CMSC 425: Lecture 15
Animation for Games: Animation and Skinning
Tuesday, Apr 2, 2013

Reading: Chapt 11 of Gregory, *Game Engine Architecture*.

Recap: In the last couple of lectures we introduced the principal elements of skeletal models and how to represent animations. 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 a the following relevant transformations:

Local-pose transformation: $T_{[p(j)\leftarrow j]}$ converts a point in j's coordinate system to its representation in its parent's coordinate system. Its inverse, $T_{[j\leftarrow p(j)]}$, converts from the parent's frame to the joint's frame.

Bind-pose transformation: (also called the *global pose transformation*) $T_{[M\leftarrow j]}$ 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, $T_{[j\leftarrow M]}$, converts from the model's frame to the joint's frame.

Animation by forward kinematics: We assume that an animation is represented by a function that maps time to joint angles, or more generally, the transformations that relate one joint to another. This function is represented by means of a collection of *channels*, each of which stores a sequence of sampled transformations, one per joint. Through the use of various interpolation methods (linear or spherical, piecewise or smooth) we can determine the exact joint-to-joint transformation at any desired time instant.

Skinning: Now that we know how to specify the movement of the skeleton over time, let us consider how to animate the skin that will constitute the drawing of the character. The first question is how to represent this skin. The most convenient representation from a designer's perspective, and the one that we will use, is to position the skeleton in the reference pose and draw the skin around the resulting structure (see Fig. 1).

In order that the skin move smoothly along with the skeleton, we need to associate, or *bind*, vertices of the mesh to joints of the system, so that when the joints move, the skin moves as well. (This is the reason that the reference pose is called the bind pose.)

As we mentioned earlier, if we were to bind each vertex to a single joint, then we would observe *cracks* appearing in our skin whenever neighboring vertices are bound to two different joints that are rotated apart from one another. To overcome this, we allow each vertex of the mesh to be bound to multiple joints. When this is done, each joint to which a vertex is bound is assigned a *weighting factor*, that specifies the degree to which this joint influences the movement of the vertex. Thus, each vertex of the mesh is associated with:

Joints: A list of *joint indices* to which this mesh vertex is bound

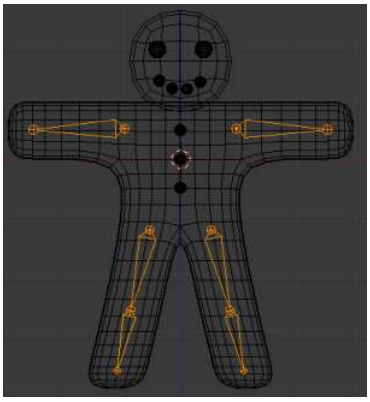

Fig. 1: Binding skin to a skeletal model in the reference pose.

Weights: A corresponding list of *weights*, which determine the influence of each joint on this vertex. These weights are assumed to be nonnegative and sum to 1.

For example, if you were to imagine the vertices of a forearm mesh, those that lie close to the elbow might be bound to the shoulder, elbow, and wrist joints, but with the elbow having the highest weight. As we move down the forearm towards the wrist, the vertex weights for the shoulder would vanish, the weights for the elbow would decrease, but the weights for the wrist would increase, and we would also see weight factors for the fingers to become positive as well. The number of joints to which a typical vertex is bound is typically small, e.g., from two to four. Good solid modelers provide tools to automatically assign weights to vertices, and designers can query and adjust these weights until they produce the desired look.

The above binding information can be incorporated into the mesh information already associated with a vertex: the (x, y, z)-coordinates of its location (with respect to the model coordinate system), the (x, y, z)-coordinates of its normal vector (for lighting and shading computation), and its (s, t) texture coordinates.

Moving the Joints: In order to derive the computations needed to move a vertex from its initial position to its final position, let's start by introducing some notation. First, recall that our animation system informs us at any time t the current angle for any joint. Abstractly, we can think of this joint angle as providing a *local rotation*, $R_j^{[t \leftarrow 0]}$, that specifies how joint j has rotated. For example, if the joint has undergone a rotation through an angle θ about some axis, then $R_j^{[t \leftarrow 0]}$ would be represented by a rotation matrix by angle θ about this same axis. (The analysis that we perform below works under the assumption that $R_j^{[t \leftarrow 0]}$ is any affine transformation, not necessarily a rotation.)

