4.15

Filling the Gaps—Advanced Animation Using Stitching and Skinning

Ryan Woodland

As hardware becomes faster and more feature-laden, game developers are searching for ways to make characters look more compelling. Of the many categories that can be improved, character animation is perhaps one of the most important.

Currently, most 3D games are starting to use some sort of skeletal representation for their characters as their topology for animation. These systems attach geometry to "bones" in a character. The bones are then animated and, consequently, the attached geometry inherits the motion creating adequate animation. Usually, however, the geometry used to represent characters is rigid in nature, which is not the most useful representation for modeling organic creatures that are definitely not rigid in nature.

Because the geometry is completely rigid, any two pieces that are supposed to be connected to each other (an upper arm and a forearm, for example) display blatant discontinuities at the joint at which they are connected. This obviously can become a problem, since the characters we are trying to represent are more often than not made up of a continuous skin that does not show any cracks or separations.

In this article, I will discuss the topics of *stitching* and *skinning* as ways to create more realistic organic animation. Stitching is actually just a less computationally expensive subset of skinning and will therefore be discussed first. Both of these techniques assume one continuous mesh that is attached to a bone structure for a character as opposed to many meshes attached to a single bone in traditional rigid-body animation. This continuous mesh is deformed relative to the character's bone structure, yielding a character that does not create visible (and often very annoying) gaps at joints when animating.

In the following sections, I will be using the example of an arm to demonstrate various features of stitching and skinning. The basic mesh used is picture in Figure 4.15.1.

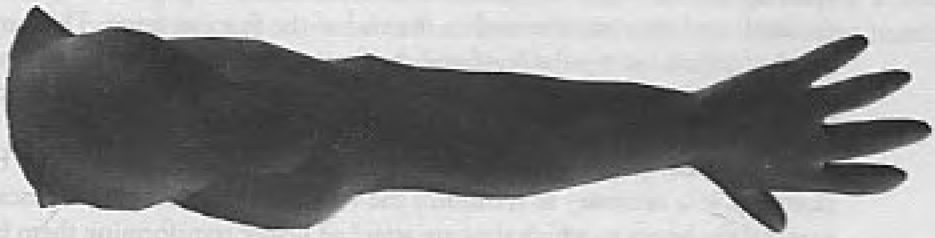

FIGURE 4.15.1 Our basic arm mesh.

Stitching

As mentioned earlier, stitching operates on a continuous mesh attached to a bone structure. In rigid-body animation, a polygon is transformed by one matrix representing the bone to which that polygon is attached. With stitching, each vertex in a polygon can be transformed by a different matrix representing the bone to which the individual vertex is attached. This means that we can create polygons that "stitch" multiple bones together simply by attaching different vertices in the polygon to different bones. When the bones are manipulated, this polygon should fill the gap you would see in rigid-body animation.

One of the major differences between stitching and rigid-body animation is the data topology for representing a character. With rigid-body animation, a bone must simply have a pointer to some geometry it is to animate. The matrix yielded by the corresponding bone then transforms that geometry. For stitching, it is necessary for each *vertex* in the character's skin to keep track of the bone to which it is attached.

```
struct Vertex
{
    float s, t;
    float x, y, z;
    unsigned long color;

    unsigned long boneIndex;
};
```

Before animating a character that has been correctly bound to this data topology, we need to deal with the problem that our vertices are not in the correct space to be properly transformed. The problem is this: a matrix used to transform a bone for animation assumes that the bone starts with its pivot point at the origin of the coordinate space of the character. This makes sense if we consider a hand bone in a normal human. This bone should start with its pivot point at the origin of its coordinate space so that we can easily rotate the bone around that point. The bone is animated (rotated) and then transformed to the end of the forearm bone. This process repeats for the forearm bone—the hand and the forearm are then animated and moved out to the end of the upper arm bone. This continues down through the hierarchy until the entire skeleton has been properly transformed.

