Efficient Collision Detection for Skeletally Animated Models in Interactive Environments

Vadim Macagon[1] and Burkhard Wünsche[2]
Graphics Group, University of Auckland, Auckland, New Zealand
[1]vadim_mcagon@hotmail.com, [2]burkhard@cs.auckland.ac.nz

Abstract

Skeletally animated polygonal models are common in interactive 3d environments such as computer games. This paper presents an efficient technique for performing collision detection for such models with the possibility of integrating a skeletal animation system (based on pre-generated animations) with an existing physics engine in order to provide physically realistic responses to collisions. The results are useful for 3d simulations in the areas of computer graphics, sport science, and computer games.

Keywords: collision detection, skeletally animated models, interactive immerse environments

1 Introduction

Collision detection is of extreme importance in visual simulations of 3d environments where various objects can interact with each other. The choice of a collision detection technique depends on the complexity and 3d representation of objects and the information required for the simulation of an object's response to a collision such as elastic deformation.

Collision detection can be divided into two phases: the *broad phase* quickly eliminates all objects that cannot possibly collide within a time frame. Examples are bounding volumes, octrees, BSP trees [1] and Hubbard's space-time bounds [2]. The narrow phase examines pairs of objects identified as potentially colliding and detects (if necessary) where and how the objects collide. Examples are separating planes [2] and Lin-Canny closest features tracking [3].

Presently there are a number of algorithms and libraries that provide fast collision detection in 3d environments, however, they typically require the 3d objects to consist of static geometry and they treat each object as one polygon mesh [4] or as a collection of basic primitive shapes such as spheres, capsules and boxes. Typically when a collision between objects is detected a list of pairs of polygons that intersect is produced, including for some libraries the intersection points. These results are not sufficient to obtain a higher-level description of the interaction between objects. In this work we will suggest a solution to this problem for objects, which consist of deformable meshes that represent humanoid models.

As a simple scenario we consider a soccer game: most of the time each player is in contact with the ground, the soccer ball, or other players. Various types of collisions occur and must be handled in order to make the soccer ball fly with each kick and to prevent players from falling through the ground. For realistic simulations we need to know which limbs, and which parts of the limbs, are involved in an impact so that we can model the response of the players to the various impacts they experience. In order to obtain higher-level collision information the polygon mesh that makes up the player model must be subdivided into a number of groups representing individual limbs or limb parts.

When a collision between objects has been detected the objects need to be repositioned to ensure they do not interpenetrate each other unless required. Furthermore in the case of humanoid models it must be possible to change the pose of a model in response to impacts.

With animated articulated models there are generally two ways to respond to collisions. The simple way of producing a response to an impact involves creating a collection of pre-canned animations (i.e. pre-recorded), and playing one of these depending on which limb or body part is hit; this has been widely used in computer games. However since there is only a fixed set of animations the end user will quickly notice that the responses to some impacts are not what one would expect to see in the real world.

An alternative approach to producing more realistic responses involves the use of a physics engine. The player model can be approximated by a collection of rigid bodies that are connected together and are subjected to physical simulation. This approach doesn't restrict the player model to a set of pre-canned animations; instead the player's pose can be changed in an infinite number of ways based not only on the points of impact, but also the force of the impact.

Our work uses the Open Dynamics Engine (ODE) [5] in order to provide physically realistic responses upon impact of the humanoid player models with their environment. ODE is a free, industrial quality library for simulating articulated rigid body dynamics. The player models in our simulation will be represented in

ODE as a collection of rigid bodies, connected together by a number of joints that constrain the positions and orientations of the rigid bodies. While ODE has its own collision detection facilities they are not sufficient for interactive animations like the one described in our example scenario. Instead the results obtained with our algorithm are used by ODE, which will in turn try to ensure all constraints are satisfied and hence produce a visually realistic response to collisions in an interactive simulation.
We implemented our simulations using the Nebula Device [6], which is a free modular framework for building 3d visualizations and game engines. Nebula provides a character animation system and contains a collision detection system that deals with static geometry by making use of the Optimised Collision Detection (OPCODE) [4] library. However the collision system is currently incapable of handling animated characters and our work presents a solution for this.

2 Skeletal Animation

Each player model consists of a single mesh that is deformed based on the underlying skeleton. Animation of the character using pre-canned character animation works by changing the pose of the skeleton. It is important to understand how this system works in detail since the collision detection and response techniques presented later on are geared towards working with such models.

