Real-Time Character Animation for Computer Games

Eike F Anderson
National Centre for Computer Animation
Bournemouth University

ABSTRACT

The importance of real-time character animation in computer games has increased considerably over the past decade. Due to advances in computer hardware and the achievement of great increases in computational speed, the demand for more realism in computer games is continuously growing.
This paper will present and discuss various methods of 3D character animation and prospects of their real-time application, ranging from the animation of simple articulated objects to real-time deformable object meshes.

1 INTRODUCTION

Real-time character animation in real-time simulated virtual environments like computer games has progressed rapidly over the past few years from non-existence to a standard which now produces results that closely resembles those of off-line computer animated characters. This development has also been reflected by the changes of the visual style of computer games:
while originally game characters would either be hand-drawn or computer generated 2D images that are projected onto a plane that is parallel to the viewport (decals, as described in [15]), modern computer games now use three-dimensional game characters which are real-time animated inside the virtual environment and projected onto the screen.
Many of the techniques which only four years ago in [8] were listed as possible future developments, and had until then only been used in off-line animation, like skeletal animation and real-time deformation and skinning of 3D meshes, are now established methods for the implementation of real-time 3D character animation.
The aim of this paper is to compare some of the existing real-time character animation techniques and to discuss their advantages and disadvantages, and to provide a demonstration of the implementation of one of the algorithms, illustrating its suitability for real-time character animation.

2 RELATED WORK

Both, memory and processing power, are expensive. More realism in games can be achieved by increasing the detail of objects in the game and by introducing objects into the virtual environment that are non-essential for the game itself, but which will be recognisable as natural for the given situation by the player. This has an impact on memory utilisation.
Memory usage can be reduced by storing less data and by generating more data *on the fly*, which in turn increases the amount of necessary operations that have to be performed by the processor.
Therefore, in real-time animation the aim is to find the balance between these two factors that will result in maximum realism at the minimum cost.

2.1 3D Hierarchic Articulated Objects

In a hierarchic character as described in [8], 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 (**Figure 1**). [1] describes how the hierarchic character is set up and how it can be displayed, by recursively traversing the hierarchic data structure starting from the root, passing down the transformation of the parent objects to its children and there concatenating the passed down translation matrix and the local translation matrix.

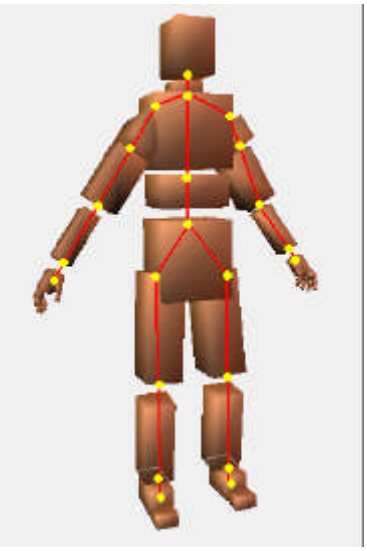

Figure 1

Older 3D games like the original **Tomb Raider**[P1] use this method for character animation.

The flexibility and adaptability of this method are its main benefits:

Animation data can be generated *on the fly* using techniques such as Inverse Kinematics (see below), and applied to the model in real-time. Memory usage is also small as the vertex and transformation information for each object contained in the model needs to be stored only once.

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. Although it is possible to hide these gaps by overlapping the objects that make up the model, this again will result in visible seams.

2.2 Inbetweening (Blending) between Character Meshes

A different approach to real-time character animation is the one described in [6], using more than one object representing the same character but each of these objects showing the character in a different pose. All objects have to contain the same number of vertices. In this case the animation is realised by interpolating the corresponding vertex positions between the different objects as illustrated in [2] (**Listing 1**) over time and blending the two objects (**Figure 2**) or just by switching between the separate objects as described in [8] (**Listing 2**).

The original **Quake**[P2] computer game used this method for animating characters and modern graphics processors have the functionality to calculate this in their hardware [2].

