# Realtime Character Animation Blending Using Weighted Skeleton Hierarchies

## Master Thesis

by Daniel Jeppsson (danne@southend-interactive.com)

Department of Computer Science
Lund Institute of Technology
Sweden

**Supervisor:** Mathias Haage, mathias.haage@cs.lth.se, Lund Institute of Technology

**August 2000**

## Abstract

It is both time-consuming and expensive to create animations, regardless if they are created by hand or by using motion-capture equipment. If the animators could reuse old animations and even blend different animations together, a lot of work would be saved in the process. The main objective of this thesis is to examine a method for blending several animations together in realtime. This will make smooth interactive animations possible while keeping the reuse of existing animations in mind. It will also solve the problem of transitioning between two overlapping animations, for example going from a walking to a running animation without resulting in jerky motion. This thesis presents and analyses a solution using Weighted Skeleton hierarchy Animation (WSA) resulting in limited CPU time and memory waste as well as saving time for the animators. The idea presented is described in detail and implemented. Finally the results are analyzed from a visual / realism viewpoint.

## 1. Introduction – About character animation

This chapter explains how most character animation systems work today and what the problems and difficulties are. First of all it describes some mathematical tools needed to interpolate animations and then the two most common methods for 3D animation used in games up until now. A discussion around the problems of connecting animations, also called transitioning, and the idea of blending several different animations together with each other end the chapter. These issues are actually part of the same problem.

The second chapter will start out by introducing the WSA method for solving the problems mentioned above. Details of this method are examined and an algorithm for blending animations together is defined. An animation system prototype using WSA, Weighted Skeleton Animations, is set up and described. The end of the chapter discusses inverse kinematics in conjunction with this system and how to synchronize looping animations together with animation blending.

The third chapter shows results from the developed prototype. The results are subjectively analyzed regarding the realistic “look and feel” of several different blended animations.

The fourth chapter presents the conclusions and ideas for future development. These include inverse kinematics constraints and an advanced collision detection subsystem for better character animation response when colliding with environment and non-environment objects.

## 1.1 Tools for an animation system

Animations can be many things and can be collectively described as pictures or objects that change or move over time. Ten years ago most people would associate the word animation with cartoons. These animations were produced by drawing the frames needed for each scene in a movie. Although many different techniques were employed to simplify the work it still took a lot of time and effort for the animators. These included creating only important frames and doing quick in-between frames as well as using static backdrops with moving characters drawn on top.

Today the word animation mostly makes people think of special effects in movies and computer games. These animations are mostly created in 3D and are much more versatile than the old 2D animations. By moving to three dimensions, the result can be so good that it fools the critical human eye in believing it is real.

A number of mathematical tools are used in the field of 3D animation. First of all, a way to describe the animation from one frame to the next is needed. Some methods use only point positions of the 3D mesh (see 1.2.1) while others use matrices or Euler angles to describe rotations of objects or bodyparts from one frame to another (see 1.2.2-1.2.3). Rotations can also be described as quaternions, which are described mathematically in paragraph 1.1.1.

A way to create frames between keyframes, also called in-between frames, is needed for smooth animation. Depending on the animation method, there are different ways to create these. Point positions can be interpolated using linear or spline interpolation, also called morphing which is covered in section 1.2. Rotations can also be interpolated but trying linear interpolation using Euler angles or rotation matrices will not work very well as can be read about in [2]. Instead quaternions can be used for fast interpolation as seen in paragraph 1.1.3.

### 1.1.1 Quaternions instead of rotation matrices

Just as a single complex number, $z = x + iy$, can be used to specify a point or vector in a 2 dimensional space, a single quaternion, $q = a + bI + cJ + dK$, can be used to specify a point in a 4 dimensional space. A quaternion with $a=0$ can be used to describe vectors in Euclidean 3 space. A quaternion can be re-written as $(s,v)$ where $s = a$, and $v = bI + cJ + dK$. s is the scalar part of the quaternion, and v is the vector part.

What has all this to do with rotations? Any rotation in Euclidean 3 space can be described with a direction vector and an rotation angle, in effect a vector on the

unit 4D sphere (s,x,y,z). For more information about quaternions, look in [2] or [3]. All quaternions discussed below are unit quaternions i.e. the length of the 4D vector is one.

It can be shown that a quaternion is equivalent to using Euler angles or rotation matrices. For instance, the quaternion q = (s,v) = $(W,(X,Y,Z))$ = $(\cos(f/2),\sin(f/2)*n)$ is equivalent to the rotation matrix (proof of this can be found in [2]):

$$\begin{bmatrix} 1-2Y^2-2Z^2 & 2XY-2WZ & 2XZ+2WY & 0 \\ 2XY+2WZ & 1-2X^2-2Z^2 & 2YZ-2WX & 0 \\ 2XZ-2WY & 2YZ+2WX & 1-2X^2-2Y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating a point using quaternions is defined as $R_q(p) = q*p*q^{-1,}$ where multiplication of quaternions are defined as:
$q_1 * q_2 = (s_1 s_2 - v_1 \cdot v_2, s_1 v_2 + s_2 v_1 + v_1 \times v_2)$
where v is the vector part of the quaternion and s is the scalar part.

Quaternions offer some advantages over rotation matrices and Euler angles:

- **Better accuracy using floating points and avoids drifting matrices**. A drifting matrix appears when rotations are added to the same matrix over time. The rounding errors of floating point math will add up and result in visual errors (drifting).

- **Interpolations of rotations are possible** (see 1.1.3). This is very hard to do using rotation matrices or Euler angles. [2] shows some problems of interpolation using Euler angles.

- **Multiple rotations can be calculated faster using quaternions**. A standard 4x4 matrix multiplication uses 64 multiplications and 48 additions while a quaternion multiplication only uses 16 multiplications and 12 additions.

- **No "Gimbal-lock" problem.** This appears when a rotation decreases the total degrees of freedom (DOF), in the animation part. Read more about "Gimbal-locks" in [2].

To be able to use quaternions there should exist fast conversion routines between Euler angles, rotation matrices and quaternions. This is because most 3D engines usually uses different ways to describe rotations depending on what is convenient at the time and therefore need quick conversions for good performance. Appendix B shows how this is done in practice. Appendix B also

shows some important mathematical operations that can be used on quaternions like inverses, multiplications and more.

### 1.1.2 Interpolation of points (morphing)

If a node is keyframed to the position $P_1$: $(x_1,y_1,z_1)$ at the time $t_1$ and the next key is positioned at $P_2$: $(x_2,y_2,z_2)$ at the time $t_2$ the easiest way to find the approximate position at a time t somewhere between $t_1$ and $t_2$ is to linearly interpolate between the two positions (keyframes).

$$P_t = P_1 + (P_2 - P_1) * \frac{t - t_1}{t_2 - t_1} \qquad (t_1 \le t \le t_2)$$

See figure 1.1.

Several ways exist to improve the results of simple linear interpolation, for example interpolation using splines. See [1] and [2].

### 1.1.3 Interpolation of rotations (SLERP)

Many methods use keyframed rotations and use these rotations to simulate motion, for instance skeleton-animation L-SKA and D-SKA. If these frames are stored as rotation-matrices, in the keys, interpolation is difficult to achieve. This is where quaternions may help. A method called SLERP (Spherical Linear intERPolation) interpolates between two unit quaternions along the shortest arc on the unit sphere. The SLERP function is defined as:

$$SLERP(q_1, q_2, u) = q_1 \frac{\sin((1-u)f)}{\sin f} + q_2 \frac{\sin uf}{\sin f} \qquad \begin{matrix} q_1 \cdot q_2 = \cos(f) \\ u \in [0,1] \end{matrix}$$

Where $q_1$ and $q_2$ are two quaternions (rotations) and u is the fraction between these rotations along the 4D sphere that we are interested in, remember that a quaternion is a vector in 4D. Appendix B shows how this is done in practice.

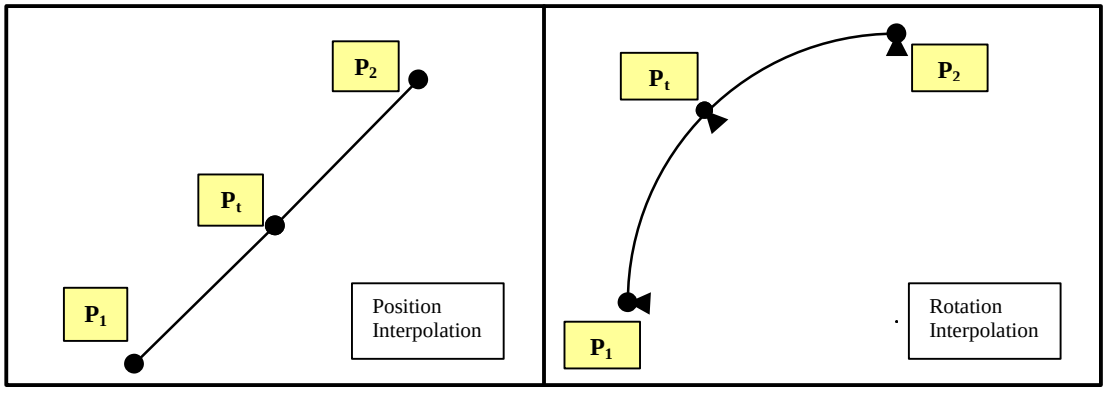



**Fig 1.1** The figure shows position interpolation between two positions in time and rotation interpolation between two rotations in time.

## 1.2 Animation techniques

To animate complex objects, such as the human body, today's 3D games mainly use two different methods. Since many different names appear in the literature I use my own terms, Vertex Animation (1.2.1) and Skeleton Animation (1.2.2-1.2.3), abbreviated in this text to VEA and SKA. VEA animates the mesh directly while SKA animates a mesh through a skeleton.

Those using SKA have two ways of binding their animated skeleton hierarchy to the mesh involved, here called Linked and Deformed Skeleton Animation (L-SKA and D-SKA).

This work focuses on the use of SKA. The information needed for animation blending is not easily available inside the VEA animation method and in VEA only the vertex positions are stored for each keyframe, making it very hard or impossible to extract animation data for different bones. Instead VEA relies on mesh morphing, to linearly interpolate meshes (see 1.2.1). Figure 1.2 shows how this kind of blending would look like with our running and waving animation.

**Fig 1.2** This figure shows a vertex morph from a running keyframe to a waving keyframe in six steps. In the figure we can see how some limbs get deformed, most noticibly the right foot and right arm.

The figure shows how deformation of bodyparts occurs, using vertex morphing. The result can be much worse if the animations are totally different. For example, if the body was rotated, the result would look like the one in figure 1.3.

**Fig 1.3** This figure shows the same morph as the one in fig 1.1 but with the wave animation turned in the opposite direction resulting in unusable interpolation steps.

### 1.2.1 Vertex Animation VEA

When using this method the animator creates his animations and saves all information for the entire mesh for every keyframe. Each keyframe includes information about changes in position for each vertex in the mesh.

The main advantage of VEA is total freedom for the animator when creating the animation. Any kind of motion and flexing deformations can be done limited only by the 3D software package used for creating the animation. Detailed and realistic animations where muscles bulge and the number of vertices increase when needed are possible.

Some major disadvantages exist with this method. The animation data is huge in size even when compressed, and is inflexible since the animator must decide on the maximum animation framerate when exporting the animation. Figure 1.8 compares the memory usage for the different methods. No general way to blend animations exists other than interpolation between vertex positions (linear or splinebased interpolation) i.e. morphing, which can deform the object severely. (See fig 1.2 and fig 1.3 above and read section 1.2)

This method was common a few years ago since it is low on processor load and is simple to implement. The demand for interactivity wasn't too high in those early days of 3D games either. If a character could not run while shooting, it wasn't a big deal because of the huge leap from 2D to 3D anyway.

