1 Introduction

I recently read the paper entitled *A direct approach for efficiently tracking with 3D Morphable Models* by Muñoz et al (published in ICCV 2009). I was curious whether their technique for precomputing some of the derivatives could extend to both multiple cameras and to a skinned mesh (as opposed to a set of basis deformations). I spent a couple days understanding, implementing, and trying to extend this idea. This document is a summary of my results and relies heavily on the ideas in that paper. This document is not self contained and references some of the symbols and formulae in the original paper.

Their factorization method is extremely neat and uses matrix operations (such as the Kronecker product and vectorization) to reorder existing matrix operations into their factorization of the Jacobian, which separates structure from motion. The factorization allows the precomputation of some of the quantities, which can help improve the timing for the tracking. Unfortunatley, these factored matrices (due to the use of indirect operations to compute the matrix product) are slightly bigger for each point's Jacobian, which means it is possible to achieve a slow-down when there are too many parameters. Their results demonstrate this as more parameters are added (they get a speedup of 6 times for a rigid body, and a speed-up of only 2 for a model with basis deformations).

The factorization does extend to multiple cameras, but I don't think it is possible to use it for a skinned mesh (although, originally I was sure that it could be). After fighting with the implementation and understanding for a few days, I started to have some concerns with the factorization approach. These concerns include some to do with the use of the gradient equivalence equation, ignoring a normalization factor in the factorization (the inverse of T), and some to do with speed.

2 Multi-cameras

The formulation for multiple cameras can follow their approach analogously, except that the projection into a camera view must take into account the camera's projection matrix. The object to be tracked will then be represented in some rest space (e.g., world coordinates), and its motion will be represented by a rigid transformation. The cameras will then have a rigid transformation to position them in this space.

When there are multiple cameras, with projection matrices, $\mathbf{P}^i = \mathbf{K}^i\mathbf{I}^{3\times4}\mathbf{E}^i$, a point $\mathbf{x}_r$ (in some known reference frame) undergoes the following transformation to take it to camera i's coordinates:

$$\mathbf{x}_t^i = (\mathbf{R}^i\mathbf{R}_t\mathbf{x}_r + (\mathbf{R}^i\mathbf{t}_t + \mathbf{t}^i)$$

which is the composition of $\mathbf{E}^i = [\mathbf{R}^i\mathbf{t}^i]$ and the object motion, $[\mathbf{R}_t\mathbf{t}_t]$

Given that the point $\mathbf{x}_r$ satisfies the plane equation $\mathbf{n}^T\mathbf{x}_r = d_r$, the homography between the reference view and the posed camera is:

$$\mathbf{u}_t^i = \underbrace{(\mathbf{R}_t^i\mathbf{R}_t + (\mathbf{R}_t^i\mathbf{t}_t + \mathbf{t}^i)\frac{\mathbf{n}_r^T}{d_r})}_{H_t^i}\mathbf{u}_r = f(\mathbf{u}_r, \mu_t)$$

where $\mathbf{u}_r = \frac{1}{z_r}K^{-1}\mathbf{x}_r$ and $\mathbf{u}_t^i = \frac{1}{z_t^i}K^{-1}\mathbf{x}_t$ are the projection in the camera views.

The brigthness constancy, $I_r(\mathbf{p}(\mathbf{u} - r)) = I_t^i(\mathbf{p}(f(\mathbf{u}_r, \mu_t)))$, is the same as the original paper (with $\mathbf{p}$ mapping projective to cartesian coordinates).

Tracking (finding the μ) is also formulated as a minimization problem, where the linearized problem is solved:

$$\mathcal{J}(\delta\mu) = \sum_i \|I_r(\mathbf{p}(\mathbf{u} - r)) - I_t^i(\mathbf{p}(f(\mathbf{u}_r, \mu_t))) - \frac{\partial I_t^i(\mathbf{p}(f(\mathbf{u}_r, \hat{\mu}_\mathbf{t})))}{\partial\hat{\mu}}\delta\mu\|$$

The authors show how the Jacobian in the above equation (later part of the equation) can be computed using a reference texture.