The result is a smooth animation with no gaps or visible seams in the character model's geometry. Also, as long as a not too complex interpolation method like linear interpolation is used, the amount of calculations required for the animation of the

Listing 1

```
for(i=0; i<VERTEX_NUM ;i++)
// linearly interpolate between 2 key-frames
{
  VTYPE kvertex=s_vert[i][k+1]-s_vert[i][k];

  disp_vert = s_vert[i][k]+(1-t)*kvertex;
  // disp_vert is the vertex that will be
  //displayed
  // t it the time delta between 2
  // keyframes with
  // 0 <= t <= 1
  // k is the current key-frame
}
```

Listing 2

```
for(i=0; i<VERTEX_NUM ;i++)
// switch between two key-poses
{
  if(t>0.5)
    disp_vert = s_vert[i][k+1];
  else
    disp_vert = s_vert[k];
  // t ist the time between 2 key-frames
  // with  0 <= t <= 1
}
```

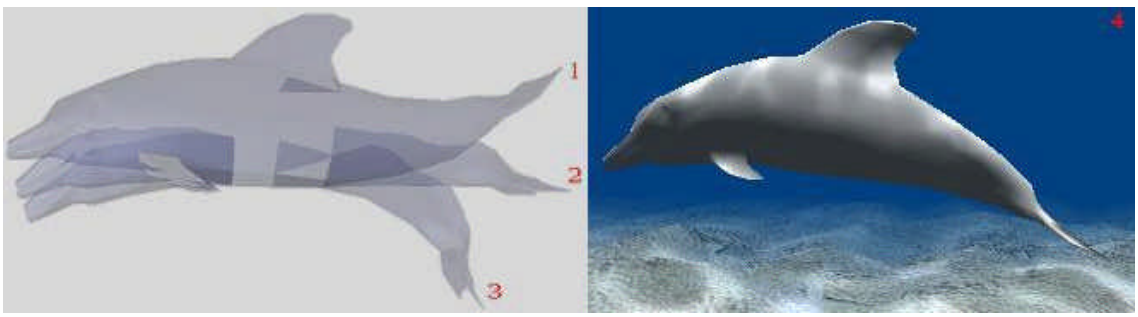

Figure 2 - vertex blending demo from DirectX 8 SDK

mesh is very small.
The downside of this technique however is that key-frames have to be very frequent to effectively control the animation (**Figure 3**) which means that the amount of data that has to be stored is very large. This has a detrimental effect on memory usage as all object copies that are needed in an animation sequence will have to be kept in memory for at least as long as it takes the animation sequence to finish. Characters that are animated with this method are also far less flexible than characters which are based on a hierarchic model representation, as the whole of the character animation has to be pre-defined off-line and no *on the fly* generation of additional animation data is possible.

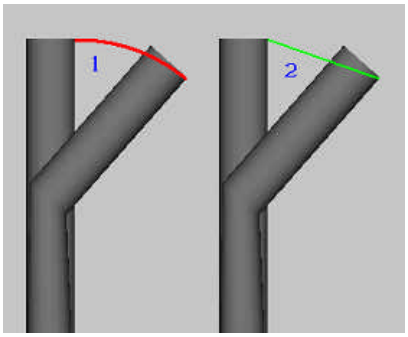

Figure 3

2.3 Skeletal Animation and Mesh Skinning

Skeletal animation was developed to simplify the animation process for dealing with articulated objects (characters like bipeds, for example) and to provide more realism through improving the looks of animated objects by making them more life-like. It is an improvement on both of the previously mentioned techniques:
it uses an endoskeleton - a hierarchic structure of joints - which drives a skin - a vertex mesh representing the shape of the object. It is this splitting of mesh data and hierarchic position information into two separate data-structures, which makes skeletal animation and mesh skinning superior to the previously mentioned techniques.
A bone is simply a transformation matrix, determining the position of the bone in relation to its parent bone, and all the bones of the articulated object together form the skeleton. Explicitly only the skeleton is animated, using an algorithm similar to that used for animating a hierarchic articulated object, which in turn implicitly animates the skin. Memory usage for skeletal animation is small, as all skin vertices have to be stored only once. It requires a significantly lower amount of information to be stored than the mesh inbetweening method discussed above.