Given the spatial relationship between the skin's vertices and the bones of the character, it is necessary to transform the vertices of the skin into the local coordinate space of the bones to which they are attached before transforming them by the bone's animation matrix. To do this, we need to keep a matrix in each bone that tells us how to transform geometry back into the local space of the bone. This matrix should be the inverse of the matrix used to transform the bone from its local space into the character's mesh, given the orientation of the mesh without any animation being applied. See Figure 4.15.2 for a depiction of the local spaces for each bone in our arm mesh.

Therefore, the data structure of our bones should look like the following:

```
struct Bone
{
    Mtx orientation;
    Mtx animation;
    Mtx inverseOrientation;

    Mtx final;

    Bone *child;
    Bone *sibling;
};
```

Once we have this data, we are ready to animate our character. To do this, we must simply step through the vertex data and transform each vertex by the orientation matrix and then the animation matrix of the corresponding bone.

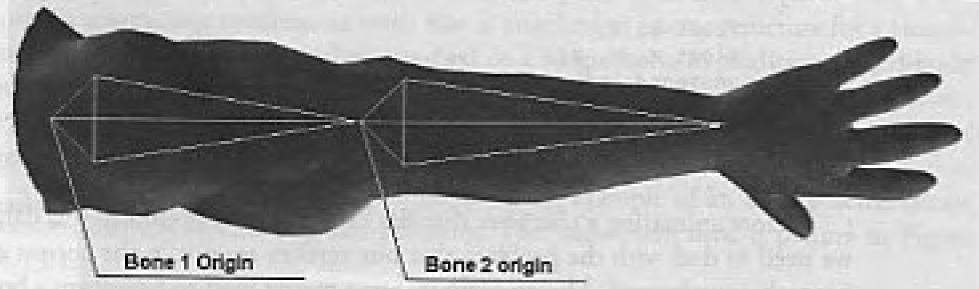

FIGURE 4.15.2 A depiction of the bones in our arm.

All of these transformations can be done faster by processing the bone hierarchy and generating a final transformation matrix for each bone concatenating a bone's inverse orientation, concatenated orientation, and concatenated animation matrices together and then transforming geometry by the resulting matrix.

```
void BuildMatrices ( Bone *bone, Mtx forward, Mtx orientation )
{
    Mtx localForward;
    Mtx localOrientation;

    // concatenate the hierarchy's orientation matrices so
    // that we can generate the inverse
    concatenate(bone->orientation, orientation ,
    localOrientation);

    // take the inverse of the orientation matrix for this bone
    inverse(localOrientation, bone->inverseOrientation);

    // concatenate this bone's orientation onto the forward
    // matrix
    concatenate(bone->orientation, forward, localForward);

    // concatenate this bone's animation onto the forward matrix
    concatenate(bone->animation, localForward, localForward);

    // build the bone's final matrix
    concatenate(bone->inverseOrientation, localForward,
    bone->final);

    if(bone->child)
        BuildMatrices(bone->child, localForward,
        localOrientation);

    if(bone->sibling)
        BuildMatrices(bone->sibling, forward, orientation);
}
```

Using the preceding technique on the arm mesh, a bend of 45 degrees and 90 degrees to the forearm bone produces the images in Figure 4.15.3.

Stitching is a very valid technique, since it easily takes advantage of any hardware that provides a transform engine. It is necessary to generate the final stitching matrix on the CPU, but the hardware can easily use these matrices to transform any number of vertices we pass it.

As an optimization to this technique, I suggest breaking up the continuous skin so that the vertices exist in the local space of the bone to which they are attached. This prevents us from having to do an extra matrix concatenation per bone per frame of animation.

