Aerial Reconstructions via Probabilistic Data Fusion

Supplemental Material

Randi Cabezas Oren Freifeld Guy Rosman John W. Fisher III
Massachusetts Institute of Technology
{rcabezas,freifeld,rosman,fisher}@csail.mit.edu

April 29, 2014

Abstract

This document is complementary to [2] and contains additional results, derivations and implementation details. It is identical to the supplemental material submitted to Computer Vision and Pattern Recognition (CVPR) on April 2014.

Contents

1 Introduction

This document supplements the paper *Aerial Reconstructions via Probabilistic Data Fusion* [2] (we henceforth refer to it as 'the paper'). It provides additional results, mathematical derivations and explanations, as well as implementation details. As such, it is not a standalone document and it assumes the reader is familiar with [2]. This document is structured as follows. Section 2 provides a brief overview of the data sources used in this work. Section 3 provides additional reconstruction comparisons that were omitted from the paper due to space constraints. These comparisons include vision-only Structure from Motion (SfM) comparisons on the *CLIF Intersection* and *CLIF Stadium Image Stack* datasets using both points (Sec. 3.1) and triangulated surfaces (Sec. 3.3). Comparisons to LiDAR-only work [7, 12] are presented in Sec. 3.4. Technical mathematical details are provided in Sec. 4, including the Lie-algebraic representation of geometric primitives, derivations of appearance updates, and use of a *Gaussian Process* to model mobile camera location. Implementation details are provided on Sec. 5, including pseudo-code algorithms for CPU and GPU implementation. For videos and source code please see the project page at: http://people.csail.mit.edu/rcabezas/.

1.1 Experiment Parameters

This section briefly outlines the parameter values used to produce the results presented both in [2] and in this document, unless stated otherwise. The image noise model is assumed to be an isotropic independent-and-identically-distributed (iid) zero-mean Gaussian with standard deviation 10 (assuming each color channel takes values in $[0, 255]$). The appearance prior model is iid Gaussian with mean 128 and standard deviation 15. The LiDAR noise model is iid Gaussian with zero mean and 12 [cm] standard deviation. The canonical appearance for each primitive was set to be 16×16 pixels. The GPS position noise parameters are modeled as iid Gaussian with zero mean and with a standard deviation of 20 [ft]; the orientation is modeled as iid Gaussian with zero mean and 5° standard deviation. Throughout the work, each observation pixel was allowed to influence its corresponding latent appearance pixel as well as its 4 neighbors.

2 Data Overview

We now describe the data sources used in this work: the Lubbock and CLIF datasets. For more details on these datasets, see [1].

2.1 Lubbock Dataset

The Lubbock Dataset (Fig. 1) consists of three images of Lubbock, Texas. The dimension of each of these images (Fig. 2) is 1336×891 pixels. The dataset also contains over 5 million LiDAR returns in a single tile (Fig. 2) of dimension 1000×1000 [m^2]. The LiDAR density is about one return per meter squared with vertical resolution of 10 [cm].

2.2 CLIF 2007 Dataset

The Columbus Large Image Format (CLIF) 2007 sample dataset [11] is used throughout this work. It consists of 50 frames, each frame contains 6 cameras, Fig. 3. Each frame is originally 2672×4016 pixels, and in this work was downsampled to 822×1326. The area covered in one frame is approximately 1700×2200 [ft^2], the approximated area of the sample set is 3500×3800 [ft^2] (*i.e.*, visible across all the frames in the entire sequence). LiDAR for the Ohio State area was obtained from [8]. The tile containing the stadium and surrounding area has over 727,000 returns. The return density is approximately one in 3×3 [ft^2]. An overview of the LiDAR tile is depicted in Fig. 3.

We focus on specific sites in the CLIF dataset, the stadium image stack (seen in top center image in Fig. 3), a cropped version of the stadium and a cropped version of the intersection (bottom center image in Fig. 3). We will discuss these three scenes next.

CLIF Stadium Image Stack consists of 49 images, one for each frame of camera one (Fig. 3, top center) from the CLIF dataset. One image was excluded as it was corrupted at collection time. An overview of the camera positions (estimated) as well as some sample images are shown in Fig. 4. This dataset uses the LiDAR tile described earlier.

CLIF Stadium Only dataset consists of a cropped version of the original CLIF frames (not downsampled). Each image in this dataset is 1024×768 pixels. Four sample images are shown in Fig. 5. This dataset uses the LiDAR tile described earlier.

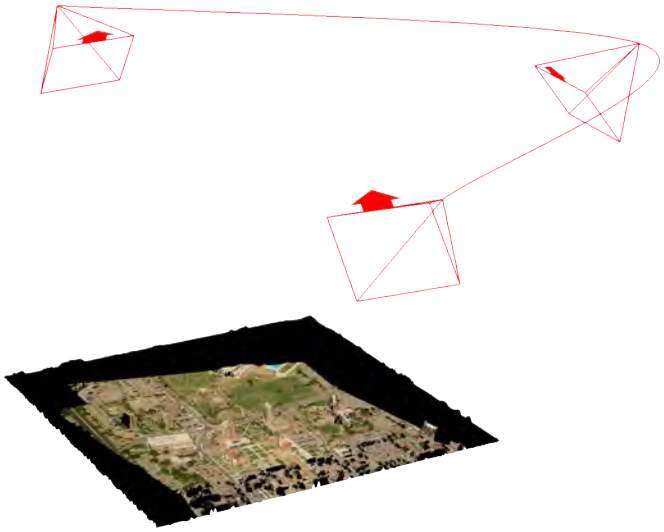

Figure 1: Overview of Lubbock dataset. Ground Truth Camera locations shown in red.

Figure 2: The three images of the Lubbock dataset and LiDAR (color coded according to height above ground)

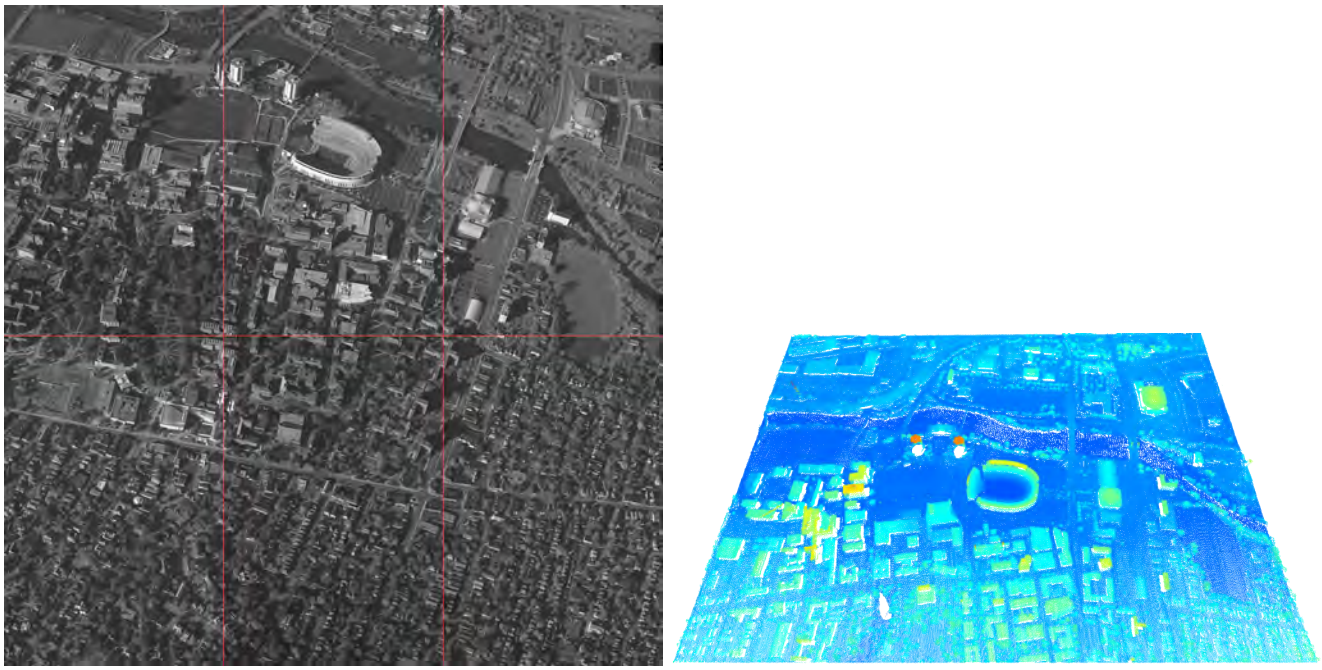

Figure 3: *Left:* Overview of CLIF frame, six cameras. The stadium image stack consists of all the images in the top center camera; the intersection can be seen in the bottom center camera. *Right:* LiDAR overview.

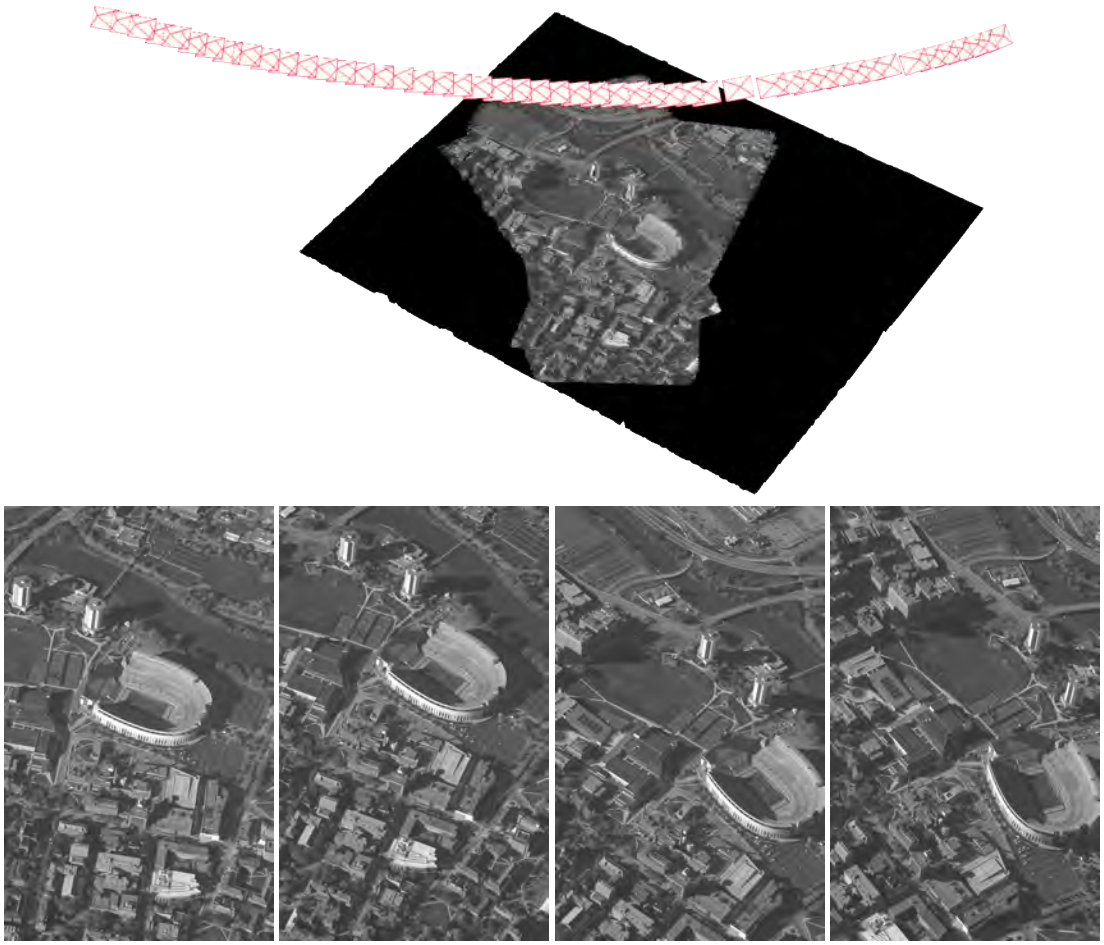

Figure 4: *Top:* Overview of CLIF Image Stack dataset, 49 images (estimated camera location shown in red). *Bottom:* Four sample images of the CLIF Image Stack dataset (left to right images 0,16,32,48)

Figure 5: Four sample images of the CLIF Stadium Only dataset (left to right images 0,16,32,48).

