6 Camera Moves

By this point, we have all the basic tools for animating objects in a scene. Rendering the animation—whether for the final shot or for interactive visualization as the animator works—needs a little more though: lights and cameras.

We won't dwell on lighting design, but simply note that placing and animating lights in a scene generally uses the same basic tools of motion curves and transforms as Forward Kinematics. There probably is more to be done here (in modeling software) on the interactive visualization side: perceptual studies have shown that shadows, for example, can play a big role in helping user perception of shape, size and relative placement. Due to the expense and complication of rendering shadows in real-time most modeling software doesn't bother with them, yet, but maybe should.

6.1 Animated Cameras

Animating a shot camera, through which the final result will be rendered, is much the same as animating any object: though there's a lot to worry about artistically, cinematographically, the most basic tools of rigid transforms aren't very different. People generally do have a strong sensitivity to upwards orientation—cameras that have "rolled", so that the image's horizontal axis is no longer aligned with the 3D world's notion of being horizontal, send an enormously powerful message to the viewer that things are off-balance or are being artistically framed for some extremely important reason, so there better be an important reason to do this. Therefore the Euler angle model of camera rotation generally uses a rotation around the world's y axis and a rotation around the camera's x axis, thus maintaining the sense of proper horizontality. A final roll angle around the camera's z axis should be left at zero unless really needed.

An alternate method of posing a camera that can be more intuitive in many

cases is the *look-at* model, which you should have seen in an earlier graphics course. Here the position of the camera $\vec{p}$, the position of a target $\vec{t}$ which the camera should be pointing at, and the world's y-axis are given, from which the rigid transform of the camera can be computed. In particular, the camera's z axis should be $\vec{p} - \vec{t}$ normalized to unit-length[3], the camera's x axis can be computed from a normalized cross-product of that with the world's y axis, and the camera's y axis computed from a final cross-product to be orthogonal. In an animation setting, the camera position and the target can then be separately animated with motion curves. (This actually touches on the subject of the next chapter, specifying motion in alternate ways based on the output instead of simple parameterized input...)

The biggest difference from camera motion curves and regular object or light motion curves is how important smoothness usually is. While occasionally a jerky camera is artistically valid—to communicate a sense of speed, disorientation, or impact, or perhaps to simulate amateur hand-held camera footage—it is not something to use casually. Apart from objects that are locked to the camera's motion, everything rendered in the scene can only appear to move as smoothly as the camera itself moves. One "kink" in the camera's motion will be perceived as every pixel in the image getting bumped, which throws audiences off balance.

In the previous chapter we saw how cubic Bézier splines can easily be made to be C^1 smooth, but this might not be quite adequate for a camera: the acceleration of the camera (the second derivative of its path) jerks discontinuously across knot boundaries in general, since we don't have C^2 smoothness. We also mentioned that B-splines permit construction of C^2 cubic curves, at the cost of losing interpolation and having to make do with approximation. An alternative is to directly apply a smoothing formula to the Bézier spline control points, making each a weighted average of the surrounding control points—there are several possibilities to consider in this

[3]Note that we want the camera's z-axis to point directly opposite from where it's pointing, if we are to keep the image x axis pointing to the right, the image y axis pointing up, and the camera transform to be right-handed.

case, including setting them to match a B-spline curve.

This is also very relevant for automatic cameras in video games. Some games naturally have a set camera: on the character for a first-person shooter, or maybe just fixed in the world for a non-scrolling 2D game. However, as soon as the camera is supposed to be in a third-person perspective and must move to follow the play, the question of smooth camera path comes up. Without getting into more sophisticated artifical intelligence methods which may try to optimize the viewpoint for a desired effect (e.g. making sure there are no obscuring objects in the way, presenting action in silhouette if possible, preserving a natural mapping from the player controls to what is seen on screen), the most basic approach is to fix the camera in the rigid coordinate system of the player's character—for example, it might always be over the left shoulder and a bit behind the player. If the character's motion is nicely responsive to player controls, it probably can be very abrupt and jerky—which we don't want to see in the camera. Therefore the actual camera path should aim to smoothly follow the desired placement instead of snapping to it at all times, either using spline contruction or even just moving, say, 20% of the distance towards the target each frame.

6.2 Interactive Camera Control

A big part of the modeling and animating process involves working on 3D models independent of what the actual shot cameras will be for the final scene; fast 3D rendering / visualization of the models is critical for the artists to be able to work, including controlling the camera for the visualization.

This problem, called generically "3D navigation", is essentially as old as real-time 3D rendering. It's tricky because we expect our camera to have 5DOF (ignoring "roll") but the classic mouse has only 2DOF: there's no way we can map a two-dimensional plane of movement to a whole five-dimensional space of rigid transforms.

