4 Matchmove

Our last topic of the chapter isn't generally classified as Inverse Kinematics, but turns out to be nearly the same problem in a slightly different setting.

The next chapter is all about rendering and compositing for animation, for example touching on how to integrate digitally rendered images into real footage (plates). While there's a lot to say about matching lighting conditions and shadows, motion blur, etc. and some interesting work in figuring out which pixels are foreground or background (opaque or transparent, or in between), the first and foremost problem is making sure the virtual camera matches up to the real camera that took the plate.

For example, suppose a digitally rendered human is supposed to be added on top of a photo of a city square, appearing as if they were standing on the ground next to a real person. If the digital addition is rendered with the wrong size, or their feet show up in the wrong spot on the image, or from more of a top-view than the real person, or more subtly with a different amount of perspective foreshortening, the illusion will utterly fail. Audiences will either see the problem and laugh at the ineptitude of the effects, or perhaps will just get subconsciously uncomfortable with something "not being right". If the camera is moving, the tolerance for error is even less — an inconsistent virtual camera will show the rendered objects sliding over the screen in a strange unreal way.

The procedure of matching up the virtual camera to the real camera is called *matchmove* (with a special emphasis on the harder problem of moving cameras). "Matching" is a little vague, however: what does it mean for a virtual camera pointed at a virtual object to match a real camera pointed at an entirely different real scene?

More precisely, we need to establish a common "world space" coordinate system between the real scene and the virtual, with the position and orientation (and any other parameters) of the real and virtual cameras with

respect to this coordinate system being equal. To make a common world space, we have to have points of correspondance: a set of points in the real world that correspond to points in the virtual world, whose world space coordinates must therefore be the same.

4.1 Markers

The points in correspondence between real and virtual scenes have to be special "landmarks": we must be able to accurately locate them in the plate images. For example, a point somewhere in the middle of the air is useless because there's no way to easily figure out where it is in an arbitrary image. A point in the middle of a clean white wall is similarly worthless — it's impossible to tell apart from any other point on the wall (at least, without first figuring out where the edges of the wall are). A point on the edge of something, where the colour in the image changes sharply between the two sides of the edge, is better, but still can't be localized to more than being somewhere on a line segment or curve in the image. A much more useful choice is a **corner** of something, for example where two edges intersect — particularly corners with a very sharp and large change in colours in different directions, such as the corners of the squares in a black and white checkerboard.

Points that we can easily and precisely locate in a plate, and that we use to solve matchmove or similar problems, are called *markers*. The two-dimensional image positions of the markers in the real footage are the primary input data to matchmove. Let's call these 2D image positions $\{\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n\}$: for n visible markers we get $2n$ numbers. It's important to be clear about what exactly these coordinates are: Normalized Device Coordinates, ranging from $(-1, -1)$ at the bottom left corner of the image to $(1, 1)$ at the top right corner, are the obvious choice.

There are a couple of ways to determine each $\vec{p}_i$. For example, a matchmove artist could manually look through the image and click on where

they judge the i'th marker to be, recording the click as $\vec{p}_i$. Alternatively, computer vision algorithms (more on this below) could automatically attempt to find it. A combination of manual and automatic methods could also be used: the artist quickly placing a blob over the rough location of the marker, and computer vision algorithms optimizing the precise location within that blob.

The larger problem is figuring out where the markers are in world space, i.e. their 3D coordinates in "world space". We'll denote these three dimensional coordinates as $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$. There are a few possibilities here too.

The simplest approach (software-wise) to getting the $\{\vec{x}_i\}$ is from *survey data*: in addition to filming the actual footage, the crew can identify good markers in the scene and measure exactly where they are with respect to an agreed coordinate system using surveying tools. For example, if a rectangular table occupies the centre of the image (on which a digital rendered object is going to be placed), one corner of the table might be selected as the world space origin, two of the edges the x and z axes; all four corners can be taken as markers, and their positions in this coordinate system accurately measured with a measuring tape. Note that keeping track of units is very important in cases like this: you don't want to get mixed up between numbers that indicate metres versus other measured numbers that indicate centimetres, or inches, or cubits.

A variation on this idea is to artificially add a *calibration target* to the scene, an object with good markers on it that has already been carefully measured. For example, a checkboard with some markings to indicate which way round it is (or something a bit more sophisticated like CALTag [AHH]) would be a good candidate for a target. This is especially useful in a scene where there are no obvious natural markers, such as on a uniform grassy field, or more typically, on a green-screen studio: instead of relying on natural markers, the crew can add their own. However, it will have to be removed from the final shot, of course, perhaps by compositing a rendered object on top of it, or simply masking it out (if it's on a green screen which

is going to be removed anyhow).

Finally, in some scenarios it may be impossible to do survey data or add a calibration target: for example, a fast-moving helicopter shot over the wilderness. In that case the problem is a lot tougher, and while computer vision techniques can suggest good likely markers from the images themselves, their 3D locations will have to be solved for simultaneously with the camera parameters: we won't tackle that problem here.

Assuming either survey data or calibration target, we end up with the markers providing two sets of point data for the matchmove problem: for each marker i a 3D world space location $\vec{x}_i$ and a corresponding 2D image space projected position $\vec{p}_i$.