2.3.1 Rigid Skinned Characters

In a rigid skinned character each of the skin vertices is associated with a single bone (**Listing 3**). The resulting animated mesh is seamless, but close to joints deformations can result in creasing (**Figure 4**), so this method of skinning should only be used for low-polygon objects that do not have a lot of modelled detail.
[5] describes a very simple and effective way of automating the generation of a skeleton and applying a rigid skin to it by creating a copy of an existing character mesh and transforming that copy into the skeleton of the character. Firstly the copy is scaled down and broken up into small parts which then are associated with the skin vertices that are closest to them, which makes these object parts the bones of the skeleton.

Listing 3

```
struct skin_vertex
{
  FLOAT    x;    // floating point x position of vertex
  FLOAT    y;    // floating point y position of vertex
  FLOAT    z;    // floating point z position of vertex
  FLOAT    u;    // floating point u texture coordinate
  FLOAT    v;    // floating point v texture coordinate
  FLOAT    nx;   // floating point x direction of normal
  FLOAT    ny;   // floating point y direction of normal
  FLOAT    nz;   // floating point z direction of normal
  bonetype *b;   // reference pointer to the bone which
                 // is associated with the vertex
};
```

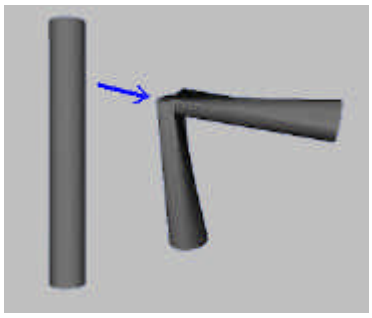

Figure 4

2.3.2 Soft Skinned Characters

Soft skinned character animation is based on rigid skinned character animation and provides

a solution for the shortcomings of that method:
the creasing of the mesh around joints that are being deformed is greatly reduced, giving the character a much more life-like look (**Figure 5**).
This is achieved by allowing more than just one bone to influence each vertex as described in [14], 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*). Although a real life bone would not really directly deform the skin but would do so indirectly by flexing and relaxing muscles that are attached to it, this effect is convincingly simulated, as the mesh vertices are offset from the bones and are influenced by transformations of more than one bone:
the final vertex position is then determined by adding up vertex transformations by the transformation matrices of all the bones that influence a vertex, each multiplied with its corresponding skin weight. For this to work, all the skin weights for a vertex must add up to 1.
A weight of 1 for a bone means that the vertex is only affected by that bone and 0 means it is unaffected by the bone (**Listing 4**).

Figure 5
skinned mesh demo from DirectX 8 SDK

Listing 4

```
for(i=0; i<VERTEX_NUM ;i++)
// for vertices that are affected by 2 bones
{
  vector v=(vert[i].x,vert[i].y,vert[i].z);
  vector sv0,sv1;
  bonetype *b0,b1;
  matrix tmat;

  b0=vert[i].b0;
  cmv(sv0,b0->t_mat,v);
  // concatenate a transformation matrix with a vector

  b1=vert[i].b1;
  cmm(tmat,b0->t_mat,b1->t_mat);
  // concatenate one transformation matrix with another
  cmv(sv1,tmat,v);
  // concatenate a transformation matrix with a vector

  disp_vert=sv0*vert[i].weight0+sv1*(1-vert[i].weight0);
  // calculate display vertex
}
```

As only a few more data elements are added to each vertex (**Listing 5**), the memory usage for this method only grows insignificantly, but it requires more computations to display than the simple rigid skinned skeletal animation described above, or the model tweening method:
for ***V*** skin vertices and n bones affecting each vertex ***V*n*** vertex transformations have to be calculated for each animation frame to calculate the vertex positions.
The more bones influence a single vertex, the more natural the skin surrounding that vertex will behave.