Figure 6: Three sample images of the CLIF Intersection dataset (left to right: images 0,14,29), as well as LiDAR tile.

Scene	Image		KeyPoint		Bundler	PMVS2	Total
	Number	Size	Detection	Matching			
Intersection	45	441x374	47	80	294	68	489
Stadium Stack	49	822x1326	461	1,414	909	330	3,114
Multi-Camera	100	660x1024	645	3,203	4,252	528	8,628

Table 1: Bundler+PMVS2 time breakdown (all times are in [sec]). Runtimes obtained using 2.8GHz i7 CPU with 24GB RAM.

Scene	Primitives	Camera Pose			Geometry			Appearance	Total
	Visible/Total	Time/Img	# Iter	Time	Time/Iter	Prim./Iter	Time	Time	
Intersection	3.2k/21k	0.977	20	904	7.3	24	1,359	0.262	2,263
Stadium Images	89k/227k	11.3	20	11,074	41	250	15,631	1.04	26,705
Multi-Camera	279k/479k	23.4	15	35,100	86	250	99,238	3.03	134,338

Table 2: Our time breakdown (all times are in [sec]; iterations in camera pose refer to updating each camera once; iterations in geometry refer to updating a set of primitives once). Runtimes obtained using 2.8GHz i7 CPU with 24GB RAM and an NVIDA GTX Titan.

CLIF Intersection dataset consists of 45 cropped portions of images from camera two and zero (bottom left and center in Fig. 3) of the CLIF dataset. Only one image for each frame was used; *i.e.*, the temporal order of the sequence was maintained. Instances where the scene was split between two cameras where discarded. Sample images of the dataset are shown in Fig. 6. A small cropped version (*e.g.*, 20k returns) of the LiDAR tile described earlier was used for this dataset (Fig. 6).

3 Additional Reconstruction Results and Comparisons

This section provides additional reconstructions which showcase the differences between the proposed model and previous work. It begins by comparing the results obtained using the proposed method with vision-only work [4, 10], namely Bundler+PMVS2. Next, the results of the proposed method are compared with the Bundler+PMVS2+PoissonRecon pipeline [5], where this last step is introduced to obtain surface reconstructions from the point cloud obtained via SfM. This section concludes in providing comparisons with LiDAR-only work of [7, 12] and additional photo-realistic reconstructions obtained using the proposed method.

3.1 Additional SfM Comparisons

Additional qualitative reconstruction comparisons for the *Intersection*, *Stadium only*, and *Stadium Image Stack* datasets are shown in this section. Note that we were not able to compare the reconstructions of the *Lubbock* scene since SfM fails to produce a reconstruction. We hypothesize that SfM cannot produce reconstruction for this scene due the small number of images used and the wide baseline between the images.

Comparisons between the reconstructions provided by the proposed method and Bundlers+PMVS2 are shown in Fig. 7-9. These comparisons highlight the main differences between the proposed method and traditional SfM; *i.e.*, the use of higher-order primitives and images to represent the scene provides a dense reconstruction (as opposed to a sparse point cloud). Moreover, small scene details can be easily seen from the reconstructions obtained using the proposed method, leading to more visually-appealing reconstructions.

3.2 SfM Computation Time

The runtime of Bundler+PMVS2 is shown in Table 1. The table shows that anywhere between 25% to 60% of the time, Bundler+PMVS2 is detecting or matching keypoints. The scene parameter optimization takes the remaining portion of the time. Overall, the computation time for each of the scenes is quite low. This is unsurprising as SfM implementations are highly efficient since they must scale well to support the large number of images needed to accurately reconstruct a scene.

The runtime for the proposed model is shown in Table 2. The table shows that the algorithm spends a significant amount of time computing scene geometry. These high run-times are mostly due to the large number of visible primitives in the scene.

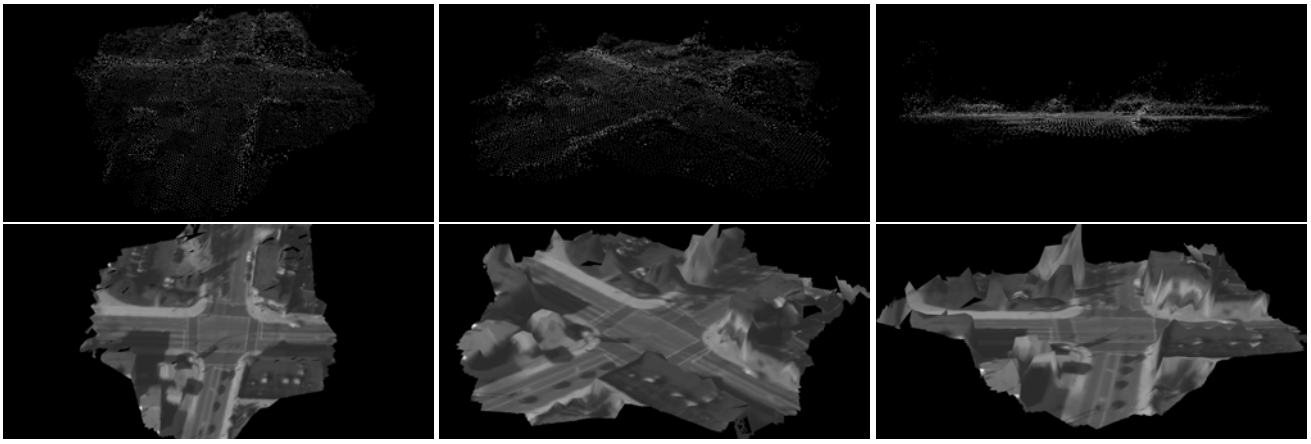

Figure 7: *Top:* Bundler+PMVS2 reconstruction of Intersection, 3 views (12k points). *Bottom:* Proposed model.

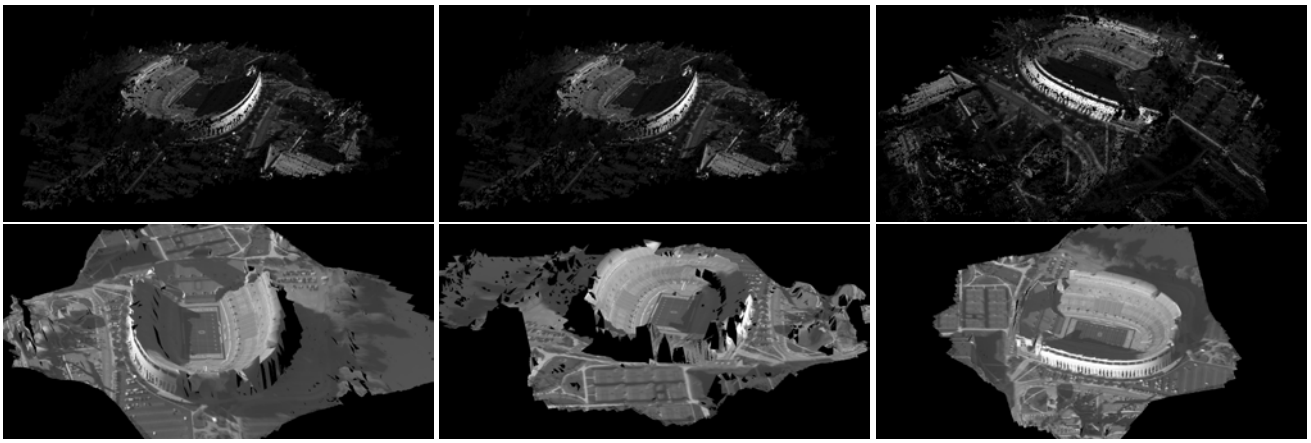

Figure 8: *Top:* Bundler+PMVS reconstruction of CLIF Stadium Only (3 views). Note that the reconstruction is inverted (156k points). *Bottom:* Proposed model.

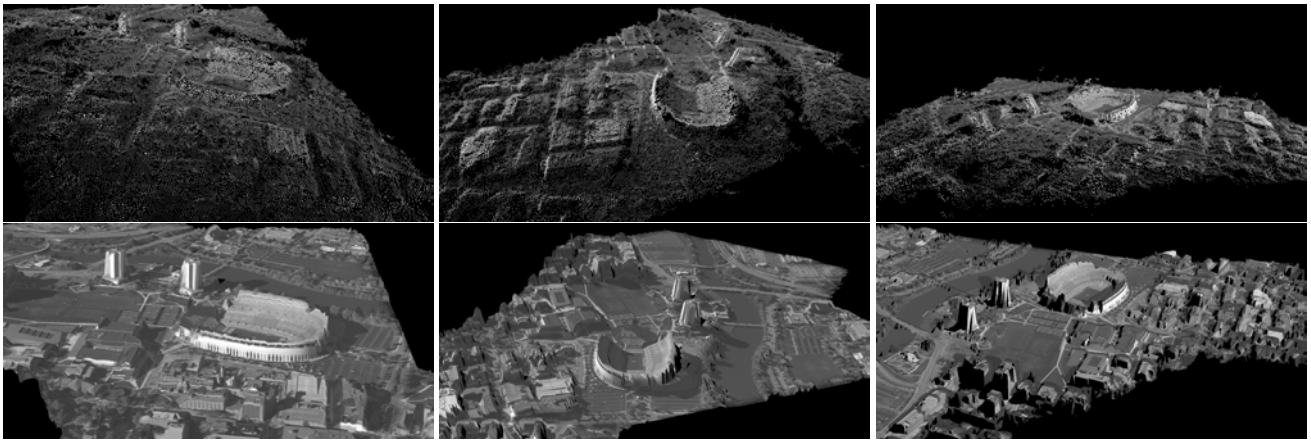

Figure 9: *Top:* PMVS reconstruction of CLIF Image Stack, 3 views. (77k points). *Bottom:* Proposed model.

