Multi-camera non-rigid SSD Tracking

Neil Birkbeck

September 16, 2009

1 Introduction

In the past little while I have been trying to determine a way that dense geometric deformations can be reconstructed and tracked over time from a vision system. The main application is to track dense geometry deformations of a human subject over time so that they can be played back under novel view and animation conditions.

It is assumed there are only a few cameras (with known pose calibration) observing the scene, so it is desirable to take advantage of as much temporal information as possible during the reconstruction. As a simpler starting point for this problem, I felt it appropriate to work on the reconstruction of dense geometry for a moving human head.

Many existing tracking methods start with some known dense geometry, or assume that the geometry can be reconstructed at each time instant (e.g., [3]). In our case, neither of these assumptions hold; the latter because we have only a few cameras. The general problem is therefore to reconstruct a dense model and its deformations over time.

In the case of a human head, or partially rigid part, once the gross motion is accounted for (or tracked) it should be possible to combine the information over time with temporal constraints in the space of the part. In this way, the temporal constraints make sense. Unfortunately, for this problem, it is still not clear what representation would be most appropriate. Consider again, a head and shoulders, observed mainly by frontal viewing cameras (see the Fig.1). In such a case we cannot reconstruct even an enclosing geometry (e.g., using SFS). At the initial time step, only part of the head can be reconstructed, but if the head were to rotate, we should be able to append the additional information into the model. If we use a mesh representation, and track only the motion of the head, we will end up with a set of noisy measurements (due to stereo and the tracking) in the reference frame of the head. If the observations circle the object, it is not clear how well a mesh representation would be to reconstruct even a static model. Some work in space-time reconstruction, e.g., for laser scanned data [6], suggests that a characteristic function representation can be augmented with a time dimension to achieve a spatially and temporally smooth reconstruction that agrees with partial input data. But it is not clear how well such implementations deal with noisy input. The single-time instant reconstructions that are similar to these approaches (e.g., Poisson surface reconstruction [4]) do not work well with this type of input data.

Other vision-based approaches use a level-set implementation and minimize an energy function that utilizes the input images directly [5] (as opposed to using secondary stereo information). Again, it is also not clear how well these methods will work with such partial information.

A part of the problem is to reconstruct the motion of the rigid parts. This can be done using all available cues, e.g., texture and stereo. And it too can be accomplished with different representations (e.g., a set of points, a mesh, or level-sets). In these notes, I considered this problem alone: given an approximate surface representation for a non-rigid object, find the transformations of the surface over time that best agree with the input images. Keep in mind, however, that these ideas are motivated by the bigger problem, where we would like to refine the surface over time.

First, we need to define our choice of model and

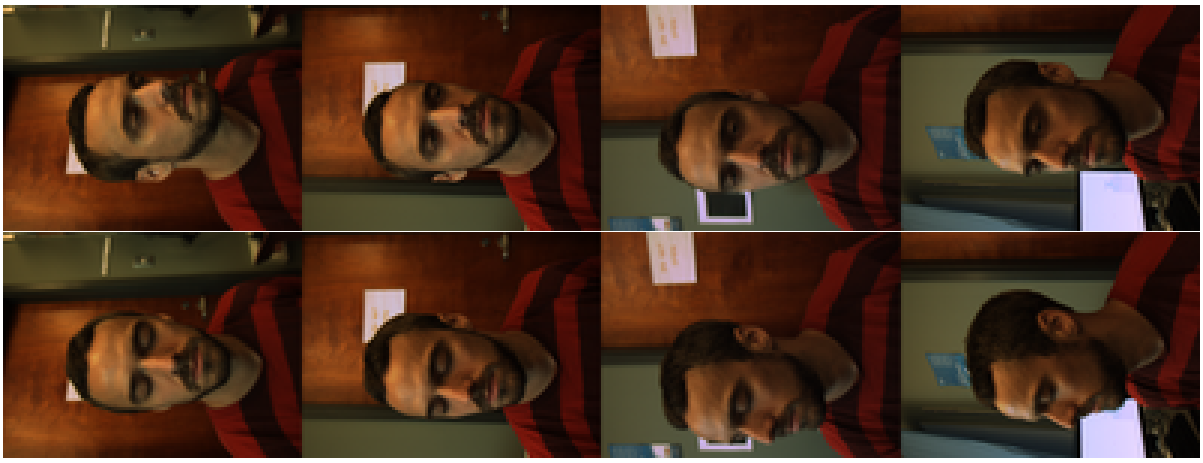

Figure 1: Example images of a head (and shoulders) taken from 4 viewpoints at two instances of time (see input-sequence.avi

how we model non-rigid deformations. In these notes, a point-based representation of the model is available (so visibility will be ignored), and the non-rigid approximation is restricted to a skinned-mesh (e.g., the parameters are only the transformation parameters of the underlying skeleton).

