Character Animation for Real-time Applications

Michael Putz, mike@bongfish.com
Klaus Hufnagl, klaus@bongfish.com

Institute of Computer Graphics
Graz University of Technology
Austria

Abstract

Many of the techniques which had recently only been used in off-line animation, like skeletal animation, real-time deformation and skinning of meshes, are now established methods for the implementation of real-time 3D character animation.
The importance of real-time character animation in computer games has increased considerably over the past decade. Due to advances in computer hardware and especially the introduction of Graphic Processing Units, the demand for more realism in computer games is continuously growing. Beside traditional pre-computed and then replayed animation data, there is use for event triggered real-time calculated animation.
Animating models by manipulating an attached skeleton is a common technique for producing lifelike animations in games. It has both a firm basis in biological reality, and a low memory requirement (compared to, for example, storing vertex and normal vectors for each frame). This work shows a way of combining both off-line animation and real-time generated procedural animation.

Keywords:

skeletal animation and mesh skinning, real-time Inverse Kinematics, bones, biped, real-time character animation, skinning, physic based animation, procedural animation, parametric animation

1. Introduction:

In the last years many attempts were taken to simulate lifelike character behaviour in real-time applications like computer-games.

In a hierarchic organized 3D character, different body parts of the articulated object are separate objects, stored in a hierarchy and joined to each other at pivot points, each referencing child objects (= attached objects of a lower hierarchic order). The flexibility and adaptability of this method are its main benefits; however this method also has a number of drawbacks: As the objects in the hierarchy are all separate, it is inevitable that gaps between these objects will appear when the character is animated *[Lander1997]*.

Blending between Character Meshes, also called vertex-key-framing, uses interpolation functions to generate the in-between positions of the vertices from different poses. Tweening is the fastest way of animating, but it requires the application to hold multiple copies of the mesh

in memory, one for each key frame, so its memory footprint is very high. A more flexible solution would allow the artist to set up the meshes for animation as they normally do, by attaching them to a skeletal system and then letting the game perform the skeletal deformation itself. This would enable the application to create unique animation sequences based on user interaction, instead of interpolating from pre-modelled and stored poses. Besides offering increased realism, this technique can be more easily integrated with various physics simulators.
After taking a look at the requirements of the mentioned animation techniques, this paper will present an implementation showing the combination of precomputed animation and real-time effects based on a skeletal system.

2. Related work

Before we start implementing a real-time solution for enhancing character movement, we take a look at related topics to fully understand the implementations coming next chapter.

2.1 Skeleton Animation Systems

Beside the mesh 3d-data representing the character, a skeletal animation system consists of a series of hierarchical transformations which represent bones. Like a real bone in a human body they influence the shape of the skin. Only the bone data needs to be stored for every frame of the animation. Usually, the bone data is represented by quaternions or a transformation matrix. This eliminates the need to store vertex positions for all the vertices for every frame of the animation, as in the vertex based animation. We can derive intermediate frames by using spherical interpolation.

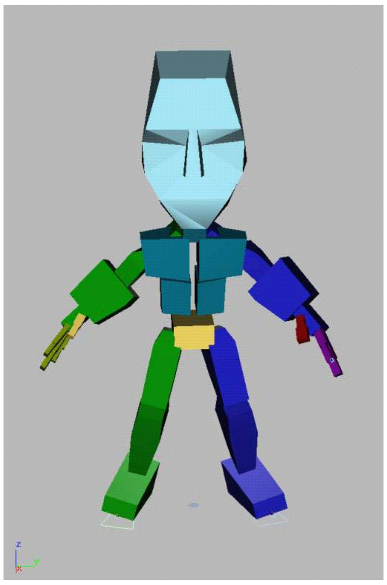


Fig. 1: Schematic skeleton view

Advantages of skeletal animation are the smooth transitions while changing from one animation to the other. Additionally any number of animations can be added whereas the mesh remains constant and the same animation-cycles could be applied to different meshes while conserving a relative small memory footprint.

2.2 The need for skinning to generate a vertex-hull

Mesh (vertices) or skin is attached to the bones. Now when the bones move or rotate, the vertices attached to the bone also move or rotate according to their representative bones.One way of attaching vertices to bones is by using a single offset per vertex. So every bone tightly influences a group of faces from the character-mesh.

2.2.1 Single weight vertices & their disadvantages

The following images show the relationship of the skeleton / bone with its mesh. The semi-transparent object is the mesh. As you see, they are tightly coupled. When a bone moves, the vertices attached to it also move *[Anderson2001]*.

