Recap: Last time we introduced the principal elements of skeletal models. Today we will discuss methods for animating these models. We will also discuss the issue of how to cover these models with “skin,” and how to move the skin smoothly as part of the animation.

Recall that a skeletal model consists of a collection of joints, which have been joined into a rooted tree structure. Recall that a pose is defined by rotating each of the joints in a specific way. Rotations are defined relative to a default pose, called the bind pose or reference pose. The mesh defining the character’s skin is defined relative to this pose. (Typically this is the character standing upright with arms outstretched to the sides.) Each node $j$ of this tree is (either implicitly or explicitly) associated with the following relevant transformations:

**Local-pose transformation**: $T[p(j)←j]$ converts a point in $j$’s coordinate system to its representation in its parent’s coordinate system. Its inverse, $T[j←p(j)]$, converts from the parent’s frame to the joint’s frame.

**Bind-pose transformation**: (also called the global pose transformation) $T[M←j]$ converts a point in $j$’s coordinate system to its representation in the model’s global reference coordinate system $M$. It can be computed as the product of the local-pose transformations from $j$ up to the root. Its inverse, $T[j←M]$, converts from the model’s frame to the joint’s frame.

**Animating the Model**: There are two natural ways to specify a skeletal’s model pose:

**Forward kinematics**: You specify the position of the model (both its location and rotation) and the angles by which the joints are to be rotated. The system then places the model at the desired position and applies these rotations, which fixes the positions of all the model’s joints.

**Inverse kinematics**: The problem with forward kinematics is that forces the designer to determine which joint angles achieve the constraints that he/she wants to impose. For example, if we know that the object is to have both feet at certain positions on the ground and is reaching with its right hand for a door knob, shouldn’t the system be able to determine the joint angles for us?

In *inverse kinematics*, the designer provides the system with a set of constraints that are to be satisfied, and the system’s job is to determine a set of joint angles so that the resulting pose achieves these constraints. Typically, these constraints take the form of specifying where the endpoints (that is, leaves of the skeleton tree) are to be placed, and it determines the intermediate joint angles to achieve these constraints.

Inverse kinematics is clearly a more convenient option for a designer, and many high-end solid modelers provide the feature. In general, inverse kinematics is much more challenging to implement than forward kinematics. There are many reasons for this. First, the inverse kinematics problem is simply a more complicated mathematical problem to solve than forward kinematics. Forward kinematics systems require little more than the ability to perform matrix multiplications. The inverse involves...
solving a nonlinear constrained optimization problem. Another reason is that a typical system of constraints is under-specified, which means that there may be multiple (often infinitely many) valid solutions. An inverse kinematics system would either need to establish which of these solutions is most “natural” (whatever that means), or it will need to request further information from the designer to add additional constraints. For example, the system may assume that joint angles that are close to their reference-pose values are of lower energy than those that involve higher turning angles, and it will attempt to find a pose that satisfies all the constraints and is of the lowest possible energy.

By the way, there is a third way of obtaining joint angles in animation. This is through the process of motion capture. Markers are placed on a subject, who is then asked to perform certain actions (walking, running, jumping, etc.) By tracking the markers using multiple cameras or other technologies, it is possible to reconstruct the positions of the joints. From these, it is simple exercise in linear algebra to determine the joint angles that gave rise to these motions. Motion capture has the advantage of producing natural motions. (Of course, it might be difficult to apply for fictitious creatures, such as flying dragons.)

Let us make the simplifying assumption that our poses and animations are specified using forward kinematics. In order to specify an animation, we need to specify how the joint angles change over time. At the most general abstract level, we can think of each local-pose transformation, $T_{p(j) \leftarrow j}$, as being a function of time. For any given time $t \geq 0$, this function would tell us how to transform a point from $j$’s local coordinate frame to its parent’s frame. If we knew these functions for all the joints of our system, at any time $t$ we could apply matrix multiplication to compute the bind pose transformation, $T_{M \leftarrow j}$. From this, we know how any point defined in $j$’s local frame is positioned relative to the model’s frame.

