The following pages are scans of pages from the Addison-Wesley and nVidia series of books *GPU Gems* and *GPU Gems III*.

- The first article, *Interpolated 3D Keyframe Animation* discusses a better method than linear interpolation to take account of continuity across keyframes;
- the second, on page 8, is *A Fast and Simple Skinning Technique*;
- the third article, on page 15, is titled *Filling the Gaps - Advanced Animation Using Stitching and Skinning*;
- from *GPU Gems III*, the next article, on page 21, is on *Improved Deformation of Bones*;
- finally, an interesting article on locomotion follows on page 31.
Interpolated 3D Keyframe Animation

Herbert Marselas

Keyframing is a simple and effective way of animating a 3D object. However, since each keyframe only represents the extremes of the object's motion, this can make the object appear to jump between positions.

Linear Interpolation

One solution is to add more keyframes to make the transition between keyframes less jarring. Another more economical method is to programatically create in-between animation frames using interpolation.

Interpolation—also known as blending, morphing, or tweening—is the process of creating a new position between two existing positions. In this case, we are interpolating two known keyframe positions \( p_0 \) and \( p_1 \) to create a new position \( p(t) \).

The easiest interpolation solution is linear interpolation. In this case, a line is drawn between the same position in two adjacent keyframes \( p_0 \) and \( p_1 \), and then we calculate where on this line the new position \( p(t) \) exists (Figure 4.13.1).

Given the desired time of the new animation position, the total number of keyframes, and the total time of the animation, the point between the two closest keyframes can be calculated.

The function `calculateFramePercentage` demonstrates this. Given the total number of keyframes in the animation, the total time of the animation, and the desired time, the keyframes on either side of the new position and the percentage between the two frames are calculated and returned.

\[
p(t) = p_0 + t(p_1 - p_0)
\]

**Figure 4.13.1** Example linear interpolation and formula.
void calculateFramePercentage(long dwTotalAnimFrames, float fTotalAnimTime, float fDesiredTime, long &dwFirstFrame, long &dwSecondFrame, float &fPercentage)
{
    // determine which frames are involved
    float fTimePerFrame = fTotalAnimTime / (float) dwTotalAnimFrames;
    dwFirstFrame = 0;
    if (fDesiredTime > fTotalAnimTime)
        fDesiredTime = fTotalAnimTime;
    for (float f = 0.0f; f <= fDesiredTime; f += fTimePerFrame)
        dwFirstFrame++;
    // set first frame
    if (f > fDesiredTime)
        dwFirstFrame--;
    if (dwFirstFrame < 0)
        dwFirstFrame = dwTotalAnimFrames - 1;
    else
        if (dwFirstFrame >= dwTotalAnimFrames)
            dwFirstFrame = 0;
    // set second frame
    dwSecondFrame = dwFirstFrame + 1;
    if (dwSecondFrame >= dwTotalAnimFrames)
        dwSecondFrame = 0;
    // calc the percentage
    fPercentage = (fDesiredTime - ((float) dwFirstFrame + fTimePerFrame)) * fTimePerFrame;
} // calculateFramePercentage

First, calculateFramePercentage increments through each frame until it finds the keyframe that is right before the desired time. This assumes that the keyframes each have the same duration. If the keyframes are not set at uniform intervals, this function will have to be changed accordingly.

With the first keyframe found, it is checked against the number of keyframes in the animation. Then, the second keyframe is determined by incrementing the first keyframe number by one. The second keyframe number is also checked against the total number of keyframes in the animation. This code assumes that the animation is going to loop back to the start of the animation after displaying the last keyframe.

It should be noted that the calculateFramePercentage function, as with all of the

4.13 Interpolation

Interpolating

With the new b
demon

Hermite Splin

One does have a comple
interpe
Sir
mines
functions in this article, are presented more for readability than performance. One easy performance improvement is to pre-compute values such as TimePerFrame.

**Interpolating Vertices and Normals**

With the two keyframes and the percentage between them identified, this data can now be used to generate the new animation frame. The `combineVertices` function demonstrates using these values to combine the vertices from the selected keyframes.

```c
void combineVertices(long dwVertexCount, float fPercentage,
                     vector3 *pFirstFrameVertices,
                     vector3 *pSecondFrameVertices,
                     vector3 *pCombinedVertices)
{
    for (long i = 0; i < dwVertexCount; i++, pFirstFrameVertices++,
         pSecondFrameVertices++, pCombinedVertices++)
    {
        *pCombinedVertices = *pFirstFrameVertices +
                             fPercentage * (*pSecondFrameVertices -
                                            *pFirstFrameVertices);
    }
}
```

The percentage that was calculated in `calculateFramePercentage` is used to combine the vertices from the two keyframes into a single new position between them.

This same method of combining the vertices of both keyframes can also be applied to combining the normals of the keyframes. If the keyframe normals were normalized before interpolating, the combined value won't need to be normalized unless there is a large difference between the normal vectors.

There can also be a performance savings if separate lists of face normals (for backface culling) and vertex normals (for lighting) are stored in each keyframe. The face normals must always be interpolated, but interpolating the vertex normals can be skipped if you're trying to improve performance. This means that the vertex lighting won't be correct, but in many situations, the user won't notice the difference.

**Hermite Spline Interpolation**

One drawback to linear interpolation is that some interpolated animation frames may have a tendency to deform to a greater or lesser extent. To solve this, a slightly more complicated interpolation system must be used. This next method, Hermite spline interpolation, takes into account the two keyframes on either side of the desired position (Figure 4.13.2).

