COLLISION DETECTION AND RESPONSE OF SKELETALLY ANIMATED MODELS

VADIM MACAGON

MARCH 11, 2003

ABSTRACT

This project set out to explore possible ways of handling the collision detection and response of skinned and skeletally animated models in an interactive and physically realistic 3d environment. A large part of the project consisted of creating a 3d soccer simulation to demonstrate the methods that were devised to handle the collision of skeletally animated models with each other and objects around them. 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 was also explored. It is hoped the techniques devised in this project will be of use to anyone interested in creating 3d simulations in the fields of computer graphics, sports science, or computer games.

1. INTRODUCTION

The aim of this project was to come up with a way to detect collisions of skeletally animated polygonal models (also referred to as characters) in a 3d environment and to provide semi-physically realistic responses upon impact of these models with the environment. Techniques presented in this report are geared towards 3d simulations in the areas of sport science, and computer games.

Collision detection has been researched for many years in computer graphics, and is of extreme importance in visual simulations of 3d environments where various objects can interact with each other. The exact method used to detect collisions will depend on the way 3d objects are described, the complexity of the objects and the results that need to be obtained from the collision detection system.

Presently there are a number of algorithms and collision detection libraries that provide fast collision detection in a 3d environment, however they typically expect the 3d objects to consist of static geometry and tend to see the object as a polygon soup. When a human looks at a 3d environment he sees objects, not just an arbitrary collection of polygons. As mentioned previously, collision detection algorithms tend to be at least to some extent application specific, and in this particular case some of the objects will consist of deformable meshes that represent humanoid models.

Existing collision detection libraries deal with polygons (mostly triangles) and primitive shapes such as spheres, capsules and boxes. When collision between objects is detected they tend to produce a list of pairs of polygons that intersect, and perhaps even the intersection points. These results are not sufficient if we want to obtain a higher-level description of the interaction between objects, such a higher-level description could be quite useful in sports science and also computer games.

A simple scenario will be used throughout this report to illustrate various points and to help describe the reasoning behind the techniques presented. Imagine a soccer game, most of the time each player is in contact with the ground, the soccer ball, or other players. In order to simulate this scenario we need to be able to detect the various collisions that occur – in order to make the soccer ball fly with each kick and to prevent players from falling through the ground. This alone however will not produce a very realistic simulation.

We'd like to know which limbs, and perhaps even which parts of the limbs, were involved in an impact so that we can better model the response of the players to the various impacts they experience. In order to obtain higher-level collision information the polygon soup 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 one would expect the pose to change in response to impacts. This is where collision response techniques come into play. 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 they 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.

For this project the Open Dynamics Engine[1] (ODE) was chosen 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. Other objects in the environment, like the soccer ball will also be represented in ODE as rigid bodies. While ODE has its own collision detection facilities a decision was made not to use them, since they alone would not be sufficient for the project. The output from the collision detection system will be fed to ODE, which will in turn try to ensure all constraints are satisfied and hence produce a visually realistic response to collisions in the soccer simulation.

The soccer simulation has been implemented using The Nebula Device[2], which is a free modular framework for building 3d visualization/game engines. Nebula already contains a collision detection system that deals with static geometry by making use of the Optimised Collision Detection[3] (OPCODE) library. Nebula also provides a character animation system, however the collision system is currently incapable of handling animated characters, so hopefully this project will be a starting point for filling in that gap.

[1] ODE can be obtained from http://opende.sourceforge.net
[2] Nebula can be obtained from http://nebuladevice.sf.net
[3] OPCODE can be obtained from http://www.codercorner.com/Opcode.htm

2. SKELETAL ANIMATION

The player models consist 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.

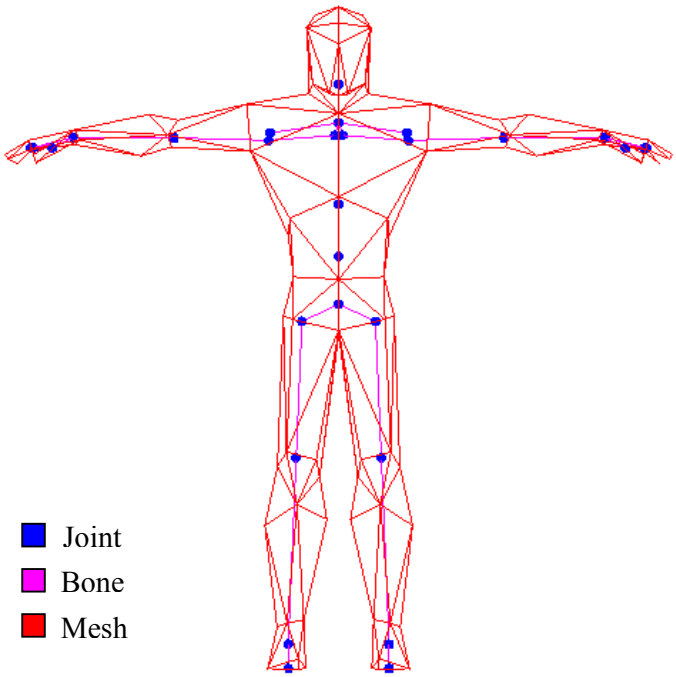

Figure 1 – Low Resolution Character Model

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[4] 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.

Throughout the remainder of this report, matrix multiplication is performed in left to right order. In this particular case each matrix is a 4x4 matrix with the translation component stored in the fourth row of the matrix.

[4] The limit of 4 joints is imposed by the implementation of skinning in Nebula.

```
for each joint j
        let Ti = Tir x Tit
        if j has a parent
                let Ti = Ti x Tip
        let Tc = Tcr x Tct
        if j has a parent
                let Tc = Tc x Tcp
        let Ts = Ti^-1 x Tc

for each vertex v
        let vc = (0,0,0)
        for each joint j that v is weighted by
                let vc = vc + (vi x Ts) x wjv
```

Listing 1 - Skinning

- $\mathbf{T_{ir}}$ is the matrix obtained by converting the initial joint rotation component into a rotation matrix.
- $\mathbf{T_{it}}$ is the matrix obtained by converting the initial joint translation component into a translation matrix.
- The final $\mathbf{T_i}$ is known as the pose matrix, and the position component of $\mathbf{T_i}$ specifies the initial position of the joint in model space. $\mathbf{T_i^{-1}}$ is the inverse of $\mathbf{T_i}$.
- $\mathbf{T_{ip}}$ is the pose matrix of the parent joint.
- The $\mathbf{T_{i*}}$ matrices need only be computed once when the character skeleton is created.
- $\mathbf{T_c}$, $\mathbf{T_{cr}}$, $\mathbf{T_{ct}}$, $\mathbf{T_{cp}}$ are obtained in a similar way from the current joint rotation and translation components.
- $\mathbf{T_s}$ 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 ($\mathbf{T_c}$).
- $\mathbf{v_i}$ is a 3-vector that contains the initial position of the vertex in model space.
- $\mathbf{w_{jv}}$ is the weight (in the range 0-1) of a joint **j** on the vertex **v**.
- The final $\mathbf{v_c}$ will contain the current position of the vertex in model space.
- For each vertex the sum of the weights should add up to 1.