Examples where it is used within the computer games industry are easy to find. The popular game Quake by ID Software and the successors Quake II and Quake III Arena uses VEA, although Q3A has a somewhat improved version where they have split their characters in three parts, head, body and legs. This way the animators can blend some animations together in a simple way.

### 1.2.2 Linked Skeleton Animation L-SKA

To decrease the size of animations and to make interpolation easier, another method for character animation is widely used today. The animator splits the character or object in a number of body parts and links them together in the joints (bones) forming a skeleton. He can then bind every bone to a separate part of the original split character mesh and rotate or move these bones of the skeleton. By doing this, only the bones position / rotation and the binding information needs to be stored to animate the mesh.

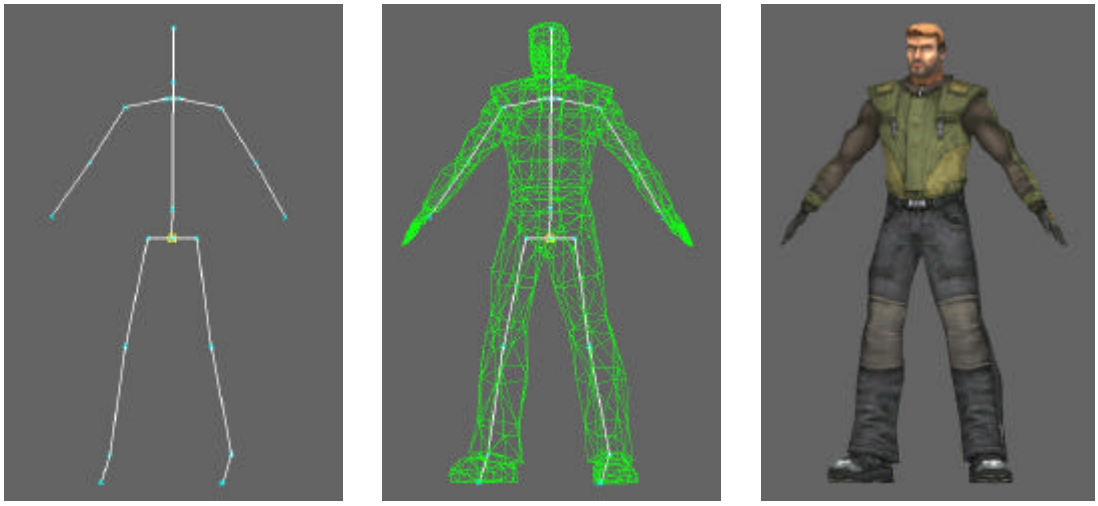


**Fig 1.4** This figure shows a character as a simple skeleton, with a mesh attached and fully textured.

The two main advantages with L-SKA are a radically decreased data size compared to VEA and the fact that the computation load is still cheap. Figure 1.8 compares the memory usage in the different methods. Another advantage is the ability to interpolate in-between frames using quaternions and SLERP interpolation between keyframes!

The greatest disadvantage is the visual errors that occur when two separate bodyparts rotate into and/or penetrate each other. Figure 1.7 shows what this visual error can look like. A reason for this is the fact that different bodyparts are not considered to affect parts connected to them. Taking this into consideration lead to deformed skeleton animation, D-SKA, which is covered in paragraph 1.2.3.

L-SKA has been used in such titles as LucasArt's Jedi Knight and Novalogic's Delta Force.

### 1.2.3 Deformed Skeleton Animation D-SKA

To remove the visual errors of L-SKA, another way to bind the skeleton to the mesh is used. The modeler or the animator binds every vertex in the mesh to different bones in the skeleton. Using weights for each binding (vertex to bone), the problem of L-SKA is reduced. Every vertex can be bound to one or more bones and the weights are used to interpolate the final position based upon the rotation and position of the bones. The figure below shows how different vertices are bound to different skeleton joints in a geometrically modeled hand. Black vertices are bound to a single bone, gray vertices are bound to several bones.

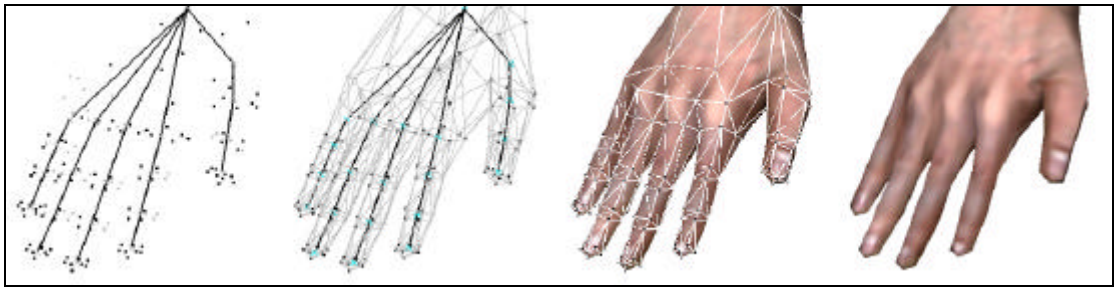


**Fig 1.5** This figure shows a character hand using 20 bones. It also shows how different vertices are bound to different bones. The vertices between bones (gray) are bound to both bones, and the rest (black) are bound to a single bone.

No partitioning of the mesh is needed and the same skeleton and animation data can be used on several different meshes. (See fig 1.6)

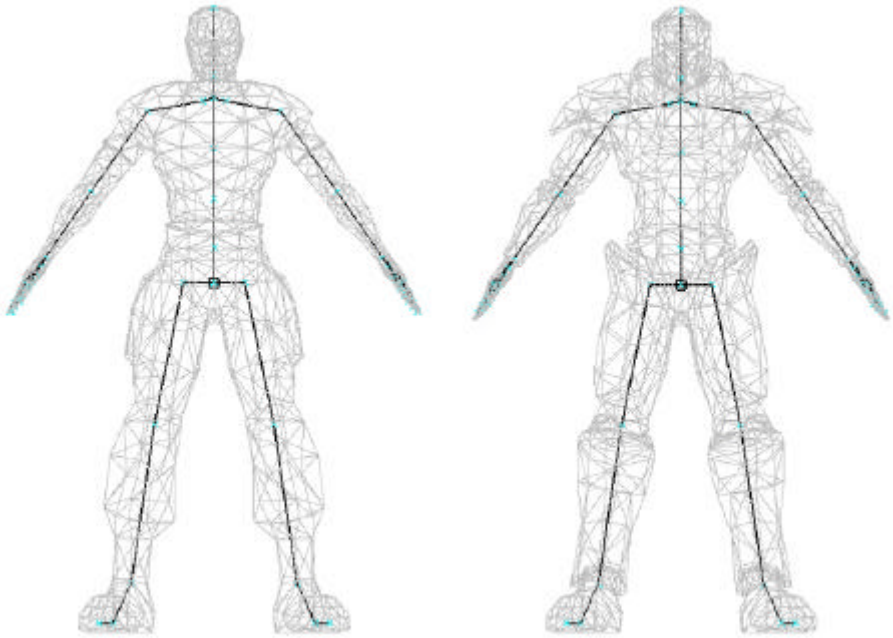


**Fig 1.6** This figure shows two different meshes using the same skeleton which means the same animation data can also be used. Since the animation data for a skeleton is based on rotations, a scaled version of the same skeleton could also be used for characters of different sizes and shapes.

The advantages of D-SKA over L-SKA are visual improvements (see figure 1.6) and less data size. (See the table in figure 1.8 for comparisons)

**Fig 1.7** This figure shows the visual difference between L-SKA (the leftmost figure) and D-SKA (the other two figures).

Two disadvantages exist. The first is the increased computational cost compared to L-SKA since all vertices bound to more than one bone must be transformed one time for each and every bone. The second is the fact that some meshes still might experience strange deformations in extreme positions if too few vertices are used near the bending joint. See the rightmost arm shown in figure 1.7.

D-SKA is becoming increasingly popular and is used in such famous titles as Valve's Halflife and Shiny Entertainment's Messiah.

### 1.2.4 Memory usage

In table 1.8 below, three different animations are tested using two different character meshes. The tests are made using VEA and SKA methods for animation.

The VEA memory usage has been calculated as the number of vertices multiplied with the number of frames in the animation loop. This number is then multiplied with 12 as in the size of three floating-point precision numbers. The "packed memory usage" column assumes that only the moving points are stored for each frame to save memory. The "interpolated memory usage" column assumes that points moving only a little distance can be interpolated and therefore skips many frames in the animation which can degrade the animation quality if compressed too much.

The SKA memory usage has been calculated as the number of bones multiplied with the number of frames in the animation loop. This number is then multiplied with 16 as in the size of four floating-point precision numbers. The "packed memory usage" column assumes that only the rotating bones are stored for each frame to save memory. The "interpolated memory usage" column assumes that only important keyframes are stored and the in-between frames are interpolated. This can degrade the animation quality if compressed too much.

The numbers in the table are approximate but should give a hint of the memory advantage of using SKA over VEA.

| Method | Animation | # Vertices | # Bones (SKA) | # Frames | Memory Usage | Packed Memory Usage | Interp. Memory Usage |
|---|---|---|---|---|---|---|---|
| VEA | Running Loop | 1000 | N/A | 16* | 192 kB | 150 kB | 100 kB |
| VEA | Running Loop | 2500 | N/A | 16* | 480 kB | 375 kB | 250 kB |
| SKA | Running Loop | 1000*** | 30 | 16** | 8 kB | 7 kB | 6 kB |
| SKA | Running Loop | 2500*** | 60 | 16** | 16 kB | 14 kB | 10 kB |
| | | | | | | | |
| VEA | Walking Loop | 1000 | N/A | 30* | 360 kB | 250 kB | 150 kB |
| VEA | Walking Loop | 2500 | N/A | 30* | 900 kB | 650 kB | 400 kB |
| SKA | Walking Loop | 1000*** | 30 | 30** | 15 kB | 14 kB | 9 kB |
| SKA | Walking Loop | 2500*** | 60 | 30** | 30 kB | 28 kB | 14 kB |
| | | | | | | | |
| VEA | Waving | 1000 | N/A | 60* | 720 kB | 350 kB | 200 kB |
| VEA | Waving | 2500 | N/A | 60* | 1800 kB | 800 kB | 350 kB |
| SKA | Waving | 1000*** | 30 | 60** | 30 kB | 19 kB | 8 kB |
| SKA | Waving | 2500*** | 60 | 60** | 60 kB | 37 kB | 15 kB |

* VEA would use even more memory if a higher number of frames per second was desirable.
** SKA can interpolate new frames without experiencing the visual artifacts of linear morphing.
*** Number of vertices in the mesh does not increase memory size for SKA animations.

**Fig 1.8** A table of approximate figures comparing a calculated memory usage of vertex animation (VEA) with skeleton animation (SKA). Please note that the figures are approximate calculations and maybe not fully optimized. The Packed Memory Usage column uses some scheme for packing the animation data and the Interpolated Memory Usage column uses the fact that small movements can be interpolated for even smaller data size.