Listing 5

```
struct vertex
{
  FLOAT     x;           // floating point x position of vertex
  FLOAT     y;           // floating point y position of vertex
  FLOAT     z;           // floating point z position of vertex
  FLOAT     nx;          // floating point x direction of normal
  FLOAT     ny;          // floating point y direction of normal
  FLOAT     nz;          // floating point z direction of normal
  bonetype *b0;          // reference pointer to the first bone
                         // associated with the vertex
  bonetype *b1;          // reference pointer to the second bone
                         // associated with the vertex
  // ... as many bones as can influence a single vertex
  FLOAT  weight0;        // bones-1 weights - as all weights have
                         // to add up to 1, the last bone's weight
                         // can be calculated by subtracting the
                         // sum of all other weights from 1
};
```

```
        (bone_0_transformations     *   bone_0_weight)
+       (bone_1_transformations     *   bone_1_weight)
+                          ...
+       (bone_N-1_transformations *     bone_N-1_weight)
+       (bone_N_transformations     *    bone_N_weight)
_____________________________________________________

=       final_vertex_transformation
```

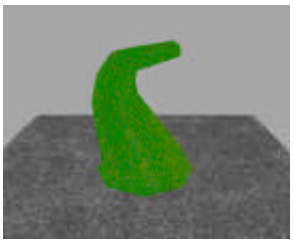

Figure 6

It is even possible to simulate the presence of muscles within a skinned mesh, by adding dummy bones whose sole purpose is to deform parts of a skin by adding further weights to the skin.

The "Bendy Tentacle Demo" program that was written to demonstrate some of the methods described in this document contains an implementation of a soft skinned mesh with 2 bone influences for each skin vertex (**Figure 6**).

Modern graphics processors contain hardware solutions that are able to optimise the performance of this mesh skinning technique [2].

2.4 Real-Time IK

Inverse Kinematics is a technology which originated in robotics. It has since found its way into off-line animation as demonstrated by [10], but has so far only rarely been used for real-time animation. The fact, that the IK functionality of off-line animation systems is applied in real-time is proof, that it is possible to have real-time IK. However the number of calculations which have to be carried out for the majority of IK algorithms is too great to be of any use in a modern computer game, where a big part the processing capability is used for maintaining the virtual environment [13]. The main reason for this is that often the IK algorithms used are direct copies of the robotics IK algorithms, but whereas only few industrial robots have more than six joints, a humanoid character in computer animation can easily have one hundred or more joints, which can slow down these IK algorithms considerably. In [11] the authors present an innovative approach to IK algorithms, tailored to computer animation, unlike many other IK algorithms, which is targeted at off-line animation and therefore still too complex for generating real-time animation for computer games. [9] implements a 2D geometrical IK solution that is transferred into 3D, which can create acceptable results, but by far the best solution for use in games is the one described in [4]. In this algorithm a force is applied to the end-effector of the IK chain, pulling it towards the IK target, trying to reach it, or - if that is impossible because it is out of the range which the articulated object can reach to - pointing towards it. For each bone in the IK chain, back tracking the chain from its end-effector to the start of the chain, the IK solver has to be invoked for each axis of rotation for that bone.

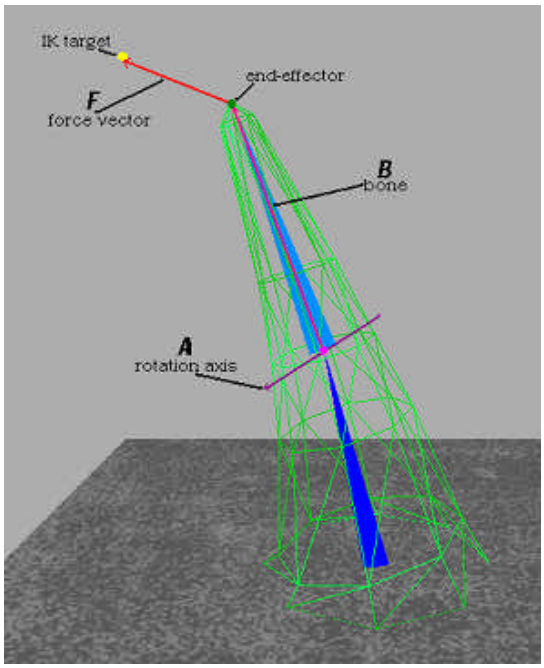

Figure 7

The pulling force towards the IK target is equivalent to the length of the vector pointing from the end-effector of the IK chain to the IK target. If *F* is the force vector, *A* is the rotation axis for the current bone and *B* is the bone itself (**Figure 7**), the basic formula for finding the angle which has to be added to the current rotation angle of the current

bone is the following:

$$Angle = |F| * A\mathbf{x}F * B\mathbf{x}F * (((A\mathbf{x}B)\mathbf{.}F) \ / \ |((A\mathbf{x}B)\mathbf{.}F)|)$$

. = dot-product
x = cross-product

The end-effector will not snap to the IK target though, but move there over a short period of time.
A variation of this algorithm is implemented in the "Bendy Tentacle Demo" program that was written to demonstrate some of the methods described in this document.