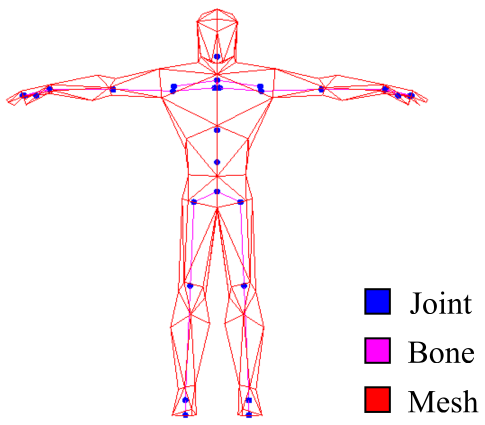

Figure 1: Low Resolution Character Model.

A model's skeleton is made up of a collection of joints, arranged in a hierarchical structure. Figure 1 shows the make-up of a player model. The bones are just a visual aid to make it easier to see the relationships between the joints and are typically only used by animators during the creation of the animations. Every vertex in the mesh is weighted by one to four joints (which is the maximum allowed number by Nebula) and the final position of each vertex (in model space) will be determined by the current pose of the skeleton.
Each joint in the skeleton has two rotation and two translation components, and all joints except for the root joint have a parent joint. A rotation component is described by a quaternion, and a translation component is described by a 3-vector. One pair of rotation/translation components contains the initial position of the joint relative to its parent, and its initial orientation. The second pair of rotation/translation components contains the current position of each joint relative to its parent, and its current orientation. The algorithm for determining the final position of each vertex in the mesh is known as *skinning* and works as shown in Listing 1.

```
for each joint $\mathbf{j}$
    let $\mathbf{T_i} = \mathbf{T_{ir}}\,\mathbf{T_{it}}$
    if $\mathbf{j}$ has a parent
        let $\mathbf{T_i} = \mathbf{T_i}\,\mathbf{T_{ip}}$
    let $\mathbf{T_c} = \mathbf{T_{cr}}\,\mathbf{T_{ct}}$
    if $\mathbf{j}$ has a parent
        let $\mathbf{T_c} = \mathbf{T_c}\,\mathbf{T_{cp}}$
    let $\mathbf{T_s} = \mathbf{T_i}^{-1}\,\mathbf{T_c}$
for each vertex $\mathbf{v}$
    let $\mathbf{v_c} = (\mathbf{0},\mathbf{0},\mathbf{0})$
    for each joint $\mathbf{j}$ that $\mathbf{v}$ is weighted by
        let $\mathbf{v_c} = \mathbf{v_c} + \mathbf{v_i}^{\mathbf{T}}\,\mathbf{T_s}\,\mathbf{w_{iv}}$
```

Listing 1: Skinning

- $\mathbf{T_{ir}}$ and $\mathbf{T_{it}}$ represent the initial joint rotation and translation components, respectively.
- $\mathbf{T_i}$ is known as the *pose matrix* and specifies the initial position of the joint in model space.
- $\mathbf{T_{ip}}$ is the pose matrix of the parent joint.
- $\mathbf{T_c}$, $\mathbf{T_{cr}}$, $\mathbf{T_{ct}}$, $\mathbf{T_{cp}}$ are the equivalent matrices for the current joint rotation and translation.
- $\mathbf{T_s}$ is known as the *skinning matrix*, and represents the transformation that needs to be applied to the initial joint pose in order to obtain the current joint pose ($\mathbf{T_c}$).
- $\mathbf{v_i}$ and $\mathbf{v_c}$ are the initial and current position of the vertex $\mathbf{v}$ in model space.
- $\mathbf{w_{jv}}$ is the weight (in the range 0-1) of a joint $\mathbf{j}$ on the vertex $\mathbf{v}$. For each vertex the sum of the weights should add up to 1.

In this listing and throughout the remainder of this paper matrices are homogeneous and are defined row-wise. The $\mathbf{T_{i*}}$ matrices need only be computed once when the character skeleton is created
The current rotation and translation components of each joint are obtained every frame from one or more *animation curves*. Each animation curve is obtained by recording the rotation/translation components of each joint at key frames of the animation. Rotation and translation components are obtained from separate curves. If multiple curves are used the samples obtained from each curve are blended together. If the skeleton bones remain the same length for each frame then only an animation curve for the rotation component is necessary for most joints. Hence an animation that runs at 30 fps and lasts for 2 seconds would have an animation curve for the rotation component that contains 60 entries (assuming there are 60 key frames), with each entry specifying a

quaternion that describes the rotation applied by the joint at that key frame.