2 Theory

2.1 Problem Definition

Given a set of input images, $I_{i,t}$, taken from $i \in 1, C$ cameras at stationary viewpoints over time $t \in \{1, T\}$, reconstruct the motion of a set of points $\{v_j\}$ (or mesh) over time. The camera calibration is given by the matrices P_i, and we will use the symbol $\Pi(\mathbf{x})$ to denote the projective division by z. As indicated earlier, the motion of the points $\{v_j\}$ is governed by the motion of an underlying skeleton, using the ubiquitous linear-blend skinning

$$\mathbf{v}_{j,t}(\theta) = \sum_b w_{b,j} M_{b,j}(\theta) M_{b,j}(\mathbf{0})^{-1} \hat{v}_j$$

where $w_{b,j}$ is the attachment weight of vertex j to bone b, and $M_{b,j}(\theta)$ is the transformation from bone space to world space, and $\hat{v}_j$ is the vertex in a rest pose. The transformation matrix of a bone, $M_{b,j}(\theta) = M_{p(b),j}(\theta) T_b A(\theta_b)$, is a concatenation of the relative transformation of the bone, b, in the parents $p(b)$ frame, T_b, with the parameters of the bone $A(\theta_b)$, which for most links are just a set of rotations. The transformation matrix $M_{b,j}$ is dependent on the parameters of the bone, b, and all of its ancestors, but it is convenient to treat the transformation as a set of all the parameters.

Inspired by 2D SSD tracking (e.g., [1]), it is straightforward to derive a tracking algorithm for such a situation. If we consider the appearance of the 3D points to be that at the original time instant, we end up with a brightness constraint:

$$f_j(\theta_\mathbf{t}) = I_{i,t}(\Pi(\mathbf{P}_i \mathbf{v}_{j,t}(\theta_t)) - I_{i,1}(\Pi(\mathbf{P}_i \mathbf{v}_{j,1}(\theta_1)) = 0 \quad (1)$$

for any $t > 1$.

2.2 A simple rigid transformation case

If we replace the skinned mesh representation with a Euclidean transformation, we can derive a tracking algorithm (here is the forward compositional approach, there has been some debate about

the existence of the inverse compositional)[1] If we parametrize our rigid transformation with an axis-angle representation and translation, e.g., $\mathbf{theta} = [t_x, t_y, t_z, v_x, v_y, v_z]^T$, and assume we are given some approximation of the current parameters θ_t^h, we can represent an updated transformation as composition of $R(\theta_t^h)R(\delta\theta_t^h)$.

Eq. 1 is a set of non-linear equations, which we can solve using Gauss-Newton method, solving for updates to the following system of linear equations:

$$J\delta\theta_t^h = -f(\theta_{\mathbf{t}}^{\mathbf{h}}) \tag{2}$$

where J is the jacobian of the cost function, and f is the stacked residual of all the equations. The Jacobian is straighforward to compute. Letting $p \in [t_x, t_y, t_z, v_x, v_y, v_z]$,

$$\frac{\partial f_j}{\partial p} = \frac{\partial}{\partial p} I_{i,t}(\Pi(\mathbf{P}_i R(\theta_t^h)R(\delta\theta_t^h)\hat{\mathbf{v}}_{j,t})) \tag{3}$$

$$= \nabla I_{i,t}(\Pi(\mathbf{P}_i R(\theta_t^h)\hat{\mathbf{v}}_{j,t}))\frac{\partial}{\partial p}[\frac{x}{z}, \frac{y}{z}]^T \tag{4}$$

where

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = P_i R(\theta_t^h)R(\delta\theta_t^h)\hat{\mathbf{v}}_{j,t}$$

differentiating x/z, y/z, requires only the partial derivatives of $R(\delta\theta_t^h)$ w.r.t. p.

$$\frac{\partial}{\partial p}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial p}x \\ \frac{\partial}{\partial p}y \\ \frac{\partial}{\partial p}z \end{bmatrix} = P_i R(\theta_t^h)\left(\frac{\partial}{\partial p}R(\delta\theta_t^h)\right)\hat{\mathbf{v}}_{j,t}$$

Differentiating $x/z, y/z$ is trivial with the chain rule.

