CMSC 425: Lecture 10
Skeletal Animation and Skinning

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

Recap: Last time we introduced the principal elements of skeletal models and discussed forward kinematics. Recall that a skeletal model consists of a collection of joints, which have been joined into a rooted tree structure. Each joint of the skeleton is associated with a *coordinate frame* which specifies its position and orientation in space. Each joint can be rotated (subject to sum constraints). The assignment of rotation angles (or generally rotation transformations) to the individual joints defines the skeleton's *pose*, that is, its geometrical configuration in space. Joint rotations are defined relative to a default pose, called the *bind pose* (or *reference pose*).

Last time, we showed how to determine the skeleton's configuration from a set of joint angles. This is called *forward kinematics*. (In contrast, *inverse kinematics* involves the question of determining how to set the joint angles to achieve some desired configuration, such as grasping a door knob.) Today we will discuss how animation clips are represented, how to cover these skeletons with "skin" in order to form a realistic model, and how to move the skin smoothly as part of the animation process.

Local and Global Pose Transformations: Recall from last time that, given a joint j (not the root), its parent joint is denoted $p(j)$. We assume that each joint j is associated with two transformations, the *local-pose transformation*, denoted $T_{[p(j)\leftarrow j]}$, which converts a point in j's coordinate system to its representation in its parent's coordinate system, and the *inverse local-pose transformation*, which reverses this process. (These transformations may be represented explicitly, say, as a 4×4 matrix in homogeneous coordinates, or implicitly by given a translation vector and a rotation, expressed, say as a quaternion.)

Recall that these transformations are defined relative to the bind pose. By chaining (that is, multiplying) these matrices together in an appropriate manner, for any two joints j and k, we can generally compute the transformation $T_{[k\leftarrow j]}$ that maps points in j's coordinate frame to their representation in k's coordinate frame (again, with respect to the bind pose.)

Let M (for "*Model*") denote the joint associated with the root of the model tree. We define the *global pose transformation*, denoted $T_{[M\leftarrow j]}$, to be the transformation that maps points expressed locally relative to joint j's coordinate frame to their representation relative to the model's global frame. Clearly, $T_{[M\leftarrow j]}$ can be computed as the product of the local-pose transformations from j up to the root of the tree.

Meta-Joints: One complicating issue involving skeletal animation arises from the fact that different joints have different numbers of degrees of freedom. 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. This allows us to assume that each joint has just a single degree of freedom.

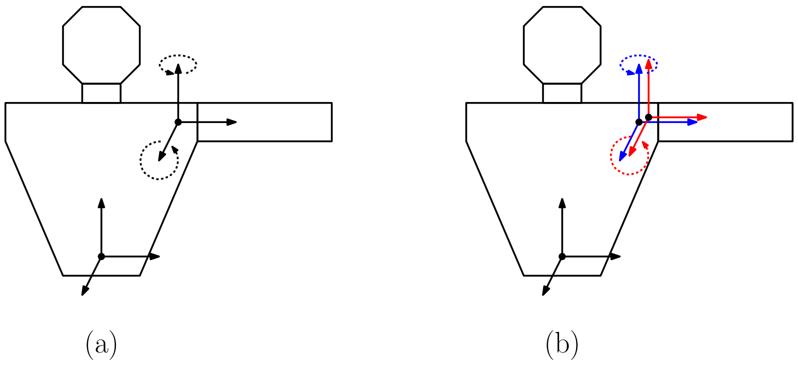

Fig. 1: Using two meta-joints (b) to simulate a single joint with two degrees of freedom (a).

Animating the Model: There are a number of ways to obtain joint angles for an animation. Here are a few:

Motion Capture: For the common motion of humans and animals, the easiest way to obtain animation data is to capture the motion from a subject. 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.

Key-frame Generated: A design artist can use animation modeling software to specify the joint angles. This is usually done by a process called *key framing*, where the artists gives a detailed layout of the model at certain "key" instances in over the course of the animation, called *key frames*. (For example, when animating a football kicker, the artist might include the moment when the leg starts to swing forward, an intermediate point in the swing, and the point at which the leg is at its maximum extension.) An automated system can then be used to smoothly interpolate the joint angles between consecutive key frames in order to obtain the final animation. (The term "frame" here should not be confused with the use of term "coordinate frame" associated with the joints.)