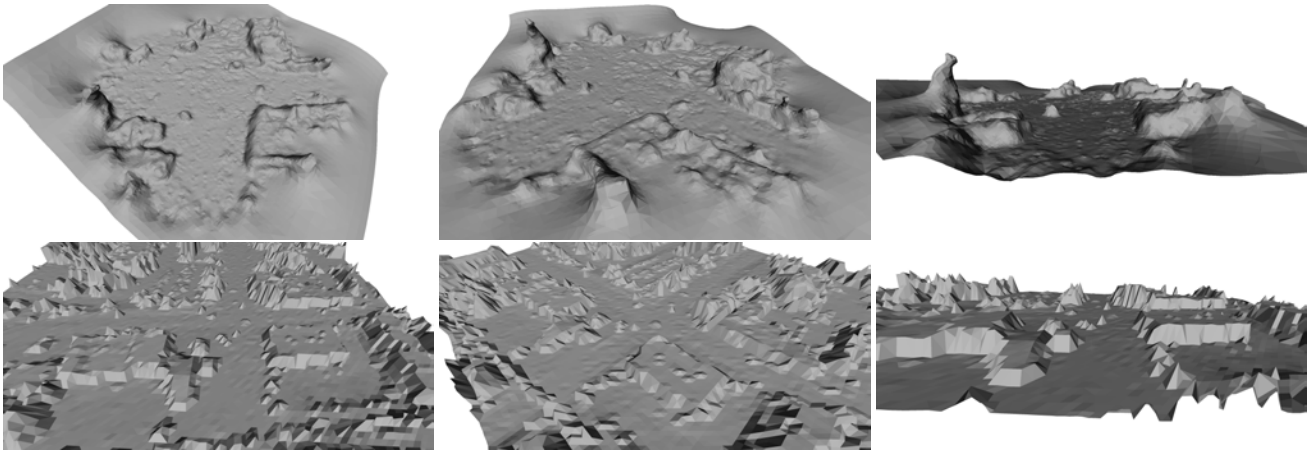

Figure 10: *Top:* Bundler+PMVS2+PoissonRecon reconstruction of Intersection, 3 views. *Bottom:* Proposed model.

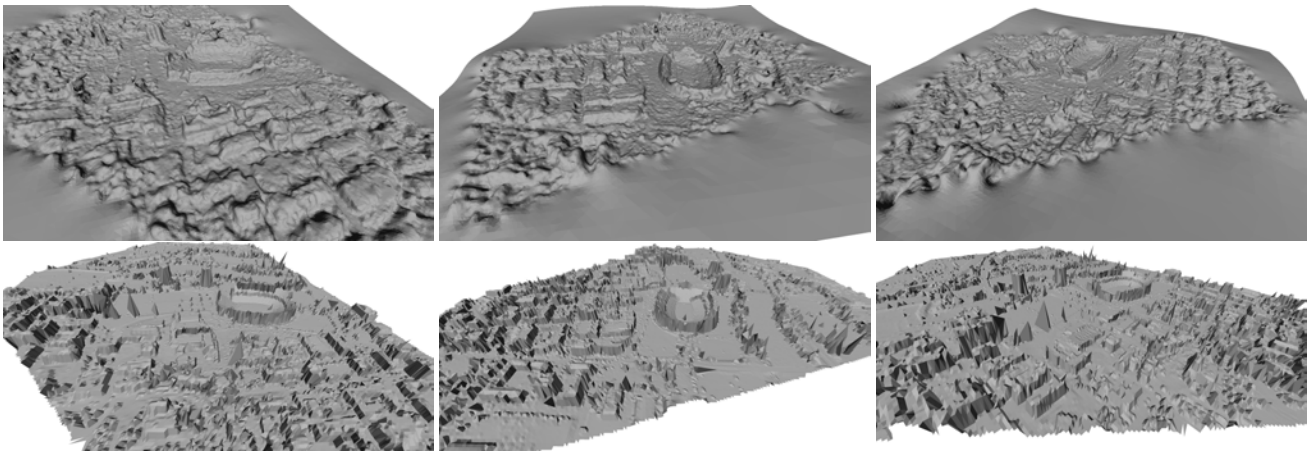

Figure 11: *Top:* Bundler+PMVS2+PoissonRecon reconstruction of Stadium Stack, 3 views. *Bottom:* Proposed model.

We further note that on average the time per iteration for a single plane is under half a second. In order to reduce computation time a closer look at the rendering pipeline is required.

3.3 Surface Comparisons

The results of the proposed method were also compared with the surfaces generated by the Bundler+PMVS2+PoissonRecon [4, 5, 10] pipeline. Adding PoissonRecon to the processing pipeline produces a watertight surface from the oriented points generated by PMVS2. The results are shown in Fig. 10 and Fig. 11. The figures highlight the benefits of the proposed approach over the multi-step SfM pipeline; *e.g.*, both horizontal and vertical surfaces are highly planar in the reconstructions obtained using the proposed work as opposed to overly smooth as produced using the multi-stage SfM pipeline.

3.4 LiDAR-Only Comparisons

The results of the proposed model are compared with the LiDAR-only work of [7, 12]. Note that [7] does not attempt to recover geometry, that work focuses on registering images and LiDAR; however, it introduces the concept of triangulation of LiDAR at the ground level and uses it to demonstrate their registration algorithm. This method of producing a watertight mesh is simple and provides suitable models.

The results of the comparisons are shown in Fig. 12, where the proposed method (with and without the Lie-algebraic representation) is compared with [7] and [12]. When compared with the simple method of [7], our method produces smoother surfaces with less jagged edges. This is expected since the method in [7] does not account for noise in the measurements and

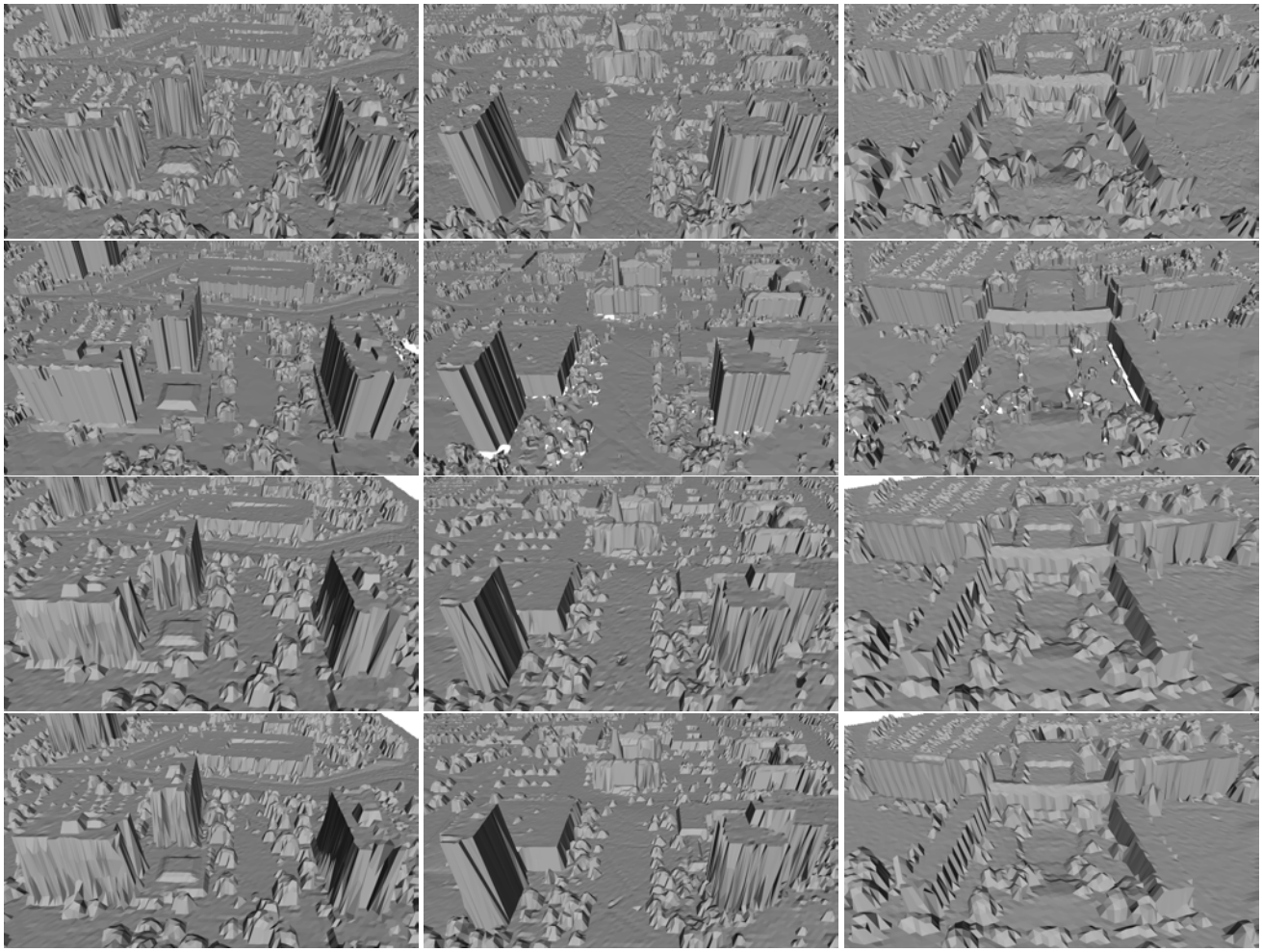

Figure 12: Comparison to LiDAR-only work for the Lubbock dataset, 3 different views (columns). *Algorithms (top-bottom)*: [7], [12], Proposed Method (using lie-algebraic representation), Proposed Method (no lie-algebraic representation).

fits the surface to every LiDAR point. The reconstructions produced by the proposed method are more similar to [12], where planar surfaces are fairly smooth yet allowing sharp transitions between ground and roofs.

Note that some of the building sides produced by [12] are qualitatively better than those produced with the proposed method (*e.g.*, last column of Fig. 12); however, the proposed method can be easily improved to handle these cases by introducing better scene priors, *e.g.*, an explicit assumption of smoothness will regularize the building sides and produce satisfactory results.

3.5 Additional Reconstruction Results

Figure 13 shows additional reconstruction views of the *Stadium-Only* dataset. Each column in the figure corresponds to a reconstructed view (*top*) as seen from a given observation view (*bottom*). The figure highlights the fact that the proposed method produces photo-realistic reconstructions where even fine scene details, (*e.g.*, parking-lot lines and marking in the football field) can be clearly seen.

