COVENTRY UNIVERSITY
Faculty of Engineering and Computing

REAL TIME CHARACTER ANIMATION: A GENERIC APPROACH TO RAGDOLL PHYSICS

Author	Mark Watkinson
Student ID	1518461
Course Title	MSc Computer Science
Module	M29CDE
Date	23/09/2009

MSc Projects/Dissertations

Declaration of Originality

This project is all my own work and has not been copied in part or in whole from any other source except where duly acknowledged. As such, all use of previously published work (from books, journals, magazines, internet etc.) has been acknowledged within the main report to an item in the References or Bibliography lists.

I also agree that an electronic copy of this project may be stored and used for the purposes of plagiarism prevention and detection.

Copyright Acknowledgement

I acknowledge that the copyright of this project report, and any product developed as part of the project, belong to Coventry University.

Signed: Date:

Office Stamp

Abstract

Ragdoll physics represents an increasingly common method to increase players' perceived interaction with 3D computer games (and other virtual worlds) by providing dynamic character response to physical impacts. Traditionally a character would not 'recover' from entering a ragdoll physics mode, consequently ragdoll was only useful for creating character death animations. Combining dynamic character physics simulation with pre-defined sources of animation (keyframe/motion capture) is a current area of research.

Most games that use ragdoll physics use an implementation specific to their game engine, and generic approaches do not exist. This project aimed to investigate and implement the concept of applying ragdoll physics in a very abstract, generic manner, and to synthesise new animation by combining ragdoll simulation with a pre-defined animation data source.

The system implemented employed skeletal retargeting, using a generic ragdoll skeleton structure to abstract the problem of ragdoll physics. The ragdoll skeleton was simulated using a constrained particle dynamics approach to rigid body dynamics, and used a Verlet-Velocity integration system. The concept of a 'target' ragdoll skeleton constructed from pure pre-defined animation data is used to allow a character's motion to be derived from both ragdoll and pre-defined sources, giving the result of allowing the character to believably recover from ragdoll effects. The ragdoll system is also used as the basis of an inverse kinematics solver.

The method and system presented shows strong results in ragdoll/pre-defined animation synthesis, as well as inverse kinematics, and further provides an effective (but inconvenient) method for applying a single ragdoll simulator to diverse models through retargeting. However, due to time constraints it was unclear whether the ragdoll method is sufficient in itself for generally realistic ragdoll effects.

Table of Contents

1 Introduction

A significant part of the immersion a virtual world provides is through its inhabitants. The characters who exist within the world are expected by the player/user to appear real and believable, and their movements, their animation, is a large part of achieving this goal. Traditionally, real time animation was entirely pre-recorded then played back as the character moved through the world. The animation did not drive the movement; the animation and the movement were merely synchronised to appear that they were the same.

With the recent, rapid increase in the hardware resources available to the average computer game, animation began to include real time simulation of character model movement in the form of ragdoll physics; ragdoll gives another approach to animation in that it allows a character model to become limp and respond to impacts accordingly. Contrary to traditional pre-defined animation, within ragdoll physics, the character's movement through the world is inseparable from its animation. Invocation of ragdoll typically meant a character had died and was now a lifeless body that could be manipulated by application of impacts (through whatever mechanism the game allowed). Recently, the concept of merging ragdoll effects into an animation cycle has been a growing area of research; the idea of which is to provide a dynamic reaction to a smaller (i.e. non-fatal) impact/effect: to use the ragdoll simulation to provide a response then blend the character model's resultant position back into the original motion capture animation. This technique is relatively new and is not (yet) commonplace within games.

Implementations for ragdoll physics are often integrated into game engines in such a way that they are difficult to re-use across different applications. Whilst standalone animation engines and physics engines exist, standalone engines that handle physics-based animation are either very rare or non-existent, despite the fact that many modern games now incorporate ragdoll physics. Such a library would have definite value.