To do so, they first define a mapping from the texture space $\mathbf{T}[\mathbf{p}(\mathbf{v})]$ to the reference image (this reference image does not have to be a real image; I have taken it to be an image with the identity for the external matrix). Brightness constancy becomes, $\mathbf{T}[\mathbf{p}(\mathbf{v})] = \mathbf{I}_t^i[\mathbf{p}(\mathbf{f}'(\mathbf{v}, \mu_t))]$, where $\mathbf{f}'(\mathbf{v}, \mu_t) = H_t^i H_0\mathbf{v}$ and $\mathbf{v}$ is the texture coordinate. The Jacobian can then be written as:

$$J(\mu_t) = \underbrace{\left[\frac{\partial\mathbf{T}[\mathbf{p}(\hat{\mathbf{x}})]}{\partial\hat{\mathbf{x}}}\bigg|_{\hat{\mathbf{x}}=\mathbf{v}}\right]}_{D}\underbrace{\left[\frac{\partial\mathbf{f}'(\hat{\mathbf{x}}, \mu)}{\partial\hat{\mathbf{x}}}\bigg|_{\hat{\mathbf{x}}=\mathbf{v}}\right]}_{T^{-1}}\underbrace{\left[\frac{\partial\mathbf{f}'(\hat{\mathbf{x}}, \hat{\mu})}{\partial\hat{\mu}}\bigg|_{\hat{\mu}=\mu_t}\right]}_{T^{-1}}$$

The authors then factorize the Jacobian so it can be written of as a product of a constant component dependent on shape, $\mathbf{S}$, and a varying component based on motion, $\mathbf{M}$. Their factorization utilizes Sherman-Morrison formula to get an analytic derivative of T^{-1}

The factorization is similar in the multi-camera case, although due to the composition of the rotation matrices, there is a different $\mathbf{M}$ for each camera. This does not change the algorithm, as the final results can then be accumulated. Care must be taken when computing the derivatives, to ensure that the $\mathbf{M}$ matrix does include the camera rotation (due to the composition of the objects transformation with the camera transformation).

Also, there are some minor errors in their manuscript (e.g., missing an $\mathbf{H}_0$ in Eq. 10).

2.1 Skinned Meshes

My original thought was that each vertex of a skinned mesh, while undergoing some linear combination of Euclidean transformations (due to the skinning weight attachment to the mesh). This meant that each vertex could still be decomposed into a similar homography, for which derivatives could be taken w.r.t. either the $\mathbf{R}$ and $\mathbf{t}$ components and then through the chain rule, the derivatives

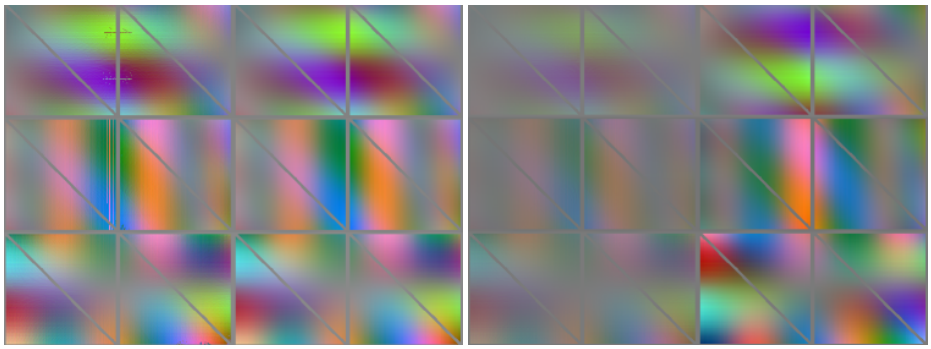

Figure 1: The derivative images when using normalization (left) vs. no normalization right. This is the derivatives for the rotation components (each row is a different rotational derivative represented in texture space). In each case the contrast for the image derivatives (which is the left hand side of each image) is scaled to be the same as the factorized derivatives. The change in scale visible on the right hand side shows that normalization does matter

w.r.t the joint angles could be obtained. These derivatives would be the same as the single or multiple camera case. Unfortunately, I think that the application of the Sherman-Morrison formula, which will obviously give a valid representation in terms of $\mathbf{R}^{-1}$, will not be useful in isolating the individual terms (which would be necessary to factor). In the end, I think the factorization would end up needing a different $\mathbf{M}$ for every point, which is useless.

3 Concerns

3.1 Normalization

I have some issues with the $\propto$ in the sherman-morrison approximation. This is obviously done to avoid the division by a normalization factor that is dependent on both rotation and structure: I call this the normlization factor. I don't think that it makes sense to just through out this factor. I have verified this experimentally: this factor does affect the scale of the derivatives (see Fig 1). Luckily, this scale could easily be updated and introduced into their algorithm on each iteration (into the structure terms before composing them). Unfortunately, I also was so far unable to reproduce their speedup factor. The computation of the Λ matrix seems to be one of the most expensive parts. The good news is that the summed Λ matrix only changes when visibility changes, so the expense could be reduced dramatically. The bad news is if the normalization factor is taken into account, then the Λ terms would need to be updated as they depend on $\mathbf{S}$ (e.g., individual terms should be normalized by the square of the normalization factor).