Similar to `calculateFramePercentage`, `calculateFramePercentageSpline` determines which keyframes are on either side of the desired animation time. Additionally,
FIGURE 4.13.2 Linearly interpolated position vs. spline interpolated position.

the frames immediately before and after these two keyframes are also calculated. These additional keyframes are used to refine the calculation for the new position.

```c
void calculateFramePercentageSpline(long dwTotalAnimFrames,
    float fTotalAnimTime, float fDesiredTime,
    long &dwFirstFrame, long &dwSecondFrame,
    long &dwThirdFrame, long &dwFourthFrame,
    float &fPercentage)
{
    // determine which frames are involved
    float fTimePerFrame = fTotalAnimTime / (float) dwTotalAnimFrames;
    dwSecondFrame = 0;
    if (fDesiredTime > fTotalAnimTime)
        fDesiredTime -= fTotalAnimTime;
    for (float f = 0.0f; f <= fDesiredTime; f += fTimePerFrame)
        dwSecondFrame++;
    // set second frame
    if (f > fDesiredTime)
        dwSecondFrame --;
    if (dwSecondFrame < 0)
        dwSecondFrame = dwTotalAnimFrames - 1;
    else
        if (dwSecondFrame >= dwTotalAnimFrames)
            dwSecondFrame = 0;
    // set frame before second frame
    dwFirstFrame = dwSecondFrame - 1;
    if (dwFirstFrame < 0)
        dwFirstFrame = dwTotalAnimFrames - 1;
    // set upper frame
```
Interpolated 3D Keyframe Animation

```c
if (dwThirdFrame >= dwTotalAnimFrames)
    dwThirdFrame = 0;
// set frame after the third frame
dwFourthFrame = dwThirdFrame + 1;
if (dwFourthFrame >= dwTotalAnimFrames)
    dwFourthFrame = 0;
// get the upper percent
fPercentage = (fDesiredTime - ((float) dwSecondFrame * fTimePerFrame)) / fTimePerFrame;
} // calculateFramePercentage
```

The positions from the four keyframes are used to calculate the new \( p(t) \) using the following equation:

\[
p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (t^3 - t^2)m_1 + (-2t^3 + 3t^2)p_3
\]

\[
m_i = \frac{1 - \alpha}{2} \left( (p_i - p_{i-1}) + \left( p_{i+1} - p_i \right) \right)
\]

The first and fourth keyframes are used to calculate the tangents \( m_i \) between the first and second keyframes, and the third and fourth keyframes, respectively.

**Interpolating Vertices**

The `combineVerticesSpline` function demonstrates calculating the tangents and then the Hermite spline interpolated position \( p(t) \).

```c
void combineVerticesSpline(long dwVertexCount, float fPercentage,
                           vector3 *pFirstFrameVertices,
                           vector3 *pSecondFrameVertices,
                           vector3 *pThirdFrameVertices,
                           vector3 *pFourthFrameVertices,
                           vector3 *pCombinedVertices)
```

```c
{ float t = fPercentage;
  float t2 = t * t;
  float t3 = t2 * t;
  vector3 m0, m1;
  const float alpha = 0.0f;
  for (long i = 0; i < dwVertexCount; i++, pFirstFrameVertices++, pSecondFrameVertices++,
```

6
Another new addition to this calculation is the variable \( \alpha \). \( \alpha \) controls the tension of the tangent to the spline that being calculated. While \( \alpha \) can be changed to make the tension higher (positive values), or lower (negative values), leaving \( \alpha \) at zero is good enough for most animations.

If you've determined that a fixed value for \( \alpha \) is sufficient for your animation, you can pre-calculate the first part of the tangent equation \( n_t = \frac{(1 - \alpha)}{2} \), and replace it with a constant, 0.5 in this case.

**Why Hermite Splines?**

At first glance, it may seem an odd choice of a Hermite spline over a better known spline such as a B-spline. While B-splines offer additional continuity, this comes at the cost of less control over the tendency of the interpolated curve.

**Summary**

Interpolating keyframe animations is an easy and inexpensive way of improving animation quality. Linear interpolation can be performed for very little cost per vertex. Hermite spline interpolation improves the quality of interpolated keyframes over linear interpolation, but comes at a greater per-vertex cost.

**References**

4.14

A Fast and Simple Skinning Technique

Torgeir Hagland

This article describes a skinning method that is most beneficial for lower polygon characters (less than 500 polygons), where the artists and animators need 100% control over what their vertices are doing. The method can in short be described as a clever way of modifying and sorting an object's vertex list and re-mapping the face list accordingly.

Why Low-Polygon Count?

When dealing with low-polygon count models, each vertex has a big visual impact on how the model's silhouette looks. As an example, let's look at your elbow. The only bones influencing it would be the upper and lower arms. When flexing your biceps, the lower arm influences how the vertices on the inside of your elbow move. It pushes those vertices up from the direction of your lower arm and averages it with the orientation of your upper arm. The end result looks like you have a very thick elbow. This technique only takes into account one bone per vertex.

The Method

The artist creates a single skin model; for example, a space soldier. They then duplicates this skin, scales it down fractionally, and proceeds to cut this smaller skin up into even smaller bits (body parts), which are used as bones. As each bone is created, it is given the same name as the skin with a number appended to it, so it can easily be recognized as a bone by our program.

Once we have identified a skin and its bones, we take the geometry of the bones and store the vertices in one big list. Each entry in this list contains the vertex position and the bone this vertex is a part of.

Now for each vertex in our skin, we find the vertex in the bone list that is closest to it. We transform the skin vertex by the inverse matrix of the bone that the vertex was closest to. This will bring the skin vertex into the local coordinate system of the
bone (in the draw loop, the vertex is transformed back again, so even though the bone-skin is smaller, it has no impact on the end result since the position is relative). The transformed vertex is stored in a temporary list that we accumulate, where we also store the original vertex list index and a pointer to the bone that influences it. The influencing bone has a counter that keeps track of the number of vertices it transforms.

When all the vertices of the skin have a bone influencing them, we process the temporary list that we created. This list is then sorted based on the order of the bones. For each bone, the number of vertices it influences is stored in the original skin vertex list, and the faces must remap the vertices that they reference since we just changed all the vertex indices.

Listing 4.14.1 contains sample code that solves for bone influences and remaps the faces accordingly. Even though this sample code uses the 3D Studio Max file toolkit, the technique can easily be used with any 3D modeling package. I only use it to keep the source size small, and to make sure the focus of this is on the influence solving and draw loop, not the model conversion, etc.