4 Mathematical Details

This section contains important mathematical details and derivations including the Lie-algebraic representation, the derivation of the appearance computation and our use of the Gaussian Process.

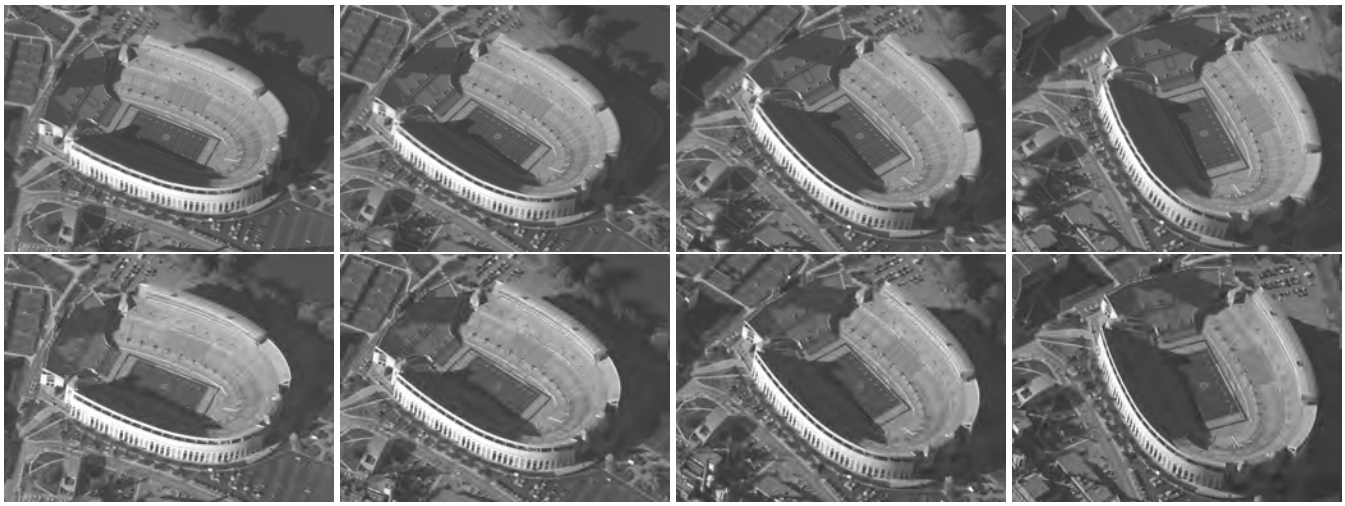

Figure 13: Stadium Reconstruction. *Top:* reconstructions; *Bottom:* original images (images: 0,16,32,48).

4.1 Lie Algebraic Representation of Primitives

This section details the Lie algebraic representation of the primitives used in this work (*i.e.*, triangles). As mentioned in the paper this representation has several advantages over naive vertex representation. It allows us to work in a linear space while at the same time provides a convenient way to decompose primitive deformations into rotation, scale and skew components; a fact that we exploit in our inference algorithms.

A triangle may be parametrized by its vertices using 9 degrees-of-freedom (DOF); viewing it as the result of a transformation applied to some reference triangle leads to another 9-DOF parametrization: 3 for global translation and 6 for triangle-to-triangle deformation. Such a 6-DOF deformation can be parametrized using a 6-dimensional Lie group [3]. This has several advantages, including a principled notion of a triangle-to-triangle relationship. Importantly, while the group is nonlinear, it has an associated linear space known as the Lie algebra. The Lie-algebraic representation enables us to work on a linear space, using a map from it onto the space of all triangles sharing the same origin; the map is given as follows (see [3] for more details).

Let $[v_0, \widetilde{v_1}, \widetilde{v_2}] \in \mathbb{R}^{3\times 3}$ denote the vertices of a reference triangle. Let $X \stackrel{\text{def}}{=} [v_1, v_2] \stackrel{\text{def}}{=} [\widetilde{v_1} - v_0, \widetilde{v_2} - v_0] \in \mathbb{R}^{3\times 2}$. The pair (v_0, X) fully describes the triangle. If $Q \in \mathbb{R}^{3\times 3}$ is invertible, then (v_0, QX) is a new triangle. In general such a Q has 9 DOF, but we can impose a $6-$DOF structure on it. Let $Q = RR_X^T ASR_X$ such that $R \in \mathrm{SO}(3)$;

$$A = \begin{bmatrix} 1 & U & 0 \\ 0 & V & 0 \\ 0 & 0 & 1 \end{bmatrix} \; ; \; S = \begin{bmatrix} S' & 0 & 0 \\ 0 & S' & 0 \\ 0 & 0 & 1 \end{bmatrix} \; ; \; V = e^v \; ; S' = e^s \tag{1}$$

where U, v and s are reals, and R_X is fully defined by X:

$$R_X = \begin{bmatrix} |v_1| & t & 0 \\ 0 & \sqrt{|v_2|^2 - t^2} & 0 \\ 0 & 0 & |v_1 \times v_2| \end{bmatrix} [v_1, v_2, v_1 \times v_2]^{-1}, \tag{2}$$

where $t = \frac{v_1^T v_2}{|v_1|}$. While Q has only 6 DOF, it can be shown that if $\mathbf{Q}$ is the space of all Q's with this structure, then $\{QX : Q \in \mathbf{Q}\} = \{AX : \det A \neq 0\}$. The nonlinear $\mathbf{Q}$ is isomorphic to a 6-dimensional matrix Lie group of 6×6 matrices. Letting $[\omega_1, \omega_2, \omega_3, u, v, s]$ denote an element of $\mathbb{R}^6$ (viewed as the Lie algebra), a parametrization $Q(\omega_1, \omega_2, \omega_3, u, v, s)$ is given by

$$U = \begin{cases} u, & \text{if } v = 0 \\ \dfrac{u}{v}\left(e^v - 1\right), & \text{otherwise} \end{cases} \tag{3}$$

while $R = R(\omega_x, \omega_y, \omega_x)$ using well-known Rodrigues' formula, and s and v affect A and S as described above.

4.2 Appearance Computation

In this section we derive the update equations for the appearance conditioned on all other parameters.

4.2.1 Observation Model

As discussed in the paper, the image observation model is:

$$I_n^c(u,v) = A_{m^*}(u',v') + Q_n, \tag{4}$$

where $Q_n \sim \mathcal{N}(q;0,r_{m^*}^2)$, I_n^c is the n^{th} image of camera c, A_{m^*} is the m^* triangle appearance, and (u,v) is the projected image coordinates of the appearance at coordinate (u',v'). We note that (u,v) and (u',v'), are related implicitly via the 3D point (x,y,z), or explicitly via the homography between image A_k and I_n. For any pixels (u,v), and (u',v') that are in correspondence, we will denote the observation and underlying latent appearance by $z = I_n(u,v)$ and $a = A_{m^*}(u',v')$.

Using this notation, Eq. (4) can be written as $z = a + q$. It follows that $z|a \sim \mathcal{N}(z;a,r)$, and $z \sim \mathcal{N}(z;\mu,\sigma^2+r^2)$. If we let A have a Normal prior, such that $A \sim \mathcal{N}(a;\mu,\sigma^2)$. So then our task is to estimate A.

4.2.2 Derivation - Multiple Observations - different noise

Assume that observation is the same as before $I_n(u,v) = A_{m^*}(u',v') + Q_n$, but with $Q_n \sim \mathcal{N}(q;0,r_n)$, *i.e.*, the noise variance depends on the image. As before $z = a + q$, but let's denote the different variance explicitly as $z = a + q_n$. Then,

$$p(a|\mathbf{z}) = \frac{\prod_{i=0}^{n-1} p(z_i|a)p(a)}{\prod_{j=1}^{n} p(z_j)} \propto \mathcal{N}(a;\mu;\sigma^2) \prod_{i=0}^{n-1} \mathcal{N}(z_i;a,r_i^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(a-\mu)^2}{2\sigma^2}\right\} \prod_{i=0}^{n-1} \frac{1}{\sqrt{2\pi r_i^2}} \exp\left\{-\frac{(z_i-a)^2}{2r_i^2}\right\}$$

Let us define $\dot{r} = \prod_{j=0}^{n-1} r_j$, and $\dot{r}_{\setminus i} = \prod_{j=0, j\neq i}^{n-1} r_j$ then,

$$p(a|\mathbf{z}) \propto \frac{1}{(2\pi)^{\frac{n+1}{2}}\,\dot{r}\sigma} \exp \underbrace{\left\{-\frac{(a-\mu)^2}{2\sigma^2} - \sum_{i=0}^{n-1} \frac{(z_i-a)^2}{2r_i^2}\right\}}_{C}$$

It can be shown that

$$C = -\frac{\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2}{2\dot{r}^2\sigma^2}\left[(a-\hat{\mu})^2 + c'\right],$$

where

$$\hat{\mu} = \frac{\mu\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2 z_i}{\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2}, \quad \text{and} \quad c' = \frac{\mu^2\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2 z_i^2}{\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2} - \hat{\mu}^2 .$$

Substituting C and performing a small amount of algebra leads to

$$p(a|\mathbf{z}) \propto \mathcal{N}(a;\hat{\mu},\hat{\sigma}^2)$$

where

$$\hat{\mu} = \frac{\mu\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2 z_i}{\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2} = \frac{\mu \prod_{i=0}^{n-1} r_i^2 + \sigma^2 \sum_{i=0}^{n-1} \left(\prod_{j=0, j\neq i}^{n-1} r_j^2\right) z_i}{\prod_{i=0}^{n-1} r_i^2 + \sigma^2 \sum_{i=0}^{n-1} \left(\prod_{j=0, j\neq i}^{n-1} r_j^2\right)} \tag{5}$$

$$\hat{\sigma} = \frac{\dot{r}\sigma}{\sqrt{\dot{r}^2 + \sigma^2 \sum_{i=0}^{n-1} \dot{r}_{\setminus i}^2}} = \frac{\sigma \prod_i^{n-1} r_i}{\sqrt{\prod_{i=0}^{n-1} r_i^2 + \sigma^2 \sum_{i=0}^{n-1} \left(\prod_{j=0, j\neq i}^{n-1} r_j^2\right)}} \tag{6}$$

4.2.3 Special Case

As a sanity check, we double check that the multiple observation with same noise is a special cases of the general update derived earlier. To obtain the multiple observation case, we can let $r_j = r, \forall j \in [0, n-1]$, with this definition, $\dot{r} = r^n$, and $\dot{r}_{\setminus i} = r^{n-1}$. Plugging the new values for $\dot{r}$ and $\dot{r}_{\setminus i}$ into the general updates of equations (5) and (6) and factoring out a common term from numerator and denominator we obtain:

$$\hat{\mu} = \frac{\mu r^2 + \sigma^2 \sum_{i=0}^{n-1} z_i}{r^2 + n\sigma^2} \tag{7}$$

$$\hat{\sigma} = \frac{r\sigma}{\sqrt{r^2 + n\sigma^2}} \tag{8}$$

Thus Eq. (7) and Eq. (8) refer to the multiple observation with same noise model.