3 Collision Detection

In a simulation where one or more objects are moving, the collision detection scheme must be capable of detecting collisions between stationary and moving objects. When checking for collisions between stationary objects it is sufficient to only consider their current position at the time at which the check is made, so the collision check becomes an intersection check. However with moving objects both the current position and the position at the previous animation step must be considered.

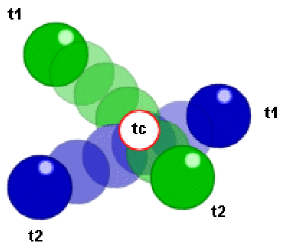

Figure 2: Collision between moving objects.

Figure 2 illustrates that an intersection test between the two spheres at time $\mathbf{t_2}$ is not sufficient since the collision at time $\mathbf{t_c}$ would be missed. Instead a number of tests along the object displacement vectors are performed to ensure a collision is detected if it has occurred between $\mathbf{t_1}$ and $\mathbf{t_2}$. In practice, detection of a contact between two moving spheres can be done using a simpler method [4]. Unfortunately most objects in our simulation consist of complex geometry so intersection tests aren't as simple as they are for spheres. To improve performance a bounding sphere can encapsulate geometry and that sphere is then used for rough collision detection (other bounding volumes could be used instead). However, relying on the sphere alone would produce phantom collisions because the sphere is only an approximation of the real object and thus there is likely to be empty space inside the sphere that is not occupied by the object.
Nebula's collision detection system associates a bounding sphere with every object which may be involved in a collision at one time or the other and provides two methods to check for collision between a pair of moving objects, the quick swept sphere approach [7] or a more accurate (but slower) approach that places an upper bound on the maximum number of intersection tests that will be done and only performs multiple tests along the displacement vector if the object has travelled more than 1/8th its bounding sphere's radius [8].
In a simulation containing *n* different objects (that may collide with each other) a brute force collision detection system will have to test for collision between every pair of objects resulting in an $O(n^2)$ algorithm. If *n* is large and the objects themselves are complex the collision detection will be unacceptably slow. Spatial subdivision is one way to speed up collision detection. For our scenario it was deemed unnecessary to use an explicit spatial subdivision scheme because the soccer simulation is relatively small and the collision system in Nebula already uses some "early out" tests as described next.
As mentioned previously the Nebula collision detection system associates a sphere with each object and keeps track of both the current and previous position of each sphere. Additionally each object belongs to a *collision class,* and the end user is able to specify the types of collision checks to be performed between each pair of classes, or whether collision between any pair of classes should be ignored entirely. Each frame the system computes an axis-aligned bounding box (AABB) that encloses the two spheres (the past and the present). A collision can only occur between two moving objects if the corresponding AABBs overlap along all 3 global axes, existence of such an overlap would indicate that the two objects might have occupied the same space at the same time and further tests would need to be performed to determine whether they actually collided. The use of AABB boxes in this way to speed up collision detection is typically known as *Sweep and Prune* [9]. All objects are kept sorted by the system along the global x-axis using the corresponding AABBs. Collisions between stationary objects can be detected by checking for an intersection between the objects, and collision between moving objects can be handled by checking for collision between so called stationary objects in a number of snapshots of the moving objects taken in the time between the last and the current frame.

3.1 The Character Collide Shape

The deformable mesh of each character in the soccer simulation consists of up to 2000 triangles. Brute-force collision detection is therefore impossible and we use instead the following two methods to improve interactivity: first the visual representation of the character (figure 3 left) is separated from the representation used for intersection tests (the *collision mesh*). The collision mesh is a low-resolution version of the original character mesh and in our case consists of around 280 triangles (figure 3 right).

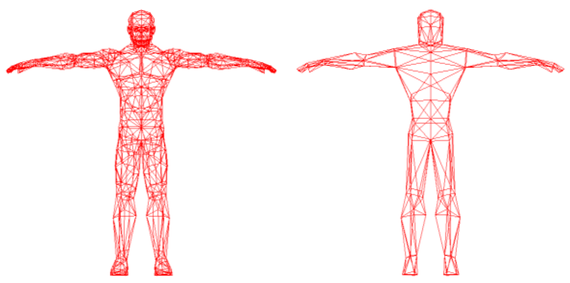

Figure 3: The high (left) and low (right) resolution mesh.

Additionally the collision mesh is subdivided so that only parts of the mesh are tested when necessary and triangle/triangle intersection tests are eliminated entirely.