After executing the code in Listing 4.14.1 we have:

- A skin, with each vertex transformed into the local coordinate system of the bone influencing it. The vertex list is sorted by the order of the bones.
- A list of bones, with a counter for how many vertices each bone should transform.

The draw loop for the skin can then be as simple as Listing 4.14.2.

Summary

This method is fast and simple, and works especially well for low-polygon characters. For higher-polygon characters, the edges are smoother, and you will need several bones influencing each vertex. You will also then most likely store two or three pointers for each vertex to the bones that influence them. This means you can no longer pre-store the inverse transformed vertices, and for each frame you need to apply the inverse transform and a percentage-based rotation for each bone that influences it. This causes more of a problem for the tool that creates the influence data. Commercial packages that export bone information do exist, and you no longer have to worry about how the influencing is done, just how to create your draw loop. If you do decide to create the influence tool yourself, I highly recommend making a tool that allows the artist to “paint” influences directly onto the geometry. This way he does not have to second guess a mathematical algorithm.

Listing 4.14.1

```c
void SolveBoneInfluences(database3ds *db, Skin *skinptr)
{
    /* Allocate a big workbuffer */
```
4.14 A Fast and Simple Skinning Technique

BonePoint *bonepointptr=
(BonePoint*)malloc(30000*sizeof(BonePoint));
BonePoint *curbonepoint=bonepointptr;
long NBoneVerts=0;

/* Make all the bones' vertices into one big vertex list with all information on what bone each point came from */

MATRIX tmptmat;
Bone *boneptr=skingptr->BonePtr;
while(boneptr)
{
    mesh3ds *bonemesh=NULL;
    GetMeshByName3ds(db,boneptr->Name,&bonemesh);
    assert(bonemesh);
    Copy3dsMatrix(tmptmat,bonemesh->locMatrix);
    InverseMatrix(tmptmat,bonemesh->Matrix);

    point3ds *bonemeshpoints=bonemesh->vertexarray;
    NBoneVerts+=bonemesh->nvertices;
    assert(NBoneVerts<30000);
    for(int i=0;i<bonemesh->nvertices;i++)
    {
        curbonepoint->Point.x=bonemeshpoints->x;
        curbonepoint->Point.y=bonemeshpoints->y;
        curbonepoint->Point.z=bonemeshpoints->z;
        curbonepoint->BonePtr=boneptr;
        bonemeshpoints++;
        curbonepoint++;
    }
    Rel3dsObj3ds(&bonemesh);
    boneptr=boneptr->NextPtr;
}

mesh3ds *skinmesh = skingptr->MeshPtr;
point3ds *skinemeshpoints = skinmesh->vertexarray;
BonePoint *skinpointptr = (BonePoint*)malloc(

skinemesh->nvertices*sizeof(BonePoint));
BonePoint *curskinpoint = skinpointptr;

/* Find the closest bone vertex to each skin vertex */
for (int i=0;i<skinemesh->nvertices;i++)
{
    curskinpoint->Point.x = skinmeshpoints->x;
    curskinpoint->Point.y = skinmeshpoints->y;
    curskinpoint->Point.z = skinmeshpoints->z;
    /* need to store original vertex index, for face remapping */
    curskinpoint->Index = i;
/* no bone is influencing this bone yet */
curskinpoint->BonePtr = NULL;
curbonepoint=bonepointptr;
float mindist=1e6;
for(int i=0;i<NBoneVerts;i++)
{
    float dist=
        CalcDistNotSquared(skinmeshpoints,
        Acurbonepoint->Point);
    if(dist<mindist)
    {
        mindist=dist;
        curskinpoint->BonePtr=
            curbonepoint->BonePtr;
        curbonepoint++;
    }
curskinpoint++;
    skinmeshpoints++;
}

/* Sort all the vertices of the skin by bone, 
and remap the faces accordingly */
skinmeshpoints = skinmesh->vertexarray;
facets = skinmesh->facearray;
long CurIndex=0;
boneptr=skinptr->BonePtr;
while(boneptr)
{
    curskinpoint=skinpointptr;
    for (i=0;i<skinmesh->vertices;i++)
    {
        if(curskinpoint->BonePtr==boneptr)
        { 
            Transform(boneptr->Matrix,
                        (float*)&curskinpoint->Point,
                        (float*)skinmeshpoints);
            RemapFacelist(skinmesh,
                          curskinpoint->Index,
                          CurIndex++); 
            boneptr->NextVert++;
            skinmeshpoints++;
        }
curskinpoint++;
        boneptr=boneptr->NextPtr;
    }
    /* Clean up after the remapping */
    CleanupFacelist(skinmesh);
    free(skinpointptr);
    free(bonepointptr);
    glEnd();

References

[Lander98] Deform
/* no bone is influencing this bone yet */
curskinpoint->BonePtr = NULL;
curbonepoint=bonepointptr;
float mindist=1e6;
for(int j=0;j<NrBoneVerts;j++)
{
    float dist=
        CalcDistNotSquared(skinmeshpoints,
        &curbonepoint->Point);
    if(dist<mindist)
    {
        mindist=dist;
        curskinpoint->BonePtr=
        curbonepoint->BonePtr;
    }
    curbonepoint++;
}
curskinpoint++;
skinmeshpoints++;

/* Sort all the vertices of the skin by bone, and
   remap the faces accordingly */
skinmeshpoints = skinmesh->vertexarray;
face3ds *skinfaces = skinmesh->facearray;
long CurIndex=0;
boneptr=skinfacet->BonePtr;
while(boneptr)
{
    curskinpoint=skinpointptr;
    for (i=0;i<skinmesh->nvertices;i++)
    {
        if(curskinpoint->BonePtr==boneptr)
        {
            Transform(boneptr->Matrix,
                (float *)&curskinpoint->Point,
                (float *)skinmeshpoints);
            RemapFaceList(skinmesh,
                curskinpoint->index, CurIndex++);
            boneptr->NrVerts++;
            skinmeshpoints++;
        }
        curskinpoint++;
    }
    boneptr=boneptr->nextptr;
}