3 FILE FORMAT CONSIDERATIONS

As mentioned before, the more data is available, the fewer calculations are required. However while file storage on hard disk is cheap, and file sizes no longer play a significant role because of the high computer specifications that nobody would have dreamed of only a few years ago and which nowadays are commonplace in home user computers, it still takes a relatively long time to transport the necessary data from a file on disk into the much faster RAM of the computer. It is therefore advisable to only store on disk the data that cannot be generated *on the fly* by the program.
This section of this paper will try to provide a simple analysis describing what data has to be stored in a 3D model file to support the animation of that model.

3.1 REQUIREMENTS

Skeleton
The data required to store a skeleton for an articulated object - as described earlier - on disk, apart from positional information for its joints itself, is the relationship between those joints. By far 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 (**Listing 6**).
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. The exception to this is the highest order joint in the hierarchy, the root of the skeleton, which should contain global positioning data for the whole object instead of the local positioning data.

Listing 6

```
HEAD <- file header
|
+- TEXT <- global texture chunk
+- MATE <- global material chunk
+- OBJE <- object chunk - highest chunk in object
|          hierarchy
|
+- NAME <- the name of the object (for example,
|         'torso' etc)
+- VECT <- the vector pool
+- LODD <- LOD information of sequence in which
|          vectors will be collapsed
+- SKIN <- a skinned sub-object as sub-chunk of
| |        object chunk
| |
| +- NAME <- For example, 'Upper left arm'
| +- MATE <- local material chunk
| +- POLY <- geometry for 'Upper left arm'
| +- PIVO <- joint data for 'Upper left arm'
| +- SKIN <- a skinned sub-object as sub-chunk of
|   |        object chunk
|   |
|   +- NAME <- For example, 'Lower left arm'
|   +- MATE <- local material chunk
|   +- POLY <- geometry for 'Lower left arm'
|   +- PIVO <- joint data for 'Lower left arm'
```

In addition to that, it may be useful to store constraints for each joint which limit the freedom of transformations which can be applied to the joint, so that a program which generates animation *on the fly* can distinguish illegal movements which might cause the object to collapse (**Figure 8**).
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.

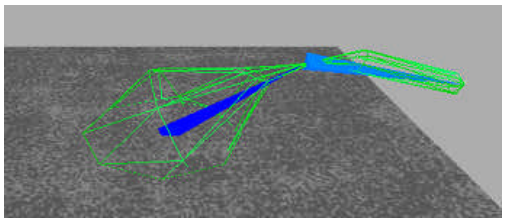

Figure 8

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.

IK chains
To use IK with an articulated object, that object will have to include IK chains, defining which will be affected by the IK. The information required for this would be some sort of list, describing which of the joints in the IK chain is the start of the chain, and which of the joints is the end-effector on the other side of the IK chain. That information could easily be stored inside the bone structures themselves, provided that the joints can point towards its parent joints, so that the IK chain can be back-tracked from its end-effector to its starting joint.
For more complex IK operations one could also introduce a weight or sensitivity value which would define by how much a bone in the IK chain would be affected by the IK.

Animation Cycles
The easiest solution by far for this is the one implemented in id Software's Quake II[P3] file format:
All models have an identical number of frames, every single frame is a key-frame and the length of each animation cycle is pre-defined. While this reduces the calculations which are necessary for displaying the animation to a minimum, it also reduces the flexibility of the file format. Usually though, transformation information for all the joints of the articulated structure is stored in each key-frame of an animation.