Goal Oriented/Inverse kinematics: In an ideal world, an animator could specify the desired behavior at a high level (e.g., "a character approaches a table and picks up a book"). Then the physics/AI systems would determine a natural-looking animation to achieve this. This is quite challenging. The reason is that the problem is under-specified, and it can be quite difficult to select among an infinite number of valid solutions. Also, determining the joint angles to achieve a particular goal reduces to a complex nonlinear optimization problem.

Representing Animation Clips: In order to specify an animation, we need to specify how the joint angles or generally the joint frames vary with time. This can result in a huge amount of data. Each joint that can be independently rotated defines a *degree of freedom* in the specification of the pose. For example, the human body has over 200 degrees of freedom! (It's amazing to think that our brain can control it all!) Of course, this counts lots of fine motion that would not normally be part of an animation, but even a crude modeling of just arms (not including fingers), legs (not including toes), torso, neck involves over 20 degrees of freedom.

As with any digital signal processing (such as image, audio, and video processing), the standard approach for efficiently representing animation data is to first *sample* the data at sufficiently small time intervals. Then, use some form of interpolation technique to produce a smooth *reconstruction* of the animation. The simplest manner to interpolate values is based on *linear interpolation.* It may be desireable to produce smoother results by applying more sophisticated interpolations, such as quadratic or cubic spline interpolations. When dealing with rotated vector quantities, it is common to use *spherical interpolation.*

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) indicating the joint frame's position and a 4-element quaternion vector (s, t, u, v) to represent the frame's rotation. Each row of this representation is a sequence of scalar values, and is called a *channel.*

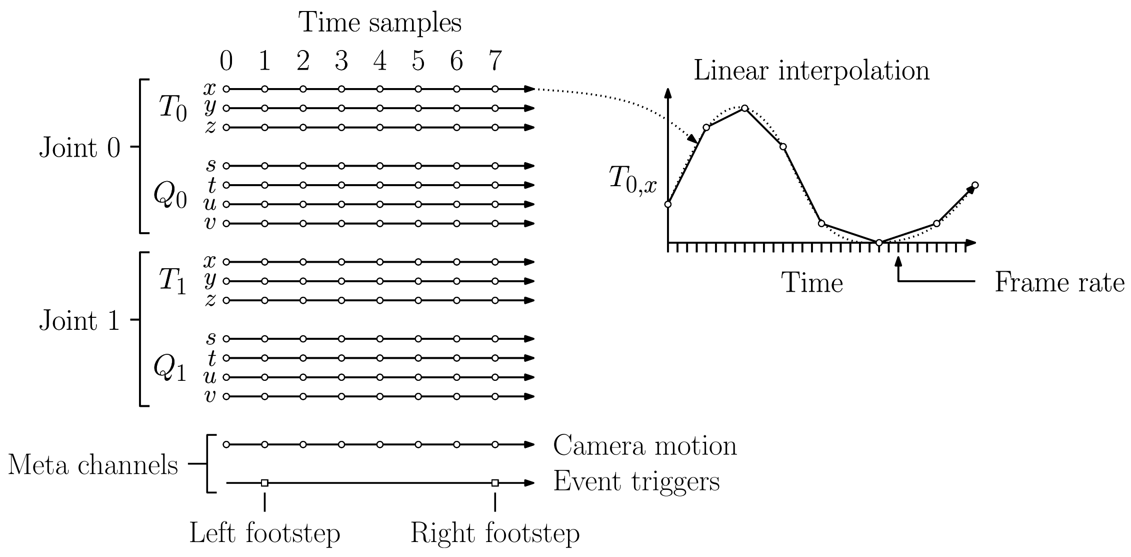

Fig. 2: An uncompressed animation stream.

It is often useful to add further information to the animation, which are not necessarily related to the rendering of the moving character. Examples include:

Event triggers: These are 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.

Skinning and Vertex Binding: 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. 3(a)).

