Basic Skeleton And Muscle Animation

Robbert B Wijtman

Abstract

Ragdoll Physics is a cornerstone of modern gaming. Most commonly used in First Person Shooters, it can also be used in many other kinds of games, as well as other projects that may not necessarily be intended for entertainment. This paper will go over how to create a basic skeleton model and the formulas required for basic 3D movement of limbs.

Introduction

Most video games, and especially First Person Shooters (or FPS), attempt to bring the player into their world, in order to enhance the gameplay experience and ensure the user does not lose interest. In FPS games, especially, realistic character animation is a must. In many older games, the characters would, when killed, go through a static, pre-loaded death animation. While this was fine for some cases, there were cases when the death animation resulted in a decidedly unrealistic situation, such as when an enemy is killed near a cliff, then dies with half the animated body hanging directly off the cliff, in violation of the laws of gravity. This gives the player an unintentional reminder that they are playing a game, which is an undesirable outcome. In addition, when games brought in explosive weapons, such as rockets or grenades, the death animations became complicated to pre-generate. To combat these issues, games added a new level of realism: Ragdoll Physics.

Ragdoll Physics, in brief, is the procedural (real-time) simulation of a body and associated limbs in motion. The most notable example is the popular Half Life 2 series. In Half Life 2, picking up an enemy soldier with the Zero Point Energy Manipulator (or Gravity Gun) and swinging it around results in very realistic movement of the body. Another good example is using an explosive weapon, as indicated above. Enemies are sent through the air with a sprawling of limbs that adds that next level of realism for the player, and allows a much greater depth of enjoyment for the player.

There are four different kinds of Ragdoll Physics:

1. Verlet Integration. In Verlet Integration, each *bone* in the character is connected to other points via a simple set of contraints.
2. Inverse Kinematics, as used in Half Life 1. Inverse Kinematics takes a pre-generated death animation and modifies it based on the environment, preventing dead characters from falling through walls, and allowing them to tumble off cliffs.
3. Blended Ragdoll, as used in Halo 2. Blended Ragdoll also uses a pre-generated animation, but contains it within a particle based system, allowing correct environmental interaction as well as keeping the character from suddenly going limp. Though expensive (requiring both extensive physics and graphics processing), Blended Ragdoll results in an impressive level of realism.
4. Procedural Animation. Rather than using a pre-generated animation, Procedural Animation uses a complicated system of bones, muscles, and sometimes even nerves to achieve a completely movable body that can correctly react to all aspects of the environment and appear fluid and as realistic as possible. Procedural Animation often makes use of Skinning, a process involving drawing skin over muscles and bones, to generate the model that is drawn to the screen. [Wikipedia, Ragdoll Physics]

Of the four above, Procedural Animation, while likely one of the most complicated, also produces the most realistic result, as well as allowing an impressive amount of control over the movement of the character animation.

In this project I did not implement any external effects, such as the character hitting a wall (the expected result would be for the body to continue moving forward even when one of the limbs has impacted the wall) or being hit with a solid object (the expected result would be for the limb struck to be pushed along the path of the object, any child and parent limbs also moving, possibly even knocking the character over). I also did not implement any translation, rotation, or scale of the body as a whole.

Basic Classes

There are five basic classes that make up the project:

1. Skeleton. The Skeleton class contains linked lists of all bones, muscles, and tendons. It is tasked with making sure all are updated and drawn. In a more complicated Ragdoll Physics system it would also store translation, scale, and rotation data for the body. The Skeleton is the only "forward facing" class-all requested manipulation to any bones, muscles, or tendons must be done through it.
2. Bone. The Bone class, very simply, is a pair of 3D points that must always maintain the same distance from each other. A more complicated system could also allow bones to be broken, destroying this bond. The Bone basically exists to show the muscles where to be drawn.
3. Muscle. A muscle belongs to a Bone, and will always be drawn from Bone start point to Bone end point. Muscles have a unique front-facing ID that is used by the main program for contracting/releasing them.
4. Tendon. The Tendon class has a start Bone and an end Bone. It is tightened or loosened by a connected muscle. The Tendon class does most of the work-its length is interpreted as the required length between the midpoints of two Bones that are joined at one end.
5. XML. Skeleton and animation data are stored via XML data. The XML class contains functions to facilitate the loading and parsing of this data. It has no other purpose and will not be discussed in depth in this paper.

Skeleton

The Skeleton is tasked with updating, drawing, and passing along motion requests to the Bones, Muscles, and Tendons. The Tendon update method, having the main workload of deciding where the bones should be moved, is called first. The Bone update, in a more complicated system, would handle the Bone's local response to the body as a whole being rotated, scaled, or translated, and is called next. However in this particular system it does nothing. The Muscle update, being little more than a formula that decides the rotation, position, and scale of the Muscle based on its parent Bone, is called last, after the final positions of the Bones have been finalized.