One possibility that's becoming central today is to use hardware with more degrees of freedom. For specialized scenarios, more exotic 3D devices like virtual reality gloves or accelerometer-equipped objects (now found in games consoles and smartphones) are a perfect fit. However, these and older ideas have a common failing in that they don't integrate easily with the less 3D parts of a standard user interface—trying to select menus or buttons by pointing in mid-air is tough. Most 3D modeling and animation programs have a multitude of regular menus and buttons, exist within a regular GUI system with standard file dialox boxes etc., and are expected to run concurrently with non-3D software like mail clients, web browsers, movie players and text editors. No device that makes that too hard can succeed.

The two non-mouse input devices which are in common usage already, and offer real advantages over mice for 3D navigation while still being usable for regular GUI tasks, are stylus-equipped tablets and touch screens.

Almost all 3D artists already have a stylus and tablet, which is heavily exploited in paint and paint-like programs. In addition to the two coordinates of where the stylus touches the tablet, a good device will also precisely measure the pressure being applied at the tip, two angles of tilt of the stylus body with respect to the tablet, and even the "barrel" rotation along the axis of the stylus, for a total of five or six input DOF. While it takes a little getting used to, many artists use the stylus exclusively even in regular applications, as a mouse replacement. However, directly mapping the device's DOF to the translation and Euler angles of a simple camera is probably useless: for one, human muscle control of the different DOF varies significantly [XBR11]. See Bridson's SpikeNav interface [Bri09] for an example of how to better exploit a stylus for this purpose, for example.

Another higher DOF input device which is common on some platforms, and increasing in popularity on others, is multi-touch screens. It's a little hard to even define how many DOF this represents. In some scenarios an application may only get a small list of contact points (one per finger), but

this still represents potentially 12 DOF assuming two fingers and a thumb from each hand can be moved independently in x and y enough to be useful. Other devices or API's can report a surprisingly high resolution image of the pressure applied across the entire screen, such as Rosenberg et al.'s UnMousePad [RP09]. Reisman et al. [RDH09] provide an example approach to using a few fingers from each hand to intuitively manipulate the location and orientation of rigid bodies—which is essentially equivalent to camera control (thinking of the entire scene as a rigid frame relative to the camera).

Another possibility on the horizon is to use computer vision approaches to accurately track the user's hands in front of the computer from camera input. Schlattmann and Klein [SK09] demonstrate how their markerless hand-tracking technology can be used for 3D object manipulation. While this kind of interaction doesn't map easily into existing GUI controls that assume a mouse or similar input, one could imagine this coexisting easily with a mouse or stylus at an artist's workstation—much as the keyboard already coexists.

Returning to the status quo today, however, we are left with the problem of mapping a 2DOF mouse (or equivalent) to controlling a 5DOF camera. We clearly can only control two of the five at a time, so there must be a way to switch between "modes" of interacting, selecting which two DOF are actively controlled. This could be as obvious as displaying a widget on-screen with different control areas—e.g. click on the straight arrows to drag the camera with translation, or on the curved arrows to drag the camera's rotation angles.

A more opaque but potentially faster (for users who have learned it) method is to use different mouse buttons or different keys to hold down (shift/control/alt/etc.) to change what the $x - y$ movements of the mouse mean. This leaves the choice of what exactly those $x - y$ mouse movements should do. For a "translation" mode, a direct mapping to translation in a plane parallel to the camera's image plane, is clearly the most obvious.

There's more to write here: it's a big and tricky subject. Refer to lecture for some more specifics.

References

[Bri09] Robert Bridson. SpikeNav: using stylus tilte in three-dimensional navigation. In *Proc. ACM UIST 2009 Posters*, 2009.

[KCZO08] Ladislav Kavan, Steven Collins, Jiri Zara, and Carol O'Sullivan. Geometric skinning with approximate dual quaternion blending. volume 27, page 105, 2008.

[LCF00] John P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proc. SIGGRAPH 2000*, pages 165–172, 2000.

[MZS+11] Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 30:37:1–37:12, 2011.

[RDH09] Jason L. Reisman, Philip L. Davidson, and Jefferson Y. Han. A screen-space formulation for 2d and 3d direct manipulation. In *Proc. of the 22nd annual ACM symposium on User interface software and technology*, UIST '09, pages 69–78, New York, NY, USA, 2009. ACM.

[RP09] Ilya Rosenberg and Ken Perlin. The UnMousePad: an interpolating multi-touch force-sensing input pad. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 28:65:1–65:9, July 2009.

[SK09] Markus Schlattmann and Reinhard Klein. Efficient bimanual symmetric 3d manipulation for markerless hand-tracking. In *Virtual Reality International Conference (VRIC)*, 2009.

[XBR11] Yizhong Xin, Xiaojun Bi, and Xiangshi Ren. Acquiring and pointing: an empirical study of pen-tilt-based interaction. In *Proc. 2011 annual conference on Human factors in computing systems*, CHI '11, pages 849–858, New York, NY, USA, 2011. ACM.