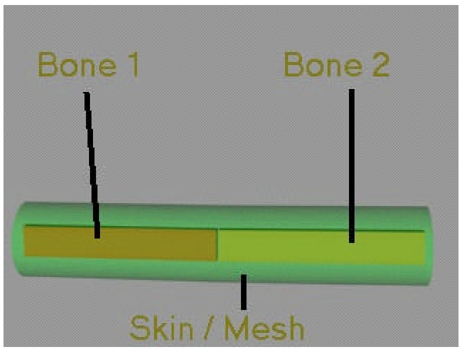


Fig. 2: Schematic bone / skin view

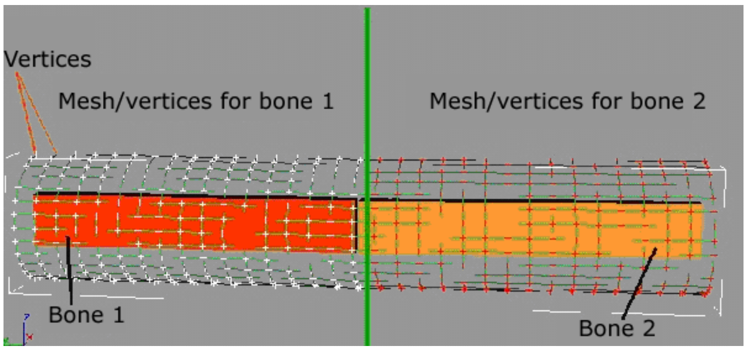


Fig. 3: Vertex / bone relation

Fig. 3 shows the same relationship as explained above. The only difference is that the mesh is shown in wire frame. The green line divides the vertices between the first and second bone. Vertices are represented by a small "+" sign. Vertices on the right are marked red indicating that they belong to the second bone, where as the vertices on the left are marked white indicating that they belong to the first bone. The following figure (4) highlights the main drawback of the skeletal animation system using a single weight per vertex. When rigid bodies move, the vertices attached to them also move rigidly.

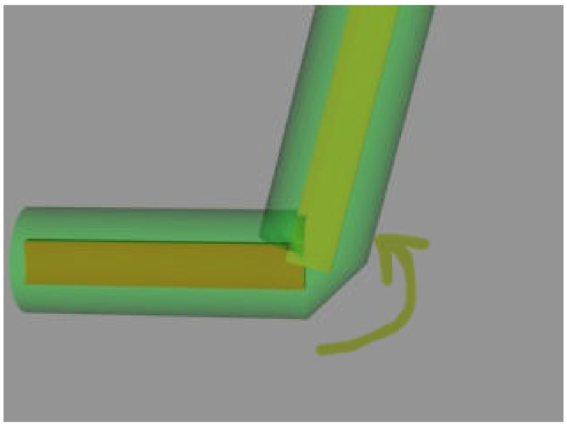


Fig. 4: Single weighted vertex drawback

2.2.2 Multiple weighted vertices for smooth skinning

Introducing multiple weights per vertex improves the smoothness of the mesh-hull. This process is called vertex blending. This is achieved by allowing more than just one bone to influence each vertex, effectively mimicking the way that a bone in the real world would affect the skin of a living being. Each vertex is given information about which of the bones in the skeleton influence it and how great the influence of those bones is (skin weight). What we need is a skin that will blend smoothly between the rigid joints.

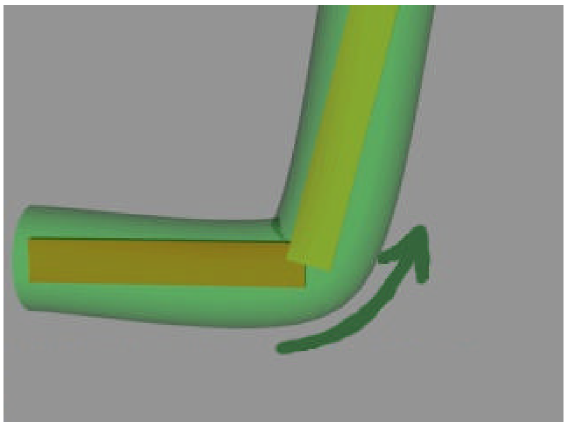


Fig. 5: multiple weights per vertex

Now every vertex is attached to multiple bones with different weights. This results in a smooth moving skin adapting itself to muscle-bulges and corners.