3.2 A LOOK AT THE DIRECTX X-FILE FORMAT

[3], [7] and [15] all contain descriptions of the DirectX X-File object format and provide implementation information for loading, and displaying files stored in the X-File format. The X-File format is based on templates, which makes it very flexible and adaptable to specific problems. There are already a number of pre-defined templates that allow data that is compliant with a number of the above mentioned requirements to be stored in an X-File. The unfulfilled requirements can easily be included by adding custom templates to the file format. A full description of this file format and its capabilities can be found in [P4]

Skeleton
The DirectX X-file format supports skeletons in hierarchic articulated objects in the form of the so-called ***Frames*** of reference, which are the X-File equivalent for bones.

Each Frame can contain 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*** as described in [3]. (**Listing 7**)

Listing 7

```
// a sample X-File
// assumption: parent_mesh and child_mesh,
//             parent_matrix and child_matrix
//             are already defined

Frame Object_Root {
   {parent_mesh}
   {parent_matrix}

   Frame Object_Child {
      {child_mesh}
      {child_matrix}
   }
}
```

IK chains

IK chains are not part of the X-file format, although the modular and expandable nature of the file format, which allows the addition of new *templates*, would make it relatively easy to create an extension to the file format which could then be parsed and interpreted by the loading program.

An IK chain template for the X-file format could easily be implemented (**Listing 8**).

Listing 8

```
template IKchain {
Frame
// any number of Frame objects can become a
// child of an IKchain. These objects must be
// specified by reference
}

// example
IKchain MyIK {
  {Object_Root}
  {Object_Child}
}
```

Animation Cycles

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.

4 SAMPLE IMPLEMENTATION OF ALGORITHMS

This paper is provided with a sample application using OpenGL, which demonstrates the use of some of the algorithms which have been discussed earlier in this paper. The "Bendy Tentacle Demo" (**Figure 9**) shows a simple articulated object (tentacle) which contains two bones which influence the weighted skin mesh.

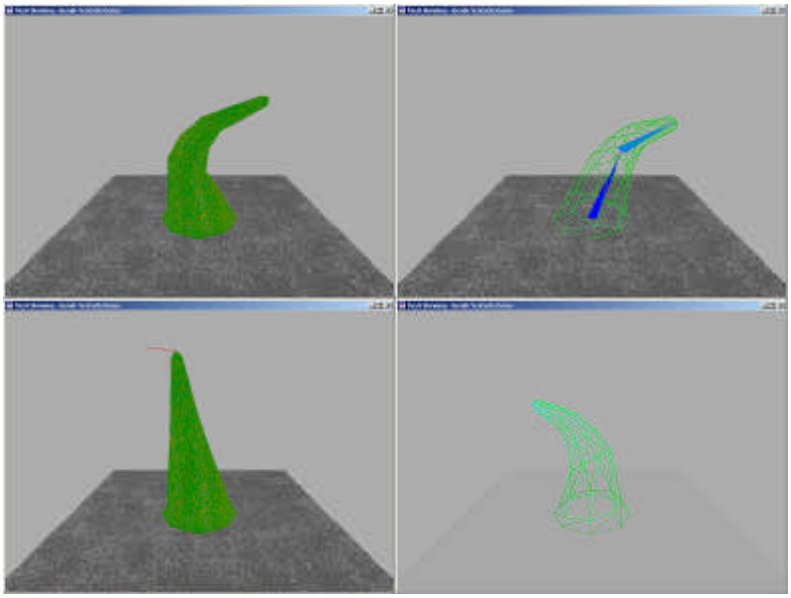

Figure 9

Interactive Forward Kinematics, similar to the "robotic arm demo" in [12] were included in this application to allow for the testing of the effects of the weighted mesh skinning function. The user can also interactively move an IK target around the scene, which exerts a pulling force on the IK chain's end-effector which is situated in the tip of the tentacle. The full source code of this demo application is included in the source distribution file of the demo program.

The fewer instructions that have to be executed, the faster the program will run.

One of the important steps in calculating real-time animation data is therefore the identification and elimination of bottlenecks in the animation program.
For skeletal animation once every frame, the program would first calculate the transformation matrices for each joint and then update the transformation matrices that are part of the joint. If a hierarchic dynamic structure is used to store the skeleton, the joints would usually be traversed recursively for updating (**Listing 9**). If a linear, fixed-size data structure is used instead, the number of instructions necessary for updating the joints of the skeleton can be reduced (**Listing 10**).