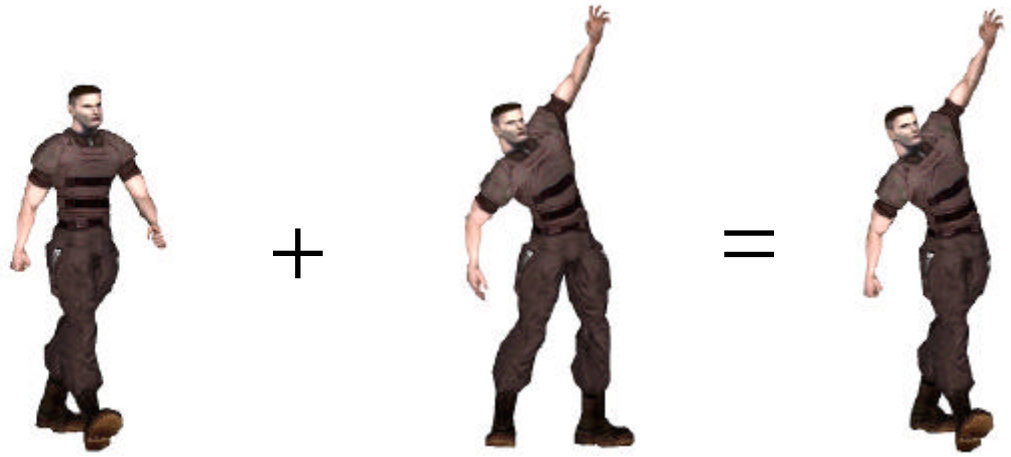
## 1. 3 Animation blending and transitioning

Games in the past have relied upon hard work and advanced animation software as the main solution to animation blending. If a running, waving character was needed, the animators had to animate a running, waving character without being able to reuse old exported animations.

If a transition between a walking and running character was needed it had to be hand-made. This also required the character to be in a special transition state before being able to make the switch from walking to running, destroying interactivity by introducing delays. If interactivity was important, the jerk between the running animation and the walking animation was very apparent and ruined the smoothness of the animation.

An ability to blend different animations together and being able to smoothly transit between animations are very useful tools in computer animation, especially for interactive graphics such as needed in the computer games industry.

In the next chapter, a method will be presented to solve both transitioning and more generic blends with only minor preparatory work needed by the animator.



**(1)** Walking Animation **(2)** Waving Animation **(3)** Blended Animation **(1)+(2)**

**Fig 1.9** This figure shows the result of a (1) Walking animation, (2) Waving animation and the (3) Blended animation.

## 2. Animation blending and transitioning using WSA

This chapter will introduce WSA as a way to blend several animations together without jerky motion between frames or different animations. The human eye is very astute in noticing jerky or unnatural movement, especially in human characters, and it is therefore important to remove such behavior in an animation system. Jerkiness in a skeleton animation system can be defined as movement of bones in too large steps between frames, or movement of bones at different speeds during a short time. The first case occurs when the computer has a constant low framerate and the second case when framerate changes quickly or when an animation makes alternating quick and slow movements.

An animation using D-SKA stores the rotation of each bone in a skeleton hierarchy, preferably in quaternion format for easy interpolation between frames using SLERP. All rotations are stored on a per-frame basis.

The problem with D-SKA is that not enough information is available to enable blending. For example, it is not known how important different animations are compared to each other. In an animation where the character waves his arm, the arm is very important but his lower body is not.

WSA solves this problem by attaching a weight to each bone that tells how important it is in the overall animation. This means the animator must prioritize different bodyparts (bones) in an animation. By defining weight functions w(t) for the bones in an animation the extra information needed to blend different animations together can be obtained. The functions can be constant for the entire animation sequence or more advanced if needs be.

Approximately the same idea is applied when binding a skeleton to the mesh in D-SKA where vertices belonging to different bones are weighted between joints. WSA extends D-SKA with weight functions w(t) to enable blending and transitioning of animations.

For example, a running animation will have "heavy" weight functions for the legs and hips but only "light" weight functions for the arms. The arm-waving animation will have "heavy" weight functions for the waving arm and "light" weight functions for the rest of the skeleton. See figure 2.1.

| | Lower Arm Left | Upper Arm Left | Lower Leg Left | Upper Leg Left | Head | Neck | Spine |
|---|---|---|---|---|---|---|---|
| Running | L(t) | L(t) | H(t) | H(t) | L(t) | L(t) | L(t) |
| Waving | H(t) | H(t) | L(t) | L(t) | L(t) | L(t) | L(t) |

**Fig 2.1** The figure shows a table of weight functions for different animations and bodyparts. L(t) is a ”light” weight function and H(t) is a ”heavy” weight function.

Given WSA, what kind of function should w(t) be; constant, linear or some sort of spline function over time? Another question is how to interpolate two animations based on these weights to achieve a blended animation?

## 2.1 Transitional blending

Transitional blending means smoothly switching from one animation to another. Figure 2.2 illustrates an example where a switching occurs from walking to running in an animation. During a transition from one animation to another, the first animation is gently increased in weight as the other gently decreases.

**Fig 2.2** The figure shows a transition blending from a walking to a running animation.

Transitional blending is a specialized case of the more general animation blending technique illustrated in figure 2.3.

## 2.2 Animation blending

The more general case of animation blending is to blend together two or more animations at the same time. By using weight functions to weight interpolations (SLERP) among different animations, blending and transitioning can be achieved.

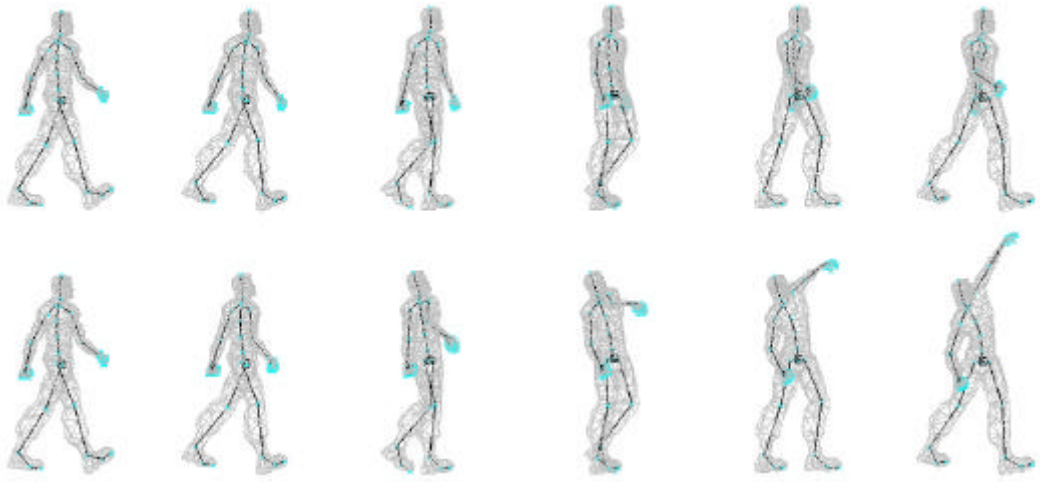


**Fig 2.3** The first row shows a walking animation. The second shows a blend between a walking and a waving animation.

## 2.3 The interpolation of animation sequences

An animation sequence consists of a sequence of quaternion rotation data for each bone in a skeleton hierarchy. How can we blend different animation sequences together if their weight functions are known?

Two quaternion rotations coming from two different animations can be blended by utilizing the SLERP function:

$Q_{total}(t) = SLERP( Q_1(t), Q_2(t), I(w_1(t),w_2(t)))$

where $Q_{total}(t)$ is the end rotation of the bone and Q(t) is the rotation of the same bone for two different animations. $I(w_1(t),w_2(t))$ is a function between 0.0 and 1.0 that controls how different weights should be compared. If $I(w_1(t),w_2(t))$ is 0.0 the SLERP result is simply the $Q_1(t)$ rotation and if $I(w_1(t),w_2(t))$ is 1.0 the result is the $Q_2(t)$ rotation. For values in-between the SLERP function returns the linearly interpolated rotation on the unit 4D sphere.

In this same way an arbitrary number of animations can be blended together. The only remaining problem is deciding how $I(w_1(t),w_2(t))$ should look for different weight functions $w_1(t)$ and $w_2(t)$. More on this subject later.

SLERP-interpolation of animations may result in unnatural movement, but adding IK constraints increases the computational cost so we will begin by trying out WSA on the original constraints inherited from the basic animations and see how it works out.

## 2.4 The character animation system prototype

This section introduces the developed prototype that utilizes WSA for character animation blending.

The design goals for this animation system were:

- To be able to transition between different animations
- To be able to blend an arbitrary number of animations together
- To perform smooth transitions, lead-ins and lead-outs

A Lead-In is a function that smoothly increases the weight of an animation until it is running with full weight. The Lead-Out is the reverse, a function that smoothly decreases the weight of an animation until it is removed.

The system should work as a black box. We should only tell it when to start animations and end animations, with parameters like lead-in time, lead-out time, loop, etc. The system should then produce smooth animations.

A typical code example using this system should be something like:

```
// Play a walking animation loop with Lead-In time 0.1s.
animationSystem.playAnimationLoop(walkingAnimation, 0.1);

// Play a single waving animation cycle with Lead-In time 0.1s and Lead-Out 0.1s.
animationSystem.playAnimationSingle(wavingAnimation, 0.1, 0.1);
```

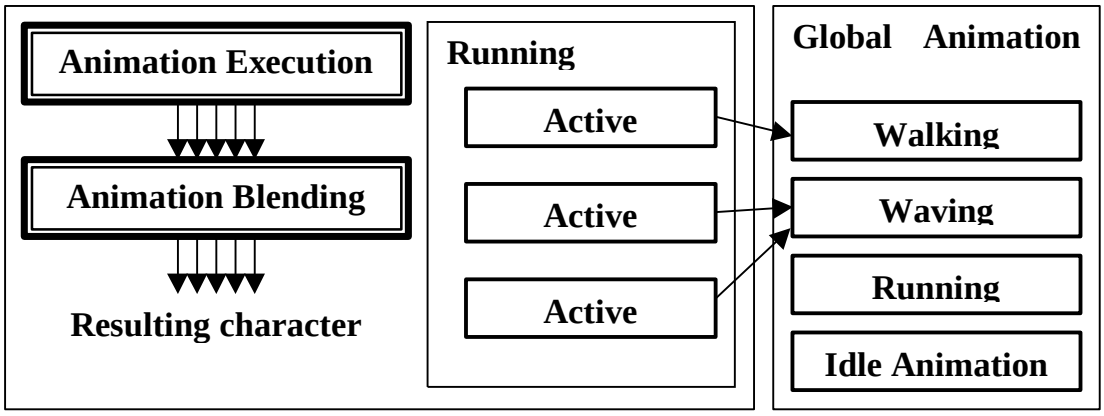



**Fig 2.4** This figure shows an overview of the animation system.

A running animation is an animation that should be blended into the resulting animation. If it is active it currently goes from one frame to the next using a set speed or framerate and if it is deactivated the animation stays on the same frame until it is activated again. An animation either has a starting and an ending point if it should run once, or only a starting point if it is a looping animation. Looping animations can also have an ending point but is usually stopped as an event in the game, not at a specific time decided from the start of the animation.

Figure 2.4 shows an overview of the main components in the prototype. To the left is the Animation Execution unit that takes care of the active animations and removes finished animations from the Running Animations list as well as handles looping and makes them go from one frame to the next. The system then blends these running animations, shown in the middle of figure 2.4 using the Animation Blending unit, and delivers the resulting character animation. The set of global animations to the right in the figure can be shared by different characters, each with its own animation system. Being able to use the same animation data on several characters saves a lot of memory in most applications using multiple character animations.

The most interesting component of this setup is the Animation Blending Unit and the problem of how it should blend the active animations together without jerky movement between frames or animations.

### 2.4.1 Animation Blending Unit

As mentioned earlier we have a powerful mathematical tool at our disposal, the SLERP function, (interpolation of rotations on the unit sphere). With it the system can interpolate between two different resulting rotations. The only problem using it is calculating the $I(w_1(t),w_2(t))$ function, a function between 0.0 and 1.0 that decides where between the two rotations the resulting rotation should be, i.e. the blending factor.