Unfortunately, this abstract view is far too general and lends itself to practical implementation. There are many approaches that can be applied to reduce the complexity to an acceptable level:

**Sample:** Rather than storing the animation in functional form, we will compute a collection of discrete samples at specific times. We can think of these as corresponding to the key frames of a key-frame animation. We then use some form of interpolation to smoothly fill the gaps between successive key frames.

**Eliminate redundancy:** Storing a full $4 \times 4$ linear transformation can be space inefficient. If we know that a joint can rotate only about a single fixed axis, then we need only store the angle of rotation. Since different joints have different rotation characteristics, is there a general way to handle this? A clever trick that can be used to store joints with multiple degrees of freedom (like a shoulder) is to break the into two or more separate joints, one for each degree of freedom. These meta-joints share the same point as their origin (that is, the translational offset between them is the zero vector). Each meta-joint is responsible for a single rotational degree of freedom. For example, for the shoulder one joint might handle rotation about the vertical axis (left-right) and another might handle rotation about the forward axis (up-down) (see Fig. 1). Between the two, the full spectrum of two-dimensional rotation can be covered.

**Compress:** If movements are small, the difference of joint angles over successive key frames may be very small. If so, it is not necessary to use all 32 bits of a floating-point value to store the rotation angle. Instead, it may suffice to store a low-precision integer that indicates the number of degrees of rotation. Expressed in terms of information theory, smooth motions have
Fig. 1: Using two meta-joints (b) to simulate a single joint with two degrees of freedom.

low information content, that is, low entropy. There are many data compression techniques that have been developed in the fields of signal processing and information theory for storing low-entropy streams efficiently.

**Simple Animation Representation:** There are a number of different file formats that are used for describing animations. These are based on various combinations of the above ideas. In Fig. 2 we give a graphical presentation of a animation clip. Let us consider a fairly general set up, in which each pose transformation (either local or global, depending on what your system prefers) is represented by a 3-element translation vector \((x, y, z)\) and a 4-element quaternion vector \((s, t, u, v)\) to represent the rotation.

![Time samples and Linear interpolation diagram](image)

Fig. 2: An uncompressed animation stream.

The clip is discretized by taking samples at equally spaced intervals. (The interval size is generally longer than the frame rate.) We may then interpolate linearly between consecutive samples at the frame rate of the display to obtain the transformation at any given point in time.

If the sampling is sufficiently dense, and the motion is sufficiently smooth, then linear interpolation
works well enough. If you want to save more space by sampling more sparsely, you may find it useful to consider more sophisticated methods of sampling.

**Spline reconstruction:** There are methods in geometric modeling for extracting a smooth curve from a set of sample points. These systems go under various names, Bezier curves, B-splines, Hermite splines.

**Spherical interpolation:** Some quantities, such as quaternions, are by definition, vectors of unit length. When you interpolate linearly between two vectors of unit length, the result is a vector that is of less than unit length. (The linear interpolant travels along the straight line curve between the point, not along the great circle of the unit sphere as it should.) There are efficient computational methods for performing spherical interpolation. We will not discuss these, but just so you are aware of standard jargon, _lerp_ means linear interpolation and _slerp_ means spherical interpolation.

**Auxiliary Information:** It is sometimes useful to add further decorations to your animation, which are not necessarily related to the rendering of the moving character. Examples include:

**Event triggers:** Discrete signals sent to other parts of the game system. For example, you might want a certain sound playback to start with a particular event (e.g., footstep sound), a display event (e.g., starting a particle system that shows a cloud of dust rising from the footstep), or you may want to trigger a game event (e.g., a non-playing character ducks to avoid a punch).

**Continuous information:** You may want some process to adjust smoothly as a result of the animation. An example would be having the camera motion being coordinated with the animation. Another example would be parameters that continuously modify the texture coordinates or lighting properties of the object. Unlike event triggers, such actions should be smoothly interpolated.

This auxiliary information can be encoded in additional streams, called _meta-channels_ (see Fig. 2). This information will be interpreted by the game engine.