In the compositional case, solving Eq. 2, gives the update $\delta\theta_{\mathbf{t}}^{\mathbf{h}}$, which is then combined to get the updated transformation $R(\theta_t^{h+1}) = R(\theta_t^h)R(\delta\theta_t)^h$. Only few of these iterations are necessary to get the updated parameters.

An alternative is to use an additive rule, which is more convenient in the case of a skinned mesh.

[1] This is similar to the skinned mesh representation, where there is only one bone and all vertices are assigned a weight of 1 to this bone.

2.3 The full skinned mesh case

In the skinned-mesh case, the situation is similar, but the Jacobian computation is slightly different. Each vertex is dependent on the parameters of the bones to which $w_{b,j}$ is non-zero, and all of the parameters of their ancestors. The derivative computation is similar, and reduces to finding the partial derivative of the transformed vertex w.r.t. the bone parameters. This is again takes on the common form of a composition of the matrices that are not affected by the parameter with the partial derivative of the matrix affected by the parameter. The partial derivative of the vertex is given by its transformation by this composition of matrices. The final Jacobian is then weighted by the skinning weight.

2.4 Quasi-newton optimization

A solution to the problem for each time can be obtained with a quasi-newton method that requires only the gradient of the cost function. An equivalent cost to the non-linear equations is

$$f(\theta) = \sum_i f_i(\theta)^2$$

Quasi-newton methods require the gradient, which is straightforward to compute with the Jacobian (e.g., each row of the Jacobian is multiplied by $2f_i(\theta)$, and the columns are summed). With this approach, it is easy to modify the cost function with a more robust norm, for example a regularized L1-norm:

$$h(\theta) = \sum_i \sqrt{f_i(\theta)^2 + \epsilon^2}$$

for some small ϵ (relate to optic flow, any function of f_i^2).

3 GPU implementation

Given some input data, an implementation of the described methods is straightforward, requiring only basic linear algebra, image derivatives, and image sampling. The most CPU intensive component (for a roughly 10000 points) is the computation of the

Jacobian (or gradient) of the cost function. Luckily as the gradient involves sampling the input images, sampling the gradient images, and computing the partial derivative of the transformed vertex w.r.t. the transformation parameters, it is easily mapped to the GPU. In fact, in the quasi-newton implementation, almost all of the computation can be performed on the GPU, with only the gradient and the final cost needing to be transferred back to the CPU.

The GPU implementation first transfers all the constant data to the GPU (e.g., rest vertex positions, texture data for the current frame, bone skinning weights, and original input image samples).

The results are computed per-view and accumulated over all the views. Before computing the cost (and gradient per view), the data common to all views is computed; this consists of the current deformed vertices (could be done on the GPU as well) and also computing the partial derivatives of the vertices w.r.t. the joint parameters. This step involves concatenating the transformation matrices on the CPU, followed by a transformation of all vertices by the computed matrix. The transformed quantities accumulate a derivative temporary storage (weighted by the associated bone weights). Note that the parameters of a bone affect the transformations of all children in the kinematic chain.

Then, for each view, the first pass computes the current cost function per vertex (e.g., by looking up the projected intensities at time t and subtracting the reference values). The second pass computes the derivative of the cost with respect to the joint parameters (e.g., the Jacobian matrix). In the case of a quasi-newton approach, the gradient can then be accumulated with a reduction operation. In the case of the Gauss-newton approach, the matrices $J^T J$ and $-J^t f$ can be computed on the GPU using reduction (the update can then be computed with a quick inverse on the cpu, e.g., $(J^T J)^{-1} J^t$). In practice, we found it faster to read-back the entire Jacobian matrix and solve the system of equations using lapack.

In the end, for a two bone configuration, with 6 parameters, 10000 vertices, and 4 views, it is possible to achieve tracking performance of almost 7 FPS. Not quite real-time, but an almost 10 fold improvement on the CPU implementation. This was achieved with the Gauss-Newton approach using a fixed number of iterative updates per-frame. The performance of the quasi-newton method was slightly worse, as there was no way to limit the number of function evaluations in the quasi-newton LBFGS method we used.