4.2 The Solve

In the real photo, the "action" of the camera was to project the 3D world space marker positions $\{\vec{x}_i\}$ to the 2D image space positions $\{\vec{p}_i\}$. We can model this mathematically the usual way, as a transform T from world space to Normalized Device Coordinates (without the depth value, which we don't know). This transform can be factored into a sequence of simpler rigid transforms (the model-view matrix which goes from world space to camera space) and a perspective projection:

$$T = P_{\text{persp}} R_z(\theta_6) R_y(\theta_5) R_x(\theta_4) T_z(\theta_3) T_y(\theta_2) T_x(\theta_1). \tag{47}$$

Here I use P_{persp} for the perspective projection, $R_z(\theta_6)$ for a rotation by an unknown angle parameter θ_6 around the local z-axis, $T_z(\theta_3)$ for translation by an unknown distance parameter $\theta_3)$ along the local z-axis and so forth. In essence, this is just like a 6DOF joint, except for the inclusion of a perspective transform at the end.

I will assume the perspective projection P_{persp} is known. This is an intrinsic setting of the camera lens system: what the camera's field of view is.[2] It

[2]For the special case of a tilt-shift camera, P_{persp} should also encode where on the image

could in principle be recovered as another unknown paramter to solve for, but in practice this is numerically quite ill-behaved (the effect of zooming in is almost the same as moving the camera towards the scene, so errors in the data make it very hard to reliably distinguish between perspective effects and camera translation). Moreover, even if the camera moves during a shot, the lens is almost always left constant and can separately be measured or calibrated (i.e. P_{persp} can be figured out separately from matchmove).

We are left with $2n$ equations to solve in 6 parameters,

$$\begin{aligned}
\vec{p}_1 &= PR_z(\theta_6)R_y(\theta_5)R_x(\theta_4)T_z(\theta_3)T_y(\theta_2)T_x(\theta_1)\vec{x}_1, \\
\vec{p}_2 &= PR_z(\theta_6)R_y(\theta_5)R_x(\theta_4)T_z(\theta_3)T_y(\theta_2)T_x(\theta_1)\vec{x}_2, \\
&\vdots \\
\vec{p}_n &= PR_z(\theta_6)R_y(\theta_5)R_x(\theta_4)T_z(\theta_3)T_y(\theta_2)T_x(\theta_1)\vec{x}_n,
\end{aligned} \tag{48}$$

where each marker gives two equations (one for the image x coordinate and the other for the image y coordinate). Thinking of this as a 6DOF joint, this is obviously just a small variation on IK — and can be solved with exactly the same regularized nonlinear least squares form, Gauss-Newton, line search etc. It does tend to be important to start with a good initial guess for matchmove, as it's easy to get stuck in some bad local minima far from the real solution: user hints (such as a rough approximation of the position of the camera) and heuristics (the camera has to be rotated to be facing the markers visible in the image, the camera probably is oriented with its image x axis horizontal in the world) may be necessary.

One issue that is a bit different between matchmove and regular IK is the nature of the markers. With less than three markers, i.e. less than six equations, there's no way the six camera DOFs can be found; geometric consideration shows there are also some degenerate arrangements of markers which don't suffice. In some frames, some markers may also be obscured

plane the image was recorded etc. For wide-angle and zoom lenses, P may also include a nonlinear distortion to account for "barrel" or "pincushion" distortions in the real camera that deviate from the perfect perspective transform we usually assume in computer graphics. We'll ignore these cases here.

and impossible to locate. Finally, whether artist or algorithm is locating the markers in the images there is always a possibility there can be errors in the $\vec{p}_i$. (Errors in the $\vec{x}_i$ are also possible, of course, but less likely if care is taken in measurement.) All of this argues for using a lot more markers than the three which are strictly necessary. In particular, least squares "averages" across the data points in some sense, hopefully reducing the impact of errors.

Having said that, not all errors are equal. Least squares does a good job handling small and unbiased perturbations in the $\vec{p}_i$ by, say, less than a pixel. Occasionally, however, an artist's hand might twitch and click a massively wrong point, or a computer vision algorithm will fail dramatically and locate a marker far from where it is — or even switch two marks around. These wild data points are called *outliers*, and basically encode no useful information at all: it's better to take them out of the problem rather than risk them screwing up the least squares solution in an unpredictable way. The problem, however, is how to identify which points are outliers and which are valid.

The *RANSAC* algorithm is a popular solution to this problem in general. Rather than solve the least squares problems with all the data points, only a small randomly selected subset are used. The solution (camera transform in this case) that is found is then checked against the remaining data points: if a large enough fraction agree with only a very small error, we trust that the subset managed to avoid outliers — then discard any points that strongly disagree, and solve the problem again with all the good points. On the other hand, if the solution doesn't agree well with most data points, we try again with a new randomly selected subset. RANSAC can run for many iterations, keeping a record of the closest fit found (with an acceptably small number of outliers thrown out).

References

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[AHH] Bradley Atcheson, Felix Heide, and Wolfgang Heidrich.

[GMHP04] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 23:522–531, 2004.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. The Johns Hopkins University Press, 3rd edition, 1996.

[NW99] J. Nocedal and S.J. Wright. *Numerical optimization*. Springer series in operations research. Springer, 1999.