/* Clean up after the remapping */
CleanUpFaceList(skinmesh);
free(skinpointptr);
free(bonepointptr);
/* no bone is influencing this bone yet */
curbonepoint->BonePtr = NULL;
curbonepoint=bonepointptr;
float mindist=1e6;
for(int j=0; j<NrBoneVerts; j++)
{
  float dist=
  CalcDistNotSquared(skinmeshpoints,
    &curbonepoint->Point);
  if(dist<mindist)
    {
      mindist=dist;
      curskipoint->BonePtr=
        curbonepoint->BonePtr;
    }
  curbonepoint++;
}
curskipoint++;
skinmeshpoints++;

/* Sort all the vertices of the skin by bone, 
and remap the faces accordingly */
skinmeshpoints = skinmesh->vertarray;
face3ds *skinfaces = skinmesh->facearray;
long CurIndex=0;
boneptr=skiptr->BonePtr;
while(boneptr)
{
  curskipoint=skipointptr;
  for (i=0; i<skinmesh->nvertices; i++)
  {
    if(curskipoint->BonePtr==boneptr)
    {
      Transform(boneptr->Matrix,
        (float*)curskipoint->Point,
        (float*)skinmeshpoints);
      RemapFacelist(skinmesh,
        curskipoint->Index, CurIndex++);
      boneptr->NrVerts++;
      skinmeshpoints++;
    }
    curskipoint++;
  }
  boneptr=boneptr->NextPtr;
}

/* Clean up after the remapping */
CleanUpFacelist(skinmesh);
free(skinpointptr);
free(bonepointptr);
Listing 4.14.2

```c
void gIDrawChar()
{
    mesh3ds *meshptr = SkinPtr->MeshPtr;
    Bone *boneptr = SkinPtr->BonePtr;
    point3ds *vertptr = meshptr->vertexarray;
    face3ds *faceptr = meshptr->facearray;

    /* For each bone in the skin, transform X amount
    of vertices with the bone's current animation matrix*/
    point3ds *skinptr = SkinPtr->PointPtr;
    while(boneptr)
    {
        MATRIX mat;
        memcpy(&mat, boneptr->AnimPtr->CurFrame, sizeof(MATRIX));
        for (int i=0;i<vert[boneptr->NVerts];i++)
            TransformMat((float*)vertptr++,
                         (float*)skinptr++);
        boneptr = boneptr->NextPtr;
    }

    /* Then simply draw the object using the facelist*/
    skinptr = SkinPtr->PointPtr;
    glBegin(GL_TRIANGLES);
    glColor3f(1,1,1);
    for (int i=0;i<meshptr->NFaces;i++)
    {
        point3ds *v1 = skinptr[faceptr->v1];
        point3ds *v2 = skinptr[faceptr->v2];
        point3ds *v3 = skinptr[faceptr->v3];
        glVertex3f(v1->x,v1->y,v1->z);
        glVertex3f(v2->x,v2->y,v2->z);
        glVertex3f(v3->x,v3->y,v3->z);
        faceptr++;
    }
    glEnd();
}
```

References

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.
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.

```c
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:

```c
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.

```c
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.
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 think of 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 $1.0 - (weight_1 + weight_2 + ... + weight_n)$.

```c
struct Vertex
{
    float s, t;
    float x, y, z;
    unsigned long color;
};
```
FIGURE 4.15.4 Ugly stitched arm mesh bent to 120 degrees.

```c
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:

\[(\text{vertex } \times \text{ matrix0 } \times \text{ weight0}) + (\text{vertex } \times \text{ matrix1 } \times \text{ weight1}) + \ldots + (\text{vertex } \times \text{ matrixN } \times \text{ weightN})\]

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.

```c
Vector3D TransformVertex ( Vertex *vert, Bone *boneArray )
{
    Vector3D temp;
    Vector3D final;
```
Improved Deformation of Bones

Jason Weber, Intel Corporation
jason.p.weber@intel.com

Artists can produce beautifully realistic meshes to represent the actors in their games. To bring them to life, these models need to be animated for a wide variety of behaviors. Storing a full set of vertex positions for every frame of animation is not only prohibitively memory-consuming, it restricts movements to only those actions explicitly created by the artist. So, many applications use a hidden hierarchy of segments that looks and acts very much like a subset of a natural skeleton.

By transforming each vertex of a mesh through multiple matrices instead of just one, and then by doing a carefully weighted average of the results, we are able to smoothly deform these meshes to any position and play back the results at any frame rate. This approach permits a significant reduction in the required animation data and allows the skeleton, and thereby the mesh, to be spontaneously adjusted to any pose, perhaps reacting to unpredictable events in its environment.

However, the popular deformation algorithm has some problems when used in its original form. We will demonstrate how large deflection angles cause joints to shrink, potentially even to a point. Fortunately, this can be overcome by adding a small chain of additional bones at troublesome joints, such as the elbows and knees. By carefully reworking the weighting data to account for these "links," we can use the same simple core deformation algorithm and only incur the small additional burden of a few extra bones.

Background

The skeletal structure is a hierarchy of transforms, like a scene graph. At each transform, we define a bone length, which is really a displacement along that transform's local x-axis. By default, the origin of all child bones is positioned at the displaced point on the end of the parent bone. We allow for an arbitrary additional displacement, but in most cases where bones connect end-to-end, it is zero.
4.8 Improved Deformation of Bones

A reference pose of the skeleton describes the state of the hierarchy where it aligns with the given undeformed mesh. (Biped, in 3DS Max, calls this the "figure mode.") The motion of the bones away from the reference pose is used to deform the mesh to any arbitrary position. The motion of these bones comes from some driving source, such as motion capture, authored motion, or inverse kinematics [Weber02]. (For a longer description of the background material, refer to the GDC 2000 proceedings [Weber00].)