In short, when the system arrives at this point, it has a list of active animations, which frame and sub-frame they are in and Lead-In / Lead-Out times for them. (See figure 2.5 for an example). With sub-frame I mean the point between two frames in an animation. For example, a running animation could be in frame 1.5 that means halfway between frame 1 and 2.

| Active Animation | Time (frames) | Length (#frames) | Loop | Lead-In (#frames) | Lead-Out (#frames) |
|---|---|---|---|---|---|
| Walking | 5.5 | 10.0 | TRUE | - | - |
| Waving | 3.6 | 20.0 | FALSE | 5.0 | 5.0 |

**Fig 2.5** An example of two running animations to be blended, a walking and a waving animation.

In the figure above, the character is currently walking and waving at the same time. He happens to be exactly halfway between keyframe 5 and keyframe 6 of the walking animation which is 10 frames long, and has just begun waving his arm being little more than halfway between keyframe 3 and 4 in the 20 frames long waving animation.

The walking animation is a looping animation and the waving animation should be smoothly blended in with a Lead-In since the Time is less than the Lead-In time.

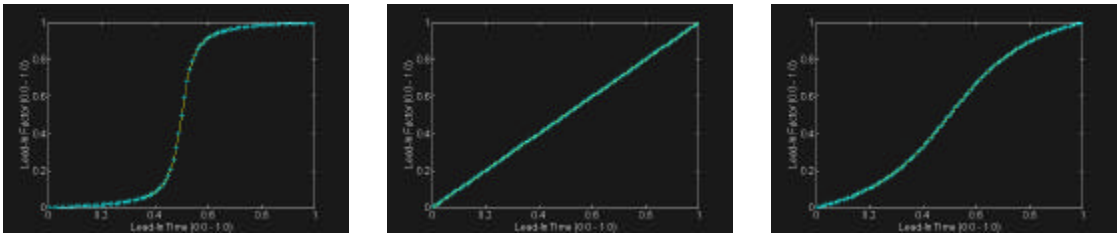


**Fig 2.6** Three different lead-in functions going from 0.0 to 1.0.

This leads to the question what the Lead-In / Lead-Out functions should look like. The three functions in figure 2.6 shows different possible functions to use. All of them result in jerky motion in special cases though the first one worked for almost all test cases.

A lot of experimenting arrived at the conclusion that the Lead-In / Lead-Out functions for a specific animation should depend on the currently active set of animations. This is because of the large differences in weight functions between some skeleton joints and certain animations.

For example, the arm in the waving animation had a way of jerking when starting the Lead-In transition. This happened because of a factor difference between the arm weight in the walking animation, which was already active and the new waving animation which was in a Lead-In transition. In this particular case the arm had a very small weight, 10 for some frame in the walking animation, and an extremely heavy weight, 100.000 for some frame in the waving animation. The Lead-In function should either take this into account, or the system would need better weight-functions for the animation taking Lead-In and Lead-Out into account as well. The latter case would be too specialized and wouldn't work for some cases so the first alternative should be used which changes with the animations already active.

A method for blending animations together will now be described. The method uses a summation function that for parameters takes any number of quaternion rotations, one for each animation $A_1$, $A_2$, etc., their weight for a specified frame and then calculates a final output rotation.

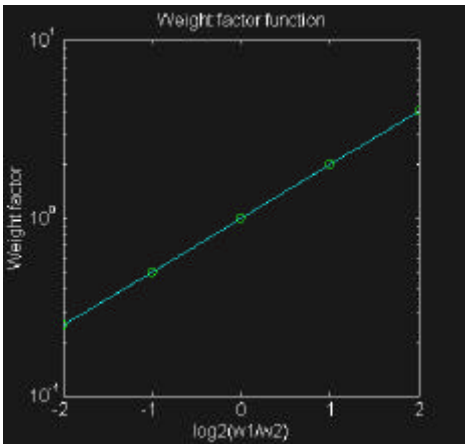
$$Qres = \sum Q(A_i) = Q(A_1) + Q(A_2) + Q(A_3) + \ldots$$

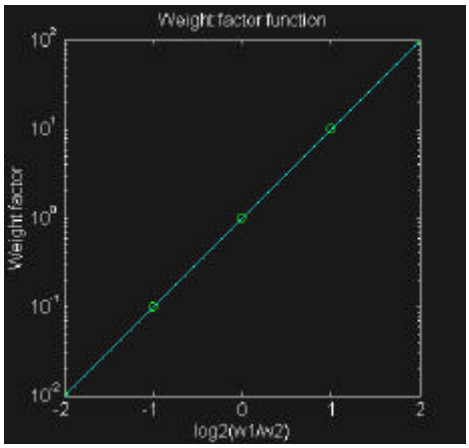
$$Q(A_1) + Q(A_2) = SLERP(Q(A_1), Q(A_2), I(w(A_1), w(A_2)))$$

The last problem of blending quaternion rotations together is to define $I(w(A_1),w(A_2))$ which is a function from 0.0 to 1.0 deciding how to blend two rotations into a single rotation. This rotation will have a new weight $w_{res}$ which can be calculated as the combined weight $w(A_1+A_2) = w(A_1)+ w(A_2)$ as well. This can be used for adding more than two animations together:

$$Q(A_1) + Q(A_2) + Q(A_3) = SLERP(SLERP(Q(A_1), Q(A_2), I(w(A_1), w(A_2))), Q(A_3), I(w(A_1 + A_2), w(A_3)))$$

How should $I(w_1(t),w_2(t))$ be defined? When first introduced I only said it was a function that compares to weights and returns a value from 0.0 to 1.0. If an animation is running with a ten times higher weight than another, should it be a linear factor 10 times more in that direction than the other? This is one way to do it of course but to almost totally take over an animation in some bones, the weights would have to be set ridiculously high for those bones in that animation. Typically you would want to set the weight to only two or three times the weight of another animation and still expect it to be the totally dominating one when compared to each other.




(1) $I_1(w_1, w_2) = \frac{1}{2^{\log(w_1 / w_2) / \log(2)} + 1}$

(2) $I_2(w_1, w_2) = \frac{1}{10^{\log(w_1 / w_2) / \log(2)} + 1}$

**Fig 2.7** Two different $I(w_1(t),w_2(t))$ functions.

Figure 2.7 shows two different $I(w_1(t),w_2(t))$ functions (the figures only show the denominator part of the equations for clarity) with function (2) being much stronger in the sense that a small difference in weight means a much greater dominating presence of the animation with the heavier weight.

## 2.5 Inverse Kinematics Joints

In addition to using prerecorded animation data, the system allows for other animation controllers to be used. One is the Inverse Kinematics Joint controller (IKJ) which can animate a character interactively. For example, the character neck could be controlled by an IKJ and interactively make the character look in different directions. With WSA the controller just blends into the other animations.

Real inverse kinematics could also be attached to the system to enable the character to point with his hand and arm or do other interactive animations blended with motion-captured animations. This would be really useful with several interacting characters as well.

## 2.6 Synchronized transitions between looped animations

To be able to transition between looping animations in a realistic manner, for example switching from a walking to a running animation, the two animations must comply with a single rule. To always start and stop in the same way. For example, starting out with the left foot going forward in both the walking and the running animation.

If this rule is followed, the animation prototype is able to do a transitioning between two looping animations. The system calculates the speed of both

animations and interpolates a transition from the first animation to the other. The speed is used to calculate what frame should be shown in both animations.

To see why this is so, consider a transition from a walking animation loop to a running animation loop. The walking animation is slower than the running animation and has therefore more frames. A transition will look natural if the speed of the two animations slowly blend together. If we have a framerate of 20 frames per second, a walking animation with 20 frames and a running animation with 10 frames, the speeds of the animations are 1 complete walking cycle per second and 2 complete running cycles per second. These speeds must now be interpolated from the first to the second if a transition is to be made. Using this scheme for looped animations removes the strange motions which can appear when transitioning.

# 3. Animation engine prototype result

This chapter shows the program developed to try out the WSA animation system. Some resulting animations that were created are shown and analyzed.

## 3.1 The WSA Animator

Figure 3.1 shows the output window of the BDA[1] Animator program written by the author using the WSA animation system. The viewport to the right shows a character system as vertices, edges and bones. This is the user view where the user can rotate the character around to get a better view while binding the mesh to the skeleton and testing various animations. The three smaller viewports to the left in figure 3.1 shows the top, front and left of the character with different view settings. Top and left view shows only the skeleton while the front view shows polygons and light with no texture attached. These viewports can be used for a better view of the animations and the user may configure them according to preference.

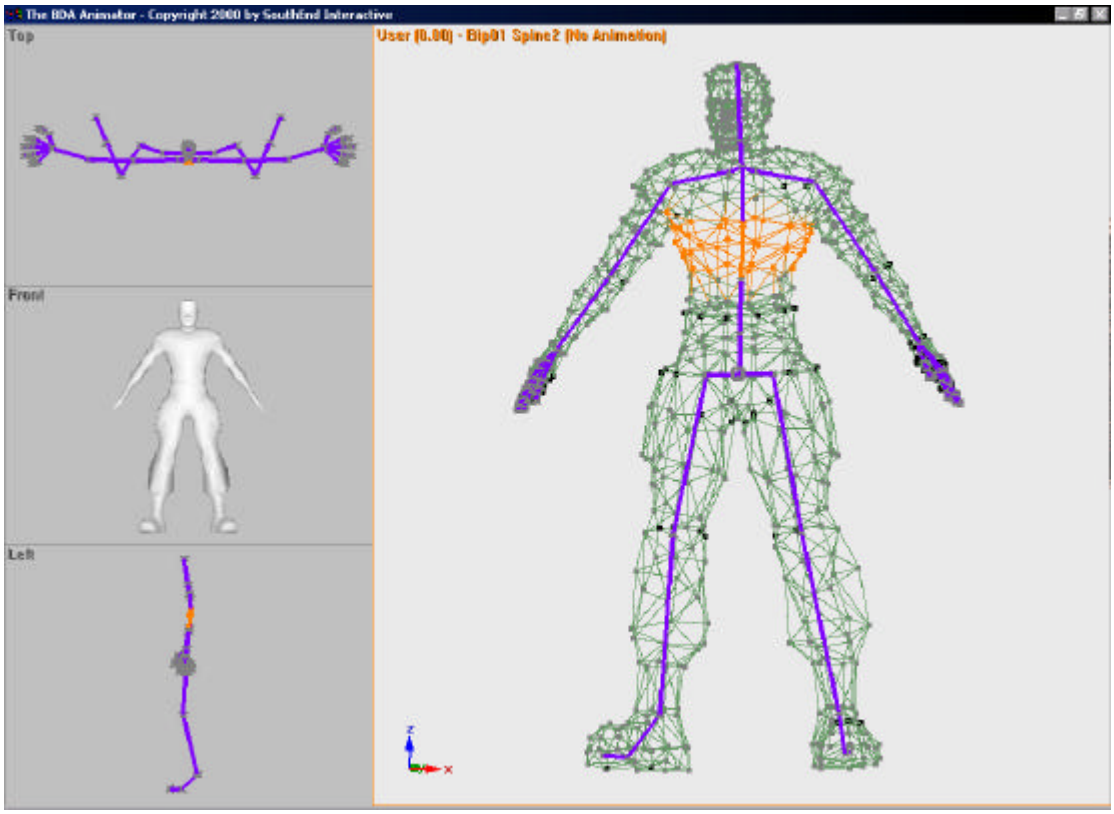



**Fig 3.1** The BDA Animator program's main output window.

Figure 3.2 shows the main control dialog where the user selects viewport settings, bind the mesh to the skeleton and toggles from mesh binding mode to animation mode. The user can set the desired framerate for the resulting animation. It is in this dialog that the user sets the WSA weights for the bones used in different animations.