The generic blending formula:

$$vBlend = V_1W_1 + \ldots + V_{n-1}W_{n-1} + \sum_{i=1}^{n} V_n(1.0-W_i)$$

Here vBlend is the output vertex, V_n the n-th vertex and W_n is the n-th vertex's weight. For two, three and four weighted matrices, the above formula becomes:

$$vBlend = V_1W_1 + V_2(1.0 - W_1)$$
$$vBlend = V_1W_1 + V_2W_2 + V_3(1.0 - (W_1+W_2))$$
$$vBlend = V_1W_1 + V_2W_2 + V_3W_3+V_4(1.0 - (W_1+W_2+W_3))$$

Typically, the vertex structure for a blending vertex looks like this:

```
#define MAX_BLEND_WEIGHTS    2
typedef struct_BlendVertex
{
D3DVECTOR Position;
D3DVECTOR Normal;
FLOAT           fWeights[MAX_BLEND_WEIGHTS];     // blending weights
BYTE            btIndices[MAX_BLEND_WEIGHTS];    // index to bone-matrix-array
DWORD           Diffuse;      //color
FLOAT           tu,tv;        // texture coordinats
}
```

2.3 File Format

As mentioned before, the more data is available, the fewer calculations are required. It is advisable to store only the data on disk that cannot be generated on the fly by the program. Let’s take a quick look what data has to be stored in a 3D model file to support the animation of that model:

2.3.1 Skeleton

The data required to store a skeleton for an articulated object, apart from positional information for its joints itself, is the relationship between those joints. The easiest way to do this is by nesting the information for joints of a lower order in the hierarchy just below the joint which they are supposed to be connected to. For the joints themselves, all that needs to be known is the relative position of the joint which occupies the next higher order in the joint hierarchy of the skeleton. Bones (connections between the joints – vectors pointing from a joint to that joint's child joints) do not have to be explicitly saved in the file, as they are implicitly defined by the joints which they connect.

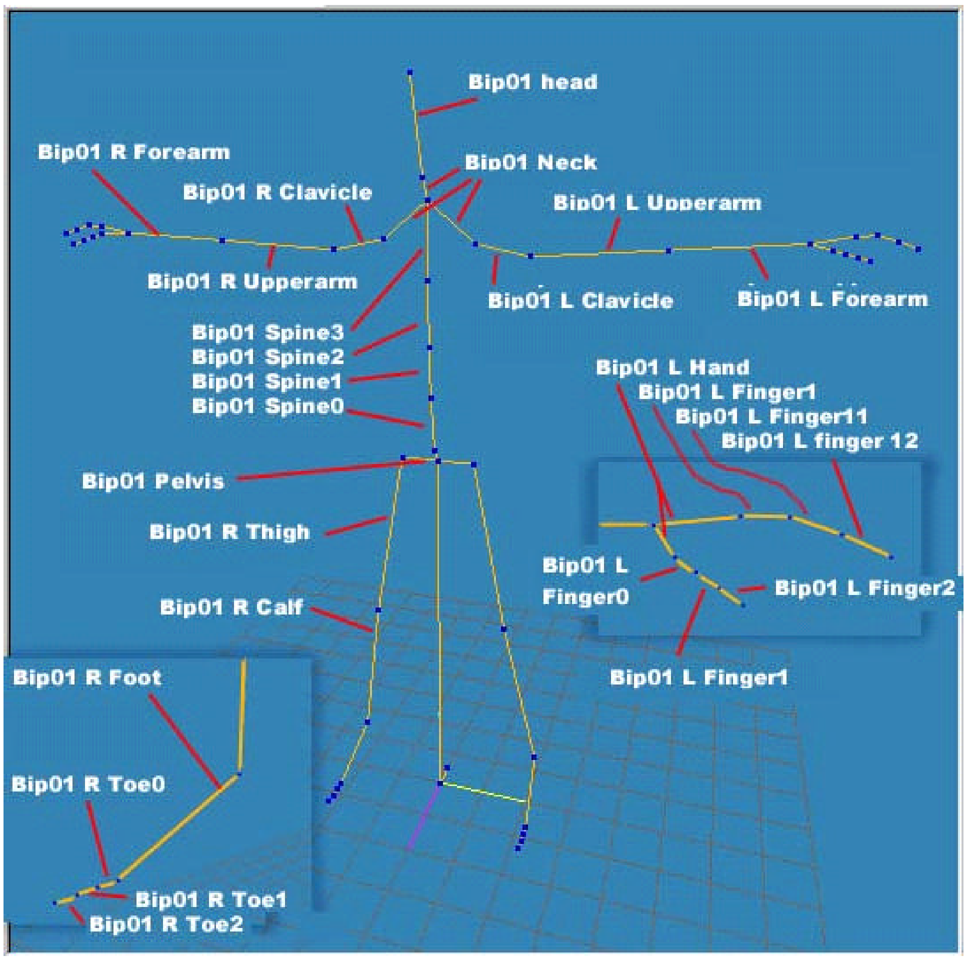


Figure 6: Hierarchic name convention for bones in X-files

Each Frame includes a *FrameTransformMatrix*, containing local transformations for that Frame. A Frame can also contain *Mesh* objects defining the vertices that form a 3D model, and child *Frames*.