Simple Methods

The basic skinning algorithm is demonstrated in *Game Programming Gems* by [Woodland00] and is also nicely explained in Jeff Lander's *Game Developer* article [Lander98]. As an example of this technique, consider vertices on an elbow whose position needs to be affected by the current transform of both the upper and lower arm bones. As we pass each vertex in the elbow through each of the two transforms, we get two different resultant vertices. If we then do a weighted average of these transformed vertices, we get a reasonable "in-between" position. An important guideline here is that these weightings for the vertices should smoothly transition from 100% upper arm to 100% lower arm from the top of the elbow to the bottom. Otherwise, the mesh will stretch abnormally and might appear to tear.

As the deflection angle of the lower arm increases and the difference in the results from each bone increases, serious visual abnormalities could arise. For example, the worst case might be if you twist the child bone 180° about its lengthwise axis. The two transformed resultants are on opposite sides of the elbow, so a 50/50 average is at a point inside the elbow. The overall effect is much like twisting or bending a cardboard paper towel tube. Figures 4.8.1 and 4.8.2 show illustrations of problem cases.

![Twisted elbows: a simple skinning method demonstrated on the left arm contrasted with an enhanced technique on the right arm.](image)
Adding the Bones

If large angles cause defects, one solution is to limit all deflections to a small angle. We cannot really restrict the motion of the existing bones, but we can distribute these large angles over several smaller bones conveniently placed at the joints. Since a deflection of 60° seems to be a safe limit, using three or so of these ‘links’ should usually be sufficient. The length of the links can be left to the author of the model. We are often content with a default value derived from the mesh’s cross-sectional radius at the joint and the length of the child bone:

\[ \text{total\_link\_length} = 0.3 \times \text{child\_length} + 1.5 \times \text{joint\_radius} \]

The placement of the bones should look something like a spline. The bones should stay arranged end-to-end, but they may slide together a little as the joint flexes. We don’t have to actually shorten the bones, but the overlap will naturally compress the mesh slightly. Our solution is demonstrated in `GPGBoneNode::CalcBoneLinks()` in `GPGBoneNode.cpp` on the CD-ROM, and it works as follows.

To compute the position of the bones, first consider three points, as shown in Figure 4.8.3a—the original joint connection A, the first link center B, and the last link center C. They will form a triangle as the joint flexes. For each link, take two points, one along BC and the other along either BA or AC, depending on whether the link is in the first or second half of the chain. The displacement of these points along these lines is proportional to the ordering of the link along the chain. Once you have these two points, take a weighted average to find the center for the desired link. The weighting goes linearly from 100% of the point on BC at either the first or last link to...
FIGURE 4.8.3 (A) A small chain of bones added to the joint. (B) Example of new weighting relationships.
50/50 for a link in the exact center. Rotations of the links are computed as the linear interpolation of the overall change in angle so that the total deflection is evenly distributed.

The instantiation and weighting of the links only need to occur once. The position and rotation of the links need to be recomputed every time the parent or child moves, potentially every frame.

**Changing the Weights**

Most of the weights around a bone-linked joint will need to be reassigned. The weights should transition smoothly from the original parent to the first link, then along to each element in the chain, and finally to the original child bone. If there is no fork at this joint and all the original influences were assigned only to this parent and child bone, it is sufficient to use the longitudinal (s) position to find which two bones to attach to (including the new link bones). If we look at the links as an integer number line, with the parent bone as zero and the original child as \((\text{number_of_links} + 2)\), then we use the local s position of the vertex along the chain, scaled to the size of the number line. We find the integer numbers above and below where that vertex falls and use the fractional part to determine the weights. For example, if our local s position scales to 2.3 in 'bone link space,' the new weights will be 70% for the second link bone and 30% for the third link bone.

If there is a fork in the bone hierarchy nearby, more information needs to be considered, even if there is not an explicit sibling to the particular child bone. The hip and shoulder areas are two good examples of this type of situation. The goal for any particular link chain is to only reassign a fair fraction of the parent's influence and leave the remainder of the influence for other bones to consider. For example, a vertex on the chest near the shoulder might be partially influenced by the motion of the upper arm, but it is also well anchored to one or more spinal bones. When adding bones between the clavicle and upper arm, we don't want to reassign the portion that belongs to the spine.

To determine this fair fraction, all the weights are found for the particular vertex that have the same parent as an ancestor to the child in question. These 'competitors' reduce how much influence we can reassign for that child. To make sure the effect is not dependent on the order that the weights are processed, only competitor entries that follow the current weight entry on the weight list are considered. Adding up these competitor weights, we determine the fraction as:

\[
\text{fraction} = 1 - \frac{\text{competitor}}{\text{competitor} + \text{childweight}}
\]

The weight to be reassigned is then:

\[
\text{fraction} \times \text{parentweight} + \text{childweight}
\]

At this point, we continue as in the unforked case. We adjust the existing stored parent weight in place. For our two new weights, we can first overwrite the previous
child entry, which has now been entirely reassigned, and then add a new weight entry for the second influence.

Figure 4.8.4a shows a shoulder joint with reassigned weights. This diagram shows which of the given links the vertices are weighted to. It does not show the magnitude of the weights or any weights to other bones, like the spine. A chain of three links connects the clavicle bone to the upper arm. Lines are drawn from each vertex to the

FIGURE 4.8.4 New weighting with three added links (A) for a shoulder mesh (B).
center of each bone that the vertex is influenced by. The shade of the line matches the shade of the bone, varied for clarity. Note how the links have influences deep into the chest. These create a gradual stretching of the skin when the arm is moved.

**Normal-Derived Influence Fade**

There is one side effect we cannot ignore. Since we are reassigning based purely on position, joints with apparent right angles in the mesh can incur excessive bulging. The shoulders tend to display this problem. When the upper arm moves up, the side of the chest will push out sideways, as though it were a very wide region of the arm (see Figure 4.8.5).

We can use the inherent difference in the normal direction to correct for this. We take the dot product of a vertex's normal with respect to the longitudinal bone axis. From this result, we subtract the x displacement of the vertex along the bone, divided by the approximate radius at the joint (the subtracted value and the result both have a floor of zero). This reduction has the effect of isolating the correction toward the parent side of the joint. The secondary result is a fraction from 0.0 to 1.0, which we square for good measure. This fraction is applied to the weight that was previously destined to reassignment. That portion of the weight is not reassigned, but directly added to the parent's weight. In our example code, see `GPGSkin::RollInWeights()` in `GPGSkin.cpp` on the CD-ROM.