[1] BDA is the engine developed by SouthEnd Interactive AB (www.southend-interactive.com)

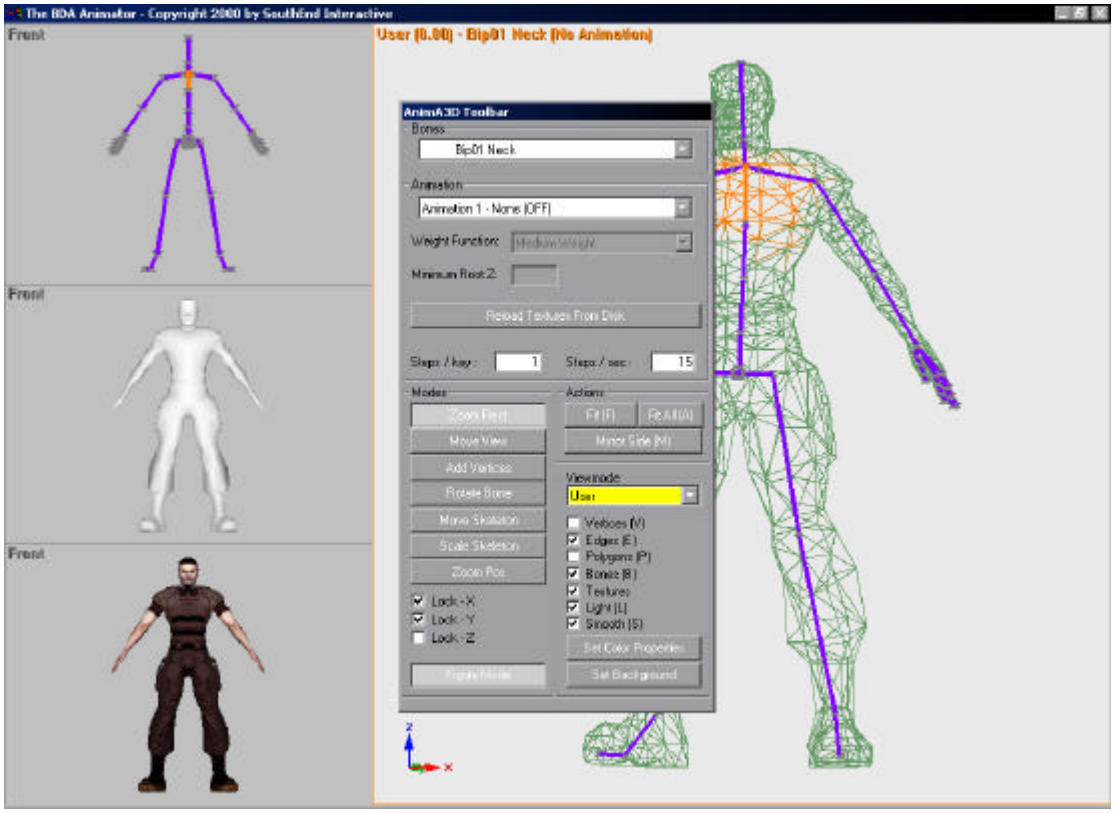



**Fig 3.2** The main control dialog for the BDA Animator.

Figure 3.3 shows the animation blending control dialog where the user can load animations into one of four slots. He can set starting times for the animations and lead-in / lead-out times for them as well as number of loops. At the bottom of the dialog are the animation controls which can start, stop and rewind an animation. There are buttons to save and load an animation setup for later.

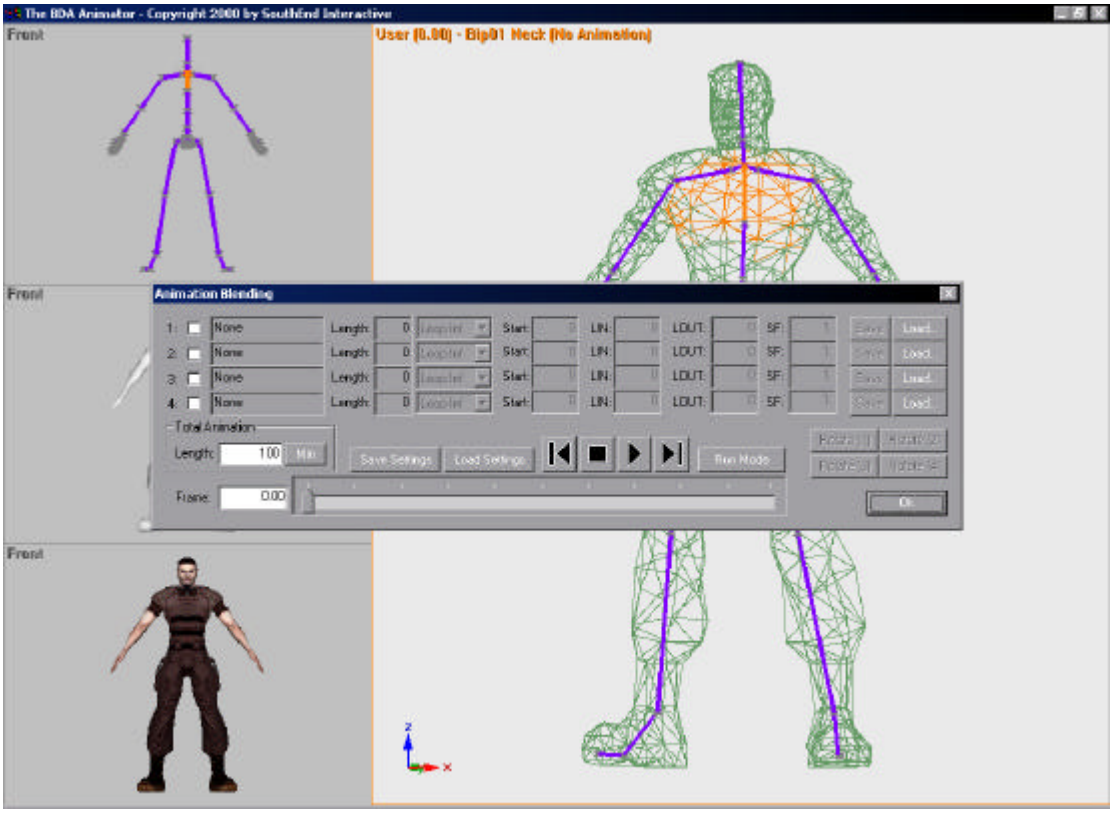



**Fig 3.3** The animation control dialog for the BDA Animator.

The last figure 3.4 shows the animation blending control dialog loaded with the blended walking and waving animation. It shows the character fully textured in the large user viewport.

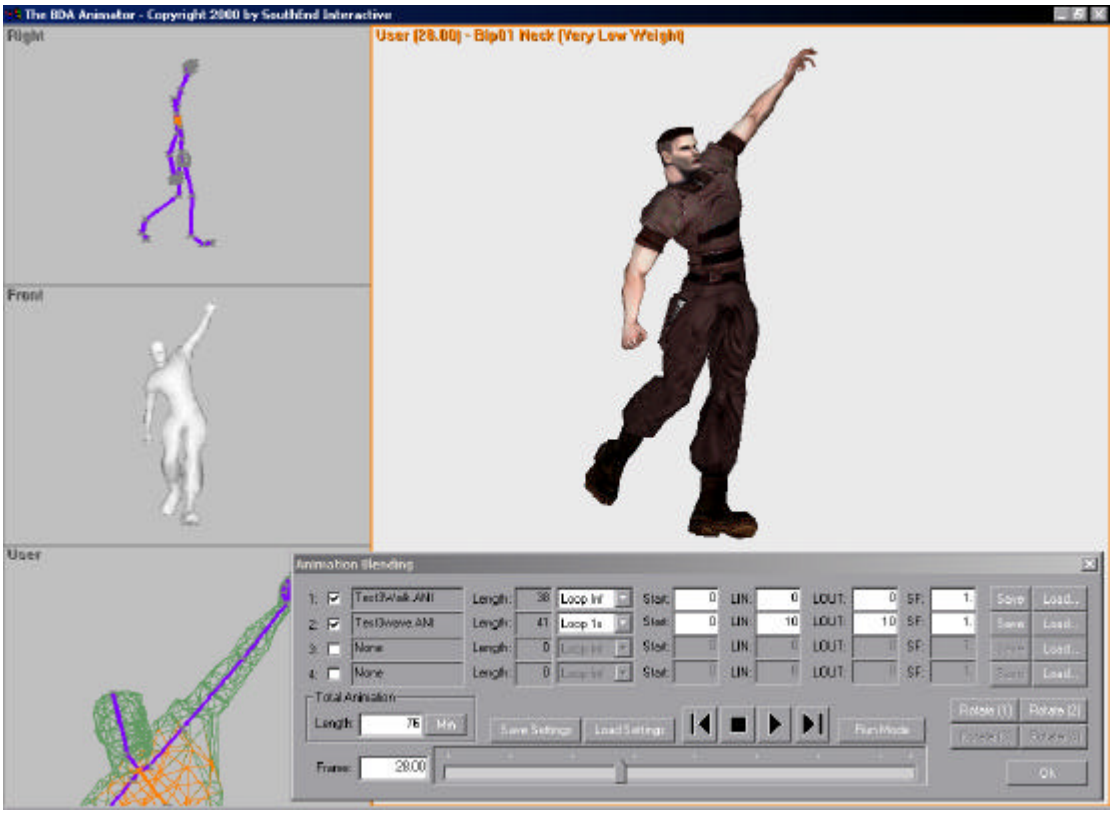



**Fig 3.4** A walking and waving blended animation in the BDA Animator program.

## 3.2 The blending results

This section analyzes WSA by conducting a series of tests that are empirically evaluated using subjective criteria.

### 3.2.1 The basic animation set

A number of motion-captured animations were chosen to test the animation system. Below is the complete list of the test animations.

| Animation | # frames | Description |
|---|---|---|
| (A1) Standing Idle | 60 | The character stands idle, breathing and moving his fingers and head a little. |
| (A2) Walking | 38 | The character walks forward with a little attitude. |
| (A3) Running | 17 | The character sprints really fast with large arm and leg movement. |
| (A4) Jumping | 101 | The character makes a jump from a standing position. |
| (A5) Rolling | 67 | The character throws himself forward from a standing position and rolls on his shoulder and back to a standing position. |
| (A6) Waving | 41 | The character waves his left arm high above his head. |
| (A7) Guards Up | 30 | The character moves his arms and fists in front of his chest and face for protection (like a boxer). |
| (A8) Punch Right | 14 | The character punches a straight punch using his right arm. |
| (A9) Punch Left | 14 | The character punches a straight punch using his left arm. |

**Fig 3.5** A list of basic animations used for conducting WSA evaluation.

Below is a detailed description of the nine basic animations covering how they are weighted to different bodyparts and why.

**A1 Standing Idle**
This animation is a simple animation of the character standing still, his chest heaving when he breathes, and his fingers flexing from time to time. It is looped. The entire skeleton is only weighted with a low weight since almost any animation supercedes this one.

**A2 Walking**
This animation is a walking loop with a little attitude in the way that the character moves. The legs of the skeleton are weighted with heavy weight since they are important for this animation, and the arms are weighted with medium weight since they are of little importance for the balance while walking. The rest of the skeleton is low weight.

**A3 Running**
This animation is a running loop with the character sprinting at full speed using almost an exaggerated motion. As in the walking animation, the legs are very important and get a heavy weight while the arms get a medium weight and the rest is low weight.

**A4 Jumping**
In this animation the character goes from a standing still position to a jump and back to standing still again. He uses his arms and legs to get momentum. It is not looped. In this animation, the spine and legs are heavy weight with arms of medium weight. The rest is low weight.

**A5 Rolling**
This animation is a forward roll. The character throws himself forward and rolls over his shoulder and back to a sitting and then standing position. At the end of the animation he uses his arms to get the momentum he needs to rise quickly and with balance. It is not looped. The entire skeleton is important in this animation and therefore everything is heavy weight.