3.2 Gradient Equivalence Equation

I also have some problems with the use of the gradient equivalence equation. This first says that you can use the derivative in image I_r to approximate the derivative in image I_t. It further says that the image derivative in I_t is equivalent to one taken in the texture space. To me this equation only holds when the images are aligned. I have also verified this experimentally. This doesn't seem to be as strong as the condition you would have in e.g., the inverse compositional method. I have a feeling this could give worse convergence. Not sure why this isn't mentioned or perhaps I have misinterpreted the paper.

3.3 Speed/Timing/Matrix Operations

Even if it were possible to perform the factorization for the skinned mesh, I suspect that there would be little speed gain; the authors report only a 2 times speed-up for the deformable mesh. Unfortunately, they don't mention how many basis functions they use, and it seems that the size of the factorization will directly depend on these numbers. They also don't mention the sizes of the factorization matrices in their argument, nor do they mention the number of discrete points used in the tracking; I suspect that the benefits of the factorization starts to show as their become more points. Unfortunately, if my calculations are correct, the Λ matrices are 210×210, meaning that it is going to start to become memory intensive with too many points.

In the case of color images, with n points and inoring visibility. $\mathbf{S}$ is 3×210, and $\mathbf{M}$ is 210×6. $\mathbf{M}$ is clearly mostly sparse, meaning that $\mathbf{A} = \mathbf{M}^T\mathbf{S}^T\mathbf{e}$ can be computed using only the blocks of $\mathbf{M}$. In the end, their algorithm will be dominated by the computation of $\mathbf{e}$ and Λ. In non-sparse case $\mathbf{e}$ will be on the order of $n(6 \cdot 210 + 210 \cdot 3) = 1830n$, and in sparse it will be $n(210 \cdot 3 + 3 \cdot 63 + 3 \times 21) = 883n$ (not including the additions to compute big $\mathbf{A}$. But Λ is 210×210 which requires $n(210 \cdot 210) = 44100n$ additions. Λ is kind of sparse (about 50%), so this gives $22000n$ additions. Again, if Λ is only updated when visibility of a point changes, we can assume that its influence is negligable. So the computation is dominated by computing $\mathbf{A}$; things outside the loop, including computation of $\mathbf{N}$ and its inverse will take on the order of $6 \cdot 210 \cdot 210 + 6 \cdot 210 \cdot 6 + 6^3$, which again is negligable if $n > 210$.

In the unfactorized case, we need to compute J^TJ and J^Te. J is $3n \times 6$, so $J^TJ = 3 \cdot 6^2 n = 324n$ and $J^te = 6 \times 3n$ for a total of $342n$. This of course doesn't take into account the time needed to compute the actual jacobian matrix. If Eq. 8 from the paper is used, J can still be computed in texture space with the multiplication of a 3×3 matrix D with a 3×3 matrix T^{-1} and a 3×6 matrix F, for a total of $(27 + 63)n$. So the unfactorized case is going to take roughly $(342 + 90)n = 432n$. This is roughly twice as few operations as the factorized case, so unless there is some other way to gain from sparsity, I cannot see how this factorization can aid. We can even compute J using the image derivatives directly, which avoids the gradient equilvalence (which I believe to be more stable). In this case, even if we perform finit difference to get the

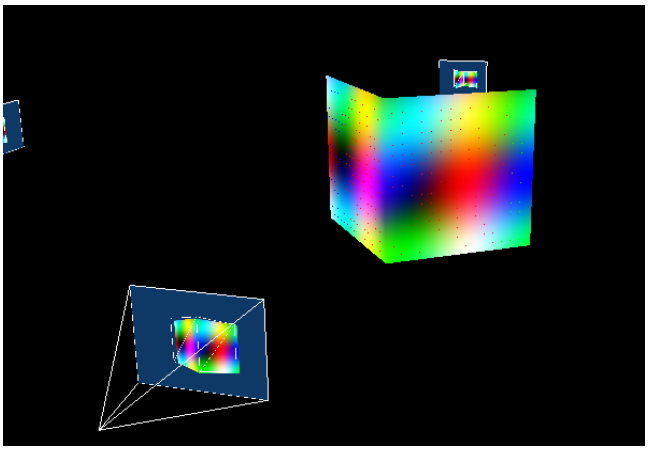

Figure 2: The synthetic setup illustrating a few of the cameras and the shape of the synthetic object

derivative of the point in the image, it will only take 24 multiplications, 18 additions, 4 divisions, and 2 subtractions to get the point derivative, plus $3 \cdot 2$ operations to modulate the derivative with the point derivative, for a total of $6(54)n = 324n$, which is even less than using the texture derivative (with the exception computing image derivatives, which need not be computed over the entire image).