Bone

The Bone class contains two 3D points, start, and end. In a more complex system these points, after being determined locally by the Tendon update method, would be scaled, rotated, and translated based on where the body as a whole was. Since I did not implement this, I do not have the formulas required to accomplish this task. It also contains an algorithm for secondary rotation of child bones if the parent has been shifted:

```
// inputs angle a and 3D axis axis

for each child bone
    let xDist = child->start_x-end_x
    let yDist = child->start_y-end_y
    let zDist = child->start_z-end_z

    // shift the bone over by this distance, to ensure they are still connected
    child->start_x -= xDist
    child->start_y -= yDist
    child->start_z -= zDist
    child->end_x -= xDist
    child->end_y -= yDist
    child->end_z -= zDist

    // rotate child around same axis, with same angle
    child->end->rotate(a,axis)

    // request that child updates its own children
    child->updateChildren(a,axis)
endfor
```

Muscle

The Muscle class uses gluDrawSphere to draw a stretched oval completely covering its parent Bone. The Muscle gets its rotation, scale, and translation by the following formula:

```
let xDist be the distance between start_x and end_x
let yDist be the distance between start_y and end_y
let zDist be the distance between start_z and end_z

let dist be the total distance between points. IE sqrt(xDist^2+yDist^2+zDist^2)

// find the midpoint of the bone-this will serve as the center of the muscle sphere
```

```
let x be start_x + xDist/2
let y be start_y + yDist/2
let z be start_z + zDist/2

// the scale of the muscle is half the distance (radius of sphere). The muscle is scaled along the x axis
let scale be dist/2

// shift the end point so that the start point is the origin
let new_end_x be end_x-start_x
let new_end_y be end_y-start_y
let new_end_z be end_z-start_z

// calculate rotations. Arctan2 is an extension of the arctan function that correctly handles quadrants.
let yRot be arctan2(new_end_x,new_end_z)
let zRot be arctan2(new_end_x,new_end_y)
let xRot be arctan2(new_end_y,new_end_z)

// if start point is less than end point then the rotation needs to be flipped around the axis
if start_x < bone_x then zRot += (90-zRot)*2
if start_z < bone_z then xRot += (90-xRot)*2
if start_x < bone_x then yRot += (90-yRot)*2 // could use x or z here
```

Muscles also bulge when they are contracted. To do that I use the following two code snippets:

In creation code:

```
let minContract be the current tightness of the tendon
```

Here we assume that at creation the model is in a position such that all limbs are fully extended, thus all muscles are fully relaxed.

In draw code:

```
let t_scale be contracted/minContract
glScale3f(scale,0.5+t_scale,0.5+t_scale)
```

Tendon

The Tendon class is the most complicated of all. Each update, we first find the current distance between the midpoints of its start and end bones. If this distance is different from the current tautness, then the end point of the end bone needs to move (the start point of the end bone being fixed to the end point of the start bone, which does not move). Both start and end points of the start and end Bones are translated so the end point of the start Bone is the origin. This gives two separate vectors, start_bone->start_point, and end_bone->end_point. Then the the distance of the start Bone and the distance of the end Bone is calculated. Using these two lengths (divided by two), and the current distance between the midpoints, we can use the law of cosines to find the angle between the two vectors. Then we can use the same two lengths (again divided by two) and the tautness of the tendon-the *desired* distance, to find the angle that we want the two vectors to have between them. Then we do a rotation around the cross product of the two vectors, by the amount of the distance between the two angles.

Or, in code:

```
// get midpoints
let start_mid_x = start->start_x+(start->end_x+start->start_x)/2
let start_mid_y = start->start_y+(start->end_y+start->start_y)/2
let start_mid_z = start->start_z+(start->end_z+start->start_z)/2

let end_mid_x = end->end_x+(end->end_x+end->start_x)/2
let end_mid_y = end->end_y+(end->nd_y+end->start_y)/2
let end_mid_z = end->end_z+(end->end_z+end->start_z)/2

// find distance between endpoints
let dist = sqrt((start_mid_x-end_mid_x)^2+(start_mid_y-end_mid_y)^2+(start_mid_z-end_mid_z)^2)

if dist != tightened then

// translate all points to use start->end as origin
lets Start = start->start-start->end
let sEnd = (0,0,0)

let eStart = (0,0,0) // end->start should be the same as start->end. The two bones are joined by
                         // these points
let eEnd = end->end-start->end

// get distances of each vector
let sDist = distance between sStart and sEnd
let eDist = distance between eStart and eEnd

// get angles
let oldAngle be angleBySides(sDist,eDist,dist)
let newAngle be angleBySides(sDist,eDist,tightened)
let angleDistance be newAngle-oldAngle

// get axis
let axis be sStart x eEnd
normalize(axis)

// rotate
end->end->rotate(axis,angleDistance)

// request end bone to update its child bones
end->updateChildren()

endif

angleBySides:
input: a,b,c as lengths. Returns angle opposite side c
```

```
let top = c^2-b^2-a^2
let bottom = -2*a*b
return arccos(top/bottom)
```

Point

Point has one function of note: the 3D rotation function. The formula involves a complicated rotation matrix, and could be accomplished by a dozen smaller steps, but this makes it easier.

```
// inputs angle t, point with coordinates u,v,w
// returned point coordinates
x = (u*(u*x+v*y+w*z)+(x*(pow(v,2)+pow(w,2))-u*(v*y+w*z))*cos(t)
+sqrt(pow(u,2)+pow(v,2)+pow(w,2))*(-w*y+v*z)*sin(t))/(pow(u,2)+pow(v,2)+pow(w,2));
y = (v*(u*x+v*y+w*z)+(y*(pow(u,2)+pow(w,2))-v*(u*x+w*z))*cos(t)
+sqrt(pow(u,2)+pow(v,2)+pow(w,2))*(w*x-u*z)*sin(t))/(pow(u,2)+pow(v,2)+pow(w,2));
z = (w*(u*x+v*y+w*z)+(z*(pow(u,2)+pow(v,2))-w*(u*x+v*y))*cos(t)
+sqrt(pow(u,2)+pow(v,2)+pow(w,2))*(-v*x+u*y)*sin(t))/(pow(u,2)+pow(v,2)+pow(w,2));
```