The current rotation and translation components of each joint are obtained every frame from one or more *animation curves*, if multiple curves are used the samples obtained from each curve are blended together. Rotation and translation components are obtained from separate curves. Each animation curve can be obtained by recording the rotation/translation components of each joint at key frames of the animation. If the skeleton bones remain the same length 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.

Further details on how character animation is performed can be obtained by studying the source code for the character animation system in Nebula, the information provided so far should be sufficient for understanding the rest of this report.

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 collision between stationary objects it is sufficient to only consider their current position at the time at which the check is made and the collision check essentially just becomes an intersection check. However with moving objects checking for collision requires taking into consideration not only their current position at the time of the check, but also their previous position at the time of the last check.

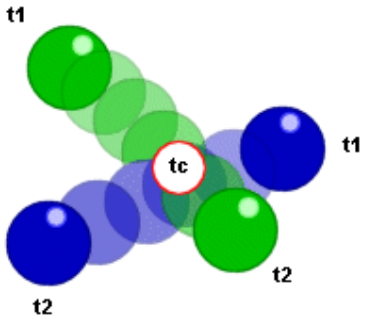

Figure 2 – Collision between moving objects

Figure 2 illustrates the situation that may occur when checking for collision between two moving objects. It would not suffice to check for intersection between the two spheres at $\mathbf{t_2}$, because no intersection would be detected at that time and 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 [1]. 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 [1] or a more accurate (but slower) approach in Listing 2. The accurate approach 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 $1/8^{th}$ it's bounding sphere's radius.

```
let num = max(v0len ÷ (rad0 × 0.125), v1len ÷ (rad1 × 0.125))
let maxChecks = 16
if num is zero then
   Let num = 1
else if num > maxChecks
   num = maxChecks
let d0 = v0 ÷ num
let d1 = v1 ÷ num

repeat num times
   p0 = p0 + d0
   p1 = p1 + d1
   check for intersection between object 0 at position p0 and
   object 1 at position p1, if intersection found break out of the loop
```

Listing 2 – Detecting Collisions Between a Pair of Moving Objects

- **p0** is the position of **object 0** last frame
- **p1** is the position of **object 1** last frame
- **v0** is the displacement vector for **object 0** (i.e. current position – last position)
- **v1** is the displacement vector for **object 1**
- **v0len** is the length of **v0**
- **v1len** is the length of **v1**
- **rad0** is the radius of bounding sphere associated with **object 0**
- **rad1** is the radius of bounding sphere associated with **object 1**

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, hence it would be $\mathbf{O(n^2)}$. If **n** is large and the objects themselves are quite complex the collision detection may take up too much time each frame. Fortunately there are ways to reduce the number of pairs that need to be tested each frame and also to avoid complex collision tests when simpler tests can be used to indicate early on whether a collision could've possibly occurred.

Spatial subdivision is one way to speed up collision detection. By subdividing the world within which the simulation occurs into smaller spaces it is possible to eliminate tests of pairs of objects that reside in non-adjacent sub-spaces. There are numerous spatial subdivision schemes one could choose from, depending on the type of simulation. For this particular project 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.

Previously it was mentioned that the Nebula collision detection system associates a sphere with each object, the system also keeps track of both the current position of the sphere its previous position. 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 could 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* [7]. All objects are kept sorted by the system along the global x-axis using the corresponding AABBs, and the system is typically requested to check for collisions between all objects it keeps track of every frame. The exact algorithm used by Nebula is summarized in Listing 3.

```
for each object obj
   for each AABB_x that overlaps with AABB_obj along the x-axis
      if user requested collisions between class_x and class_obj to be ignored
         move onto the next object
      else
         if AABB_x overlaps with AABB_obj along the y & z axes.
            perform further collision tests between x and obj (Listing 2)
```

Listing 3 – Checking for collisions.

- $\mathbf{AABB_x}$ is the AABB that corresponds to object **x**.
- $\mathbf{class_x}$ is the *collision class* that object **x** belongs to.

Essentially collision 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. If the check for intersection between characters used a simplistic method that simply tested each triangle in one mesh against all the triangles in the other mesh, the method would have to go through approximately 4 million pairs. Despite today's blazingly fast CPUs, processing two soccer teams each frame of the simulation while also handling user input, rendering, logic, and perhaps even sound while maintaining a descent frame-rate is going to be impossible using this method. Therefore to achieve a reasonable frame-rate the number of pairs that are tested each frame needs to be reduced.

To cut down on the number of triangles that need to be processed two methods where used. First of all the visual representation of the character (Figure 3) was 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 4). Secondly the collision mesh was subdivided so that only parts of the mesh would be tested when necessary and triangle/triangle intersection tests where eliminated entirely.

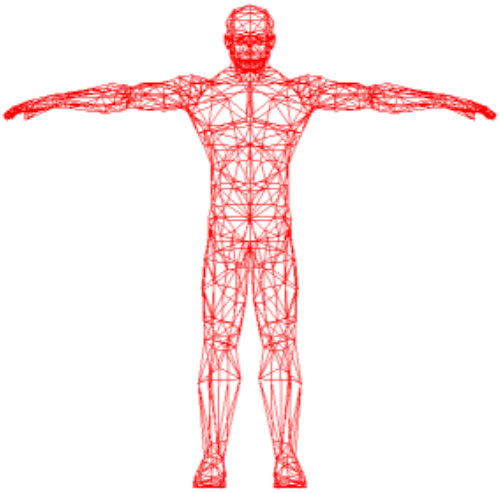

Figure 3 – High Resolution Mesh

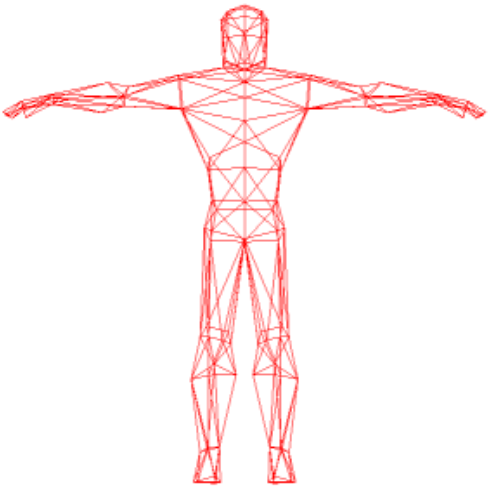

Figure 4 – Low Resolution Mesh

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. As mentioned previously Nebula is currently only capable of dealing with collide shapes that consist of non-deformable geometry, these shapes consist of a triangle soup 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 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 consists of another collection of volumes (spheres and capsules).