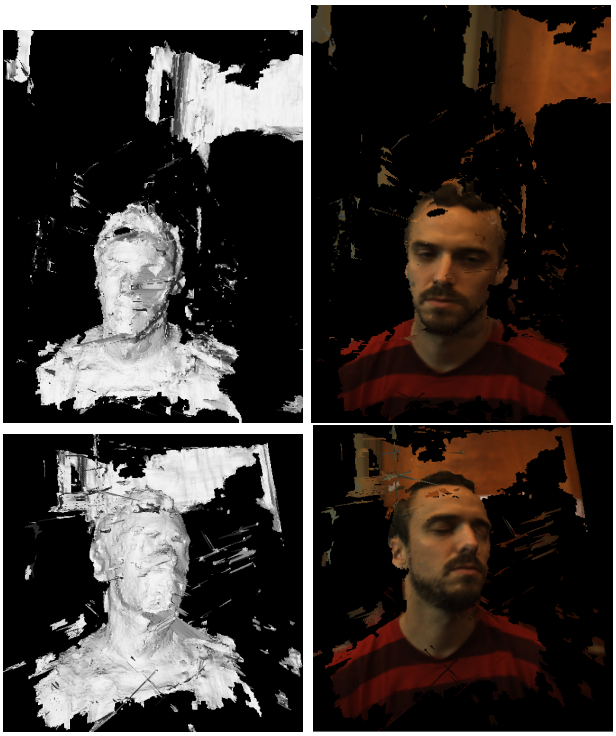

Figure 2: Two views of noisy input depth meshes extracted from stereo.

4 Experiments

To test the implementation, a sequence of my head and shoulders was captured using 4 synchronized cameras (400x300 resolution). The sequence had 213 frames (see http://www.neilbirkbeck.com/?p=1394 for videos). The algorithm requires an initial starting mesh; this mesh was obtained by doing stereo on the first frame (see Fig.4 for examples of depth created from stereo). The skeleton, which consists of two bones (one for the shoulders and one for the

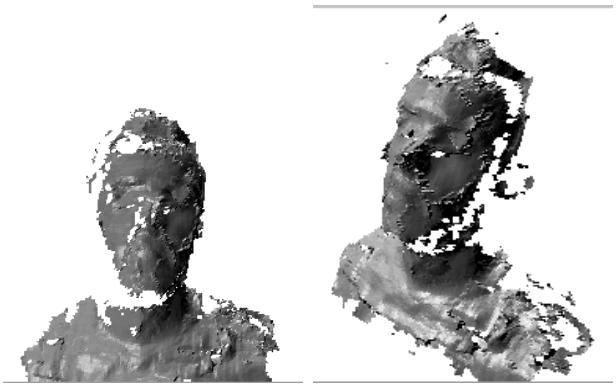

Figure 3: Two views of the input mesh extracted from depth at time 1

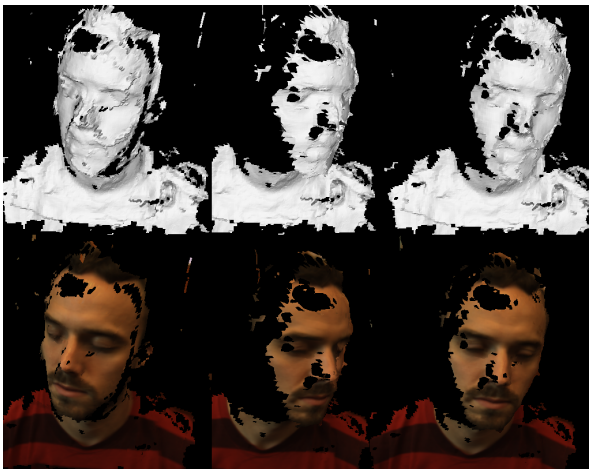

Figure 4: The tracked results. The sequence mainly contained head motion.

head) was manually aligned and skinning weights were initialized using the Laplacian heat approach of Popovic [2]. The initial pose estimate for time $t = 1$ is known, as the mesh was initialized in this frame. Subsequent frames were initialized using the optimized result from the previous frame. Tracking using both the LBFGS and Newton method worked well (see Fig. 4). The timings for a Gauss-Newton method with 4 iterations were 4m36.976s on the CPU and 38s on the GPU, including loading the images and gradient computations.

Another sequence involved more motion (including non-rigid motions, smiling, etc.). In this case, the tracker was lost due to visibility not being taken into account (see the videos).

5 Outlook and Conclusion

The tracked coordinate frame may be a more useful location to register information and enforce temporal smoothness. If we just track the rigid portion of the face in the examples, and register the dense stereo in this coordinate frame it is clear that we should be able to get a more complete model (see Fig.5) Again, the problem is that it is not clear what is the best representation; it might be slightly easier if there were more cameras (and a visual hull could be used). Also, it is not clear if it is best to merge geometries or use a representation that can reconstruct a consistent geometry over time.