![Figure 4.8.5](https://example.com/image.png) **Figure 4.8.5** Raised arms (A) without and (B) with normal-derived influence fade.
4.8 Improved Deformation of Bones

Packing It Up and Making It Fast

In order to process the weights and generate the deformed mesh as quickly as possible, it is critical that we lay out our runtime data in a cache-friendly manner. The biggest decision to make is whether to process the weights in a 'bone-major' or 'vertex-major' fashion.

For the bone-major method, you have to first clear all the vertex positions and normals in the mesh. Then for each bone, you process all the vertices that the bone influences, accumulating fractional components to their stored positions and normals. While this will probably keep the current matrix in cache, it reads and writes the mesh in a very scattered manner.

In the vertex-major method, we process the vertices in order, pulling in matrices as necessary. While this may incur some scattered access to the matrix array, the advantages are numerous. First of all, there is no clearing stage. We know when the first write to a vertex occurs, so that access can be a pure set instead of an add. Since the weights for a vertex are clumped together, we can accumulate the results in a local variable and dump them out with one write per vertex, instead of one read and one write per weight per vertex.

We have tried it both ways, and for all our measurements, the vertex-major approach was at least twice as fast, and probably much faster, even with all the optimizations we were able to add later on due to the flexibility of the layout.

Transform Matrices

Every bone in the skeleton has a transform, including the added bone links. Until the actual deformation stage, we use quaternions because of their superior interpolative qualities [Bobick98]. However, for raw vertex transforms, using the matrix form is almost four times faster. So, just before the core deformation loop, we fill in a nicely packed array of \(3 \times 4\) matrices, one for each bone. Each matrix is assigned the inverse of the bone's reference transform, multiplied by the bone's current transform. In this way, we can transform directly from the original, undeformed mesh without having to store vertex offsets relative to each bone.

The core deformation loop contains only about 50 lines of code. See GPGSkin::ComputeDeformedVerticesPacked() in GPGSkin.cc on the CD-ROM.

Packweights

Since the vertex weights will usually be a larger data structure than the matrix array or even the mesh, it is important to keep the weight list small in order to reduce the amount of data we need to process each frame. This not only saves space, but it should really optimize our cache usage. However, we also need to be aware that excessive byte conservation might throw off the word alignment, which would be just as detrimental to the process.
The `packweight` structure is a big byte block with alternating sections of one vertex definition and one or more boneweight influences. The vertex definition contains the vertex index, a copy of the undeformed vertex position and normal, and the number of weights to follow. The boneweight block contains just the bone index into the matrix array and the fractional weight. As we read any block, we can prefetch the next one.

Note that storing the vertex position and normal in the weight list means that we can continuously deform to an output mesh without having to retain an undeformed input mesh. If we wanted to allow outside modifications to the input mesh, such as with a morphing modifier, we would not store that data in the weight list, but we would have to take the penalty of rereading vertices from the input mesh every frame. See the file `GPGPackWeights.h` on the CD-ROM for our example code.

**Normal Renormalization**

We can perform a weighted average of multiple normals just like we do with the positions, but the result will have a reduced magnitude. These differences are easy to fix. Since the reduced normals are known to have a range of magnitude from 0 to 1, you can use a modest table to eliminate the `sqrt()` operation. If you do not renormalize them, they could cause a reduction in lighting intensity. Our observations show very little difference, so you might want to consider leaving them as is or hook the option to a quality toggle.

**Conclusion**

Bone-based animation can be a key to reducing animation overhead and allowing for spontaneous and unique behaviors. Existing deformation techniques can be made very fast and are easily extended to overcome some inherent limitations.

Additional topics could cover the generation and manipulation of the original vertex weights. Color Plate 7 demonstrates improvements achieved by using a completely automated procedure to generate raw weights, remove anomalies, smooth the distribution, and add bone links.

The improvement in the waist is mostly due to the regenerated weights. The upper leg benefits dramatically from using the links to reduce shrinkage. Even the shoulder and knees improve significantly by eliminating excessive stretching. Also, the chest looks more realistic, since using links in the shoulder permits a wider spread of influences over the surrounding mesh.

**References**


The author will make an effort to maintain a long-term archive and link site for some related resources at http://www.imonk.com/baboon/bones.
4.9

A Framework for Realistic Character Locomotion

Thomas Young, PathEngine
thomas@pathengine.com

With today's hardware, we can render extremely realistic-looking characters. High-fidelity motion-capture systems are widely available for providing animation data. However, in most games, as soon as a character starts to walk, the illusion of reality is destroyed. The character's feet slide against the ground, the character is rotated arbitrarily in mid-animation, or the animation jumps suddenly to a completely different state.

Even a small amount of foot-sliding is noticeable. As soon as we see a foot slide against the floor, we know that breaking friction has been overcome. Without friction between the foot and the floor, there is no mechanism that a character's forward motion can be attributed to, so the movement of the character is strongly perceived as unrealistic.

It is difficult to solve all the constraints for realistic animation at the same time. We can ensure smooth transition between animations with a tweening modifier. This modifier calculates the in-between (or tween) positions in animation poses. We can solve the problem of arbitrary targets for locomotion by modifying the translation resulting from each animation segment. However, a straightforward implementation of these modifiers will result in foot-sliding.

This gem presents a solution to this problem based on adjusting the position of the feet only when they are already in motion. A framework is described for applying this idea to the problem of realistic character animation by using independent modifiers for different parts of a skeleton.

Problem: Locomotion to an Arbitrary Target

Toward the end of the 2D era, we started to see games such as *Prince of Persia* and *Fate of Black*, which featured beautifully sequenced animations. The trick was that the game environment was built from unit-length tiles on a fixed grid. Since the animation was designed around exactly the same unit-length, it was possible to guarantee that the animation ended perfectly, just pixels in front of a wall or a cliff.
As soon as the possibilities for movement include turning in arbitrary angles and moving forward, the situation becomes more complicated because we can no longer constrain this movement to a fixed grid. Now we are stuck with the problem of characters needing to move arbitrary distances and turn in arbitrary angles.