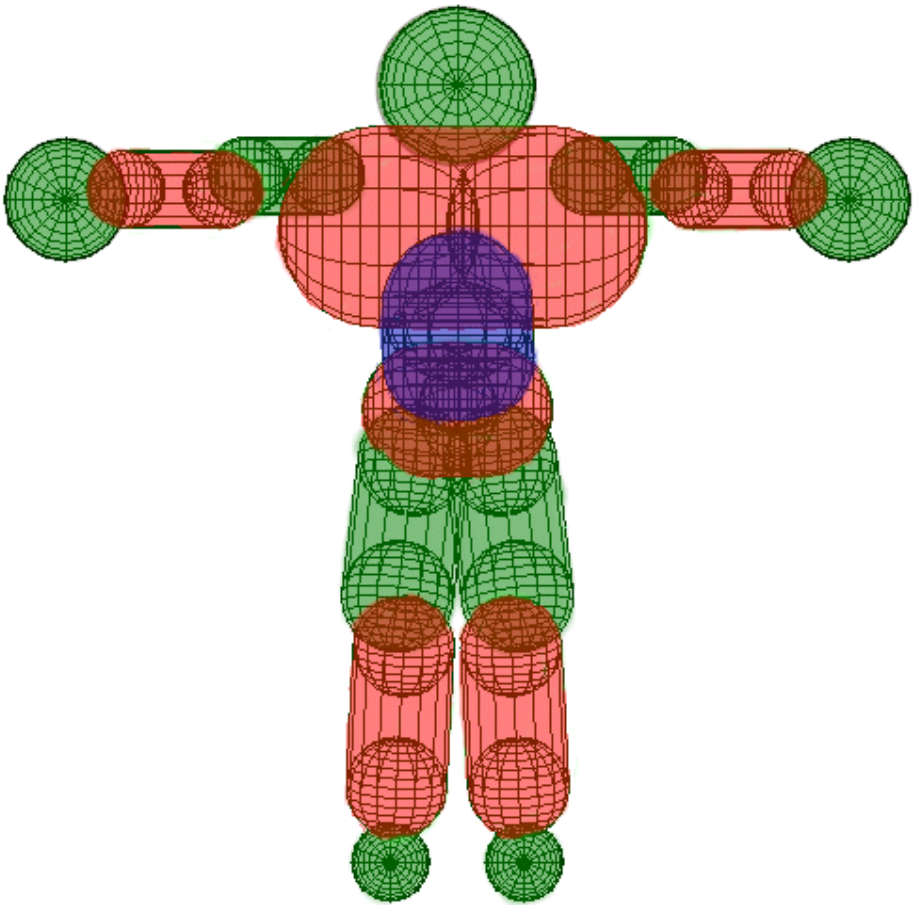

Figure 5 – Level 1 bounding volumes for a character.

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.

These bounding volumes are attached to the character skeleton so they move with the character. This is achieved by associating each bounding volume with a skeleton joint and positioning/orienting the volume relative to that joint. Figure 6 provides 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). 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.

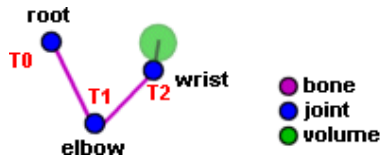

Figure 6 – Attaching a volume to a skeleton joint

$$\mathbf{T}_{flat} = \mathbf{T2} \times \mathbf{T1} \times \mathbf{T0}$$
$$\mathbf{v}_{sphere} = \mathbf{u}_{sphere} \times \mathbf{T}_{flat}$$

Listing 4 – Computing volume position in model space.

- $\mathbf{T}_{flat}$ is the result of flattening the joint hierarchy at the wrist joint.
- $\mathbf{u}_{sphere}$ is a 3-vector containing the coordinates of the centre of the sphere volume relative to the wrist joint.
- $\mathbf{v}_{sphere}$ is the 3-vector containing the coordinates of the centre of the sphere volume in model space.

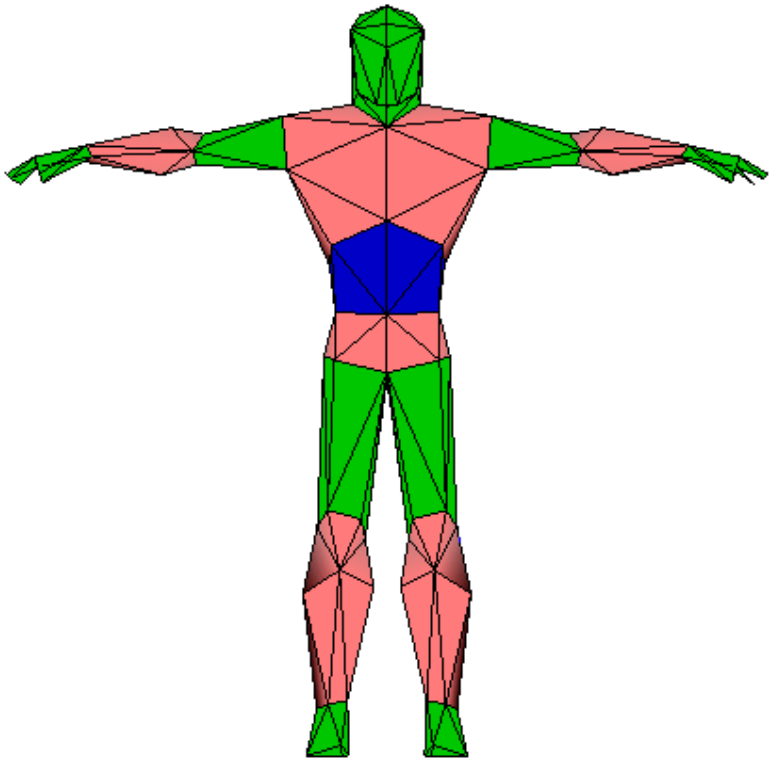

Figure 7 – Level 2 Collision Mesh and Triangle Groups

The collision mesh for a character is classified as Level-2, and can be used to obtain finer grained information about which areas of the character came into contact with an object. This can be 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 could 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. It could be taken even further to uniquely identify each triangle. All this 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.