4.3 Gaussian Process Prior

Oblique aerial images used in this work are taken from a moving platform. This type of data has certain characteristics that can be incorporated as prior knowledge. For example, we can expect that that plane collecting the measurements follows a smooth trajectory. This implies that the camera collecting the images will also have a smooth position as a function of time (assuming a rigid attachment of camera to plane). We can incorporate this information as a prior in the form of a *Gaussian Process* Prior. For more on Gaussian Processes see [6, 9].

4.3.1 Definition

A *Gaussian Process* is a collection of infinite random variables, any finite number of which have a Gaussian distributions. A Gaussian process (GP) is a generalization of a multivariate Gaussian distribution to infinitely many variables. A GP can be fully specified by a mean function $m(x)$ and a covariance kernel $k(x, x')$, so that $f(x) \sim \mathcal{GP}\left(m(x), k(x, x')\right)$ where x is one-dimensional variable (chosen for simplicity of exposition, GPs can easily be extended to multidimensional case).

In practice we do not need to instantiate the infinite collection of random variables, instead we can focus on a finite collection of them $\mathbf{f} = [f(x_1), f(x_2), \ldots, f(x_n)]^\top$, if we let $m(x) = 0$ for all x, then $\mathbf{f} \sim \mathcal{N}(0, \Sigma)$ where $\Sigma_{ij} = k(x_i, x_j), \forall i, j \in [1, n]$. For many applications $m(x)$ can be made zero and $k(x, x')$ can take the squared exponential function:

$$k(x, x') = \upsilon^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right) \tag{9}$$

The hyper-parameters, υ and ℓ, in the square exponential function, Eq. (9), have specific interpretation. The lengthscale, ℓ, controls the level of variability expected in the input. Larger lengthscales indicate that the function is expected to vary slowly, while shorter lengthscales indicate rapid changes in the function values. The signal variance, υ, defines the vertical scale of variations of a typical function, see Fig. 14 for samples of a GP drawn with different hyper-parameters.

4.3.2 Inference

Typically, GPs are used as priors. In the case when the likelihood is Gaussian, this results in a closed form, Gaussian, posterior. We will present the prototypical predictive case here. Let us assume we have noisy observations of a smooth function $\mathbf{t}_N = \{x_n, y_n\}_{n=1}^N$, the task is to infer the function $f(x)$. Assume you have a prior over functions that is a Gaussian Process, such that $f(x) \sim \mathcal{GP}(0, k(x, x'))$, and that the observation model is $p(y|x, f(x)) = \mathcal{N}(\mathbf{f}, \sigma^2 I)$, where I is the identity matrix and σ^2 represent some suitable noise. Then, the posterior over $f(x)$ is also a GP:

$$f(x)|\mathbf{x}, \mathbf{y} \sim \mathcal{GP}\left(\mu_{post}(x), k_{post}(x, x')\right)$$

where

$$\mu_{pos}(x) = k(x, \mathbf{x})\left[K(\mathbf{x}, \mathbf{x}) + \sigma^2 I\right]^{-1}\mathbf{y} \quad \text{and} \quad k_{post}(x, x') = k(x, x') - k(x, \mathbf{x})\left[K(\mathbf{x}, \mathbf{x}) + \sigma^2 I\right]^{-1} k(\mathbf{x}, x').$$

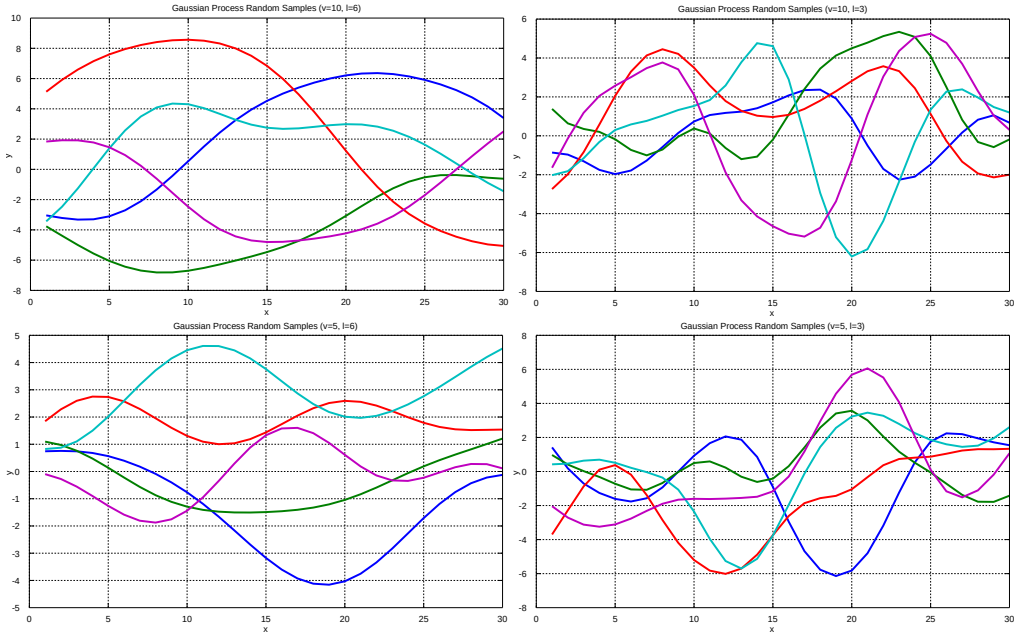

Figure 14: Random Samples from a Gaussian process with different hyperparameters (5 samples per plot). *Left:* column has a large lengthscale, producing smooth samples; *Right:* column has smaller lengthscale resulting in rapidly varying samples. *Top* and *bottom* rows have different signal variances, hence their amplitude ranges vary according.

4.3.3 Hyperparameters

The covariance kernel typically has a few hyper-parameters. For example, the square exponential covariance in Eq. (9), has two hyper-parameters: the lengthscale and the scale variance. These parameters can be manually tuned or learned from the data. An effective way of learning parameters involves looking at the posterior probability of the hyper-parameters $\boldsymbol{\theta}$:

$$p(\boldsymbol{\theta}|\mathbf{x},\mathbf{y},\mathbf{f}) \propto p(\mathbf{y}|\mathbf{x},\mathbf{f},\boldsymbol{\theta})p(\boldsymbol{\theta})$$

where the first term on the RHS is data likelihood and the second is a prior over the parameters. Taking the log of the marginal likelihood, we obtain

$$\log p(\mathbf{y}|\mathbf{x},\mathbf{f},\boldsymbol{\theta}) = -\frac{1}{2}\log|K| - \frac{1}{2}\mathbf{y}^\top K^{-1}\mathbf{y} - \frac{n}{2}\log(2\pi)$$

(where K is shorthand for the covariance matrix) which we can optimize over to get the hyper-parameters.

Typically the gradient useful in the optimization, we can compute the gradient of the log likelihood with respect to the i^{th} hyper-parameter as:

$$\frac{\partial}{\partial\theta_i}\log p(\mathbf{y}|\mathbf{x},\mathbf{f},\boldsymbol{\theta}) = -\frac{1}{2}\text{Trace}\left(K^{-1}\frac{\partial K}{\partial\theta_i}\right) + \frac{1}{2}\mathbf{y}^\top K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}\mathbf{y}$$

Taking the partial derivative of the matrix K with respect to θ_i is just taking the partial derivatives of each entry. Alternatively we can take the partial of the covariance kernel, and build the new ∂K using the modified kernel. For the square exponential covariance, the partial derivative of the kernels with respect to the parameters, $k_\upsilon(x,x')$ and $k_\ell(x,x')$, are

$$\frac{\partial k(x,x')}{\partial\upsilon} = \frac{\partial}{\partial\upsilon}\left[\upsilon^2\exp\left(-\frac{(x-x')^2}{2\ell^2}\right)\right], \quad \text{and} \quad \frac{\partial k(x,x')}{\partial\ell} = \frac{\partial}{\partial\ell}\left[\upsilon^2\exp\left(-\frac{(x-x')^2}{2\ell^2}\right)\right]$$

$$= \underbrace{2\upsilon\exp\left(-\frac{(x-x')^2}{2\ell^2}\right)}_{k_\upsilon(x,x')} \qquad = \underbrace{\upsilon^2\frac{(x-x')^2}{\ell^3}\exp\left(-\frac{(x-x')^2}{2\ell^2}\right)}_{k_\ell(x,x')}.$$

Algorithm 1 Image Likelihood Evaluation (all cameras) – CPU/GPU.

Compute image noise variance for all primitives in all images. ▷ See Algorithm 2
Set $ll = 0$ and $C = 0$; ▷ Log-likelihood and Partition Function initialization
for c = 0:1:N_C **do**
  for i = 0:1:N_I^c **do**
    Render texture map world as viewed from image i, transfer framebuffer image to main memory, denote $\hat{I}$.
    Render Triangle map as viewed from image i, transfer framebuffer image to main memory, denote I_{tri}.
    for p=0:1:size(I_i) **do**
      Map I_{tri} to plane number, denote m.
      Set $\sigma = w_i^m$
      Set $ll = ll - \frac{(I_i(p) - \hat{I}(p))^2}{2\sigma^2}$ ▷ Computes the log-likelihood
      Set $C = C - \log \sigma\sqrt{2\pi}$ ▷ Computes partition function
    end for
  end for
end for
return $ll + C$. ▷ Returns a normalized log-likelihood

5 Implementation Details

This section discusses important technical details for implementing the proposed model. Importantly, efficient implementations are provided for the image and LiDAR likelihoods. Furthermore, efficient implementation of latent appearance and geometry are discussed. Throughout this section pseudo-code algorithms for CPU and GPU implementation are provided.

5.1 Efficient Image Likelihood Computation

Estimation of camera pose and world geometry relies heavily on the evaluation of the image likelihood. Furthermore, since geometry estimation takes a significant portion of the runtime, fast and efficient computation of this likelihood is crucial. In this section we describe how we implement the image likelihood computation and how we leverage the power of graphics hardware to obtain high performance.

Recall the image likelihood is given by:

$$p(\mathbf{I}|\mathbf{G}, \mathbf{A}, \mathbf{K}, \mathbf{T}) = \prod_{c=1}^{N_C} \prod_{n=1}^{N_I^c} \prod_{k \in \mathcal{S}_c^n} p(I_k^{n,c}|\mathbf{G}, \mathbf{A}, K^c, T^c) \ . \tag{10}$$

The evaluation of the data term in Eq. (10) requires identifying the association between an observation pixel and a world appearance pixel, then evaluating a Gaussian distribution. This computation can be done efficiently in the OpenGL pipeline with two renders: the first maps observation pixels to primitives (in order to identify noise level); while, the second maps appearance pixels to observation pixels. With the information provided in these two renders we can turn one image evaluation into:

$$p(I_n^c|\mathbf{G}, \mathbf{A}, K^c, T^c) = \prod_{k \in \mathcal{S}_n^c} \mathcal{N}(i_k - \hat{i}_k; 0, r_{m^*}^2) \ , \tag{11}$$

where $\hat{i}$ is the textured mapped world as viewed from image n.