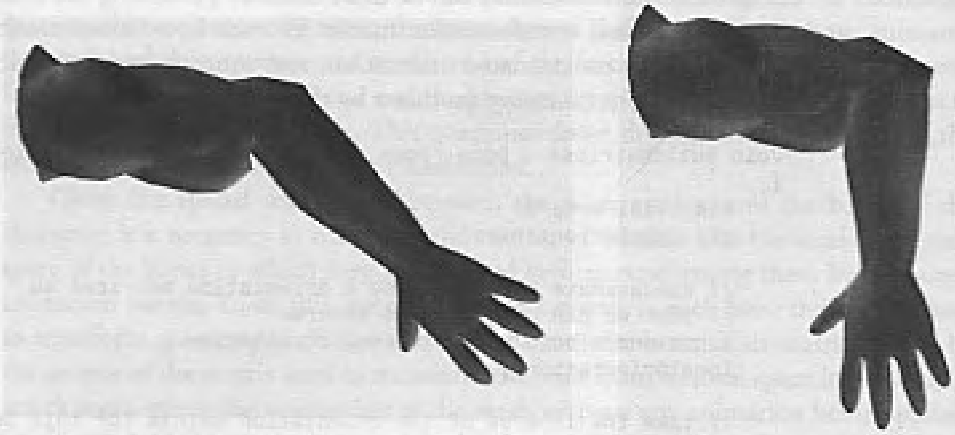

FIGURE 4.15.3 *a:* Stitched arm mesh bent to 45 degrees. *b:* Bent to 90 degrees.

Skinning

While stitching is a valid technique, it has some problems. In cases of extreme joint rotation, geometry tends to shear massively and appear quite unnatural. Using the techniques discussed earlier, a forearm rotation of 120 degrees displays quite a nasty shear effect at the elbow. This results because we only have one polygon to span the entire gap between the upper arm and the forearm. The larger this gap becomes, the worse the solution looks, as shown in Figure 4.15.4.

To prevent this, we can implement a full system of skinning where a vertex is not limited to being affected by a single bone; it can instead be influenced by multiple bones. This makes sense if we look at the behavior of the human body. The skin on a person's elbow is not affected by the orientation of just one bone. The movements of both the upper and lower arm bones affect it. Similarly, skin in the neck and shoulder is affected by the orientations of the arm, neck, and chest.

To enable this, each vertex in a skinned mesh must contain a list of bones that affect it. Each vertex must also carry a weight per bone that tells us how heavily affected the vertex is by the bone. For this example, we will assume linear skinning, which means all of the weights of a vertex must add up to 1.0. Because of this, given *n* bones by which a vertex is affected, we need to store *n*–1 weights, since the remaining weight should be 1.0 – ($weight_1$ + $weight_2$ + ... + $weight_{n-1}$).

```
struct Vertex
{
    float s, t;
    float x, y, z;
    unsigned long color;
```

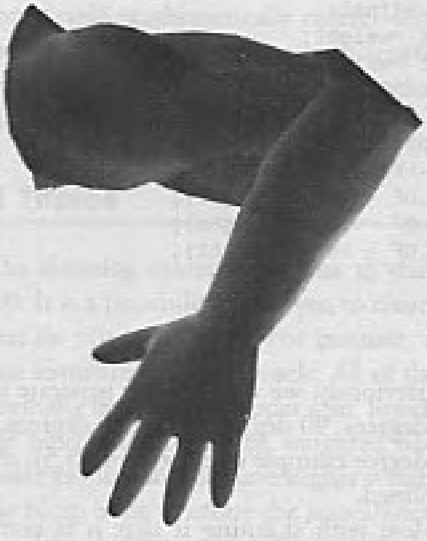

FIGURE 4.15.4 Ugly stitched arm mesh bent to 120 degrees.

```
    unsigned long boneIndex1;
    unsigned long boneIndex2;

    float weight;
};
```

As mentioned earlier, stitching is a subset of skinning, and therefore suffers from the same local-space transform issues as stitching. Therefore, we should use the same bone representation as shown previously.

In order to do full skinning, we need to transform each bone by each matrix affecting it, then multiply the result by the corresponding weight, and, finally, accumulate the results. The equation for skinning looks like:

*(vertex * matrix0 * weight0) + (vertex * matrix1 * weight1) + ... + (vertex * matrixN * weightN)*

where the sum of all weights 0...*N* = 1.0.