2.3.2 Skin

The information which has to be stored for the skin of an articulated object, are the vertices which make up the skin. Each vertex structure has to contain data regarding the untransformed position of the vertex itself, the vertex normal, the UV texture coordinates of the vertex for texturing the model, a list of bones and skin weights, which define which of the joints of the skeleton are able to influence the vertex and by how much each of these joints influences the vertex.

2.3.3 Animation Cycles

Usually transformation information for all the joints of the articulated structure is stored in each key-frame of an animation. In the X-File format animation cycles are saved in the *AnimationSet* structure. Within an AnimationSet one can define a separate *Animation* for each part of the model which animates within the time frame of that particular animation cycle. Each Animation contains an *AnimationKey* structure which in turn contains a list of timed key transformations which will affect the part of the model referenced by the animation *[Anderson2001]*.

3. Approach & Implementation

3.1 Offline-animation Implementation

The SkinnedMesh sample, that is part of the DirectX8 SDK, already offers functions to load the mesh, its texture and animation sets from X Files *[Freid2001]*. The sample-code makes use of a framework that handles device initialisation and shows how to render a skinned mesh with software, hardware T&L indexed and non-indexed vertex processing. The mesh and animation information in the X Files is stored in frames, called *sFrames* here. Animation is handled by linked lists of key frames for the given joint and numbers of key-frames. To play an animation, the given function SetTime is called with a time-value within a loop through all the frames in FrameMove.

```
while ((pdeCur != NULL)                              // pointer to current sub object (mesh)
{
pframeCur = pdeCur->pframeAnimHead;                  // current frame is head of hierarchy
    while (pframeCur != NULL)
        {
         pframeCur->SetTime(pdeCur->fCurTime);            //go to timestamp in the animation
         pframeCur = pframeCur->pframeAnimNext;           //advance to next bone or joint
         }
pdeCur = pdeCur->pdeNext;                            //advance to next sub object in the x-file
}
```

To get the related frame number of the 3dsmax animation, we have to multiply the current timestamp by 4800 and divide through 30. (In case the animation is saved with 30 fps)
1/4800sec is the 3dsmax time unit. It is chosen to be a multiple of the standard frame rates 24 (movies), 60 (NTSC) and 50 (PAL).

The *SetTime* function searches the corresponding matrix key to the given timestamp at the current joint. If the timestamp lies between two matrix keys, the key-frame that is closer in time to the given value is displayed. This results in jumping and “jaggy” animation at lower frame-rates, since the keys usually resemble positions further apart in space. The same technique is used with scale and rotation keys. To smooth the animation, interpolation between the key-frames is necessary. A linear interpolation between the keys in question happens to be the best solution to guarantee constant movement. To enhance the basic animation functions and make them usable for a real-time gaming application, it is not enough to play custom animations forward and backward; other functions are needed as well.

One example for enhanced offline-animation is the blending of the end of an animation with the beginning of a new one. Additionally the interpolation is implemented either cubic or linear.
In a playing animation, the timing can be critical, so no additional blending frames are wanted. In this example 4 original frames are "lost" to guarantee the motion-flow.
The interpolated key-frames are generated “on the fly” and not stored in memory.

3.2 Physic-based Implementation

The current use of physic based animation in real-time applications is limited to special problems like animating chains (Fig. 7) or vegetation. But there are many more opportunities to breathe life into objects *[Walter1997]*.

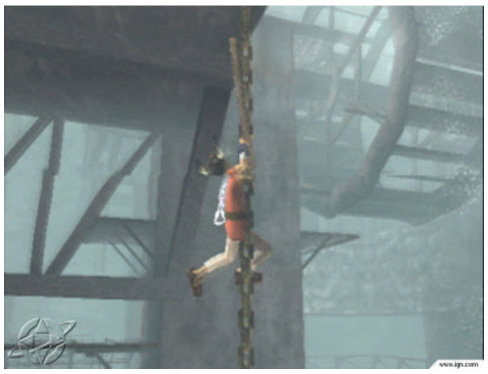


Fig. 7: Sony’s ICO real-time IK chain

Due the recent availability of hardware transform & lightning, we are now able to devote processor time to tasks other than environments. There are no more excuses for bad character animation *[Lander2000]*.