The character in Figure 4.9.1 has an animation for starting to walk, a walk cycle animation, and an animation for stopping. In Figure 4.9.1a, two walk cycles will not take the character far enough, but in Figure 4.9.1b, three will take the character too far. Figure 4.9.1c shows how the same problem applies when turning arbitrary angles. In this case, the character only has one turn animation. Playing the animation once does not turn enough, but playing it twice turns the character too far. We can improve the situation by providing more animations for a character to choose from, but the basic problem remains.

**Plan and Modify**

We can solve this problem by modifying the translational or rotational offsets for animations as they play. In Figure 4.9.2, the closest set of animations in length is chosen programmatically from the available animations. The offset required to take the end point of the animations to the target is split across the animations and applied as a modification to each animation. The animation sequence now ends exactly at the target point.
In order to modify the translational or rotational result of an animation, we simply apply an offset to the position or orientation of the character origin with a 'ramped' multiplier that goes from 0 to 1 smoothly as the animation is played. However, since this multiplier will change during parts of the animation, such as when a foot is supposed to be stationary, the result is foot-sliding.

**Problem: Smooth Transition Between Animations**

Motion-captured moves will never start or end in exactly the right stance, no matter how good the actor is. Concentrating on hitting the right stance can also have a negative effect on the quality of motion. It is a shame to discard a capture in which the actor got the movement right, just because the capture ends a bit out of stance. With motion-captured animation, we need to transition smoothly between captures that are out of stance. Even if we are working with hand-animated moves that start and end in perfect stance, we still need to be able to transition early out of a move.

**An Approach**

The best result will be achieved by transitioning directly between the state of the skeleton at the end of one motion and the state at the start of the next motion, as opposed to transitioning in and out of a predefined stance. Since we do not want to interrupt the flow of movement, a transition should be made while the animation is playing. For the greatest flexibility, we should not project which animation will play next until we must play that animation. Based on these considerations, a good approach to the problem is to modify the start of each animation to be the same as the
FIGURE 4.9.3  Transition between stances. (A) State of skeleton at end of previous animation. (B) State of skeleton at start of following animation. (C) Applying the same modification later in an animation causes problems at a foot.

end of the previous animation, and then ease out this modification as the animation is played.

The character in Figure 4.9.3a shows the state of the skeleton at the end of the previous animation. Figure 4.9.3b shows the skeletal state at the start of the following animation. We need a modification that will transform the skeleton from the state in Figure 4.9.3b to the state in Figure 4.9.3a. We also need to be able to smoothly transition back to the original position as the second animation plays.

A common approach to solving the transition problem is to store the state of a skeleton as a set of hierarchical relative orientations and to interpolate between these orientations. Orientations are often interpolated with the use of spherical linear interpolation of quaternions. (See [Shankel00] for a more detailed description of quaternion interpolation.) If we use this interpolation method, then our modification takes the form of a set of quaternion offsets for each joint. This modification can be eased out by multiplying the offsets by a 'tween ratio' that goes from 1.0 to 0.0, over the duration of the current animation. Figure 4.9.3b shows how a set of rotations at the joints can take the skeleton to the desired position. The origin for the character in Figure 4.9.3 is between the hips. As the height of the origin can vary with animation, our modification should also include an offset to this height.

Problems with this Approach
The first problem with this kind of straightforward interpolation approach is that for any modification that affects the position of the feet, easing out that modification will also affect the feet. If this happens while the foot is supposed to be stationary against
the floor, then we get foot-sliding. The amount of sliding will depend on the size of
the original discrepancy and the length of time over which our modifier is eased out.

The second problem is that our modifier will give us undesirable results when it
is applied to the skeleton later on in the animation. Figure 4.9.3b shows the rotations
that will be applied by our modifier to the left leg. We know that these rotations result
in a ‘correct’ position for the skeleton when applied to the first frame of animation
because the rotations were chosen to achieve a specific target position when applied at
this point. Figure 4.9.3c shows how the application of the same rotations to the leg
later on in the animation results in the foot hovering above the floor. The interface
between the foot and the floor over a period of animation depends on a combination
of translation at the origin and rotation of the leg. Applying a modifier to the rotation
of the leg breaks this interface. This will result in feet hovering or interpenetrating the
floor, feet being positioned at the wrong angle with respect to the floor, and feet slid-
ing against the floor when they are supposed to be stationary. The problem gets worse
as the skeleton gets further from its start position. We can improve the situation by
easing out our modification more quickly, but this will affect the smoothness of the
transition and make the foot-sliding more pronounced over the ease-out duration. So,
once again, while we have an approach for transitioning between two animations, the
sliding-feet problem still plagues our result.

**A Framework for a Solution: Local Modifiers with Independent Tween Ratios**

By modifying an animation slightly as it is played, we can solve some problems in
character locomotion. However, if this modification affects the positions of the feet
while they are supposed to be stationary against the floor, then there is still a problem.

We can choose when to reduce our tween ratio. If a character jumps in the air
halfway through an animation, then we can delay tweening until both feet leave the
floor and finish tweening by the time the character lands. This would eliminate prob-
lems resulting from changing tween ratio while the feet are on the floor. Unfortu-
nately, most animations will have at least one foot on the floor most of the way
through the animation.

**Affecting Feet Only When They Are Already Moving**

The trick is to use independent modifiers for different parts of a skeleton. This way,
we can ease out each modifier over different sections of the animation. Thus, to mod-
ify an animation without introducing foot-sliding, we use a separate modifier for each
leg. For a walk animation in which the left foot moves first, followed by the right foot,
we ease out the left-leg modifier while the left foot is moving forward, then we wait
until the right foot starts to move before easing out the right-leg modifier.