Consider a vertex v of the mesh. Let $v^{(0)}$ denote v's position in the initial reference pose, and let $v^{(t)}$ denote its position at the current time. We assume that this information is provided to us from the solid modeler. We can express v in one of two coordinate systems. Let v_j denote its initial position with respect to j's coordinate frame and let v_M denote its initial position with respect to the model's frame. (Throughout this section, we will assume that v is associated with the single joint j, rather than blended between multiple joints.) Given the above local rotation transformation, we have $v_j^{(t)} = R_j^{[t \leftarrow 0]} v_j^{(0)}$. (Because we assume that v rotates with frame j, its representation with respect to

j does not really change over time. Instead, think of $v_j^{(t)}$ as its representation relative to the joint's unrotated reference frame.)

Recall that $T_{[M \leftarrow j]}$ denotes the bind-pose transformation, which we introduced earlier. In general, let $T_{[M \leftarrow j]}^{[t \leftarrow 0]}$ denote the transformation that maps the vertex $v_j^{(0)}$ (which is in j's coordinate frame at time 0) to $v_M^{(t)}$ (which is in the model's frame at any time t).

Let's consider how to compute $T_{[M \leftarrow j]}^{[t \leftarrow 0]}$. As we did earlier, we'll build this up in a series of stages, by converting a point from its frame to its parent, then its grandparent, and so on until we reach the root of the tree. To map a vertex at time 0 from its frame to its parent's frame at time t we need to do two things. First, we apply the local joint rotation that takes us from time 0 to time t with respect to j's local frame, and then we transform this to j's parent's frame. That is, we need to first apply $R_j^{[t \leftarrow 0]}$ and then $T_{[p(j) \leftarrow j]}$. Let us define $T_{[p(j) \leftarrow j]}^{[t \leftarrow 0]}$ to be the product $T_{[p(j) \leftarrow j]} R_j^{[t \leftarrow 0]}$. We now have

$$v_{p(j)}^{(t)} \;=\; T_{[p(j) \leftarrow j]} v_j^{(t)} \;=\; T_{[p(j) \leftarrow j]} R_j^{[t \leftarrow 0]} v_j^{(0)} \;=\; T_{[p(j) \leftarrow j]}^{(t)} v_j^{(0)}.$$

An Example: These matrix relations can be rather confusing, so let us consider a simple example to make this a bit more concrete. Consider the pair of joints shown in Fig. 2(a), where joint j lies 5 units to the right of its parent $p(j)$.

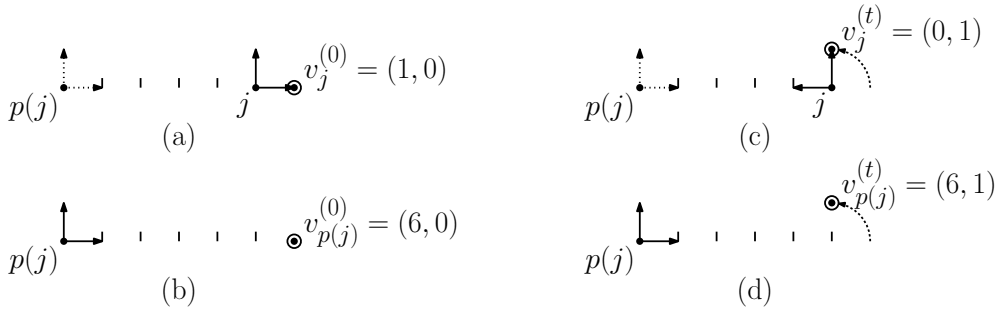

Fig. 2: Mapping a joint to its parent's frame. (a) and (b): before rotating the joint and (c) and (d): after rotating the joint.