**A6 Waving**
In this animation the character waves his left arm from a standing position. It is not looped. The left arm is important and is weighted heavy, part of the spine gets medium weight, the rest is low weight.

### A7 Guards Up

In this animation the character protects himself with his arms and hands like a boxer. It is looped. Only the arms and shoulders are important in this animation and get heavy weight. The spine gets a medium weight and the rest low weight.

### A8 Punch Right

This animation is a straight, forward punch using the characters right arm and shoulder. The upper body is also used somewhat for positioning. The right arm is important and gets a heavy weight, the spine gets medium weight and the rest low weight.

### A9 Punch Left

This animation is a straight, forward punch using the characters left arm and shoulder. The upper body is also used somewhat for positioning. The left arm is important and gets a very heavy weight, the spine gets medium weight and the rest low weight.

### 3.2.2 Evaluation of blended animations

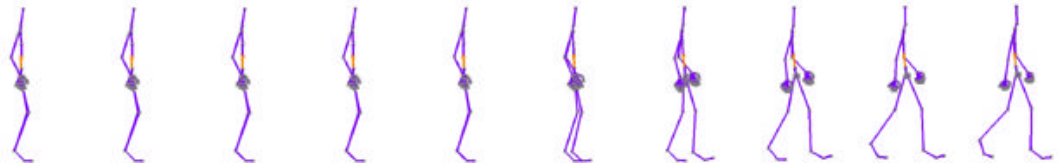
| Animations to test | Description |
|---|---|
| A. Idle to walking | Transition from standing idle to walking. |
| B. Walking to idle | Transition from walking to standing idle. |
| C. Walking to running | Transition from walking to running. |
| D. Running to walking | Transition from running to walking. |
| E. Walking to rolling to walking | Transition from walking to a roll and back to walking again. |
| F. Walking to jumping | Transition from walking to jumping. |
| G. Running to jumping | Transition from running to jumping. |
| H. Walking and waving | Blend of a walking and arm waving animation. |
| I. Running and waving | Blend of a running and arm waving animation. |
| J. Walking and running | Blend of walking and running at the same time. |
| K. Idle and guards up | Blend between an idle animation and a guard up animation (ready to punch). |
| L. Walking and guards up | Blend between a walking animation and a guards up animation (ready to punch). |
| M. Running and guards up | Blend between a running animation and a guards up animation (ready to punch). |
| N. Walking, waving and guards up | Blend between a running animation and a guards up animation (ready to punch) and a waving animation. |
| O. Running and jumping | Blend of running and walking at the same time. |
| P. Idle and walking | Blend of standing idle and walking at the same time. |
| Q. Jumping and punching | Blend of a punching and jumping animation at the same time. |
| R. Jumping and rolling | Blend of a rolling and jumping animation at the same time. |

**Fig 3.6** The set of test cases used for evaluation of the WSA method.

Since it is hard to show the quality of animations in still pictures, I will discuss the different test cases. What are the problems and how should it be evaluated? Some of the animations can be found on the web as mpeg movies at my homepage: www.efd.lth.se/~d94dj. The final result is shown in figure 3.7.

### A. Idle to walking (transition)

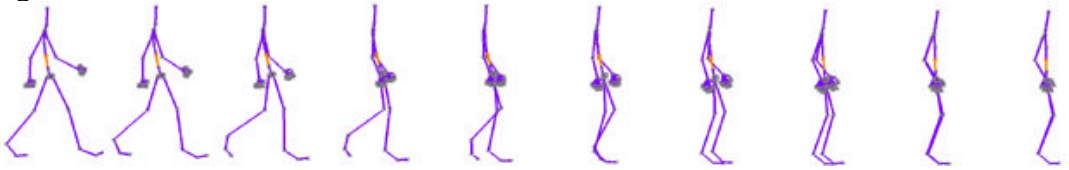
The first test case is a simple transition from standing idle to walking in a looped walk using only two small animations (A1) and (A2) from figure 3.5.



This transition worked very well with both small and large lead-in times, but about ten frames seemed to be best since the start frame of the walking animation was so different from the idle animation.

### B. Walking to idle (transition)

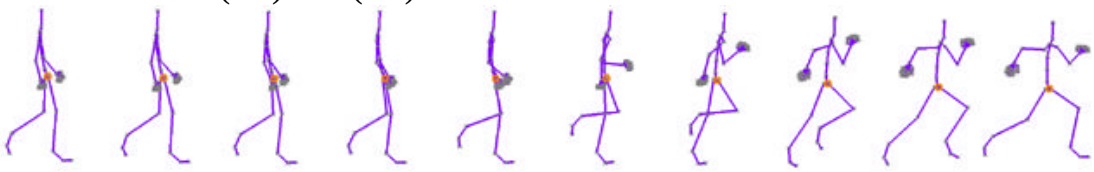
This test case is the reverse of test A above, a transition from walking to standing still using only the two same small animations (A1) and (A2) from figure 3.5.



This transition also worked very well with both small and large lead-in times but about ten frames seemed to be best for the reason as in test case A.

### C. Walking to running (transition)

This test makes a transition from walking to running. Both animations start with the same foot and both animations complete one cycle of animation but with different number of frames since the walk is slower than the run. The test used animations (A2) and (A3).



The two animations used for this transition had to be synchronized together as discussed in section 2.6 to achieve good results in all startup cases. A lead-in / lead-out time of six to ten frames looked best.

### D. Running to walking (transition)

This test makes a transition from running to walking. The test used animations (A2) and (A3).

The two animations used for this transition also had to be synchronized together to achieve good results in all startup cases. A lead-in / lead-out time of about ten frames looked best.

**E. Walking to rolling to walking (transition)**
This test makes a transition from walking to rolling and back to walking again. The test used animations (A2) and (A5).

The resulting animation looks very good even though the rolling animation begins in a standing position. A lead-in and lead-out time of about ten looks best.

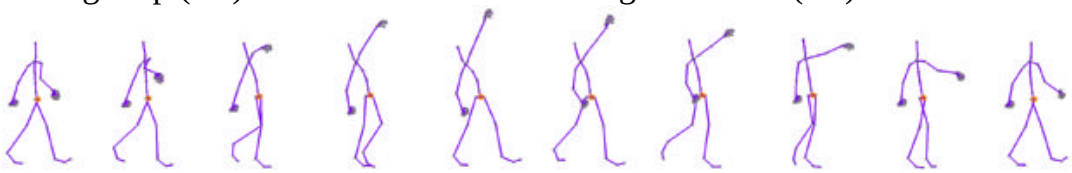
**F. Walking to jumping (transition)**
This test makes a transition from walking to jumping. The test used animations (A2) and (A4).

The resulting animation looks very good even though the jumping animation begins in a standing position. A lead-in and lead-out time of about ten looks best.

**G. Running to jumping (transition)**
This test makes a transition from running to jumping. The test used animations (A3) and (A4).

The resulting animation looks very good even though the jumping animation begins in a running position. A lead-in and lead-out time of about ten looks best.

**H. Walking and waving (blending)**
This is the first blending test case. The character walks using the cycled walking loop (A2) and wave his left arm using animation (A6).



The resulting animation looks exactly as predicted. The character walks and waves his arm in a natural way. A lead-in and lead-out time of about five to ten for the waving animation looks best.

**I. Running and waving (blending)**

This is also a non-transitioning test case. The character runs using the cycled running loop (A3) and wave his left arm using animation (A6).

Yet again the resulting animation looks exactly as predicted. The character runs and waves his arm independent of each other in a natural way. A lead-in and lead-out time of about five to ten for the waving animation looks best.

**J. Walking and running (blending)**

This is an interesting test case since the animations to blend are very interdependent. The animations used are the walking animation (A2) and the running animation (A3).

A variation of the synchronization discussed in section 2.6 was used to get the same number of frames in the two looping cycles. If no synchronization is used the resulting animation still works but the legs move in an unnatural way since the walking animation might retract a leg at the same time as the running animation extends it which results in unnatural movement.

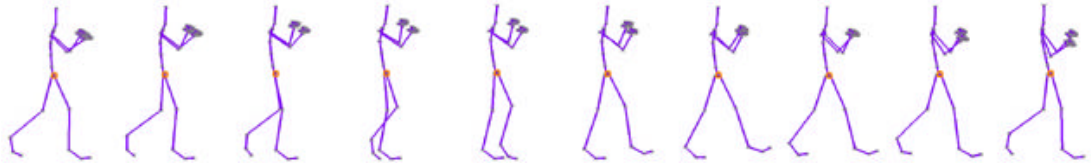
**K. Idle and guards up (blending)**

This is another example of animation blending with an animation of the character standing about idle (A1) and another animation where the character protects himself using arms and hands like a boxer (A7).

The resulting animation looked very good and natural. A lead-in time and lead-out time of ten to fifteen frames looked best for the guard animation.

**L. Walking and guards up (blending)**

This is another example of animation blending with an animation of the character walking (A2) and another animation where the character protects himself using arms and hands like a boxer (A7).



This resulting animation also looked very good and natural. A lead-in time and lead-out time of ten to fifteen frames looked best for the guard animation.

**M. Running and guards up (blending)**
This is yet another example of animation blending with an animation of the character running (A3) and another animation where the character protects himself using arms and hands like a boxer (A7).

The resulting animation looked very good although it was a bit stiff in the upper body. This could be corrected by changing the weights of the upper body in any one or both of the source animations.

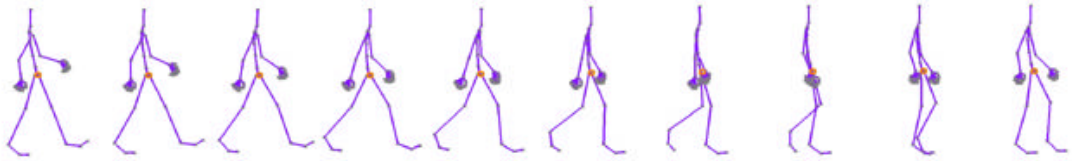
**N. Walking, waving and guards up (blending)**
This animation will blend three different animations together. It uses the running animation (A3), the waving animation (A6) and the guards up animation (A7).

The resulting animation looked very good and natural. Lead-in times of about ten to fifteen were best for the guards up animation and about ten for the waving animation.

**O. Running and jumping (blending)**
This animation will blend the jumping animation (A4) and running animation (A3) at the same time. The predicted result should be a character continuing to run in the air while jumping.

The resulting animation looked exactly as predicted and the character made a running jump instead of one standing still. A lead-in time of ten for the jumping animation and a lead-out of about five to ten looked the best.

**P. Idle and walking (blending)**
This animation will blend the animation of the character standing idle (A1) and the walking animation (A2).

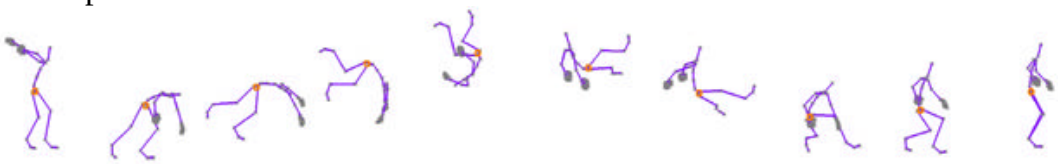


Since no part of the idle animation is truly important in the animation itself we would suspect the result to be the walking animation itself, and this was exactly what happened.

**Q. Jumping and punching (blending)**
This animation will blend the animation of the character jumping (A4) and the two punching animations (A8) and (A9).

The result looked natural although it was a bit tricky to set appropriate lead-in and lead-out times for the punching animations. A lead-in of about 2 frames and a lead-out of five frames seemed to work best. The punches looked a bit awkward since the character uses his arms so much in the jumping animation too.