Real-time would be cool, but it doesn't necessarily scale well (e.g., more parameters will necessarily slow down the implementation). A good thing is that extra bones may only add one or two parameters (not a full 6 d.o.f. transformation).

Future improvements include incorporating visibility into the objective function, and making the objective function less sensitive to illumination changes (cross-correlation or mutual information). Using triangles instead of the vertices would also be an improvement. A more efficient implementation might consider using only the salient points on the model. It would be interesting to see if adding more bones to see if this type of tracking can be used to track more non-rigid motions (e.g., the motion cheeks and jaw).

As for tracking the dense non-rigid changes, it may be useful to use optic flow (e.g., [3]).

5.1 Historical note

An original implementation of a single rigid body transformation was completed in roughly one day (Early September, a Friday I think). The Tuesday of the next week was pretty much a write-off, and I only tried some visualization and investigated using

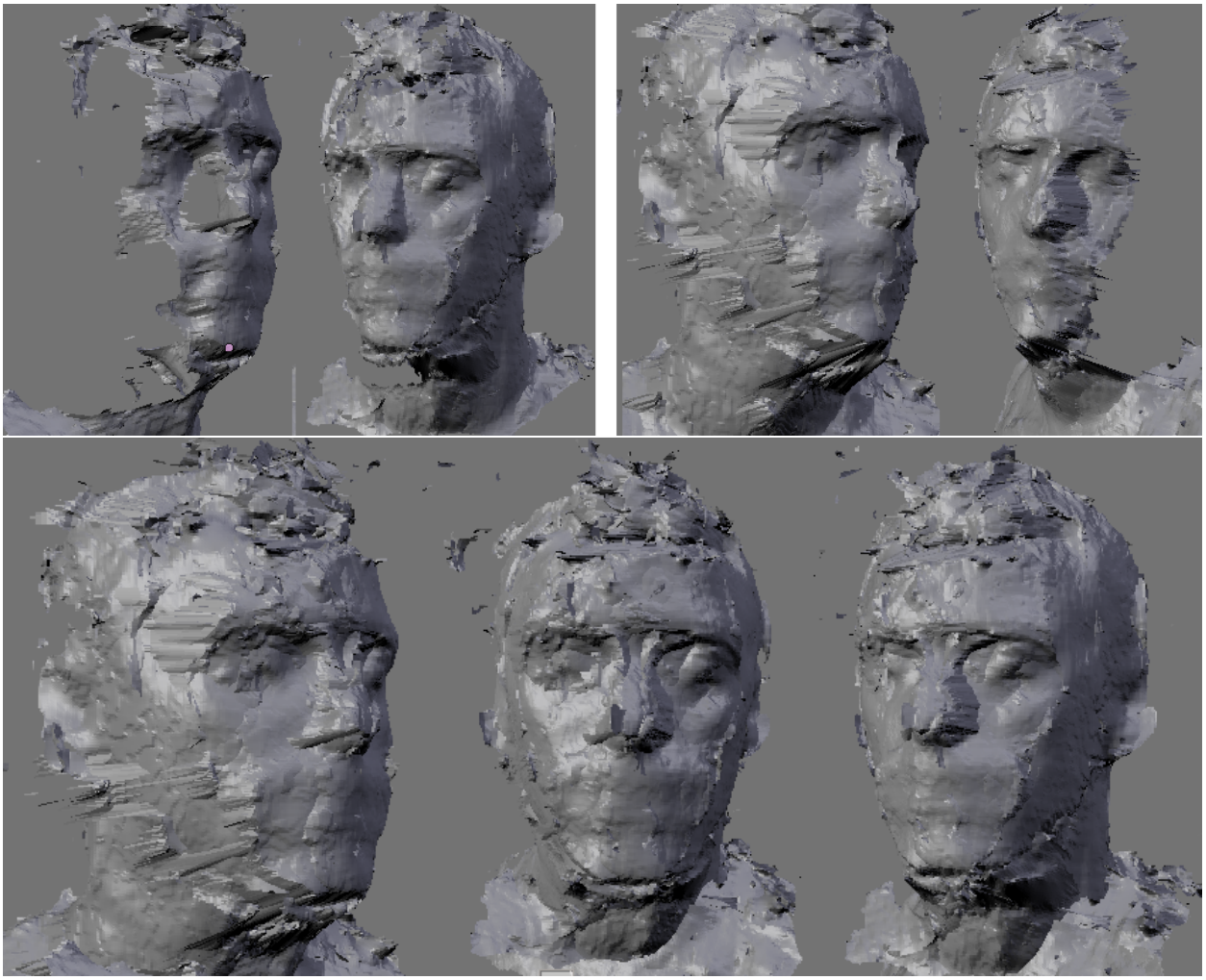

Figure 5: Top row: two geometries from different time steps. Camera configuration implies only part of the surface is reconstructed well (e.g., left or right shown above). When the tracking information is used to register these meshes in the coordinate frame of the head, the geometry can be merged (bottom row). Tracking inaccuracies and non-rigid motions will cause slight misregistration, but it is a good start.