We can generalize this to any kind of animation and automate the process of
determining when to perform the ease-out for each modifier. Figure 4.9.4 shows a
two-step animation with the corresponding movement profile for each foot. We can
4.9 A Framework for Realistic Character Locomotion

Take the tween ratio for each leg directly from this movement profile. The tween ratio at a given point can be set as the movement up to that point divided by total movement over the course of the animation.

If a foot does not move at all during the animation, or if there is insufficient total movement (and so the rate of twerking would be too fast), then we can choose either to allow some foot-sliding for that animation or allow the modifier to remain at the end of the animation without being eased out completely.

Because the feet will move slightly as a result of error accumulation down the skeleton hierarchy and/or because of error in the original motion capture, it helps to set a threshold for foot movement and ignore any movement below that threshold.

**Application: Locomotion to an Arbitrary Target**

For the problem of locomotion to an arbitrary target, we need the animation to be unmodified at the start, but uniformly offset at the end. By applying an offset at the character origin, we already bring the feet and legs to the correct position by the end of the animation. However, to apply our framework to this problem, we need some way to apply this offset at different times for each foot.

The solution is to keep track of three tween ratios: a straightforward, ramped tween ratio controls a global offset applied to the character origin. Tween ratios, determined from the movement profiles of the feet, keep track of the desired amount of offset at each foot. A modifier is then applied at each leg to correct the difference between the tween ratio already applied by the global modifier and the desired tween ratio for that foot. Figure 4.9.5 shows how this would apply to the two-step animation in Figure 4.9.4. Halfway through the animation, the global tween ratio is 0.5; the left foot has finished its step and therefore should be at 1.0, and the right foot has not moved yet, so it should be at 0.0. To correct the positions of the feet, the left leg needs to be modified by 0.5 and the right leg by −0.5.
A rotational or translational modifier will need to be set up as required to create the same effect as the global modifier, when the modifier is applied with a positive value, or to cancel that effect when the modifier is applied with a negative value. If the character origin is at the hips, then a rotational modifier can simply rotate the leg by changing the orientation of the hip joint. A translational modifier will be more involved, and we have some choices about how to implement this.

**Translational Modifiers**

A translational modifier for a foot needs to apply an offset to the position of that foot and set up the rest of the leg appropriately, without affecting the position of the hips. This is a classic problem for inverse kinematics (IK) (see, for example, [Tolani00]). With an IK approach, a set of constraints is solved for the leg, with the goal of putting the foot in the desired position. We have to take into account the possibility that the IK can fail. In this case, we could put the foot at the closest position that we can achieve within the given constraints. Figure 4.9.6a shows a required offset for the foot. Figure 4.9.6b shows how an IK solution might achieve this offset as a combination of rotations at the joints.

A simpler alternative is to point the ankle in the desired direction by rotating the hip joint and then apply scaling to the leg to bring the ankle to the correct position, as shown in Figure 4.9.6c.

At first glance, the IK solution looks better because it maintains skeletal constraints correctly throughout the animation, but there are some problems with this approach. A straightforward IK solution does not take into account the need for consistency across frames. Small changes in the position of the IK target can lead to big
changes in the position of the leg and inconsistency between frames. These inconsistencies between frames can result in unnatural animation.

The most important constraint for consistent animation is that the modifier should have a very small effect on the leg for a small offset. In order to enforce this constraint in the IK solution, we need to reformulate the problem for IK. We can redefine the problem and find a modification to the angles in the leg in order to achieve an offset to the position of the foot. Unfortunately, there is no guarantee that the position of the leg in the original animation conforms to the constraints of our IK in the first place.

In practice, the simpler approach is recommended (applying a single rotation at the hip and then scaling). This gives us smoother animation and an acceleration at each point in the leg that corresponds better to the original animation. For a small offset, we are guaranteed a small modification. The angle at the knee is also preserved with this method. Because we do not enforce constraints at the hip, however, the skeleton can get into some strange positions; and while using a small amount of scaling on the leg will not be noticeable, any significant amount will look very odd. For these reasons, we should try to avoid using large values for modifiers and also try to avoid leaving modifiers on characters when they are stationary.

Sometimes, even a small amount of scaling can mess up the skinning. One trick we can use as an alternative to a simple scaling is to stretch the leg along the direction of the bones without scaling in the other directions.
Application: Transitions

We can apply the same framework for transitions between animations. By using separate modifiers for each leg, we can eliminate problems resulting from changing the tween ratio while a foot is supposed to be stationary. However, applying a modifier to the angles of the legs while the character origin is moving will still cause problems at the feet.

The solution is to use an ‘anchored modifier.’ This essentially does the same thing as the translational modifiers previously discussed, but also affects the orientation of the foot. Instead of specifying an offset, we specify a target position for the foot in world space (or in the character’s local coordinate system, if that does not change as the animation plays). The target position for the anchored modifier is the position of the foot at the end of the previous animation. Assuming that the foot is correctly placed with respect to the ground at that point, the modifier will ensure that the foot remains correctly placed with respect to the ground until that foot starts moving. As soon as the foot starts moving, the modifier can be eased out.

An anchored modifier will ensure that the position of the feet corresponds to the end of the previous animation, but it does not do anything about the rest of the leg. Even if the hip and the leg do not move across the transition, if the position of the leg in-between those points does not match up across that transition, then the animation can still appear jerky. We can solve this problem by simply applying spherical interpolation to the leg as before, but with the anchored modifier applied to the result of that interpolation.

Further Details

Single-Step Animations

In a two-step animation where both feet move over the course of the animation, we get the chance to ease out modifiers on both legs while the animation is playing. Transition modifiers enable us to generalize the technique also to single-step animations. Any modifiers not eased out by the end of an animation will be dealt with by modifiers at the start of the next animation.

Keeping Characters Moving

In order to make characters look alive, we must keep them moving. We can use a collection of moving-on-the-spot animations to avoid characters standing completely stationary. If these animations include moving the feet slightly, then this gives the animation system a chance to ease out any remaining modifiers.

The player will most likely notice irregularities in a character’s posture when that character comes to a stop. Therefore, it is a good idea to ease out any modifiers when a character stops. If there is a pause key, then the same concerns apply for paused action.