It follows that $T_{[p(j) \leftarrow j]}(x, y) = (x + 5, y)$. We can express this in the form of homogeneous matrices as follows. Recall that the homogeneous representation of (x, y) is $[x, y, 1]$. Thus, we have

$$T_{[p(j) \leftarrow j]} \;=\; \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Let's consider the point lying at the tip of the x-axis in j's coordinate frame in the reference model. That is, $v_j^{(0)} = (1, 0)$. This point's representation relative to $p(j)$'s coordinate system is

$$v_{p(j)}^{(0)} \;=\; T_{[p(j) \leftarrow j]} v_j^{(0)} \;=\; \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \;=\; \begin{bmatrix} 6 \\ 0 \\ 1 \end{bmatrix},$$

which is the point $v_{p(j)}^{(0)} = (6, 0)$, relative to $p(j)$'s coordinate system at time 0, as expected (see Fig. 2(b)).

Suppose that between time 0 and t we rotate the joint j by $90°$ counterclockwise (see Fig. 2(c)). The associated rotation matrix maps a point (x, y) to $(-y, x)$. This is represented by the following rotation matrix.

$$R_j^{[t\leftarrow 0]} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Since $v_j^{(0)} = (1, 0)$, after rotation it is mapped (relative to j's unrotated coordinate system) to

$$v_j^{(t)} = R_j^{[t\leftarrow 0]} v_j^{(0)} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix},$$

which is the point $(0, 1)$ at the tip of the y-axis as expected (see Fig. 2(c)).

Finally, let's consider the combination of these two steps. We want to know where the vertex v is mapped to at time t by in $p(j)$'s coordinate frame. To obtain this we compose the two transformations by multiplying the corresponding matrices

$$T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]} = T_{[p(j)\leftarrow j]} R_j^{[t\leftarrow 0]} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Therefore, we can obtain v's representation at time t relative to $p(j)$'s coordinate frame, that is, $v_{p(j)}^{(t)}$, by applying this transformation to $v_j^{(0)}$. That is,

$$v_{p(j)}^{(t)} = T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]} v_j^{(0)} = \begin{bmatrix} 0 & -1 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix},$$

which is the point $(5, 1)$, just as we would expect (see Fig. 2(d)).

The Current-Pose Transformation: Hopefully, you now believe that $T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]}$ maps a vertex from j's frame at time 0 to $p(j)$'s frame at time t, accounting for the rotation of joint j. To obtain the position of a vertex associated with j's coordinate frame, we need only compose these matrices working back to the root of the tree. Suppose that the path from j to the root is $j = j_1 \to j_2 \to \ldots \to j_m = M$, then transformation we desire is

$$T_{[M\leftarrow j]}^{[t\leftarrow 0]} = T_{[j_m\leftarrow j_{m-1}]}^{[t\leftarrow 0]} \cdots T_{[j_3\leftarrow j_2]}^{[t\leftarrow 0]} T_{[j_2\leftarrow j_1]}^{[t\leftarrow 0]} = \prod_{i=1}^{m-1} T_{[j_{m-i-1}\leftarrow j_{m-i}]}^{[t\leftarrow 0]}.$$

We refer to this as the *current-pose transformation*, since it tells where joint j is at time t relative to the model's global coordinate system. For the sake of making our notation more concise, we'll refer to it henceforth as $C_{[M\leftarrow j]}$. Observe that with each animation time step, all the matrices $R_j^{(t)}$ change, and therefore we need to perform a full traversal of the skeletal tree to compute $C_{[M\leftarrow j]}$ for all joints

j. Fortunately, a typical skeleton has perhaps tens of joints, and so this does not represent a significant computational burden (in contrast to operations that need to be performed on each of the individual vertices of a skeletal mesh).

Blended Skinning: Next, we consider how to compute the positions of blended vertices. We assume that for the current-pose transformation $C_{[M \leftarrow j]}$ has been computed for all the joints and we assume that each vertex v is associated with a list of joints and associated weights. Let $J(v) = \{j_1, \ldots, j_k\}$ be the joints associated with vertex v and let $W(v) = \{w_1, \ldots, w_k\}$ be the associated weights. Typically, k is a small number, ranging say from 2 to 4. For i running from 1 to k, our approach will be compute the coordinates of v relative to joint j_i, then apply the current-pose transformation for this joint in order to obtain the coordinates of v relative to the (global) model frame. This gives us k distinct points, each behaving as if it were attached to a different joint. We then blend these points together, to obtain the desired result.