LBFGS and a robust measure over the next two days. The Thursday day (and night) I was thinking about using CUDA for the GPU implementation for no reason other than to see if I could make it real-time (and maybe learn some CUDA along the way). The Friday day was pretty much all spent transferring the Jacobian computation and cost function computation to the GPU. At this point, I think I knew that I was at best going to see a 10 times speed-up.

On Saturday I cleaned up and merged this code into its final resting position. Saturday night, I played around with the Gauss-newton implementation (first started doing the gradient on the GPU). Also cleaned up the makefiles and tested more of the code. Come Sunday afternoon, after doing some necessary closet clean-up and re-filling (via a few purchases at winners), I started writing this document. It is now Sunday night, and the document is almost readable. Need to add some pictures, and clean it up.

A Derivatives of Euclidean transformation

Parital derivatives of the transformation matrix parametericed by $\theta = [t_x, t_y, t_z, v_x, v_y, v_z]^T$:

$$T(\theta_t) = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}([v_x, v_y, v_z]^t) & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix}$$

$$\mathbf{R}([v_x, v_y, v_z]^t) = \mathbf{I} + \omega sin(\phi) + \omega^2(1 - cos(\phi))$$

$$\phi = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

$$\omega = \begin{bmatrix} 0 & -\frac{v_z}{\phi} & \frac{v_y}{\phi} \\ \frac{v_z}{\phi} & 0 & -\frac{v_x}{\phi} \\ -\frac{v_y}{\phi} & \frac{v_x}{\phi} & 0 \end{bmatrix}$$

The partials w.r.t. t_x is (t_y and t_z similarly)

$$\frac{\partial}{\partial t_x} T = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\frac{\partial T}{\partial v_x} = sin(\phi)\frac{\partial \omega}{\partial v_x} + \omega cos(\phi)\frac{\partial \phi}{\partial v_x} + \omega^2 sin(\phi))\frac{\partial \phi}{\partial v_x} + (1 - cos(\phi))(\omega\frac{\partial \omega}{\partial v_x} + \frac{\partial \omega}{\partial v_x}\omega)$$

$$\frac{\partial \phi}{\partial v_x} = \frac{v_x}{\phi}$$

$$\frac{\partial \omega}{\partial v_x} = \frac{1}{\phi^2}\begin{bmatrix} 0 & v_z\frac{\partial \phi}{\partial v_x} & -v_y\frac{\partial \phi}{\partial v_x} \\ -v_z\frac{\partial \phi}{\partial v_x} & 0 & -(\phi v_x - v_x\frac{\partial \phi}{\partial v_x}) \\ v_y\frac{\partial \phi}{\partial v_x} & \phi v_x - v_x\frac{\partial \phi}{\partial v_x} & 0 \end{bmatrix}$$

The derivatives w.r.t. v_y and v_z are similar.

References

[1] Simon Baker and Iain Matthews. Lucas-kanade 20 years on: A unifying framework. *Int. J. Comput. Vision*, 56(3):221–255, 2004.

[2] Ilya Baran and Jovan Popović. Automatic rigging and animation of 3d characters. *ACM Trans. Graph.*, 26(3):72, 2007.

[3] Carlos Hernández, George Vogiatzis, Gabriel J. Brostow, Bjoörn Stenger, and Roberto Cipolla. Non-rigid photometric stereo with colored lights. In *Proc. of the 11th IEEE Intl. Conf. on Comp. Vision (ICCV)*, 2007.

[4] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 61–70, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[5] Marcus Magnor and Bastian Goldlucke. Spacetime-coherent geometry reconstruction from multiple video streams. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 365–372, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Andrei Sharf, Dan A. Alcantara, Thomas Lewiner, Chen Greif, Alla Sheffer, Nina Amenta, and Daniel Cohen-Or. Space-time surface reconstruction using incompressible flow. *ACM Trans. Graph.*, 27(5):1–10, 2008.