**R. Jumping and rolling (blending)**
This animation will blend the animation of the character jumping (A4) and the rolling animation (A5). Since both animations uses the entire skeleton it could lead to problems.



The result as seen above was way above expectations and could certainly be tweaked by setting rules in the interactive system as to when the animations could be triggered. This testcase shows a weakness of the system. The blending unit doesn't use any constraints or laws of physics to determine how a blended move should result! Future additions in this area could possibly solve this problem. Most users of this animation system should be able to work around this weakness and use WSA without these kind of blended animations.

### 3.2.3 The results

| Animation | Type | Result |
|---|---|---|
| A. Idle to walking | Trans. | Very good results. |
| B. Walking to idle | Trans. | Very good results. |
| C. Walking to running | Trans. | Very good results if synchronized. Otherwise only acceptable. |
| D. Running to walking | Trans. | Very good results if synchronized. Otherwise only acceptable. |
| E. Walking to rolling to walking | Trans. | Very good results. |
| F. Walking to jumping | Trans. | Very good results. |
| G. Running to jumping | Trans. | Very good results. |
| H. Walking and waving | Blend. | Very good results. |
| I. Running and waving | Blend. | Very good results. |
| J. Walking and running | Blend. | If synchronized together the result is very impressive, otherwise very unnatural. |
| K. Idle and guards up | Blend. | Very good results. |
| L. Walking and guards up | Blend. | Very good results. |
| M. Running and guards up | Blend. | A bit stiff in the upper body, otherwise good results. |
| N. Walking, waving and guards up | Blend. | Very good results. |
| O. Running and jumping | Blend. | Very good results. |
| P. Idle and walking | Blend. | Very good results. |
| Q. Jumping and punching | Blend. | Good results although a bit awkward. |
| R. Jumping and rolling | Blend. | Way above expectations. Especially if constraints are set as to when the animations can be triggered. |

**Fig 3.7** The results of the blending animation system.

As can be seen in the table above (figure 3.7) the results are quite impressive, for the chosen test cases. To make the animation system fail, you would have to blend totally impossible animations together such as walking in two directions at the same time.

## 4. Conclusions and future work

The previous chapter showed great results by using the WSA method and still having moderate computational cost and memory footprint. We solved the difficulties with Lead-In / Lead-Out functions, added Inverse Kinematics joints and weighted an arbitrary number of animations together.

The system is currently implemented into the BDA Engine by SouthEnd Interactive (www.southend-interactive.com), and is running perfectly without seams or jerkiness.

**Fig 4.1** A successful blend between a walking and a waving animation.

Future additions to the animation engine will include IK restraints and advanced collision detection subsystems. By adding IK restraints, it will be impossible to blend into impossible poses, for example bending the arm so that it would break. Advanced collision detection will make it possible for the character to for instance walk up a staircase and interact in a natural way with his environment. Building in the laws of physics and dynamics should make it possible to blend even the most complex animations together and will also be added in the future.

The method also lends itself to other forms of animation, like facial animation, which will also be incorporated into the system in the near future. Facial animation is a set of bones inside the character head, which can be be animated using WSA. Therefore we can simply apply the same blending rules and SLERP functions to interpolate different facial expressions.

To optimize animation, reduced detail in the skeleton and mesh will be added so that characters far from the viewpoint or not in focus will use less CPU time.

This can be achieved by having several levels of detail (LOD) of both the skeleton and the mesh itself.
Other future development include ways to flex bones and create effects like bulging muscles, etc. But it will of course cost some additional CPU time. With faster and faster computers and 3D hardware, we will soon see detailed characters easily rivaling the ones seen in movies and cut-scenes.

## 5. References

[1] Foley, J.D., van Dam, A, Feiner, S.K. and Hughes, J.F.
Computer Graphics – Principles and Practice – Second Edition in C
Addison-Wesley, Reading MA
1996

[2] Watt, A. and, Watt, M.
Advanced Animation and Rendering Techniques – Theory and practice
Addison-Wesley, Reading MA
1995

[3] Bobick, N.
Rotating Objects Using Quaternions
Game Developer Magazine article, Miller Freeman Inc.
February 1998

[4] Lander, J.
Slashing Through Real-Time Character Animation
Game Developer Magazine article, Miller Freeman Inc.
April 1998

[5] Corley, S.
Architecting A 3D Animation Engine
Game Developer Magazine article, Miller Freeman Inc.
April 1998

[6] Steed, P.
The Art of Quake 2
Game Developer Magazine article, Miller Freeman Inc.
April 1998

[7] Knight, D. and Tolles, T.
When Motion Capture Beats Keyframing
Game Developer Magazine article, Miller Freeman Inc.
September 1997

[8] Ridgway, W.
Incorporating Motion Capture Animation into an AI Engine
Game Developer Magazine article, Miller Freeman Inc.
June 1998

[9] Rodgers, J.
Animating Facial Expressions
Game Developer Magazine article, Miller Freeman Inc.
November 1998

[10] Lander, J.
Oh My God, I Inverted Kine!
Game Developer Magazine article, Miller Freeman Inc.
September 1998

[11] Lander, J.
Making Kine More Flexible
Game Developer Magazine article, Miller Freeman Inc.
November 1998

[12] Steed, P.
No Pain, No Gain: Implementing New Art Technology in Quake 3: Arena – G.D. Magazine article
Game Developer Magazine article, Miller Freeman Inc.
December 1999

[13] Lee, A.W.F., Dobkin, D., Sweldens, W. and Schröder, P.
Multiresolution Mesh Morphing
Princeton University

[14] Lander, J.
Read My Lips: Facial Animation Techniques
Game Developer Magazine article, Miller Freeman Inc.
June 1999

[15] Steed, P.
Mo-Cap and Keyframing, Sittin' in a Tree
Game Developer Magazine article, Miller Freeman Inc.
November 1999

[16] Kalra, P., Magnenat-Thalmann, N., Moccozet, L., Sannier, G., Aubel, A. and Thalmann, D.
Real-Time Animation of Realistic Virtual Humans
IEEE Computer Graphics and Applications
September / October 1998

[17] Earnshaw, R., Magnenat-Thalmann, N., Terzopoulos, D. and Thalmann, D.
Computer Animation for Virtual Humans
IEEE Computer Graphics and Applications
September / October 1998

[18] Rose, C., Cohen, M.F. and Bodenheimer, B.
Verbs and Adverbs: Multidimensional Motion Interpolation
IEEE Computer Graphics and Applications
September / October 1998

[19] Willis, P.J.
Computer Animation and Human Animators
Proceedings of the Winter School of Comp. Graph. and Visualization
1995

[20] Sowsy, D.
A Survey of Animation Techniques (Computer Graphics 2: 91.547)
Published on the web.
(http://www.cs.uml.edu/~dsowsy/coursework/AnimPaper.html)

[21] The Character Animation FAQ
Maintained by Hexapod (hexapod@netcom.com)
Published on the web.
(http://www.flipcode.com/documents/charfaq.html)

# Appendix

## A. Dictionary

| | |
|---|---|
| Animation Blending | Combining several animations to form a new one. Example, using a walking and a waving animation to create a walking and waving animation at the same time. |
| Animation Transition | The transition from one animation to another. Example, going from a walking to a running animation. |
| Bone | A bone is a part of a skeleton hierarchy. |
| D-SKA | Deformed Skeleton Animation – Skeleton Animation with a single mesh skinning the skeleton. |
| Forward Kinematics (FK) | The reverse of IK (Inverse Kinematics). Using FK you have to start in the root node of a skeleton and animate each bone in turn before traversing deeper down into the skeleton hierarchy. See Inverse Kinematics. |
| FPS | Frames per second. |
| Framerate | Frames per second. |
| In-between | An in-between frame is an interpolated frame between two keyframes. See Interpolation. |
| Interpolation | To calculate positions or rotations between keyframes. |
| Inverse Kinematics (IK) | A way to set a position for a bone and having the skeleton hierarchy follow along. For example, using IK you can just set the hand of a character in a certain way and the arm bones will automatically be positioned. The reverse of IK is FK (Forward Kinematics). IK is expensive for the CPU because of the math involved. See Restraints and Forward Kinematics. |

| | |
|---|---|
| Keyframe | A keyframe (or key) is set for each change in an animation on discreet times depending on the animation framerate. If a higher output framerate is desired the program will have to interpolate between keyframes. |
| L-SKA | Linked Skeleton Animation – Skeleton Animation with separate bodyparts / objects for each bone. |
| Mesh | The polygons and points describing a character or object. |
| Morphing | Interpolation of vertices in a mesh to animate the change from one mesh to another. |
| Node | A node is the place where two or more bones meet in a skeleton hierarchy. |
| Quaternion | Quaternions are a mathematical tool and are really unit vectors on the 4D sphere. They can describe 3D rotations in a better way than ordinary rotation matrices. |
| Restraints | Restraints are used to set rules for rotations in a skeleton so that unnatural movements can't be done. See Inverse Kinematics. |
| SKA | Skeleton Animation (see Skeleton). A way to animate a character using a skeleton put inside the mesh. The skeleton controls the mesh and if a bone is moved, part of the mesh is moved as well. |
| Skeleton | A skeleton is a "tree" of bones building the necessary joints inside the character. Instead of animating the mesh, a program can animate the skeleton. |
| Skinning | Another word for having a skeleton inside a mesh and then animating the skeleton to make the skin (mesh) move naturally. |
| VEA | Vertex Animation. A way to animate a character using stored keyframe positions for each vertex in the mesh. |

## B. Quaternion Example

This appendix will show an example quaternion class written in C++ and compiled with Microsoft Visual Studio 6.0. Any ANSI-compliant compiler should be able to compile the code.

### Defining the quaternion class

Below is the header code for the class itself.

```
//////////////////////////////////////////////////////////////////////////
// Quaternion
//////////////////////////////////////////////////////////////////////////
// This class is a quaternion rotation. It can be described as an axis
// (vector) and an angle around that axis (scalar) to rotate.
//
// Constructors with parameters to create quaternions from include:
// * Vector
// * Vector and an angle
// * Yaw-Pitch-Roll angles
// * A rotation matrix
// * Two different Quaternions and an interpolation value
//
// The last one is very interesting since it uses SLERP, interpolation
// along the unit 4D sphere to create the new quaternion.
//////////////////////////////////////////////////////////////////////////
class Quaternion
{
public:
  float w;      // Scalar
  float x,y,z;  // Axis vector

  // Create an uninitialized quaternion
  Quaternion();

  // Create the quaternion with a simple vector (w=0.0f)
  Quaternion(Vector &v);

  // Create the quaternion with an axis and an angle around that axis
  Quaternion(Vector &v, float angle);

  // Create a quaternion from yaw-pitch-roll values
  Quaternion(float yaw, float pitch, float roll, BOOL anti=FALSE);

  // Create the quaternion from a matrix
  Quaternion(Matrix &m);

  // Create a quaternion using SLERP interpolation
  Quaternion(Quaternion &q1, Quaternion &q2, float step);

   // Normalize the quaternion
   void normalize();

   // Invert the quaternion
   void inverse();

   // Multiply quaternion
   Quaternion multiply(Quaternion &quat);
};
// Quaternion
//////////////////////////////////////////////////////////////////////////
```

(**x**,**y**,**z**) is the vector part and **w** is the scalar part of the quaternion.

Note the constructor to the left uses **SLERP interpolation** of two other quaternions to create a new one.

The class itself uses a few other classes such as Matrix and Vector that are defined later in this chapter.