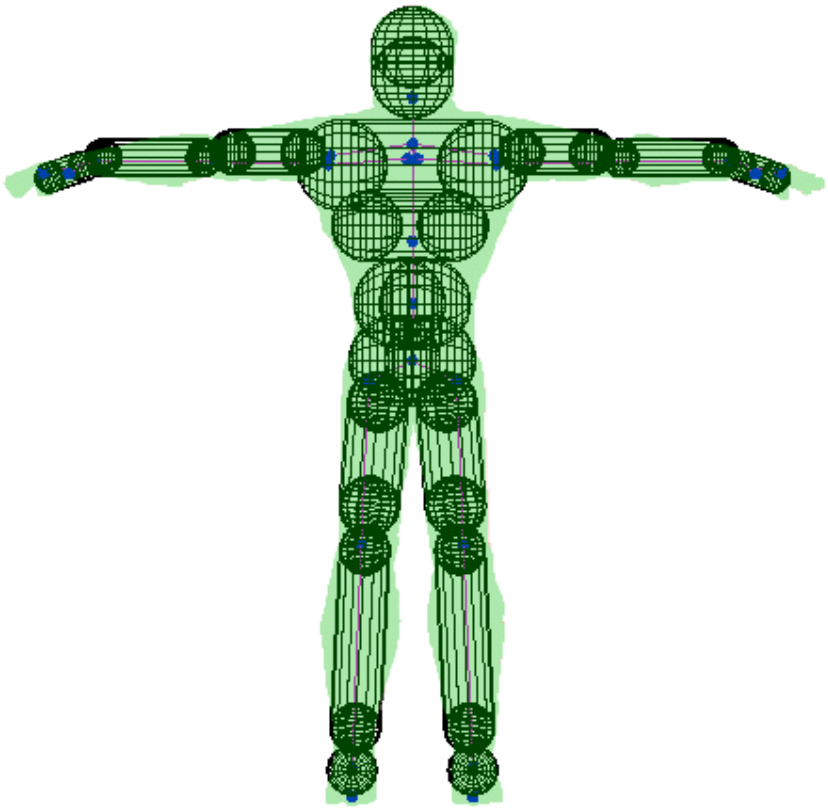

Figure 8 – Level 3 volumes (Front View)

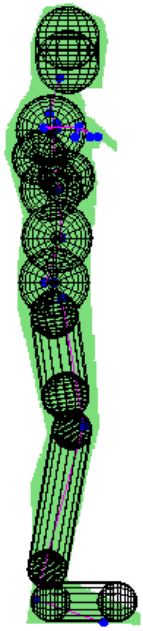

Figure 9 – Level 3 volumes (Right View)

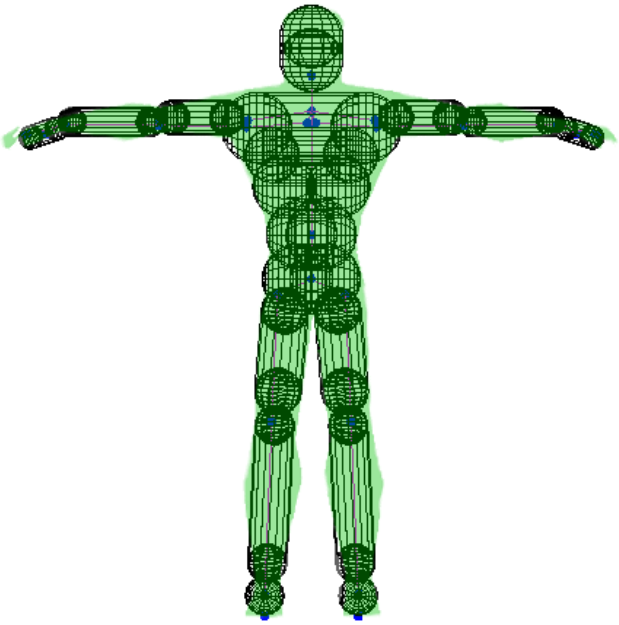

Figure 10 – Level 3 volumes (Back View)

The final level is made up of a collection of volumes, these volumes are bound to the character skeleton just like Level 1 volumes, and there may be multiple Level 3 volumes for each Level 1 volume. Unlike the Level 1 volumes Level 3 volumes don't contain any 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, this is used for collision response and is explained in section 4. Ideally the Level 3 volumes should approximate the collision mesh as closely as possible, but as can be seen in Figures 8-10 that's hard to do without using lots of small volumes.

As already mentioned, during the search for an intersection between a character and some object, the bounding volumes provide spatial subdivision that allows for 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 any significant advantage over the brute force approach to finding intersections the bounding volumes need to satisfy a number of properties. The bounding volume should encapsulate triangles as tightly as possible in order to minimize the number of "false positives" which lead to checking all triangles in the volume. The test to check whether a volume overlaps with another should be as quick as possible. And finally it should be computationally cheap to transform each volume as the character is animated.

After some consideration the sphere and capsule were chosen. 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. Only the two endpoints of the line segments need to be transformed during animation. Capsules provide a better fit than spheres in some 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).

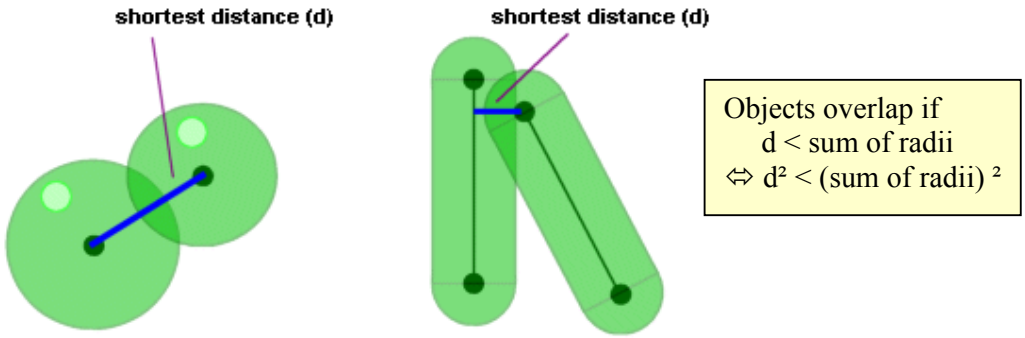

Figure 11 – Overlap of spheres and capsules.

Axis aligned bounding boxes (AABB) [5] and oriented bounding boxes (OBB) [6] 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 [8].

3.2. CHARACTER INTERSECTION

Once the collide shape for a character has been defined the method in Listing 5 is used to find intersections with another character (intersection tests are done in world space).

```
intersected <= IntersectCharacters( characterA, characterB )
{
   Transform all Level-1 and Level-3 volumes to world space
   For each Level-1 volume volA1 from characterA
      For each Level-1 volume voB1 from characterB
         If volA1 and volB1 intersect store information about the contact
         i.e. IntersectLevel1( volA1, volB1 )
}

intersected, contact <= IntersectLevel1( volA1, volB1 )
{
   if volA1 and volB1 overlap
      transform all triangles in volB1 to world space (i.e. skinning)
      for each Level-3 volume volA3 in volA1
         for each triangle tri in volB1
            check for intersection between volA3 and tri
            if intersection exists store the contact point, contact normal and depth
               intersected = true
      if contact points were found then
         combine all contacts into a single contact
      else
         transform all triangles in volA1 to world space
         for each Level-3 volume volB3 in volB1
            for each triangle tri in volA1
               check for intersection between volB3 and tri
               if intersection exists store the contact point, contact normal and depth
                  intersected = true
         if contact points were found then
            combine all contacts into a single contact
}
```

Listing 5 – Finding an intersection between characters.