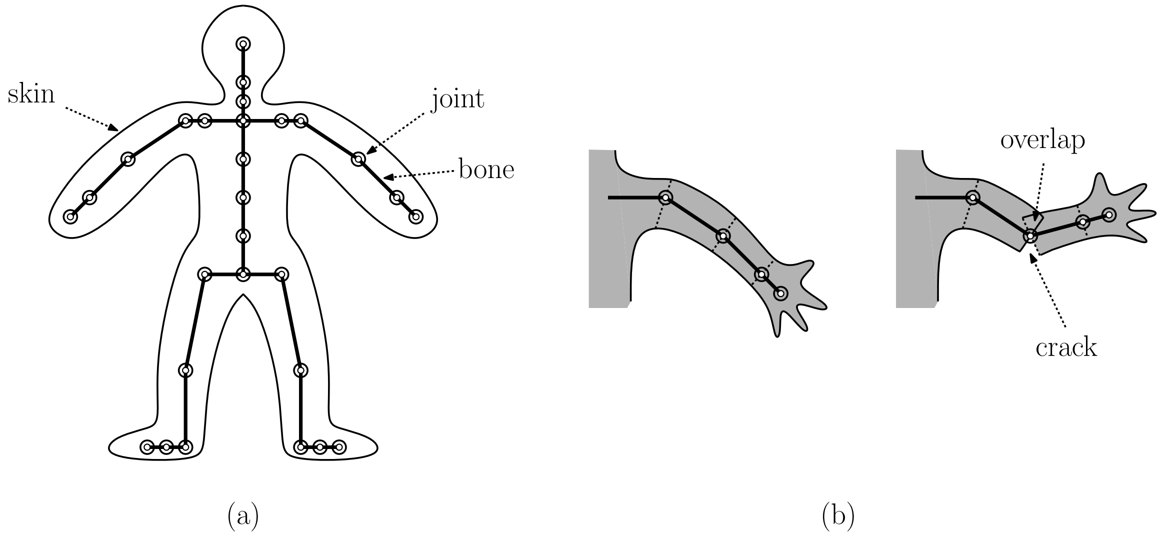

Fig. 3: (a) Binding skin to a skeletal model in the reference pose and (b) cracks and overlaps.

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.)

If we were to bind each vertex to a single joint, then we would observe *cracks* and *overlaps* appearing in our skin whenever neighboring vertices are bound to two different joints that are rotated apart from one another.

Dealing with this problem in a realistic manner will be too difficult. (The manner in which the tissues under your skin deform is a complex anatomical process. Including clothing on top of this makes for a tricky problem in physics as well.) Instead, our approach will be to find a heuristic solution that will be easy to compute and (hopefully) will produce fairly realistic results.

Linear-Blend Skinning: The trick is to 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.

For example, the mesh vertices near your elbow will be bound to both the shoulder joint (your upper arm) and the elbow joint (your lower arm). As we move down the arm, the shoulder weight factor diminishes while the elbow weight factor increases. Consider for example, consider a vertex v that is located slightly above the elbow joint (see Fig. 4(a)). In the bind pose, let v_1 and v_2 denote its positions relative to the shoulder and elbow joint frames, respectively.[1] Since this point is slightly above the elbow joint, we give a slightly higher weight with respect to the shoulder joint. Suppose that the weights are $w_1 = \frac{3}{4}$ and $w_2 = \frac{1}{4}$.

