a hundred photos, the top 2% have over a thousand.

We strongly encourage readers to view the supplementary video to fully evaluate our results.

Figure 5 shows the size of the reconstructions of the landmarks, measured in terms of the number of photos. Note the long-tail distribution: the most popular 2% of landmarks contain between 1000 and 25000 images. The remaining average less than 100 images. The landmarks on average have 3.2 canonical views (and 6 images), while the largest landmark has 16 views (and over 40 images).

6.1. Web-based Viewing

Our goal is to stream and render photo tours to any viewer over the Internet via their web browser. We implemented the viewer as a javascript web application running on any browser that supports the HTML5 WebGL spec (Chrome, Firefox, Safari, etc.). The images, depth maps, and sequencing information is stored on a server and streamed real-time to the client. We prefer this to pre-recording movies, primarily to keep the option of interactivity, if needed. Secondly, the typical bandwidth required to play a tour of 10 views for a 1024x768 viewport is less than 3MB, which works out to 115kbits/sec, while for comparison, Youtube's standard video setting is 250kb/s and Netflix starts at 625kb/s, so again photo tours are significantly more efficient than streaming video. Most modern graphics cards have no problem keeping up; the application is able to render a tour at 30 fps on a GeForce 9400M and 60 fps on a Quadro FX 580.

6.2. Discussion

As seen in the accompanying videos, this approach yields high quality results for a broad range of scenes. Overall since launch, we have found that of all the tours rated by users, 90% of our photo tours got a thumbs up rating. As with any "at-scale" system, however, there are failure cases. We now enumerate the types of failure cases we've observed (can also be seen in the accompanying video). The first category of failure cases is due to *data problems*, i.e., problematic photos, pose, or geometry. At times due to noisy or sparse 3d data, the depth maps can be stretchy and this causes problems in the final renderings, eg the Fountain

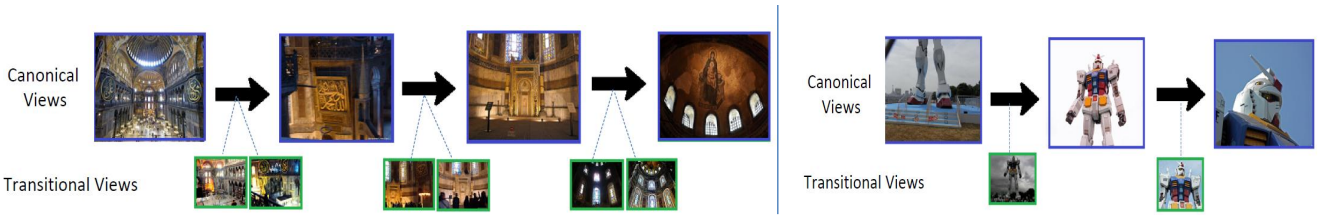

Figure 6. Sample photo tours, displayed as a sequence of images (better viewed in accompanying video). Left: Hagia Sophia, Istanbul, Right: Gundam, Tokyo. The sequence of canonical images are outlined in blue, whereas the transitional views are outlined in green.

of Neptune sequence. Errors in depth maps can also occur for thin objects (e.g., spans on bridges), reflective surfaces (water, windows), and textureless or partially occluded regions; such errors can produce warping artifacts. A second category of failure cases is due to *tour problems*, i.e., shortcomings in our graph traversal approach. Some tours are too short and do not provide much detail eg the Cathedral of Santa-Maria photo tour, often due to a scarcity of photos. Other tours are too long; a good example is the Colosseum tour in Rome which does a 360 rotation around the outside of the Colosseum, which can get tedious. As these are community photo collections, some of the photos have people occluding the main attraction, which can take away from the experience. One example of a bad photo can be seen in the Ponte-Mezzo photo-tour, where a child appears in one of the transitions.

In addition to addressing these challenges, an interesting direction of future work is to add textual annotations and image captions, providing context on what's being depicted. For example, the visualization could identify "Raphael's Tomb" inside the Pantheon, and other significant scene elements, by comparing the photographs, with others annotated on the web. This would provide for a more holistic experience.

Acknowledgements: This work was supported in part by National Science Foundation grants IIS-0811878 and IIS- 0963657, the University of Washington Animation Research Labs, and Google.

References

[1] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski. Building rome in a day. In *ICCV 09*, pages 72–79.

[2] D. Applegate, R. Bixby, V. Chvtal, and W.J.Cook. The traveling salesman problem: A computational study. In *Princeton University Press*, 2006.

[3] S. E. Chen and L. Williams. View interpolation for image synthesis. In *SIGGRAPH '93*, pages 279–288, 1993.

[4] N. Christofides. Worst-case analysis of a new heurestic for the travelling salesman problem. In *Techincal Report 338, Grad School of Industrial Administartion, CMU*, 1976.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms.

[6] P. Debevec, C. Taylor, and J. Malik. Modelling and rendering architecture from photographs : A hybrid geometry and image based approach. In *SIGGRAPH '96*.

[7] J.-M. Frahm, P. Fite-Georgel, D. Gallup, T. Johnson, R. Raguram, C. Wu, Y.-H. Jen, E. Dunn, B. Clipp, S. Lazebnik, and M. Pollefeys. Building rome on a cloudless day. In *ECCV'10*.

[8] C. R. E. Fritsch F. N. Montone piecewise cubic interpolation. In *SIAM J, Vol. 17*, pages 238–246, 1980.

[9] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski. Reconstructing building interiors from images. In *ICCV 2009*, pages 80–87, 2009.

[10] Y. Furukawa and J. Ponce. Accurate, dense, and robust multiview stereopsis. *PAMI 2010*, 32(8):1362–1376.

[11] M. Goesele, J. Ackermann, S. Fuhrmann, C. Haubold, R. Klowsky, D. Steedly, and R. Szeliski. Ambient point clouds for view interpolation. In *SIGGRAPH 2010*.

[12] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S. M. Seitz. Multi-view stereo for community photo collections. In *ICCV 2007*.

[13] S. Guha and S. Khullar. Connected facility location problems. In *DIMACS Workshop on Network Design*, 97.

[14] P. J. Huber. Robust estimation of a location parameter. In *Annals of Statistics 53*, pages 73–101, 1964.

[15] R. Kaminsky, N. Snavely, S. M. Seitz, and R. Szeliski. Alignment of 3d point clouds to overhead images. In *Workshop on Internet Vision, CVPR'09*.

[16] M. A. Lourakis and A. Argyros. SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Trans. Math. Software*, 36(1):1–30, 2009.

[17] I. Simon, N. Snavely, and S. M. Seitz. Scene summarization for online image collections. In *ICCV 07*, pages 1–8.

[18] N. Snavely, R. Garg, S. M. Seitz, and R. Szeliski. Finding paths through the world's photos.

[19] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH 2006*.

[20] S. Winter. Modeling costs of turns in route planning. In *GeoInformatica*, volume 6, pages 363–380, 2002.

[21] K. C. Zheng, A. Colburn, A. Agarwala, M. Agrawala, D. Salesin, B. Curless, and M. F. Cohen. Parallax photography: creating 3d cinematic effects from stills. In *Graphics Interface 09*.