The method for finding the intersections between a character and a non-deformable object is slightly different. 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.

```
intersected <= IntersectCharacterOpcodeShape( character, opcodeShape )
{
   transform all Level-1 and Level-3 volumes in character to world space
   for each Level-1 volume volA1 from character
      obtain a list of triangles from opcodeShape that overlap with volA1
      transform touched triangles to world space (if any)
      for each Level-3 volume volA3 in volA1
         for each triangle tri in the transformed triangles list
            check for intersection between volA3 and tri
            if intersection exists store the contact point, contact normal and depth
               intersected = true
      if contact points were found then
            combine all contacts into a single contact
}
```

Listing 6 – Finding an intersection between a character and an OPCODE shape.

During the building of the collide shape for the soccer player character it became obvious that a lot of the Level-1 volumes only contained one Level-3 volume, such was the case for legs, and arms. Therefore the methods above could be improved by checking for this case and avoiding the Level-1 overlap test altogether. Furthermore for character intersection it may 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. Collision Response

In order to provide the proper response to collision there are primarily three things the collision detection system should be capable of providing for each collision, a contact position, a contact normal and a penetration depth.

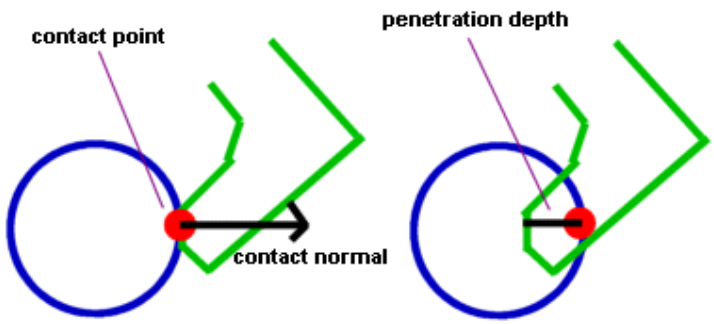

Figure 12 – Foot hitting ball.

Once a collision between two or more objects has been detected something must be done to bring the intersecting objects into a touching contact (so they just touch, but don't intersect each other). This usually involves moving one of the objects some distance back, but this is not always simple because another object may by now occupy that space so some sort of shuffling may be necessary. Alternatively objects could be allowed to move only when their final position has been verified to be valid.

To resolve an intersection between two objects two approaches can be used. The first approach relies on checking for intersection several times along the lines of movement (displacement vectors), the objects are then moved back to their positions at the time when the first intersection has occurred. A drawback of this approach is that intersection must be checked several times, and a limit needs to be placed on the maximum number of checks that can be done. The more checks are made the more precise the estimate of the time of the first intersection, but clearly more checks will also take up more CPU time.

An alternative, faster, approach can be used for dealing with collisions involving characters. Recall each character has a collection of Level-3 volumes, consisting of capsules and spheres. While multiple intersection tests along the lines of movement may still be necessary in order to move the objects back to the approximate positions at which the first intersection test has occurred not as many tests need to be done because once intersection has been detected the Level-3 volumes can be used to obtain an estimate of penetration depth between two objects. Penetration depth is the depth to which two objects interpenetrate each other, if penetration depth is zero the objects are just touching.

Simply bringing colliding objects into touching contact is not sufficient on its own to create a realistic simulation. When two moving objects collide in the real world they won't both just come to a complete stop, physics dictates that due to conservation of momentum either one object or both will deflect off each other and continue to travel in some direction. Therefore objects must be given physical properties and a physics engine needs to be used in order to correctly compute new velocities for objects after a collision.

Writing a general physics engine from the ground up is not a trivial task. Therefore the Open Dynamics Engine was chosen to assist in generating the appropriate collision response for the soccer simulation. ODE primarily deals with rigid bodies and joints. ODE joints have a different meaning to the skeleton joints mentioned in previous sections. The ODE joints are attached to rigid bodies and embody one or more constraints on the position/orientation of bodies relative to each other. Different types of ODE joints are available, and each type provides its own set of constraints. The exact shape of a rigid body is not important [2].

ODE resolves interpenetration between rigid bodies using *contact joints* [2], since the exact shape of each body is not strictly necessary for computation of the physical properties of the rigid body it is not kept track of by ODE. However the contact joints need to be provided with information about intersections by the collision detection system in order to properly resolve collisions, specifically the contact position, contact normal and penetration depth.

One of the goals of the soccer simulation was not only to allow the ball to bounce off the goal post, or the players, but also to have players realistically fall over if they are knocked down or are tripped up (usually known as *rag-doll* physics). This meant ODE would have to handle whole characters that consist of a number of connected bodies and not just simple objects like the soccer ball, which can be represented using a single rigid body. A method had to be devised for integrating Nebula's skeletal character animation system with ODE.

4.1. Deducing Rigid Body Transforms from Character Skeleton

Figure 13 demonstrates one possible configuration of ODE rigid bodies and ODE joints in order to build an articulated soccer player character. Three types of ODE joints are used in this case, hinge joint, ball joint and angular motor joint (amotor), but other types may also be suitable depending on the character being articulated. The constraints placed on rigid bodies by each joint are described in [2] (accompanied by nice images).