The evaluation of Eq. (11) can be seen in Algorithm 1. The purpose of the triangle-mapping render is to identify the noise level that should be associated with the Gaussian evaluation. Note we are not interested in knowing which primitive generated the observation, but only in its appearance value – which we obtain from the texture mapping. The pseudo-code of the image likelihood computation in GPU is identical to that of the CPU. While some implementation details do change (*e.g.*, variable increments must treated atomically), we omit those for the sake of brevity. Table 3 shows the computation time and speed-up factors for the image likelihood computation for several datasets. As can be seen, the GPU implementation is at least four times faster than CPU.

Data	Lubbock	CLIF (subset)	CLIF (all)
# Triangles	80,000	227,000	227,000
# Images	3	24	50
Image Size	1336x891	822x1326	822x1326
Texture/Atlas Size	8/512	16/1024	16/1024
CPU Time (s)	0.295	2.865	5.858
GPU Time (s)	0.066	0.424	0.873
Speed up	4.47	6.76	6.71

Table 3: CPU/GPU image likelihood evaluation speed-up (times estimated by averaging over 5 runs of the algorithm), GPU used was an NVIDIA GTX-580 graphics card.

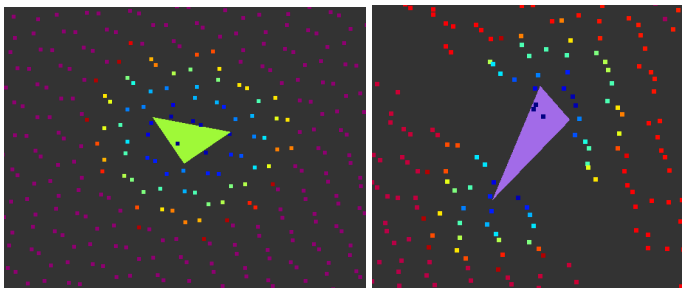

Figure 15: LiDAR and primitive association. (Points associated with input primitive colored according to distance to the primitive, other points shown in orange-magenta color).

5.2 Efficient LiDAR Likelihood Computation

Recall that for the LiDAR observation model, we need to address a data association problem. To this aim, we sample the association from a categorical distribution with probabilities inversely proportional to the distances between a measurement and the set of primitives. The desire to avoid exhaustive distance computation for each LiDAR and primitive pair coupled with the intuition that far way primitives are highly unlikely to generate an observation leads us to the idea of using a k-d tree to prune the set of potential associations to analyze.

Ideally, we would like to build the k-d tree over the set of primitives. Then, for each input LiDAR measurement, identify a set of nearby primitives. However, this idea is prohibitive as the set of primitives is changing (*i.e.*, we are learning the geometry parameters) which would require re-learning the tree every time a primitive changes. Instead we build the tree over the static LiDAR observations; this has the advantage that it only needs to be done once. We may then query the LiDAR tree with a primitive and obtain a candidate list of measurements for association. Fig. 15 shows several association examples.

The procedure outlined above, is used to provide candidate associations. The actual association is sampled from the categorical distribution as described earlier. It is important to point out that as outlined the k-d tree procedure allows multiple primitives to be associated with a given measurement and multiple measurements to be associated with a given primitive. Moreover, there could be primitives with no associated LiDAR points; however, the converse is not allowed as every LiDAR measurement must be associated by a primitive or be labeled as an outlier. To ensure this is the case, we must identify non-associated inlier measurements and attribute candidate associations via exhaustive search. Empirically we find that using the k-d tree reduces the computation time on average by two orders of magnitudes over exhaustive search.

5.3 Efficient Appearance Computation

We now discuss the implementation details for fast appearance estimation. Computationally, our goal is to utilize the power of GPUs as not only a rendering pipeline but also as a general-purpose parallel processor. Recall that A_m denotes the appearance of m^{th} primitive, an image of a user-specified size. We are interested in learning the distribution of color for each pixel according to the update equations (5) and (6). For a set of camera parameters and world structure, the appearance equation requires collecting all of the observation pixels (image pixels) that correspond to a given pixel in the primitive appearance and weighting them appropriately to produce the desired texture. Note that this procedure should be applied only to visible

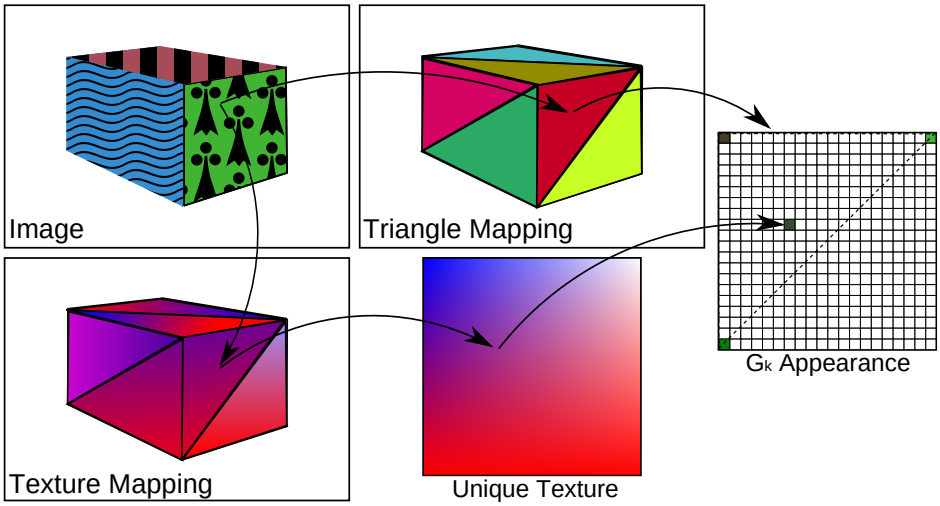

Figure 16: Appearance computation procedure. First determine which primitive generated the observation using a triangle mapping, secondly using the texture mapping determine which pixel generated the observation.

Data	Lubbock		CLIF (subset)	
# Triangles	80,000	80,000	227,000	227,000
# Images	3	3	24	24
Texture /Atlas Size	8/512	16/1024	8/1024	16/1024
CPU Time (s)	1.361	2.435	8.145	11.304
GPU Time (s)	0.201	0.193	1.826	1.9404
Speed up	6.7	12.6	4.5	5.8

Table 4: CPU/GPU speedup (times are the average of 5 estimations), all test done on an NVIDIA GTX-580 graphics card.

primitives, as appearance information is not available for occluded or out-of-view primitives.

The mapping of image observation to triangle appearance can been seen in Fig. 16. The mapping can be explained in two parts. The first part (Fig. 16, top row) maps every observation pixel to a triangle via a color mapping which depends on the primitive. This data association problem can be solved efficiently by using the GPU rendering pipeline to encode the index of the primitive as an RGB triplet. Consequently, observing a particular triplet corresponds to observing a particular primitive. The second part of the mapping involves determining which pixel in the appearance map generated the observation. This mapping can be seen in the bottom row of Fig. 16. For this mapping we create a unique texture, the same size as the appearance map where each RGB triplet encodes the pixel location, then a similar mapping as that for primitive number can be used.

We can put these two pieces together and obtain the appearance estimation method shown in Algorithm 2 for a CPU computation or Algorithm 3 for GPU computation. The main distinction between CPU and GPU implementation is that on CPU all the rendering is done ahead of time and stored, then the textures are computed; in the GPU implementation due to memory constraints, renders are done one at a time and aggregated to the texture after all images are added.

In terms of computational complexity both CPU and GPU algorithms are bounded by the number of images and the size of each image. The main limitation of the CPU algorithm is the data transfer between main memory and device memory, since it requires transferring two images from device to main memory for each observation, and then transferring all the appearance textures from main memory to device memory. The GPU implementation does not suffer from this problem since the texture maps are never transferred, *i.e.*, they remain in device memory and are accessed via the texture accumulation kernel; similarly, the texture data always remains in device memory. Hence the main limitation of the GPU computation is the render itself. Table 4 shows the computation time and speed up factors for several datasets. We can see from the table that the GPU implementation is at least four and a half times times faster than CPU.

Algorithm 2 Appearance Estimation – CPU.

1: Compute unique world projection for all images, I_{tri}^{map} ▷ associates each pixel with a primitive
2: Compute unique texture projection into all images, I_{tex}^{map} ▷ associates each pixel with a texture coordinate
3: **for** k = 1:1:N_p **do** ▷ compute image noise level
4: $w_k = 1$
5: Compute plane normal n_k
6: **for** i = 0:1:N_i **do**
7: Compute image view direction v_i
8: Compute triangle-image weight $w_k^i = \frac{1}{|v_i^\top n_k|}$
9: Compute triangle weight $w_k = w_k * w_k^i$
10: **end for**
11: **end for**
12: Allocate memory for A_k^{tex} and A_k^{weight}, set to zero ▷ the un-normalized texture values and weights respectively
13: **for** c = 1:1:N_C **do** ▷ Loop over cameras
14: **for** i = 1:1:N_I^c **do** ▷ Loop over images
15: **for** p=0:1:size(I_i) **do** ▷ Loop over pixels for i^{th} image
16: Compute triangle source of $I_i(p)$, by looking at $I_{tri}^{map}(p)$, denote T ▷ identify primitive
17: Compute texture location of $I_i(p)$, by looking at $I_{tex}^{map}(p)$, denote P ▷ identify appearance pixel
18: Compute triangle-image weight $w_T^i = \frac{1}{|v_i^\top n_T|}$
19: Compute iteration weight $w = \frac{w_T}{w_T^i}$
20: Set $A_T^{tex}(P) = A_T^{tex}(P) + wI_i(p)$
21: Set $A_T^{weight}(P) = A_T^{weight}(P) + w$
22: *Optional:* Fill in neighbors of p (as lines 20-21). ▷ allows observation to influence multiple appearance pixels
23: **end for**
24: **end for**
25: **end for**
26: Compute and save weighted texture as $A_k = A_k^{tex}/A_k^{weight}$
27: *Optional:* Fill in empty pixels in G_k, with four neighbor average.
28: De-allocate memory.

Algorithm 3 Appearance Estimation pseudo-code - GPU.