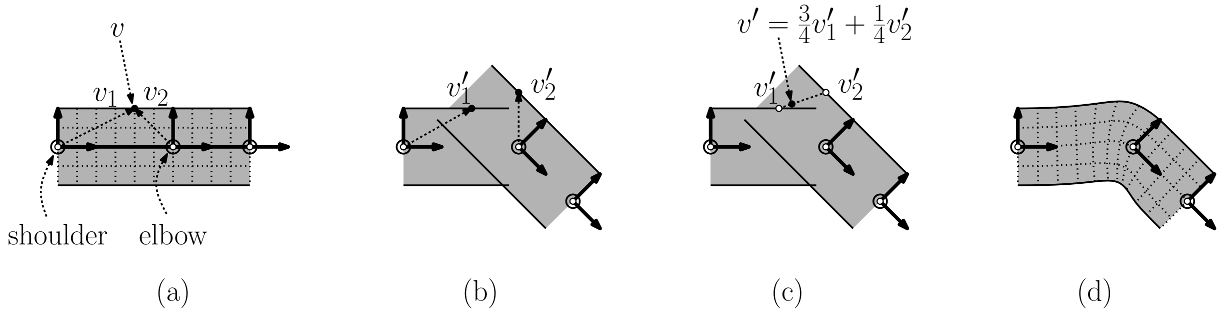

Fig. 4: Skinning with multiple joint bindings.

Now, suppose that we bend both of these joints. Let v'_1 and v'_2 denote the respective images of the points v_1 and v_2 after the rotation. They will be in the same position relative to their respective joints, but their absolute positions in space have changed due to the rotation. See Fig. 4(b). We use our weight factors to interpolate between these two positions, so the final position of the vertex is at the point $v' = \frac{3}{4}v'_1 + \frac{1}{4}v'_2$ (see Fig. 4(c)). This is the simplest way to blend vertex positions and is called *linear-blend skinning*. There are many other methods (such as *dual quaternion skinning*), which achieve more realistic results at the price of additional computation time. Because of the smooth variations in weights, the vertices of the mesh will form a smooth interpolation between the upper arm and the lower arm (see Fig. 4(d)). It may not be anatomically realistic, but it is a reasonable approximation for small rotation angles, it avoids cracks, and it is very easy to compute.

To make this more formal, we assume that each vertex of the mesh is associated with:

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

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.

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.

[1]Here we are thinking of v_1 and v_2 as the homogeneous vectors used to represent the "intrinsic" point v. If we were to express this in terms of the notation developed in the previous lecture, let i be the shoulder-joint index, and let j be the elbow-joint index. Then, $v_1 = v_{[i]}$ and $v_2 = v_{[j]}$.

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 without Blending: Let's begin by considering how to compute vertex positions assuming the simple case where each vertex is bound to a single joint. In order to derive the computations needed to move a vertex from its initial position to its final position. The notation is going to be rather heavy from this point on, but the ideas are relatively straightforward:

- Convert the vertex to the current joint's coordinate frame (local pose transformation)
- Apply the desired rotation at this joint for the current animation time
- Move on to the next joint

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*, $R^{(t)}_{[j]}$, that specifies how joint j has rotated. For example, if the joint has undergone a rotation through an angle θ about some axis, then $R^{(t)}_{[j]}$ 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 $R^{(t)}_{[j]}$ 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. Given the above local rotation transformation, we have $v^{(t)}_{[j]} = R^{(t)}_{[j]} v^{(0)}_{[j]}$.[2]

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

Let's consider how to compute $T^{(t)}_{[M \leftarrow j]}$. 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 $R^{(t)}_{[j]}$ and then $T_{[p(j) \leftarrow j]}$. Let us define $T^{(t)}_{[p(j) \leftarrow j]}$ to be the product $T_{[p(j) \leftarrow j]} R^{(t)}_{[j]}$. We now have

$$v^{(t)}_{p(j)} \;=\; T_{[p(j) \leftarrow j]} v^{(t)}_{[j]} \;=\; T_{[p(j) \leftarrow j]} R^{(t)}_{[j]} v^{(0)}_{[j]} \;=\; T^{(t)}_{[p(j) \leftarrow j]} v^{(0)}_{[j]}.$$

[2]Because we assume that v rotates with frame j, its representation with respect to j does not really change over time. Instead, think of $v^{(t)}_{[j]}$ as its representation relative to the joint's unrotated reference frame.