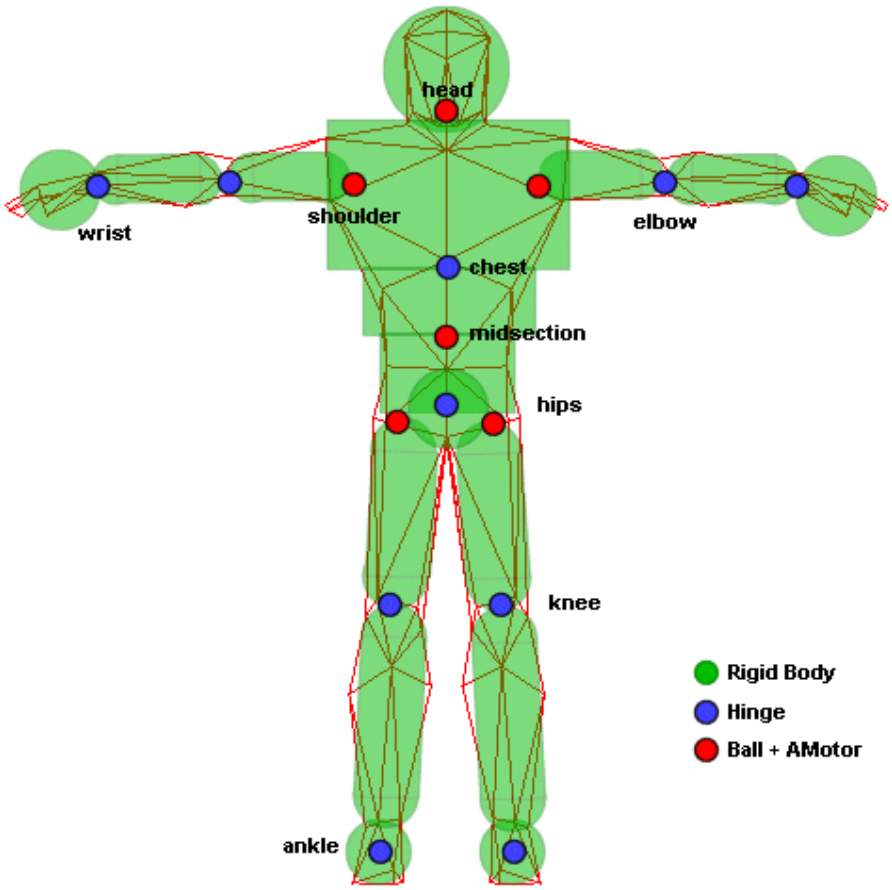

Figure 13 – Building an articulated character using ODE rigid bodies

At any one time a character's pose is determined either by the Nebula character animation system or by ODE. Whenever control switches from the animation system to ODE the rigid bodies that correspond to various body parts of the character must be

re-positioned to match the current pose of the character, furthermore the ODE joints that connect the rigid bodies must be re-attached and the angular limits the joints place on orientation of bodies relative to each other need to be recomputed. Therefore rigid bodies need to be attached to the character skeleton just like Level-1 and Level-3 volumes, the rigid body transforms can then be obtained from the skeleton using the method in Listing 7.

```
for each rigid body b
    obtain the Nebula joint j that b is attached to
    T_b = T_brj x T_flatj x T_w
```

Listing 7 – Compute rigid body transforms from the character skeleton.

- $\mathbf{T}_{brj}$ is the transform of the rigid body **b** relative to the Nebula joint **j**.
- $\mathbf{T}_{flatj}$ is the flattened joint hierarchy at **j**.
- $\mathbf{T}_{w}$ is the character model to world transform.

Since all ODE joints have a corresponding Nebula joint, obtaining the anchor position ($\mathbf{v}_{anchor}$) of the ODE joint from the character skeleton is trivial (Listing 8).

```
for each ODE joint odej
    obtain the corresponding Nebula joint nebj
    let T_flatnebj = matrix formed from flattening the joint hierarchy at nebj
    v_p = position component of T_flatnebj
    v_anchor = v_p x T_w
```

Listing 8 – Compute ODE joint anchor.

- $\mathbf{T}_{w}$ is the character model to world transform
- $\mathbf{v}_{p}$ and $\mathbf{v}_{anchor}$ are 3-vectors.

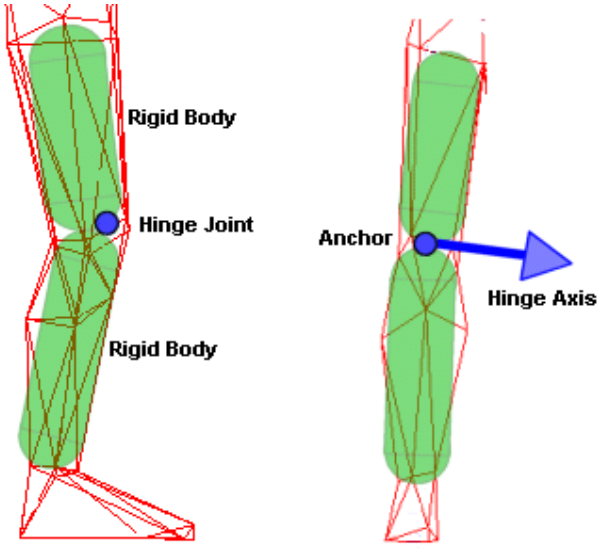

Figure 14 – Articulated Right Leg

Taking into consideration the knee as an example of a hinge joint, limits need to be set on how far the bodies are allowed to rotate relative to each other in order to prevent the thigh and the shin getting into humanly impossible positions. Whenever the hinge anchor or axis is set the current relative orientation of the bodies is stored, and from there on ODE can determine the relative angle of rotation about the hinge axis by taking into account the initial orientation and the current orientation [3]. Each time the switch is made from pre-canned animation to ODE the hinge anchor and axis are going to be updated. Hence the angular limits must be recomputed because the current relative orientation of the bodies is considered by ODE to be the initial orientation, however this is not really the case and so the limits must be adjusted to match ODE's perception.

Once the relative angle of rotation (θ) is known the angular limits that need to be given to ODE can be obtained using the formula below.

```
odeLimit = initialLimit - θ
while odeLimit < -180.0f
   odeLimit = odeLimit + 360.0f
while odeLimit > 180.0f
   odeLimit = odeLimit - 360.0f
```

Listing 9 – Obtaining an adjusted angular limit.

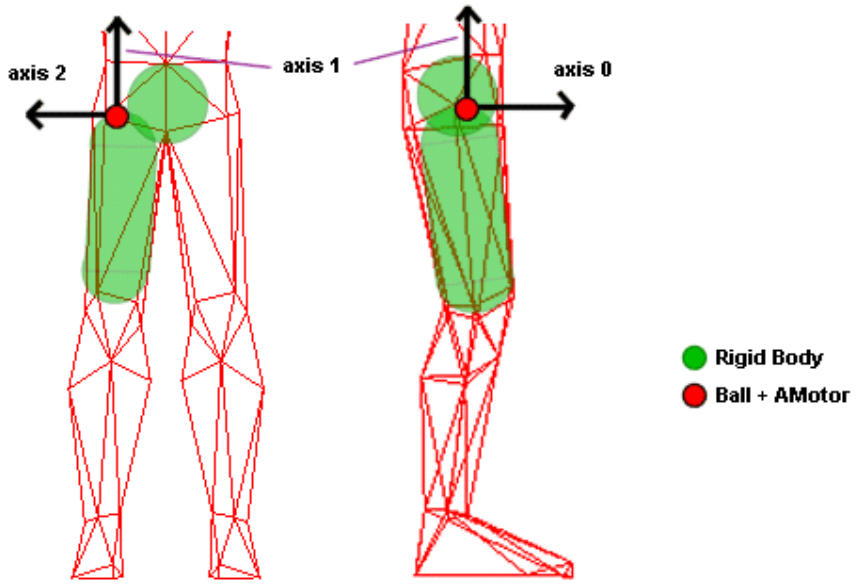

Figure 13 – Right Hip Joints

The ODE ball and amotor joints are used together for the hips, shoulders and head. The ball joint constrains the orientations of the bodies relative to each other, while the amotor joint is used to set angular limits on their rotation relative to each other [2]. For our soccer player an amotor can be used in euler mode and initial limits can be set for rotation about the 3 motor axes. Similar to the hinge joints the rotational limits on

each axis need to be updated whenever the amotor is reset (when switching from pre-canned animation to physical simulation), once the relative angle of rotation about each axis has been determined the new limits can be deduced using the method in Listing 9. The relative angle of rotation about each motor axis can be obtained similarly to the way it is obtained in [4].