Recall the inverse bind-pose transformation $T_{j \leftarrow M}$, which maps a vertex v from its coordinates in the model frame to its coordinates relative to j's coordinate frame. Recall that this transformation is defined on the reference model, and so is applied to vertices of the original model, prior to animation. Once we have the vertex in its representation at time 0 relative to joint j, we can then apply the current-pose transformation $C_{[M \leftarrow j]}$ to map it to its current position relative to the model frame. If v was bound to a single joint j, we would have

$$v_M^{(t)} \;=\; C_{[M \leftarrow j]} T_{[j \leftarrow M]} v_M^{(0)}.$$

Let us define $K_j = C_{[M \leftarrow j]} T_{[j \leftarrow M]}$. This is called the *skinning transformation* for joint j. (Note that this needs to be recomputed each time the rotation angles change.)

Now, we can generalize this to the general case of blending among a collection of vertices. Recall that v has been bound to the joints of $J(v)$. Its blended position at time t is given by the weighted sum

$$v_M^{(t)} \;=\; \sum_{j_i \in J(v)} w_i K_{j_i} v_M^{(0)}.$$

A simple example of this is shown in Fig. 3. In Fig. 3(a) we show the reference pose. In Fig. 3(b), we show what might happen if every vertex is bound to a single joint. When the joint flexes, the vertices at the boundary between to bones crack apart from each other. In Fig. 3(c) we have made a very small change. The vertices lying on the seam between the two pieces have been bound to both joints j_1 and j_2, each with a weight of $1/2$. Each of these vertices is effectively now placed at the midpoint of the two "cracked" copies. The result is not very smooth, but it could be made much smoother by adding weights to the neighboring vertices as well.

It is worth making a few observations at this point about the storage/computational requirements of this approach.

Matrix palette: In order to blend *every* vertex of the model, we need only one vertex for each joint of the skeleton, namely the skinning matrices K_j. Recall that a skeleton will have perhaps tens of joints, while it may have hundreds of vertices. Assuming that the joint are indexed by integers, the palette can be passed to the GPU as an array of matrices.

Vertex information: Each vertex is associated with a small list of joint indices and weights. In particular, we do not need to associate entire matrices with individual vertices.

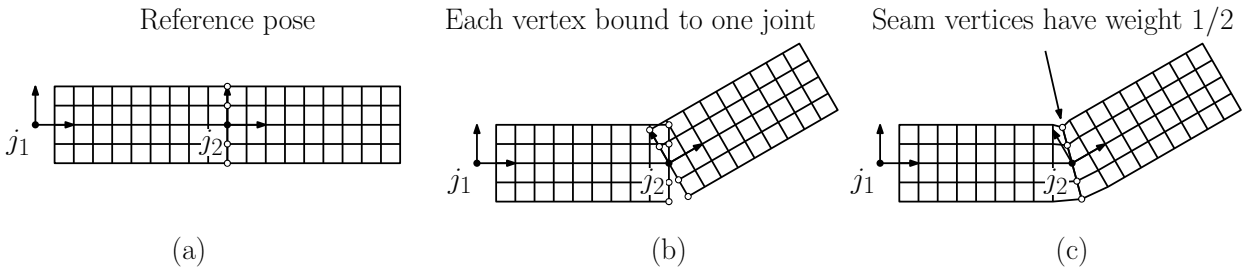

Fig. 3: A simple example of blended skinning.

From the perspective of GPU implementation, this representation is very efficient. In particular, we need only associate a small number of scalar values with each vertex (of which there are many), and we store a single vertex with each joint (or which there are relatively few). In spite of the apparent tree structure of the skeleton, everything here can be represented using just simple arrays. Modern GPUs provide support for storing matrix palettes and performing this type of blending.

Shortcomings of Blended Skinning: While the aforementioned technique is particularly well suited to efficient GPU implementation, it is not without its shortcomings. In particular, if joints are subjected to high rotations, either in flexing or in twisting, the effect can be to cause the skin to deform in particular unnatural looking ways (see Fig. 4).

Fig. 4: Shortcomings of vertex blending in skinning: (a) Collapsing due to bending and (b) collapsing due to twisting.