To obtain the position of a vertex associated with j's coordinate frame, we need only compose these matrices in a chain working back up to the root of the tree. We apply a rotation, convert to the coordinate frame of the parent joint, apply the parent rotation, convert to the coordinates of the grandparent joint, and so on. Suppose that the path from j to the root is $j = j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_m = M$, then transformation we desire is

$$\begin{aligned} T^{(t)}_{[M \leftarrow j]} &= T^{(t)}_{[j_m \leftarrow j_{m-1}]} \ldots T^{(t)}_{[j_3 \leftarrow j_2]} T^{(t)}_{[j_2 \leftarrow j_1]} \\ &= T_{[j_m \leftarrow j_{m-1}]} R^{(t)}_{[j_{m-1}]} \ldots T_{[j_3 \leftarrow j_2]} R^{(t)}_{[j_2]} T_{[j_2 \leftarrow j_1]} R^{(t)}_{[j_1]}. \end{aligned}$$

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. Observe that with each animation time step, all the matrices $R^{(t)}_{[j]}$ change, and therefore we need to perform a full traversal of the skeletal tree to compute $T^{(t)}_{[M \leftarrow j]}$ 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).

Moving the Joints with Blending (Optional): Finally, let's consider how to apply blended skinning together with the dynamic pose transformations. This will tell us where every vertex of the mesh is mapped to in the currrent animation. We assume that for the current-pose transformation $T^{(t)}_{[M \leftarrow j]}$ 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 1 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. This transformation is defined relative to the bind pose, 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 $T^{(t)}_{[M \leftarrow j]}$ to map it to its current position relative to the model frame. If v was bound to a single joint j, we would have

$$v^{(t)}_{[M]} = T^{(t)}_{[M \leftarrow j]} T_{[j \leftarrow M]} v^{(0)}_{[M]}.$$

Let us define $K^{(t)}_{[j]} = T^{(t)}_{[M \leftarrow j]} T_{[j \leftarrow M]}$. This is called the *skinning transformation* for joint j. Intuitively, it tells us where vertex v is mapped to under at time t of the animation assuming that it is fixed to joint j.

Now, we can generalize this to the 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 of the image of the skinning transformed vertices $K^{(t)}_{[j_i]} v^{(0)}_{[M]}$ for each joint j_i to which v is bound:

$$v^{(t)}_{[M]} = \sum_{j_i \in J(v)} w_i K^{(t)}_{[j_i]} v^{(0)}_{[M]}.$$

This then is the final answer that we seek. While it looks like a lot of matrix computation, remember each vertex is associated with a constant number of joints, and each joint is typically at constant depth in the skeletal tree. Once these matrices are computed, they may be stored and reused for all the vertices of the mesh.

Discussion: A simple example of this is shown in Fig. 5. In Fig. 5(a) we show the reference pose. In Fig. 5(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. 5(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.

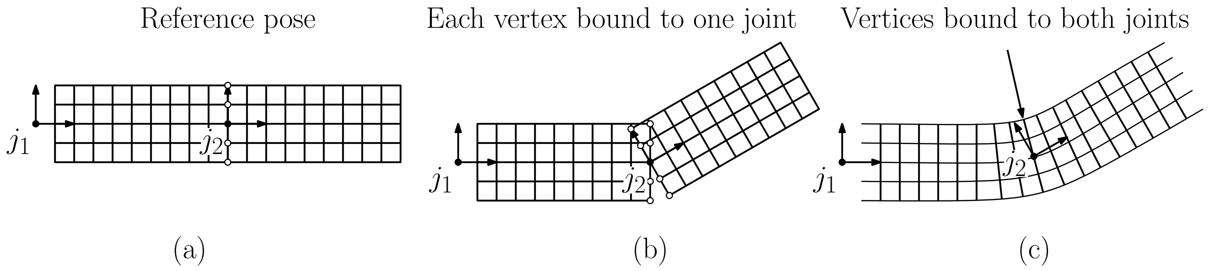

Fig. 5: A simple example of blended skinning.

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 matrix for each joint of the skeleton, namely the skinning matrices $K_{[j]}^{(t)}$. While a skeleton may have perhaps tens of joints, 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.

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. 6). This is often referred to as the *candy-wrapper effect*. Other skinning methods can be used to avoid these artifacts.

(a) (b)

Fig. 6: Shortcomings of vertex blending in skinning: (a) Collapsing due to bending and (b) collapsing due to twisting.