The Nebula collision detection system uses the term *collide shape* to refer to the data that describes the shape (i.e. geometry) of an object that may be involved in collisions. Nebula is currently only capable of dealing with collide shapes that consist of non-deformable geometry and are described by a triangle mesh and an AABB tree (that is built by OPCODE). Therefore a new collide shape was devised to represent characters.

The new collide shape consists of 3 levels, and is used by the collision detection system for performing intersection tests between characters and other non-deformable objects. Level-1, shown in figure 4, is made up of a collection of bounding volumes (spheres and capsules), each bounding volume contains a sub-group of triangles from level-2 (the collision mesh). The remaining level-3 volume consists of another collection of volumes (spheres and capsules).

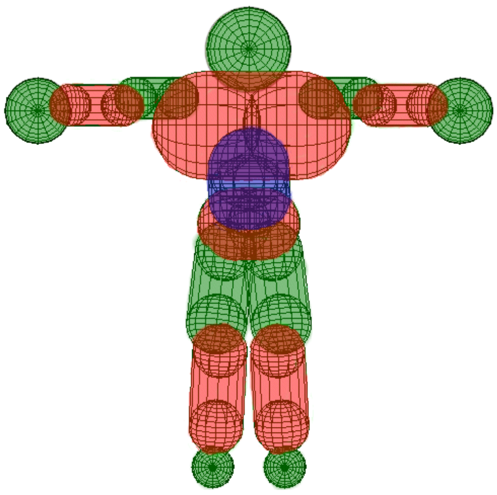

Figure 4: Level -1 bounding volumes for a character.

Level-1 bounding volumes are used to subdivide the collision mesh into groups, such that each group coincides with a body part. This subdivision serves two distinct purposes. First of all by subdividing a character into parts the collision detection method doesn't always have to process every triangle in the collision mesh, since it's rare for all body parts to be in contact with something at the same time. Secondly, higher-level collision information becomes available, so the collision system can tell the user not just whether a character collided with some object, but also which parts of the character collided with that object. This additional information is extremely useful in trying to create a realistic simulation. For instance, in the soccer simulation the soccer player could be made to limp slightly if another player hits him in the foot, or could get a bloody nose as a result of the soccer ball hitting him in the face.

The level-1 bounding volumes are attached to the character skeleton (and move with it) by associating each volume with a skeleton joint and positioning/ orienting the volume relative to that joint. Figure 5 and the listing 2 provide a simple example of computing the position of the bounding volume for the wrist (in this case we assume there are only 3 joints in the whole skeleton).

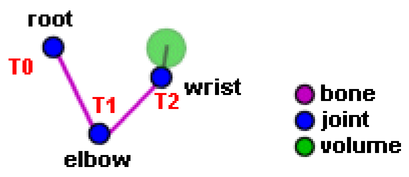

Figure 5: Attaching a volume to a skeleton joint.

The position of the sphere volume in model space is obtained by flattening the joint hierarchy at the joint to which the volume is attached, just as it is done during vertex skinning. Position and orientation is computed in a similar way for capsule volumes, the only difference being that two points are transformed instead of one.

$$\mathbf{T}_{\text{flat}} = \mathbf{T}_2\ \mathbf{T}_1\ \mathbf{T}_0$$
$$\mathbf{v}_{\text{sphere}} = \mathbf{u}_{\text{sphere}}^{\mathbf{T}}\ \mathbf{T}_{\text{flat}}$$

Listing 2: Computation of the position of a sphere in model space.

- $\mathbf{T}_{\text{flat}}$ is the result of flattening the joint hierarchy at the wrist joint.
- $\mathbf{u}_{\text{sphere}}$ and $\mathbf{v}_{\text{sphere}}$ are the coordinates of the centre of the sphere volume in the local wrist joint coordinate system and the model coordinate system, respectively.

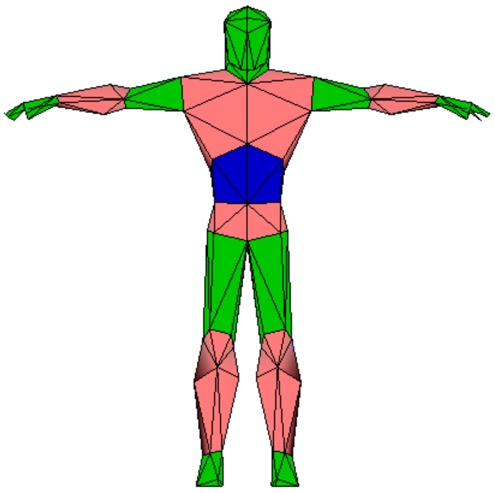

Figure 6: Level-2 collision mesh and triangle groups.