What we are effectively doing is a linear interpolation between transformed vertices. The following is the code used to perform this operation on a given mesh.

```
Vector3D TransformVertex ( Vertex *vert, Bone *boneArray )
{
    Vector3D   temp;
    Vector3D   final;
```

```
        temp = XFormVec(vert->position,
        bone[vert->boneIndex1]->final)
        final.x = temp.x * vert->weight;
        final.y = temp.y * vert->weight;
        final.z = temp.z * vert->weight;

        temp = XFormVec(vert->position,
        bone[vert->boneIndex2]->final)
        final.x += temp.x * (1.0F - vert->weight);
        final.y += temp.y * (1.0F - vert->weight);
        final.z += temp.z * (1.0F - vert->weight);

        return final;
    }
```

Using the technique outlined previously, we were able to generate the following output for forearm rotations of 45 degrees, 90 degrees, and 120 degrees, respectively. Note that even in the extreme 120-degree example (see Figure 4.15.5), the continuity of the elbow geometry is still maintained.

As you can see, a major problem with skinning is that it is computationally expensive. Unfortunately, these computations are not well supported by today's hardware transform engines. An alternative way of performing the linear interpolation calculations, which potentially takes advantage of some current hardware implementations, is to generate a skinning matrix to be passed to hardware to perform the final transform. To calculate the skinning matrix, simply interpolate the matrices linearly based on the weight:

*(matrix0 * weight0) + (matrix1 * weight1) + ... + (matrixN * weightN)*

where the sum of all weights 0..N = 1.0.

This method is only useful if the same skinning matrix can be used for multiple vertices; in other words, different vertices are weighted identically between the same bones. The less this case is true, the less the gain of this method will be.

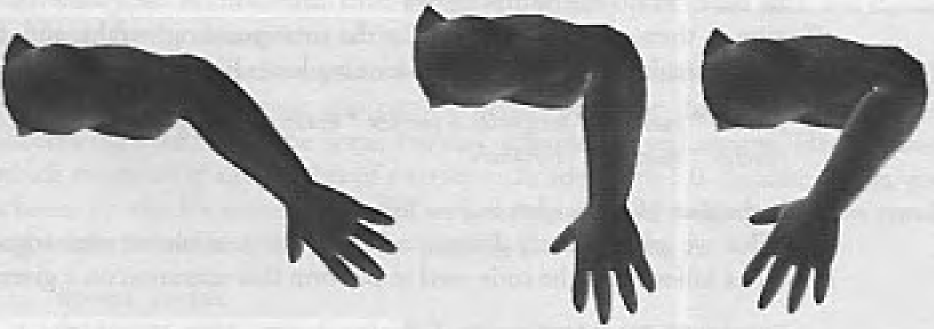

FIGURE 4.15.5 *a:* Skinned arm mesh bent to 45 degrees. *b:* Bent to 90 degrees. *c:* Bent to 120 degrees.

It is important to note that the skinning technique outlined previously is not a completely mathematically correct technique. If normals are transformed using this technique, the results are not guaranteed to be normalized. If per-vertex lighting is required for a character using this technique, post-transform normals must be re-normalized before lighting calculations.

Advanced Topics

The skinning example assumes all the weights influencing a vertex must add up to 1.0. It is a possibility, however, to create some compelling special effects with weights that do not sum to 1.0. For instance, it is possible to place an extra bone in an arm that simulates a bicep muscle. All of the vertices in the arm's skin should be weighted normally between the upper arm, lower arm, and shoulder. However, the vertices near the bicep should also be weighted based on their distance from the bicep bone—closer vertices should have higher weight values. When the arm bends, apply a scale to the bicep bone to create the appearance of a muscle flexing.

The skinning technique outlined is not mathematically correct because we are essentially linearly interpolating matrices. Instead of representing bones as matrices, it is possible to represent them as a quaternion. *SLERP* between the quaternions based on the per-vertex weights and then produce a matrix from the result. This should yield a somewhat better-looking skinned mesh.

References

[Lander98] Lander, Jeff, "Skin Them Bones: Game Programming for the Web Generation," *Game Developer Magazine* (May 1998): pp. 11–16.

[Terzopoulos87] Terzopoulos, Demetri, et al, "Elastically Deformable Models," Computer Graphics, Vol 21, no.4 (SIGGRAPH 1987): pp. 205–214.