1: Compute image noise level. ▷ lines 3-11 in Algorithm 2
2: Create Triangle Map and Texture Map framebuffers (assign 2 textures pointers with depth and color to each)
3: Allocate device memory for A^{tex},A^{weight}, I_{tri}^{map}, and I_{tex}^{map}.
4: **for** c = 1:1:N_C **do** ▷ Loop over cameras
5: **for** i = 1:1:N_I^c **do** ▷ Loop over images
6: Bind Triangle Map framebuffer, render triangle map as viewed from image i.
7: Bind Texture Map framebuffer, render texture map as viewed from image i.
8: Call texture Accumulation kernel. ▷ Same as pixel loop in Algorithm 2
9: **end for**
10: **end for**
11: Call texture normalize kernel. ▷ Computation of line 26 in Algorithm 2
12: De-allocate memory.

5.4 Efficient Geometry Computation

In this section we discuss the implementation details for inferring world-geometry parameters; *i.e.*, we are concerned with evaluating

$$p(\mathbf{V}|\mathbf{I},\mathbf{L},\mathbf{G};\theta) \propto \prod_{n=1}^{N_I} p(I_n|\mathbf{G},\mathbf{A},K,T) \prod_{l=1}^{N_L} p(L_l|\mathbf{G}) \prod_{m=1}^{N_P} p(G_m|\mathbf{V};\theta)\,. \tag{12}$$

Recall that the evaluation of the likelihoods in this equation might be computationally intensive; while the savings established in Sections 5.1 and 5.2 help in reducing the overall computation time of one primitive, the sheer number of primitives for which Eq. (12) needs to be evaluated quickly increases the computation time. This is particularly true for optimization where Eq. (12) becomes the objective function, and thus requires being evaluated thousands of times per primitives. We note that when evaluating the image likelihood as a function of a set of primitives, only a very small subset of pixels change. This observation, coupled with the desire to reduce computation time motivates evaluations of Eq. (12) for multiple primitives in parallel.

The evaluation of Eq. (12) for multiple primitives at a time has the advantage of reducing the number of draws required to compute the image likelihood, which is the computational bottleneck. However, in order to take advantage of these savings, the optimization routines used must propose independent perturbations for the triangles in question; these perturbations must be combined to produce a single render and the likelihood for each primitive must be computed separately, then the optimization scheme must make decoupled decisions based on each likelihood evaluation. Assuming that this is the case, let us investigate the necessary changes to compute image likelihoods in the multi-primitive case.

5.4.1 Image Likelihood Evaluation under Multiple Primitives

In order to extend the approach to multiple primitives we must account for image pixel association and the generating primitive. The pixels in the image likelihood, Eq. (10), can be separated into those generated by the primitives which we are optimizing over, $\mathcal{P}$, and those that are not. Then the image likelihood takes the form:

$$p(\mathbf{I}|\mathbf{G},\mathbf{A},\mathbf{K},\mathbf{T}) = \prod_{c=1}^{N_C}\prod_{n=1}^{N_I^c}\left[\prod_{k\in\mathcal{P}} p(I_k^{n,c}|\mathbf{G},\mathbf{A},K^c,T^c) \prod_{k\in\mathcal{S}_n^c\setminus\mathcal{P}} p(I_k^{n,c}|\mathbf{G},\mathbf{A},K^c,T^c)\right] \tag{13}$$

where $\mathcal{S}_n^c$ is the set pixel in image n of camera c. We are interested in the set of pixels generated by the primitives we are optimizing over, $\mathcal{P}$, and the interactions between this set and the other pixels. Importantly, we are assuming that there is no interaction between elements in the set. To better understand the bookkeeping necessary to achieve the correct calculation, let us concentrate on calculating the partial likelihood for the pixels associated with a single primitive. We begin by noticing that between any two likelihood evaluations, there are two possible changes: first, the set of pixels over which the evaluation takes place can change; second, the value of the underlying rendered image can change. Let us denote the value of the initial rendered image by $f_0(\cdot)$, and the value of the new image by $f_1(\cdot)$; furthermore, let us denote the evaluation masks by M_0 and M_1 respectively. With this notation, we can calculate the change between the image likelihood as:

$$\delta_{01}(M_0,M_1) = f_1(M_0\cup M_1) - f_0(M_0\cup M_1). \tag{14}$$

We note that $\delta_{01}(M_0,M_1)$ properly accounts for changes in the evaluation mask and the function values. If we want to obtain the new absolute likelihood, as opposed to the change, we simply add the initial total likelihood and the change,

$$f_1(M_1) = f_0(M_1\cup M_0) + \delta_{01}(M_0,M_1) - f_1(\bar{M}_1\cap M_0) \tag{15}$$

We easily see special cases of this general computation from Eq. (14), *e.g.*, if the evaluation masks remain the same, $M_0 = M_1$, then we are simply taking the difference between the image values. If in addition $f_0 = f_1$, then $\delta_{01} = 0$ and there is no change.

Now that we understand how to compute the incremental likelihood for a single triangle, the extension to the multi-triangle case follows trivially: we simply let each triangle have its own evaluation mask. Computationally, we can evaluate the multi-primitive image likelihood as shown in Algorithms 4 and 5. The computation is divided into an initialization step and a subsequent computation since the initial step is much simpler; it only requires being able to evaluate current values. On the other hand, subsequent evaluations are slightly more involved since for every pixel that we encounter we have to compute not only its current triangle association but also its past association.

Several observations can be made from the algorithms, first we notice that memory usage has dramatically increased, (*i.e.*, more than doubled using 4 byte floats for ll), since we must maintain all the previous pixel likelihood values for each image,

Algorithm 4 Multi-Primitive Image Likelihood Evaluation - Initialization - CPU/GPU.

Select N planes compute the likelihood over.
Initialize an array of N structure planeLL, with members: imgLL, and imgPxCount (set all to zeros)
Initialize array of N_i structure imEvalMask, with members: mask, ll, and llpixelToAdd. Set mask and ll to the size of the corresponding i^{th} image.
for c = 1:1:N_C **do** ▷ camera loop
  for i = 1:1:N_I^c **do** ▷ image loop
    for $p \in \mathcal{S}_i^c$ **do** ▷ pixel loop
      Compute pixel log likelihood, denote ll.
      Set imEvalMask[i].ll[p]=ll
      Identify triangle associated with current pixel, denote T.
      if optimizing over T **then**
        imEvalMask[i].mask[p]=T
        planeLL[k].imgLL += ll; ▷ the k^{th} entry is associated with triangle T
      end if
    end for
  end for
end for
Maintain imEvalMask and planeLL in memory.

Algorithm 5 Multi-Primitive Image Likelihood evaluation - Subsequent evaluations - CPU/GPU.

Set all members of planeLL to zero, except imgLL.
for c = 1:1:N_C **do** ▷ camera loop
  for i = 1:1:N_I^c **do** ▷ image loop
    for $p \in \mathcal{S}_i$ **do** ▷ pixel loop
      Compute pixel log likelihood, denote ll.
      Identify world triangle associated with pixel p, denote T.
      Identify previous triangle association at pixel p, $\hat{T}$ = imEval[i].mask[p].
      if optimizing over T or $\hat{T}$ **then** ▷ multi-plane bookkeeping
        if optimizing over T **then**
          Set index = k; ▷ where k is the position of triangle T in planeLL structure
          Set planeLL[index].imgLL += ll
          Increment planeLL[index].imgPxCount by one.
        else if optimizing over $\hat{T}$ **then**
          Set index = k;
          Set planeLL[index].imgLL += ll
          Increment planeLL[index].imgPxCount by one.
        end if
        Subtract previous ll, planeLL[index].imgLL -=imEvalMask[i].ll[p]
      end if
      Set imEvaMask[i].ll[p] = ll ▷ update likelihood and mask for the next iteration.
      Set imEvaMask[i].mask[p] = T
    end for
  end for
end for
Read off likelihood values from planeLL[·].imgLL

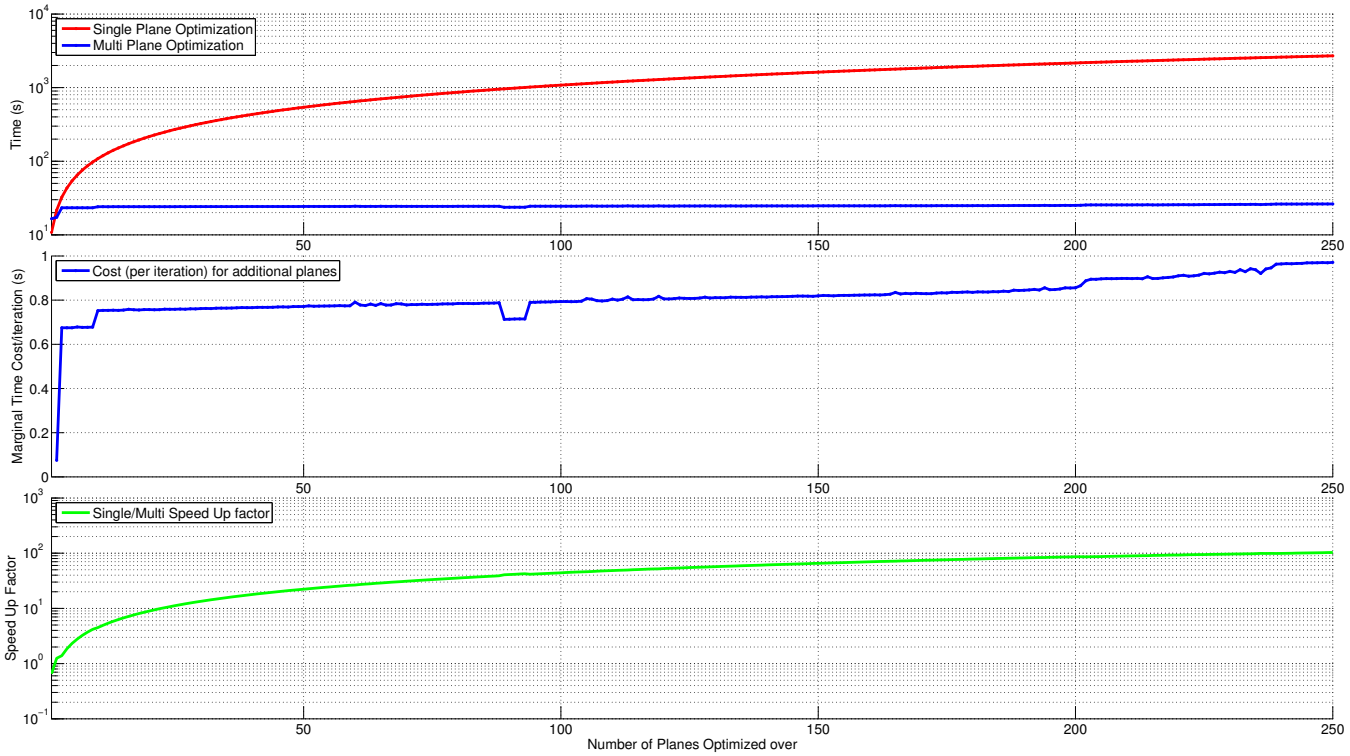

Figure 17: Computation time comparison for single and multi-plane optimization for the CLIF Intersection scene. Top plot: Cost for each method as a function of number of planes being optimized. Middle plot: per iteration additional cost as a function additional planes added (base is 1). Bottom plot: speed up factor.

as well as the evaluation mask. This large memory consumption is prohibitive for graphic hardware, where memory is limited. Furthermore, the use of structures in graphics hardware is cumbersome, since memory management can be difficult. As a result, graphics implementation requires unrolling the structures into one continuous array, creating extra bookkeeping details or creating extra kernels to handle memory operations.