The collision mesh for a character is classified as level-2 and is shown in figure 6. The mesh can be used to obtain more detailed information about which component of a character collided with an object. This is achieved by tagging each triangle in the collision mesh with a group identifier that allows for further subdivision of the collision mesh. For example the triangles belonging to the volume that bounds the left forearm can be separated into two groups, one group would consist of the triangles on the outer side of the forearm, the other would consist of the ones on the inner side. In the extreme each triangle can be identified uniquely. The extra information can be used to provide visual feedback to the user whenever the character experiences an impact by adding a decal to the character's texture at the point of impact.

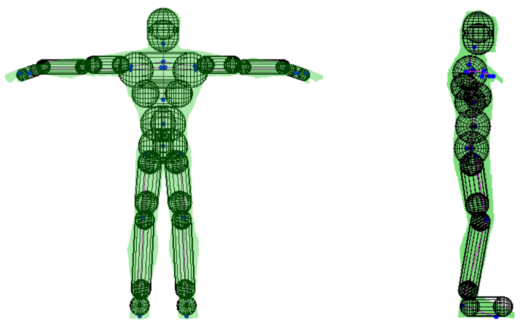
Figure 7: Level-3 volumes (front & right view).

The final level-3, shown in figure 7, is made up of a collection of volumes which are bound to the character skeleton just like level-1 volumes. However, each level-1 volume can consist of multiple level-3 volumes. Unlike the level-1 volumes level-3 volumes do not contain triangles, and exist solely for the purpose of computing an estimate of the penetration depth between a character and some other object whenever a collision occurs, which is used for determining the collision response [8]. Ideally the level-3 volumes should approximate the collision mesh as closely as possible.
During the search for an intersection between a character and some object, the bounding volumes provide a spatial subdivision that allows the fast elimination of whole groups of triangles at once if the bounding volumes enclosing these groups don't overlap with the object. In order to provide an significant advantage over the brute force approach to finding intersections the bounding volumes need to satisfy a number of properties. The bounding volumes should encapsulate triangles as tightly as possible in order to minimize the number of "false positives" which leads to checking all triangles in the volume.
The test to check whether two volumes overlap must be quick and the transformation of volumes as the character is animated must be computationally efficient. After some consideration we chose spheres and capsules as bounding volumes. The sphere has a quick overlap test and only the sphere centre needs to be transformed during animation. Unfortunately spheres usually do not provide a very tight fit. The capsule can be described by a line segment and a radius, and has a pretty quick overlap test (figure 8). Only the two endpoints of the line segments need to be transformed during animation. Capsules provide a better fit than spheres in many cases. Furthermore capsules and spheres allow for quick computation of penetration depth when two volumes or a volume and a triangle intersect (the depth value is necessary for providing proper collision response).

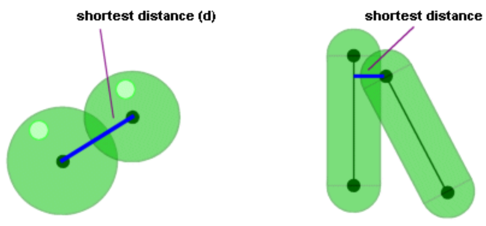

Figure 8: Overlap of spheres and capsules.

Axis aligned bounding boxes (AABB) [10] and oriented bounding boxes (OBB) [11] were also taken into consideration. However, when the AABB needs to be transformed during animation there are two options. One is to compute a new AABB by finding the extents of the transformed geometry, but doing so is computationally expensive. Alternatively the previous AABB is transformed and a new one is computed based on the transformed vertices of the old AABB, but this may produce an AABB that is twice as large as the original. Oriented bounding boxes don't suffer from this "growing" problem, but they take up more memory and are more expensive to transform [12].

```
IntersectCharacters( characterA, characterB )
{
   Transform level-1 and level-3 volumes to world space
   For each level-1 volume volA-1 from characterA
      For each level-1 volume volB-1 from characterB
         IntersectLevel1( volA-1, volB-1 )
}

IntersectLevel1( volA-1, volB-1 )
{
   if volA-1 and volB-1 overlap
      transform all triangles in volB-1 to world space
      for each Level-3 volume volA-3 in volA-1
         for each triangle tri in volB-1
            check for intersection between volA-3 and tri
            if intersection exists store the contact point,
               contact normal and depth
      if contact points are found then
         combine all contacts into a single contact
      else
         transform all triangles in volA-1 to world space
         for each level-3 volume volB-3 in volB-1
            for each triangle tri in volA-1
               check for intersection between volB-3 and tri
               if intersection exists store the contact point,
                  contact normal and depth
          if contact points are found then
            combine all contacts into a single contact
}
```

Listing 3: Finding an intersection between characters.