The alternative to directly positioning bodies and re-adjusting joints and angular limits all the time is to compute the linear and angular velocities that should be applied to the rigid bodies in order to bring them into alignment with the character skeleton. However this would require a simulation step just to bring the bodies into alignment, and any other forces/torques will need to be applied in the following simulation step.

4.2. DEDUCING POSE OF CHARACTER SKELETON FROM RIGID BODIES

After ODE manipulates the rigid bodies that make up the physical representation of the character, the character skeleton (i.e. Nebula joints) must be updated to reflect the new position and orientation of the various body parts. Rigid body position and orientation can only be obtained from ODE in world space, hence the model to world transform of each body must be decomposed in order to obtain the rotation and translation components for the skeleton joints. Furthermore the number of ODE joints is likely to be less than the number of Nebula joints, due the lack of a one to one relationship the assumption must be made that any Nebula joint which doesn't have a corresponding ODE joint remains stationary relative to its parent. The method in Listing 10 was devised to handle the task, but at present assumes the root joint has a rigid body attached to it.

For each Nebula joint $\mathbf{neb_j}$
  Obtain parent joint $\mathbf{neb_{jp}}$ of $\mathbf{neb_j}$
  If $\mathbf{neb_j}$ has no parent joint (i.e. $\mathbf{neb_j}$ is the root joint)
    If a rigid body $\mathbf{body_1}$ is attached to $\mathbf{neb_j}$
      Compute the current model to world transform for the character ($\mathbf{T_w}$)
      and the current transform of $\mathbf{neb_j}$ ($\mathbf{T_j}$)
      See Listing 11.
    else
      error
  else
    let $\mathbf{T_{pb}}$ = body transform corresponding to $\mathbf{neb_{jp}}$
    If a rigid body $\mathbf{body_1}$ is attached to $\mathbf{neb_j}$
      $\mathbf{T_j} = \mathbf{T_{brj}}^{-1} \times \mathbf{T_b} \times \mathbf{T_{pbrj}}^{-1} \times \mathbf{T_{pb}}^{-1}$
    else
      let $\mathbf{T_b} = \mathbf{T_j} \times \mathbf{T_{pb}}$
      $\mathbf{T_j}$ doesn't change

Listing 10 – Updating character skeleton from rigid bodies.

- The algorithm assumes joints are processed in order, such that a Nebula joint is always processed before its ancestor joints, that way $\mathbf{T_{pb}}$ is always up to date whenever it is used.
- $\mathbf{T_b}$ is the current body transform associated with $\mathbf{neb_j}$ (every Nebula joint still has a body transform even if it doesn't have a corresponding rigid body).
- $\mathbf{T_{pb}}$ is the current body transform associated with $\mathbf{neb_{jp}}$.
- $\mathbf{T_{brj}}$ is the transform of the body attached to $\mathbf{neb_j}$ and relative to it.
- $\mathbf{T_{bprj}}$ is the transform of the rigid body attached to $\mathbf{neb_{jp}}$, if $\mathbf{neb_{jp}}$ doesn't have a corresponding rigid body then $\mathbf{T_{bprj}}$ is set to identity.
- $\mathbf{T_j}$ is the joint transform (rotation x translation) of $\mathbf{neb_j}$ relative to $\mathbf{neb_{jp}}$.

The formulas deserve some further explanation, which is best done with an example. Assume the skeleton pose before the rigid bodies were moved about by ODE is known, and there is a single root Nebula joint (**j0**) that is the predecessor of all other joints in the skeleton.

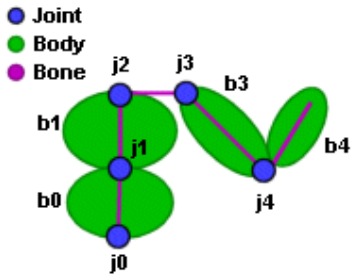

Figure 14 – Sample skeleton with attached rigid bodies.

Each joint **jn** has a corresponding joint transform $\mathbf{T_{jn}}$ and a body transform $\mathbf{T_{bn}}$, for **n** = 0 to num joints - 1. $\mathbf{T_{jn}}$ consists of the rotation and translation components of the joint relative to its parent. $\mathbf{T_{brjn}}$ is the transform of the body **bn** relative to the joint **jn**. Joint **j2** doesn't have a body attached, since there is no body there is no $\mathbf{T_{brj2}}$ (or it could be considers to be the identity matrix), however **j2** still has a corresponding body transform $\mathbf{T_{b2}}$. $\mathbf{T_w}$ is the current character model to world transform.

$$\mathbf{T_{b4}} = \mathbf{T_{brj4}} \times \mathbf{T_{j4}} \times \mathbf{T_{j3}} \times \mathbf{T_{j2}} \times \mathbf{T_{j1}} \times \mathbf{T_{j0}} \times \mathbf{T_w} \quad (1)$$

$$\mathbf{T_{b3}} = \mathbf{T_{brj3}} \times \mathbf{T_{j3}} \times \mathbf{T_{j2}} \times \mathbf{T_{j1}} \times \mathbf{T_{j0}} \times \mathbf{T_w} \quad (2)$$

$$\Rightarrow \mathbf{T_{brj3}}^{-1} \times \mathbf{T_{b3}} = \mathbf{T_{j3}} \times \mathbf{T_{j2}} \times \mathbf{T_{j1}} \times \mathbf{T_{j0}} \times \mathbf{T_w} \quad (3)$$

$$\Rightarrow \mathbf{T_{b4}} = \mathbf{T_{brj4}} \times \mathbf{T_{j4}} \times \mathbf{T_{brj3}}^{-1} \times \mathbf{T_{b3}} \quad \text{(Substitute 3 into 1)}$$

$$\Rightarrow \mathbf{T_{brj4}}^{-1} \times \mathbf{T_{b4}} = \mathbf{T_{j4}} \times \mathbf{T_{brj3}}^{-1} \times \mathbf{T_{b3}}$$

$$\Rightarrow \mathbf{T_{j4}} = \mathbf{T_{brj4}}^{-1} \times \mathbf{T_{b4}} \times (\mathbf{T_{brj3}}^{-1} \times \mathbf{T_{b3}})^{-1}$$

$$\Rightarrow \mathbf{T_{jn}} = \mathbf{T_{brjn}}^{-1} \times \mathbf{T_{bn}} \times (\mathbf{T_{brj(n-1)}}^{-1} \times \mathbf{T_{b(n-1)}})^{-1}$$

Listing 11 – Computing the Nebula joint transform from an ODE rigid body.