This argument was loose, and mostly assumed that the addition/multiplication in a matrix multiplicatoin to be one operation. I still don't see how the factorization can really improve things. Perhaps I should have been more careful in my analysis, but I think the dimensions are all accurate; the only thing missing would be the explicit separation of addition/multiplication.

In my code, with roughly 7000 points (which nearly exhausts the 4GB of memory to store $\mathbf{S}$ and $\mathbf{\Lambda}$), I found that each iteration took roughly $0.035s$ and $0.013s$ for the factorized and unfactorized code respectively. You will notice that $0.035/0.013 = 2.6923$ is very close to the ratio of times predicted by the argument above (I used a jacobian in image space), which gives a ratio of $883n/(324n) = 2.7253$. The above was without update Λ, when updating Λ the factorization approach is much slower, taking about 1.6s per iteration. Taking the ratio of the times $1.6/0.035 = 45$, we see that this ratio is close to the ratio of operations performed $(44100 + 883)/883 = 50$.

4 Experiments

To test the implementations, I generated a data set of 120 images, taken from 4 cameras (each had 120 images) of size 800x600 (see Fig.2). The model is two planes that are not parallel. Visiblity is used (only front facing). The sample points inversely proportional to number of cameras (as the factorization requires

	4 camera case	
	Image Jacobian	Factored
grad	47 ms	–
vis	37 ms	37
track	50 ms	240ms
total	134 ms	277 ms

	1 camera case	
	Image Jacobian	Factored
grad	47 ms	–
vis	10 ms	10
track	5 ms	500
total	52 ms	510 ms

Table 1: Time to track a frame for the two methods. The factorization method is actually slower. Keep in mind that these times take into account early breaks. The case is even worse for the single camera case, most likely due to updating of Λ

caching some quantities, actually run out of memory). Each tracker was run with a max iterations of 20 per time frame, and they break early if update size is small. Visibility was updated every 2nd time step. For the factorization approach, a full update of Λ was performed every Full update of Λ every 10 tracking iterations. Partial updates of Λ, corresponding to points that changed visibility, were performed whenever visibilty is estimated.

I ran the trackers on the sequence, using both the full 4 cameras and only the first camera. In the four camera case there were 420 sample points[1]. In the one camera case there were 6384 points. In earlier tests the factorization approach was able to track, but this sequence has quite fast motion and the tracker gets lost pretty easily. The image-derivative based Jacboian does have a minor mistrack but is able to recover. Qualitatively, the factorized derivative (based on gradience equivalence) seemed to perform worse (probably due to the gradience equivalence). Table 1 shows the timing results. Again, like the discussion above, the factorized approach is also slower. Timings don't take into account loading images (which was on the order of several 12's millis for 4 images and 4 millis for a single view case).

See http://www.neilbirkbeck.com/ for videos.

5 Conclusions

It is indeed possible to use the factorization approach for a multi-camera setup, but I suspect that the same is not possible for a skinned mesh.

I am not entirely sure what I have done differently to make my factorized Jacobian slower than the standard approach. I am certain that the factorized Jacobian is the same as the other Jacobian (at least when the tracking parameters are exact, as stated by the gradient equivalence equation), as this is how I started this project. I also still have residual concerns about how well the gradient when computed in texture space compares to the one that is computed in the image space. Again, the one computed in image space must be accurate,

[1]Not exactly inversely proportional, should have divided the spacing by sqrt of number of cameras

on the other hand the one computed in texture space is completely indpendent of the image!

Regarding the slowness, I can see how without computation of Λ it might be possible to get an implementation that is faster (although my results do not reflect this). Unfortunately, as I verified in my experiments, the normalization factor, which does affect the Jacobian, does change, and does influence Λ, implying Λ should be updated.

Another downside to the factorization approach is its complexity. The standard Jacobian takes only a matter of minutes to implement (if that), whereas getting the latter correct could take several hours.