All results presented here and in the paper used the multi-primitive evaluations. In terms of computational complexity, the multi-primitive evaluations are substantially faster than the single primitive evaluations. This is seen in the top plot of Fig. 17, where the computation time for the multiple-primitive estimation is always lower than single plane (the exception being when a single primitive is optimized, in this case the added bookkeeping is slightly slower). The overall speed-up factor between single and multi-primitive is seen in the bottom plot of Fig. 17. From the figure, we see that the gains are anywhere between slightly less than one and over two orders of magnitude.

5.5 Texture Atlas

As we saw in Sec. 5.1, the ability to render a textured mapped scene is crucial for the image likelihood computation. Furthermore, as hinted above, the limiting factor of the GPU implementation is the speed of the rendering itself, rather than the computation. In order to reduce overhead, we use texture atlases. These allow us to reduce the number of texture binding needed at drawing time.

According to the proposed model, each triangle has a canonical texture, corresponding to a right triangle of user-specified size. This modeling choice has several consequences: 1) texture coordinates are static; 2) computed textures are subject to different scaling for each direction; 3) rendered texture pixel need not be square. The first of these points is essential in practice, since the texture coordinates do not need to be re-calculated when the geometry of a triangle is changed, yielding substantial computational savings; however, these savings come at a cost - the latter two points. Furthermore the use of right triangles has the added benefit of allowing two canonical textures to be stored using a single square texture (the only type OpenGL allows). Thus providing us the ability of storing two textures per image.

Note that there are two user parameters: the texture size, T, (textures will be $T \times T$) and the atlas size, A, (atlases will

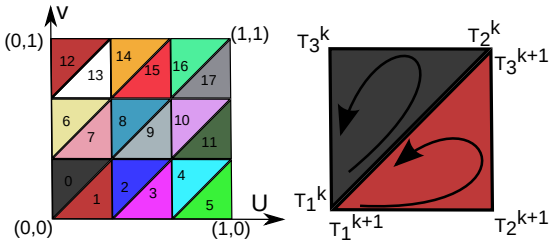

Figure 18: Texture Atlas Pictorial Depiction. *Left:* sample atlas where the number indicate which triangle the texture belongs to; *Right:* vertex drawing order (1,2,3).

be $A \times A$). Due to openGL implementation constraints both need to be powers of two and $A \geq T$. The general idea of a texture atlas is pictorially depicted in Fig. 18. In the reminder of this section we describe the initialization, update and drawing procedure for the texture atlas.

5.5.1 Initializing

Assuming that the number of triangles is known (denote M, and triangles are indexed by k, where $k \in [0, M-1]$), then the initial set up of the texture atlas is seen in Algorithm 6. The algorithm can be broken down into three main parts, first allocating main memory to contain the list of triangle properties, including the atlas associated with each primitive as well as its texture neighbor. A texture neighbor, is the triangle that shares the texture square in the texture atlas (*e.g.*, 0 and 1, or 14 and 15 in Fig. 18); this information will be needed for the texture updates later.

The second part of the algorithm computes the texture coordinate in the atlas, this computation is dependent on the triangle's ordering. The last step of the algorithm is to transfer all parameters to device memory.

Algorithm 6 Texture Atlas Initialization.

1: Set the values of A and T.
2: Compute $N = \lfloor \frac{A}{T} \rfloor$, the number of textures per dimension.
3: Compute $N_a = \lceil \frac{M}{2N^2} \rceil$, the number of atlases needed to represent all the triangles.
4: Create an array of N_a atlases in device memory, each of size $A \times A$, initialize them to zero, denote $pAtlas$.
5: Create $pTex$ an array that contains the atlas index for each triangle.
6: Create $texNeigh$ an array that contains the texture neighbor (either up or down for each triangle.
7: Compute $\delta = \frac{1}{A}$, the one pixel offset in the diagonal between the top and bottom texture.
8: Compute Texture Coordinates using the counter-clockwise order shown in Fig. 18.
9: **for** k = 0:1:M-1 **do**
10: $pTex[k] = pAtlas[k \mod 2N^2]$. ▷ assigns atlas to triangle
11: **if** $k \mod 2 == 0$ **then**
12: Compute: $x = (\frac{k}{2} \mod \frac{N^2}{A^2}) \mod \frac{N}{A}$ and $y = \frac{[(\frac{k}{2} \mod \frac{N^2}{A^2}) - x]A}{N}$
13: $T_1^k = (x, y + \delta)$
14: $T_2^k = (x - \delta, y + \frac{1}{N})$
15: $T_3^k = (x, y + \frac{1}{N})$
16: Set $texNeigh = k + 1$;
17: **else**
18: Compute: $x = (\frac{k-1}{2} \mod \frac{N^2}{A^2}) \mod \frac{N}{A}$ and $y = \frac{[(\frac{k-1}{2} \mod \frac{N^2}{A^2}) - x]A}{N}$
19: $T_1^k = (x + \delta, y)$
20: $T_2^k = (x + \frac{1}{N}, y)$
21: $T_3^k = (x + \frac{1}{N}, y + \frac{1}{N} - \delta)$
22: Set $texNeigh = k - 1$;
23: **end if**
24: **end for**
25: Create device vertex array, normal array, texture coordinate array, and transfer primitive information.

Method	Texture Size	Atlas Size	Number of Atlases	Render Time (micro seconds)
Single Textures	8	-	-	110,000 - 115,000
Texture Atlas	8	256	40	9.4 - 11.5
Texture Atlas	8	512	10	6.9 - 8.5
Texture Atlas	8	1024	3	5.5 - 7.3
Texture Atlas	8	2048	1	5.1 - 6.9
Texture Atlas	8	4096	1	5.7 - 7.2
Texture Atlas	8	8192	1	5.7 - 7.2

Table 5: Render time for different texture methods on 80k triangles (tested: on NVIDIA GTX-580).

5.5.2 Update

Suppose we want to update the texture of the k^{th} triangle, with some texture image, $data$. The update procedure is seen in Algorithm 7. The update procedure consists of first identifying the orientation of the k^{th} triangle and its neighbor; then building the replacement image $texData$ accordingly. Once the replacement image has been created, it can be uploaded to the texture atlas using glTexSubImage2D with the proper starting and ending coordinates.

Algorithm 7 Texture update for the k^{th} primitive.

if $texNeigh$ is valid **then** ▷ check status of neighbor triangle
  if $k \mod 2 == 0$ **then** ▷ Orient the texture data
    $topTex = data$, $bottomTex = texNeigh$.
  else
    $topTex = texNeigh$, $bottomTex = data$.
  end if
  Construct $texData$ using the correct portions of $topTex$ and $bottomTex$.
else
  $texData = data$.
end if
if $k \mod 2 == 0$ **then**
  $x = (\frac{k}{2} \mod \frac{N^2}{A^2}) \mod \frac{N}{A}$
  $y = \frac{[(\frac{k}{2} \mod \frac{N^2}{A^2}) - x]A}{N}$
else
  $x = (\frac{k-1}{2} \mod \frac{N^2}{A^2}) \mod \frac{N}{A}$
  $y = \frac{[(\frac{k-1}{2} \mod \frac{N^2}{A^2}) - x]A}{N}$
end if
Replace part of the atlas image (*e.g.*, using glTexSubImage2D, with x and y as offsets, and $texData$ as texture data).

5.5.3 Draw

The draw routine for texture atlases consists of Algorithm 8. It is important to point out that the draw order is counter-clockwise as shown in Fig. 18.

5.5.4 Resulting Speed-up Gain

Render times using texture atlases are shown in Table 5. From the render times we can see that using texture atlases provides a five orders of magnitude speed up over individual textures. Furthermore, from the table we can see that the atlases advantages plateau when all primitives are drawn using a single atlas. Importantly, increasing the size of the atlas beyond this point can be detrimental as the atlas might be larger than what the graphics hardware can cache (as seen in the last two rows of Table 5).

Algorithm 8 Draw routine using texture atlases.

Bind vertex, normal, texture coordinate, and index arrays.
Compute $lastDrawCount = M \mod 2N^2$. ▷ this is the number of primitives in the last atlas
for $i = 0 : 2N^2 : M - lastDrawCount$ **do**
    Bind $pTex[i]$ Texture.
    Draw $2N^2$ primitives in order. (*e.g.*, using glDrawRangeElements)
end for
if $lastDrawCount \neq 0$ **then**
    Bind $pTex[i+1]$ Texture.
    Draw $lastDrawCount$ primitives in order.
end if

References

[1] R. Cabezas. Aerial Reconstructions via Probabilistic Data Fusion. SM Thesis, Massachusetts Institute of Technology, 2013. http://people.csail.mit.edu/rcabezas/. 2

[2] R. Cabezas, O. Freifeld, G. Rosman, and J. W. Fisher III. Aerial Reconstructions via Probabilistic Data Fusion. *CVPR*, 2014. 1, 2

[3] O. Freifeld and M. Black. Lie bodies: a manifold representation of 3D human shape. *ECCV*, 2012. 9

[4] Y. Furukawa and J. Ponce. Accurate, dense, and robust multiview stereopsis. *PAMI*, 2010. 5, 7

[5] M. Kazhdan and H. Hoppe. Screened Poisson Surface Reconstruction. *ACM Trans. Graph.*, 32(3):29:1–29:13, July 2013. 5, 7

[6] D. J. C. Mackay. *Information Theory , Inference , and Learning Algorithms*. Cambridge University Press, 2003. 11

[7] A. Mastin, J. Kepner, and J. Fisher. Automatic registration of LIDAR and optical images of urban scenes. *CVPR*, 2009. 2, 5, 7, 8

[8] Ohio Statewide Imagery Program and Ohio Geographically Referenced Information Program. Ohio LiDAR Data. http://gis3.oit.ohio.gov/geodata/. 2

[9] C. Rasmussen and C. Williams. *Gaussian processes for machine learning*. The MIT Press, 2006. 11

[10] N. Snavely, S. M. Seitz, and R. Szeliski. Modeling the World from Internet Photo Collections. *IJCV*, 2007. 5, 7

[11] US Air Force. Columbus Large Image Format Dataset 2007. https://www.sdms.afrl.af.mil/index.php?collection=clif2007. 2

[12] Zhou, Q.Y., Neumann, U. 2.5D Dual Contouring: A Robust Approach to Creating Building Models from Aerial LiDAR Point Clouds. *ECCV*, 2010. 2, 5, 7, 8