3.2 The Character Intersection

Once the collide shape for a character has been defined the method in Listing 3 is used to find intersections with another character (intersection tests are done in world space).

The method for finding the intersections between a character and a non-deformable object is slightly different and shown in listing 4. Recall that non-deformable objects are handled by OPCODE, which builds an AABB tree from the mesh that is then used to quickly obtain a list of potentially colliding triangles.

```
IntersectCharacterOpcodeShape( character, opcShape )
{
  transform all level-1 and level-3 volumes in character
        to world space
  for each level-1 volume volA-1 from character
    obtain a list of triangles from opcShape that
        overlap with volA-1
    transform touched triangles to world space (if any)
    for each level-3 volume volA-3 in volA-1
      for each triangle tri of the transformed triangles
        check for intersection between volA-3 and tri
        if intersection exists store the contact point,
          contact normal and depth
    if contact points were found then
        combine all contacts into a single contact
}
```

Listing 4: Detect character/OPCODE shape intersection.

During the construction of the collide shape for the soccer player character we found that many level-1 volumes contained only one level-3 volume, such was the case for legs and arms. The methods above can be improved by checking for this case and avoiding the level-1 overlap test altogether. Furthermore for character intersection it might be beneficial to buffer the transformed triangles. This means that if a character is involved in collisions with multiple objects the relevant parts of the collision mesh only need to be skinned once after the skeleton is repositioned for each frame.

4 Results and Conclusion

We have introduced a new collision detection algorithm for skeletally animated polygonal models. The technique described in this paper has been implemented as part of a soccer simulation. Initial results are encouraging and show that the collision detection techniques discussed are effective in practice and can be integrated with a physics engine to provide physically realistic responses to collisions. Many more improvements are possible and we are particularly interested in exploiting temporal coherence. It might also be worth considering other collision detection algorithms which compute the penetration depth between complex objects and using one of them instead of the approximation provided by level-3 volumes as described in this paper. An example is a novel variant of GJK presented in [13].

Work on the integration of the ODE physics engine with Nebula's skeletal animation system for the purpose of physically-based simulations is currently still in progress. A demo and source can be obtained from www.steelronin.com.

5 References

[1] Watt, A. and Policarpo, F., *3D Games: Real-time Rendering and Software Technology*, Addison-Wesley (2001).

[2] Hubbard, P. M., "Collision detection for interactive graphics applications", *IEEE Transactions on Visualization and Computer Graphics*, 1(3), pp 218-230, September (1995).

[3] Lin, M. C., "Efficient collision detection for animation and robotics", PhD Thesis, University of California, Berkeley (1993).

[4] Terdima, P., "OPCODE home page", http://www.codercorner.com/Opcode.htm, visited on 20/08/2003.

[5] Smith, R. et al., "Open Dynamics Engine home page", http://opende.sourceforge.net, visited on 20/08/2003.

[6] The Nebula Device Wiki., "home page", http://nebuladevice.sourceforge.net, visited on 20/08/2003.

[7] Gomez, M., "Simple intersection tests for games", Gamasutra.com, October 18, (1999).

[8] Macagon, V., "Collision detection and response of skeletally animated models", *FoS Summer Scholarship Project Report*, University of Auckland, March (2003).

[9] Cohen, J.D., Lin, M.C., Manocha, D., Ponamgi, M.K., "I-COLLIDE: An interactive and exact collision detection system for large-scale environments", *Proceedings of ACM Interactive 3D Graphics*, pp 189-196 (1995).

[10] Lander, J., "When two hearts collide: axis aligned bounding boxes", Gamasutra.com, February 3, (2000).

[11] Bobic, N., "Advanced Collision Detection Techniques", Gamasutra.com, March 30, (2000).

[12] van den Bergen, G., "Efficient collision detection of complex deformable models using AABB trees", *Journal of Graphics Tools*, 2(4), pp 1-14 (1997).

[13] van den Bergen, G., "Proximity queries and penetration depth computation on 3d game objects", Proceedings of the Game Developers Conference 2001, http://www.gdconf.com/archives/2001/vdbergen/vdbergen.doc, visited on 20/08/2003.