To achieve more life-like characters typical uses for procedural animations are look-at constraints, secondary motion, inverse-kinematics, muscle-bulges, chest-heaves, ponytails or cloth-animation.

3.2.1 Implementing physic based animation using bones animation

By modifying single bones in a skeleton system there is no more need for re-calculating every single vertex for physic based animation effects.
Let’s imagine a snowboarder character mesh which arms should shake when surfing over harsh snow underground. So we need an algorithm which can parse itself through an hierarchical bone-structure and modify the needed bone-positions. Since we want the “arm-shaking” to be calculated on the fly, there is need for 2 kinds of parameters.

3.2.1.1 Initial parameters:

```
// PARAMETERS FOR COS-BASED factor ( factor = cosf(time*speed)*amplitude
        m_fAnimProceduralSpeed =40.0f;
        m_fAnimProceduralAmplitude =0.3f;
// PARAMETERS for relations between hierachies
        m_fAnimProceduralAttenuation =0.3f;
```

```
// PARAMETERS FOR TRANSLATION ( e.g. transx*factor )
        m_fAnimProceduralTransX=0.0f;
        m_fAnimProceduralTransY=1.0f;
        m_fAnimProceduralTransZ=0.0f;

// PARAMETERS FOR ROTATION ( e.g. rotx*factor )
        m_fAnimProceduralRotX=0.0f;
        m_fAnimProceduralRotY=0.1f;
        m_fAnimProceduralRotZ=0.0f;
```

3.2.1.2 Runtime parameters:

```
// to fade out/in animation
m_fAnimProceduralFADE;

// type of animation (preset)
m_gAnimProceduralTYPE;
```

3.3 Approach:

First we calculate a value depending on playback speed and the maximum allowed amplitude.

```
// calc speed & amplitude
m_fFactor=cosf(m_fTime*m_fAnimProceduralSpeed)*m_fAnimProceduralAmplitude;
```

Then the hierarchical level of the modified bone will be brought into this calculation.

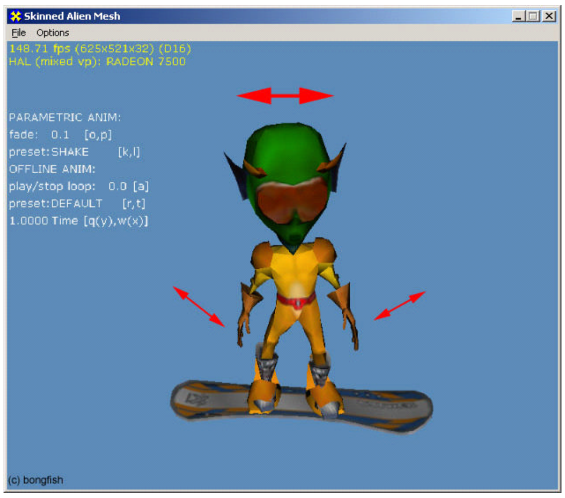
```
// increase attentutation for every hierachical level
if (m_fBoneModifier > 0.1f) m_fBoneModifier +=m_fAnimProceduralAttenuation;
if  (m_fBoneModifier < 0.0f) m_fBoneModifier = 0.0f;

// calc final modifier including hierachical level
m_fFactor = m_fFactor*m_fBoneModifier*(1-m_fAnimProceduralFADE);
```

Finally we setup a translation and rotation matrix containing the initial parameters and the appropriate runtime factors.

Those matrices are now multiplied with the original bone matrices, resulting in a blend of offline-animated & physic-based bone position.