The formula in Listing 11 only works for **n** > 0, in order to extract the character transform and the current rotation/translation components of the root joint the assumption must be made that the root's translation component hasn't changed. This means any change in the orientation of the body attached to the root is associated with the root's rotation component and any change in the position of the body is associated with the character transform. Translation is associated with the character transform

because the root joint cannot move large distances from the model space origin, otherwise when control is switched from ODE back to the pre-canned animation system the root translation component will be blended with the translation component from the pre-defined animation and as a result the character will be flung back to its previous position (due to the linear interpolation of the translation components). The method for extracting the new root joint transform ($\mathbf{T_{j0}}$) and the new character transform ($\mathbf{T_w}$) is shown in Listing 12.

$$\mathbf{T_{b0}} = \mathbf{T_{brj0}} \times \mathbf{T_{orj0}} \times \mathbf{R} \times \mathbf{T_{opj0}} \times \mathbf{T_{ow}} \times \mathbf{P} = \mathbf{T_{brj0}} \times \mathbf{T_{j0}} \times \mathbf{T_w}$$
$$\mathbf{T_{bi}} = \mathbf{T_{brj0}} \times \mathbf{T_{orj0}} \times \mathbf{T_{opj0}} \times \mathbf{T_{ow}}$$
$$\mathbf{T_{diff}} = \mathbf{T_{bi}}^{-1} \times \mathbf{T_{b0}}$$
$$\mathbf{R} = \mathbf{T_{rdiff}}$$
$$\mathbf{T_{j0}} = \mathbf{T_{orj0}} \times \mathbf{T_{rdiff}} \times \mathbf{T_{opj0}}$$
$$\mathbf{P} = (\mathbf{T_{brj0}} \times \mathbf{T_{j0}} \times \mathbf{T_{ow}})^{-1} \times \mathbf{T_{b0}}$$
$$\mathbf{T_w} = \mathbf{T_{ow}} \times \mathbf{P}$$
$$= \mathbf{T_{ow}} \times (\mathbf{T_{brj0}} \times \mathbf{T_{j0}} \times \mathbf{T_{ow}})^{-1} \times \mathbf{T_{b0}}$$
$$= \mathbf{T_{ow}} \times \mathbf{T_{ow}}^{-1} \times (\mathbf{T_{brj0}} \times \mathbf{T_{j0}})^{-1} \times \mathbf{T_{b0}}$$
$$= (\mathbf{T_{brj0}} \times \mathbf{T_{j0}})^{-1} \times \mathbf{T_{b0}}$$

Listing 12 – Computing the root joint transform and the character transform.

- $\mathbf{T_{rdiff}}$ is the rotation component of $\mathbf{T_{diff}}$.
- $\mathbf{T_{bi}}$ is the previous body transform.
- $\mathbf{T_{orj0}}$ is the last known rotation component of **j0** (the root joint).
- $\mathbf{T_{opj0}}$ is the last known position (i.e. translation) component of **j0**.
- $\mathbf{T_{ow}}$ is the previous character transform.
- **R** is the current rotation of **b0** relative to its previously known rotation.
- **P** is the current translation of **b0** relative to its previously known translation.

4.3. CHARACTER NAVIGATION

In order to allow the end user to control the soccer player a spherical rigid body is placed at the base of the character and the character transform is derived from the sphere's transform. Forces and torques are applied to the sphere in order to move and orient the character. The character can be subjected to gravity and friction as a whole without having to physically simulate each foot, which reduces the amount of work ODE has to do. And additionally sliding against walls or down inclines is automatically handled by ODE.

Whenever the user presses a key to move the character the direction of movement is determined with respect to the character's local axes. This presents some difficulties if both torques and forces are applied to the sphere during a physics time step. The problem stems from the fact that application of a torque changes the direction of the character, but the new orientation can't be determined until the physics step is taken. So if both a torque and a force are applied the character will move in the direction he was facing at the beginning of the physics step but he will also rotate about while doing so, hence producing nothing resembling proper character movement. To counteract this problem an attempt was made to allow either torques or forces (through the centre of mass, hence no torque will be generated) to be applied at any one physics step. However a satisfactory solution has not been obtained as yet, because the current implementation seems to prevent any translation whenever the user rotates the character while moving.

5. CONCLUSION

The collision detection techniques discussed in this report seem to work, however, there is little doubt there is room for improvement. One area that wasn't explored by this project is temporal coherence, various collision detection algorithms have been exploiting it with success and it may be possible to apply it to the methods presented. Also, other collision detection algorithms such as variants of GJK [9] exist that are able to compute the penetration depth between complex objects, so it may be worth considering using one of them instead of the approximation provided by Level 3 volumes described in this report.

The integration of the ODE physics engine with Nebula's skeletal animation system for the purpose of rag-doll physics using the ideas presented in this report has yet to be proven to work in practice, as the implementation has not as yet been completed.

This project is still very much a work in progress, and suggestions for improvement or corrections for any mistakes will be appreciated.

6. IMPLEMENTATION

Some of the techniques described in previous sections have been implemented as part of the soccer simulation. Unfortunately due to time constraints not everything has been implemented and things that have been implemented are somewhat of a mess. Further details regarding the existing and future implementations will be available in a separate document that will accompany the soccer simulation. The demo and source can be obtained from www.steelronin.com (sooner or later).

7. ACKNOWLEDGMENTS

Many thanks to the dedicated and underpaid Scott Lange who modelled and animated the environment and characters for the simulation. Thanks to the good folks in #nebula, especially Leaf Garland and Bruce Mitchener for providing assistance with Nebula, Per Vognsen for helping out with bits of maths and physics. And finally thanks to Radon Labs for releasing Nebula, Magic Software Inc. for their freely available implementations of various low-level intersection algorithms, Russell Smith for ODE and Pierre Terdiman for OPCODE.

8. REFERENCES

[1] *Simple Intersection Tests For Games* article in Gamasutra, Oct 18 1999
[2] Open Dynamics Engine v0.035 User Guide
[3] ODE v0.035 source code, method getHingeAngle (joint.cpp)
[4] ODE v0.035 source code, method amotorComputeEulerAngles (joint.cpp)
[5] Jeff Lander
When Two Hearts Collide:
Axis Aligned Bounding Boxes article on Gamasutra.com
Feb 3 2000
[6] Nick Bobic
Advanced Collision Detection Techniques article on Gamasutra.com
March 30 2000
[7] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi.
I-COLLIDE: An interactive and exact collision detection system for large-scale environments
1995
[8] Gino van den Burgen
Efficient Collision Detection of Complex Deformable Models using AABB Trees
Nov 6 1998
[9] Gino van den Burgen
Proximity Queries and Penetration Depth Computation on 3D Game Objects
GDC 2001