Listing 9

```
struct bone
{
  ...

  bone *child;
  bone *sibling;
};

...
void traverse(bone *b,...)
{
  ...
  if(b->child!=NULL)
    traverse(b->child,...);
  if(b->sibling!=NULL)
    traverse(b->sibling,...);
  ...
}
```

Listing 10

```
struct bone
{
  ...

  int parent;
};

void update(...)
{
  ...
  for(int i=0;
i<NUM_BONES; i++)
{
  ...
}
```

The implementation of the mesh skinning can be found in the function *skin()*

```
skin()
{
        Glfloat   tempx1,tempx2,tempy1,tempy2,tempz1,tempz2,tvec[4],vec[4],transform[16];
        build_matrices();
        for(int i=0;i<NUM_VERTICES;i++)   // calculate skin vertex positions
        {
                int bone1,bone2;
                bone1=tentacle_skin[i].bone1;          // retrieve the skin weight for the first bone
                bone2=tentacle_skin[i].bone2;          // retrieve the skin weight for the second bone
                vec[0]=vertices[i][0];
                vec[1]=vertices[i][1];
                vec[2]=vertices[i][2];
                vec[3]=1.0f;                           // homogeneous coordinate
                matxvec(tvec,bones[bone1].tmatrix,vec);// calculate the transformation of the skin vertex for
                tempx1=tvec[0];                        // the first bone
                tempy1=tvec[1];
                tempz1=tvec[2];
                matxmat(transform,bones[bone1].tmatrix,bones[bone2].tmatrix);
                matxvec(tvec,transform,vec);           // calculate the transformation of the skin vector for
                tempx2=tvec[0];                        // the second bone
                tempy2=tvec[1];
                tempz2=tvec[2];
                tentacle_skin[i].x=(tempx1*tentacle_skin[i].weight)+(tempx2*(1-tentacle_skin[i].weight));
                tentacle_skin[i].y=(tempy1*tentacle_skin[i].weight)+(tempy2*(1-tentacle_skin[i].weight));
                tentacle_skin[i].z=(tempz1*tentacle_skin[i].weight)+(tempz2*(1-tentacle_skin[i].weight));
        }
}
```

In the function above (*simplified version of source*), each skin vertex of the model is transformed by the transformation matrices of both bones which have an influence on that skin vertex, and the two resulting vertex positions are then multiplied by the corresponding bone's weight. The resulting vertex positions are then added, to give the final vertex position of the current vertex.

The implementation for the IK solver can be found in the function *solvejoint()*

```
// magnitude() returns the length of the passed vector
// sinEnclosed returns the value of the cross product of the 2 passed vectors
// cosEnclosed returns the value of the dot product of the 2 passed vectors
// sign returns the sign of the passed value
double solvejoint(GLfloat *a,GLfloat *b,GLfloat *f)
{
        GLfloat rlength;
        GLfloat r[3];
        r[0]=(a[1]*b[2])-(a[2]*b[1]);
        r[1]=(a[2]*b[0])-(a[0]*b[2]);
        r[2]=(a[0]*b[1])-(a[1]*b[0]);
        rlength=sqrt((r[0]*r[0])+(r[1]*r[1])+(r[2]*r[2]));
        r[0]/=rlength;
        r[1]/=rlength;
        r[2]/=rlength;
        return magnitude(f)*sinEnclosed(a,f)*sinEnclosed(b,f)*sign(cosEnclosed(r,f));
}
```

In the function above (*simplified version of source*), the length of the force vector F is multiplied with the cross-products of A, the rotation axis, and F and B, the bone, and F, which in turn is multiplied with the dot-product of the cross-product of A and B with F. The resulting value is the angle which has to be added to the current angle of rotation, to solve the IK problem.

The main control keys for the demo application are:

[Esc] - end the application
[B] - toggle bone visibility on/off
[F] - toggle Forward Kinematics on/off
[I] - toggle Inverse Kinematics on/off
[P] - pause the default animation of the tentacle
[R] - toggle the turntable rotation on/off
[T] - toggle between wireframe and smooth shading

Arrow Keys and PgUp and PgDn - move the tentacle in interactive modes

For further information please consult the "readme" file that is included in the source and binary distribution files for the demo application.