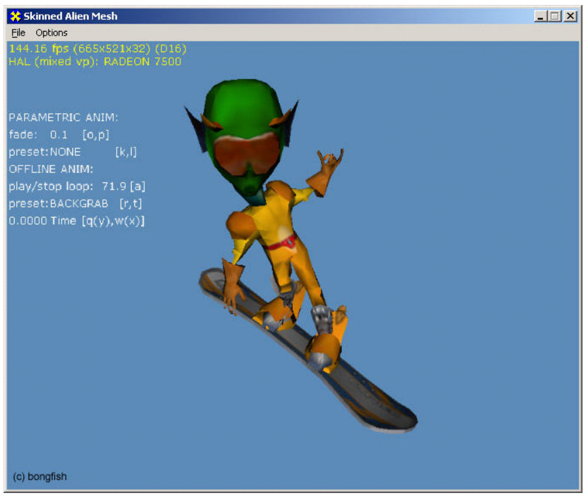
4. Results

To view different examples of offline character animations, their advantages and disadvantages as well as the combination with physics based character animation the use of our custom viewer is recommended.



```
148.71 fps (625x521x32) (D16)
HAL (mixed vp): RADEON 7500

PARAMETRIC ANIM:
fade:   0.1   [o,p]
preset:SHAKE      [k,l]
OFFLINE ANIM:
play/stop loop:   0.0 [a]
preset:DEFAULT    [r,t]
1.0000 Time [q(y),w(x)]
```

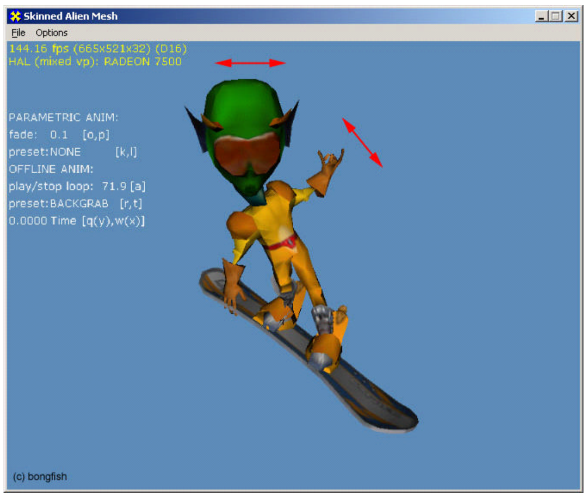
Fig. 8: Physic based "shake" animation



```
144.16 fps (665x521x32) (D16)
HAL (mixed vp): RADEON 7500

PARAMETRIC ANIM:
fade:   0.1   [o,p]
preset:NONE       [k,l]
OFFLINE ANIM:
play/stop loop:  71.9 [a]
preset:BACKGRAB  [r,t]
0.0000 Time [q(y),w(x)]
```

Fig. 9: Offline signature animation

With above implementation it is easy to trigger the "arm-shake animation" (Fig.8) with events coming from the physic-engine. Now the character mesh can be easily animated using an offline animation for signature moves (Fig.9) like personal expression or certain poses requiring a performance-expensive inverse-kinematics solution, while adding physic based animation on the fly with no extra performance cost (Fig.10).



```
144.16 fps (665x521x32) (D16)
HAL (mixed vp): RADEON 7500

PARAMETRIC ANIM:
fade:   0.1   [o,p]
preset:NONE       [k,l]
OFFLINE ANIM:
play/stop loop:  71.9 [a]
preset:BACKGRAB  [r,t]
0.0000 Time [q(y),w(x)]
```

Fig. 10: combined result of physic-based and precomputed animation

5. Conclusion & Outlook

Simulation and animation should work together in any modern computer game that makes use of character movement. On the one hand, complex, motion captured moves could be integrated to simulate character specific behavior like feelings or signature moves. On the other hand, real-time animation of certain bones can add additional realism without losing to much performance. The above implementation shows a way of combining both offline-animation and real-time procedural animation. For professional use enhanced motion blending of the animations would be useful. Like the support of multichannel precalculated animations (for example separate arm or leg movements). The realtime procedural part at this stage only supports animations based on sinus/cosinus movements. Although this works well for shaking-like animations, a more sophisticated approach would widen the range of use. An interesting application would be a real-time inverse-cinematic solution to simulate anatomical correct crash-animations using physic based triggers and terrain-information.

6. References

[Anderson2001] E.F. Anderson - Real-Time Character Animation for Computer Games, National Centre for Computer Animation, Bournemouth University, 2001

[Freidlin01] B. Freidlin, DirectX 8.0: Enhancing Real-Time Character Animation with Matrix Palette Skinning and Vertex Shaders, MSDN Magazine, 2001

[Lander1997] J. Lander, "On Creating Cool Real-Time 3D", 1997, http://www.gamasutra.com

[Lander2000] J. Lander, Using Technology to Create Believable 3D Characters, 2000

[Walter1997] M. Walter, A. Fournier, Growing and Animating Polygonal Models of Animals, Computer Graphics Forum, *16 (3)*, pp. 151-158, (August 1997)