Below follows the implementation of the various member functions and constructors declared above.

```
////////////////////////////////////////////////////////////////////////
// Quaternion
Quaternion::Quaternion(Vector &v)
{
  x = v.x; y = v.y; z = v.z;
  w = 0.0f;
}

///

Quaternion::Quaternion(Vector &v, float angle)
{
  float halfAngle = angle*0.5f;
  float t = SinHP(halfAngle);
  x = t*v.x;
  y = t*v.y;
  z = t*v.z;
  w = CosHP(halfAngle);
}

///

Quaternion::Quaternion()
{
  x = y = z = w = 0.0f;
}

///


Quaternion::Quaternion(float yaw, float pitch, float roll, BOOL anti)
{
  if (anti)
  {
    float cr, cp, cy, sr, sp, sy, cpcy, spsy;
    float rxHalf = yaw*0.5f;
    float ryHalf = pitch*0.5f;
    float rzHalf = roll*0.5f;
    cr = CosHP(rzHalf);
    cp = CosHP(ryHalf);
    cy = CosHP(rxHalf);
    sr = SinHP(rzHalf);
    sp = SinHP(ryHalf);
    sy = SinHP(rxHalf);
    cpcy = cp * cy;
    spsy = sp * sy;
    x = sr * cpcy - cr * spsy;
    y = cr * sp * cy + sr * cp * sy;
    z = cr * cp * sy - sr * sp * cy;
    w = cr * cpcy + sr * spsy;
  }
  else
  {
    float rx,ry,rz,cx,cy,cz,sx,sy,sz,cc,cs,sc,ss;
    rx = yaw*0.5f;
    ry = pitch*0.5f;
    rz = roll*0.5f;
    cx = CosHP(rx);
    cy = CosHP(ry);
    cz = CosHP(rz);
    sx = SinHP(rx);
    sy = SinHP(ry);
    sz = SinHP(rz);
    cc = cx * cz;
    cs = cx * sz;
    sc = sx * cz;
    ss = sx * sz;
    x  = (cy * sc) - (sy * cs);
    y  = (cy * ss) + (sy * cc);
    z  = (cy * cs) - (sy * sc);
    w  = (cy * cc) + (sy * ss);
  }
}

///

Quaternion::Quaternion(Matrix &morg)
{
  Matrix m = morg;
  m.transpose();
  float tr, s, q[4];
  int   i, j, k;
  int nxt[3] = {1, 2, 0};
  tr = m.a0 + m.b1 + m.c2;

  // Check the diagonal
  if (tr > 0.0)
  {
```

This is a constructor which creates a quaternion from a vector **v** with no rotation around it.

This constructor creates a rotation quaternion using a vector **v** and a rotation **angle** (using radians) around that vector.

An empty constructor setting the entire quaternion to zero.

This constructor converts Euler angles (**yaw**, **pitch**, **roll**) to a quaternion rotation. If **anti** is true the rotation is made as ZYX not XYZ order..

Here is another interesting constructor creating a quaternion from a rotation matrix **morg**.

```
    // Diagonal is positive
    s = (float)SqrtHP(tr + 1.0f);
    w = s*0.5f;
    s = 0.5f/s;
    x = (m.c1 - m.b2) * s;
    y = (m.a2 - m.c0) * s;
    z = (m.b0 - m.a1) * s;
  }
  else
  {
    // Diagonal is negative
    i = 0;
    if (m.elementAt(1,1) > m.elementAt(0,0))
      i=1;
    if (m.elementAt(2,2) > m.elementAt(i,i))
      i=2;
    j = nxt[i];
    k = nxt[j];
    s = (float)SqrtHP((m.elementAt(i,i) - (m.elementAt(j,j) + m.elementAt(k,k))) + 1.0f);
    q[i] = s * 0.5f;
    if (s != 0.0f)
      s = 0.5f / s;
    q[3] = (m.elementAt(j,k) - m.elementAt(k,j)) * s;
    q[j] = (m.elementAt(i,j) + m.elementAt(j,i)) * s;
    q[k] = (m.elementAt(i,k) + m.elementAt(k,i)) * s;
    x = q[0];
    y = q[1];
    z = q[2];
    w = q[3];
  }
}

///

Quaternion::Quaternion(Quaternion &q1, Quaternion &q2, float step)
{
  float to1[4];
  float omega, cosom, sinom;
  float scale0, scale1;

  // Calc cosine
  cosom = q1.x * q2.x + q1.y * q2.y + q1.z * q2.z + q1.w * q2.w;

  // Adjust signs (if necessary)
  if ( cosom < 0.0f )
  {
    cosom = -cosom;
    to1[0] = -q2.x;
    to1[1] = -q2.y;
    to1[2] = -q2.z;
    to1[3] = -q2.w;
  }
  else
  {
    to1[0] = q2.x;
    to1[1] = q2.y;
    to1[2] = q2.z;
    to1[3] = q2.w;
  }

  // Calculate coefficients
  if ( (1.0f - cosom) > DELTA)
  {
    // Standard case (SLERP)
    omega = Acos(cosom);
    sinom = SinHP(omega);
    scale0 = SinHP((1.0f - step) * omega) / sinom;
    scale1 = SinHP(step * omega) / sinom;
  }
  else
  {
    // q1 and q2 are very close so we do a linear interpolation
    scale0 = 1.0f - step;
    scale1 = step;
  }

  // Calculate final values
  x = scale0 * q1.x + scale1 * to1[0];
  y = scale0 * q1.y + scale1 * to1[1];
  z = scale0 * q1.z + scale1 * to1[2];
  w = scale0 * q1.w + scale1 * to1[3];
}

///

void Quaternion::normalize()
{
  float lenInv;
  lenInv = 1.0f / ((x*x) + (y*y) + (z*z) + (w*w));
  x = x * lenInv;
  y = y * lenInv;
  z = z * lenInv;
  w = w * lenInv;
}
```

This is the single most interesting constructor. It uses SLERP interpolation to create a rotation from two other unit quaternions **q1**, **q2** and a **step** value between them (0.0 - 1.0).

This function normalizes the quaternion (makes the length one). $\mathbf{x^2+y^2+z^2+w^2 = 1}$

```
///

void Quaternion::inverse()
{
  x = -x;
  y = -y;
  z = -z;
}

///



Quaternion Quaternion::multiply(Quaternion &quat)
{
  Quaternion ret;
  ret.x = w * quat.x + x * quat.w + y * quat.z - z * quat.y;
  ret.y = w * quat.y + y * quat.w + z * quat.x - x * quat.z;
  ret.z = w * quat.z + z * quat.w + x * quat.y - y * quat.x;
  ret.w = w * quat.w - x * quat.x - y * quat.y - z * quat.z;
  return ret;
}
// Quaternion
//////////////////////////////////////////////////////////////////////////
```

This function reverses the rotation. It really reverses the vector part of the quaternion which in effect means rotating in reverse.

This function multiplies the quaternion with another quaternion **quat** and returns the resulting quaternion.

As promised, below follows the (abbreviated) matrix and vector classes used in the code above.

```
////////////////////////////////////////////////////////////////////////////
// Matrix
////////////////////////////////////////////////////////////////////////////
// This class is a standard 4x4 matrix. Many different
// constructors and static creators exist to make (for example):
// * Rotation matrices
// * Translation matrices
// * Screenprojection matrices
// * Scaling matrices
////////////////////////////////////////////////////////////////////////////
class Matrix
{
public:
  float a0,b0,c0,d0;  // Matrix elements (Row-0)
  float a1,b1,c1,d1;  // Matrix elements (Row-1)
  float a2,b2,c2,d2;  // Matrix elements (Row-2)
  float a3,b3,c3,d3;  // Matrix elements (Row-3)

  // Create a rotation matrix from a quaternion
  Matrix(Quaternion const &quat);

  // Returns the elementa at position row,col
  float elementAt(int row, int col);

  // Transpose the matrix
  void transpose();

  // Make this matrix unit
  void makeUnit();

  // Invert the matrix
  void inverse();

  // ....
  // ....
  // ....
}
// Matrix
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Vector
////////////////////////////////////////////////////////////////////////////
// This class is a standard 3D vector.
// Functions exist to rotate it, make crossproducts, transform it, etc.
// Operator overloads exist for add, dot, subtract, etc.
////////////////////////////////////////////////////////////////////////////
class Vector
{
public:
  DWORD timestamp;  // A timestamp used elsewhere
  float x,y,z;      // Vector direction

  // Constructors
  Vector()                                          {x = y = z = 0.0f;}
  Vector(Vector &v)                                 {x = v.x; y = v.y; z = v.z;}
  Vector(float x, float y, float z)                 {this->x = x; this->y = y; this->z = z;}
  Vector(Quaternion &quat)                          {x = quat.x; y = quat.y; z = quat.z;}

  // Operator overloads
  Vector& operator=(Vector const &v)                {x = v.x; y = v.y; z = v.z; return *this;}
  float operator*(Vector const &v) const    {return x*v.x+y*v.y+z*v.z;}
  Vector operator-(Vector const &v)                 {return Vector(x-v.x,y-v.y,z-v.z);}
  Vector operator+(Vector const &v)                 {return Vector(x+v.x,y+v.y,z+v.z);}

  // Normalize the length of the vector == 1.0f
  Vector &normalize();

  // Create the unnormalized crossproduct from this and another vector (this X vec2)
  Vector crossProduct(Vector const &vec2) const;

  // Transform the vector with a matrix
  Vector transform(Matrix const &m);

  // Return the length of this vector
  float length();
};
// Vector
////////////////////////////////////////////////////////////////////////////
```

The 4x4 matrix components.

Of special interest in the Matrix class is the constructor creating a rotation matrix from a quaternion rotation:

```
////////////////////////////////////////////////////////////////////////
Matrix::Matrix(Quaternion const &quat)
{
  float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;
  x2 = quat.x + quat.x; y2 = quat.y + quat.y; z2 = quat.z + quat.z;
  xx = quat.x * x2;   xy = quat.x * y2;   xz = quat.x * z2;
  yy = quat.y * y2;   yz = quat.y * z2;   zz = quat.z * z2;
  wx = quat.w * x2;   wy = quat.w * y2;   wz = quat.w * z2;
  a0 = 1.0f - (yy + zz);
  b0 = xy - wz;
  c0 = xz + wy;
  d0 = 0.0f;
  a1 = xy + wz;
  b1 = 1.0f - (xx + zz);
  c1 = yz - wx;
  d1 = 0.0f;
  a2 = xz - wy;
  b2 = yz + wx;
  c2 = 1.0f - (xx + yy);
  d2 = 0.0f;
  a3 = 0.0f;
  b3 = 0.0f;
  c3 = 0.0f;
  d3 = 1.0f;
}
////////////////////////////////////////////////////////////////////////
```

## Using the quaternion class

After having defined these classes how does one use them? First of all we want to use the quaternion to simply rotate a vector:

```
////////////////////////////////////////////////////////////////////////
// Rotate a vector (called vec) 30 degrees around the z-axis using quaternion rotations.
////////////////////////////////////////////////////////////////////////
Quaternion rot30Z(Vector(0.0f,0.0f,1.0f), PI/6.0f);
Quaternion rot30Zinv = rot30Z;
rot30Zinv.inverse();
Vector result = rot30Z*Quaternion(vec)*rot30Zinv;
////////////////////////////////////////////////////////////////////////
```

The example shows how a vector can be converted to a quaternion, rotated by multiplying with the quaternion to the left and the inverse of the quaternion to the right. The resulting quaternion can then be converted back to a vector by simply removing the scalar (w) component.

Another example is shown below. It creates two different quaternions out of two rotation matrices and interpolates them using the SLERP constructor to the rotation halfway between them. Last of all it converts back to a matrix for future use in for example a 3D pipeline:

```
////////////////////////////////////////////////////////////////////////
// Create a 50-50 interpolation of two rotations defined as rotation matrices.
////////////////////////////////////////////////////////////////////////
Quaternion q1(rot1), q2(rot2);
Quaternion interpolatedQ(q1, q2, 0.5f);
Matrix result(interpolatedQ);
////////////////////////////////////////////////////////////////////////
```