4.1 FUTURE DEVELOPMENT

The source code of the "Bendy Tentacle Demo" could without many problems be converted and extended into a general 3D mesh skinning library for real-time animation. It should also be relatively simple to port the current application source code to a different graphics API like DirectX. This would be especially useful as the native support for skinned meshes by the DirectX API is less than adequate, providing little information about the implementation, which could have been realised a lot simpler, but with similar results, as demonstrated with the "Bendy Tentacle Demo".
Some more work can be done on the IK solver - weights and constraints could be added to increase the realism and improve the usability of the methods presented in this paper.
Further improvements can be made on the animation timer. Currently the program uses a standard timer event, but a high resolution timer could increase the smoothness of the animation considerably.

5 CONCLUSION

This paper and the demo program have presented a simple and efficient algorithms for realistically representing a smooth skinned articulated object in a real-time calculated virtual (game) environment.
Further techniques suitable for real-time character animation have been presented.
This paper and the demo program also demonstrate the real-time generation of additional animation data for skeleton-driven objects, using an "inexpensive" Inverse Kinematics algorithm.
This shows that many off-line animation techniques can be successfully implemented in real-time applications. With ever increasingly more powerful hardware becoming available, it is certain that these techniques and methods will find their way into real-time animation in the foreseeable future.

ACKNOWLEDGEMENTS

Robert Bergkvist
Prof. Peter Comninos
Adam Vanner (project tutor)

REFERENCES

[1] Prof. Peter Comninos, "Defining Hierarchic Objects", 3D Computer Animation. National Centre for Computer Animation, Bournemouth University, Bournemouth, UK.

[2] Cem Cebenoyan, "Efficient Animation", GDC2001 presentation, NVIDIA Corporation

[3] Robert Dunlop, Dale Shepherd and Mark Martin, *Teach Yourself DirectX 7 in 24 Hours*, Sams Publishing, Indianapolis, IN

[4] Hugo Elias, "Inverse Kinematics", *freespace.virgin.net/hugo.elias/models/m_ik.htm*

[5] Torgeir Hagland, "A Fast and Simple Skinning Technique." In *Game Programming Gems*, Charles River Media, Rockland, MA

[6] Peter J. Kovach, *Inside Direct3D*, Microsoft Press, Redmont, WA

[7] Peter J. Kovach, *The Awesome Power of Direct3D/DirectX*, Manning Publications, Greenwich, CT

[8] Jeff Lander, "On Creating Cool Real-Time 3D", *www.gamasutra.com*

[9] Adrian Perez and Dan Royer, *Advanced 3-D Game Programming Using DirectX 7.0*, Wordware Publishing, Plano, TX

[10] Steve Pitzel, "Character Animation: Skeletons and Inverse Kinematics", Intel Developer Service

[11] Deepak Tolani, Ambarish Goswami and Norman I. Badler, "Real-time inverse kinematics techniques for anthropomorphic limbs", Computer and Information Science Department, University of Pennsylvania, Philadelphia, PA

[12] Mason Woo, Jackie Neider and Tom Davis, *OpenGL Programming Guide, Second Edition*, Addison-Wesley, Reading, MA

[13] Alan Watt and Fabio Policarpo, *3D Games Real-time Rendering and Software Technology*, ACM Press, New York, NY

[14] Ryan Woodland, "Filling the Gaps – Advanced Animation Using Stitching and Skinning." In *Game Programming Gems*, Charles River Media, Rockland, MA

[15] Stefan Zerbst, *3D Spieleprogrammierung mit DirectX in C/C++*, Libri BoD, Braunschweig, Germany

PRODUCT REFERENCES

[P1] Tomb Raider™ Core Design Limited *http://www.eidos.com*

[P2] Quake® id Software, Inc *http://www.idsoftware.com*

[P3] QuakeII® id Software, Inc *http://www.idsoftware.com*

[P4] DirectX 8.0 SDK Microsoft Corporation *http://msdn.microsoft.com/directx*