All of the areas necessary for simulating in-game physics are well established, having been applied in physics and engineering fields long before their inclusion into computer games. The specific method of simulation outlined in this project is borrowed from particle dynamics, and its application to ragdoll was originally presented in Hitman2 (IOInteractive 2002).

Synthesising new animation from existing animation has always been a difficult problem, but potentially a very rewarding one to solve. By combining existing animation data to form new animation routines, animation synthesis potentially provides a much richer database than would otherwise be obtainable of animations that the game can use. In doing so, synthesis allows creation a more believable world for the player. Effective synthesis of new motions allows fewer resources to be

allocated to creation of animation data during an application's development[1] and those resources can thus be spent elsewhere.

This project implements an animation library to allow ragdoll physics and blending between ragdoll and keyframe animation sources. The methods used are suitable for use on an arbitrary humanoid bipedal input model. In this report are detailed the methods underlying the library's implementation. The library implements a standard keyframe animation library and an entirely separate ragdoll simulation library, the latter forming the bulk of the work.

In this report it is shown that ragdoll is a difficult problem to solve in a truly abstract sense as there exist many caveats which relate to character models' potential individual properties, but by using skeletal retargeting techniques to create a 'standard' ragdoll skeleton, many of the individual considerations can be removed at the ragdoll level. Combination of the two data sources is addressed and solved within the ragdoll system at the level of the ragdoll skeleton by making use of a second skeleton, which represents the library's pure keyframe state (i.e. the character model in its keyframe position, translated to an equivalently posed ragdoll skeleton). At this simplified level, merging of the two skeletons becomes almost trivial and due to skeleton constraints that enforce the skeleton remain humanly shaped, this method of blending offers greater realism than simpler animation blending techniques.

The ragdoll skeleton is shown to be surprisingly versatile in that it forms a natural basis for an inverse kinematics solver. Its appropriateness as an IK solver is explored, and exploited to provide another method of motion synthesis.

1 Given that there are so many combinations and permutations of animations (i.e. “idle”, “walk”, “run”) and sequences (i.e. “idle to walk”, “walk to run”), and as these have to be recorded individually, creation of a full animation database is potentially very expensive: costs involve 3D modelling, and actors for motion capture.

2 Literature Review, Previous Work, and Current Standards

2.1 Character Animation Standards

Before evaluating the more advanced areas with which this report is concerned, a brief discussion of standard approaches to real time animation at a basic level is provided.

Real-time animation is a well studied concept and has been included (to some degree) in almost every competitor game of the past ten to fifteen years. The work flow to include character models and animation within an application became standard:

1. A modeller creates a model within a 3D modelling package (e.g: 3DS Max, Blender, Maya).
2. Animation data is associated with the model, either by defining the animation within the modelling package, or by using motion capture. The animation data is stored within the model's files.
3. The model (and its animation) is exported to a file format[1], on disk.
4. The application (game) imports the model by parsing its file, and stores it within memory for the runtime of the duration of the application's runtime

3D character models are usually represented as a tree structure[2] with each node having associated with it a transformation matrix. 3x3 rotation matrices are a well known property of linear algebra, but in 3D graphics it is also common to introduce a fourth dimension (homogeneous co-ordinates) and use 4x4 matrices for such transformations, as these can also contain scaling and translation properties (Rotenberg 2005). This matrix orients the node in 3D space relative to its parent node (as defined in the tree). This is so called 'local-space', which makes keeping the different nodes of the model attached to each other much easier to enforce; a transformation of any node may easily be inherited by its children (so a movement at the shoulder automatically results in the upper-arm, lower-arm and hand being moved along with it). (flipcode.com 1998)

1 There exist many, many file format specifications for storing models. Each format holds roughly the same data, but encoded differently. This means that step 4's parsing will be written specifically for the format the application expects to use, and importing models in other formats will not be possible without writing extra importers.

2 The tree uses a node for each limb in the model, the hierarchy is defined by joints e.g. the hand is a child-node of the forearm, which is a child of the upper arm, etc. Counter intuitively, the most useful position for a root node ends up being around the waist.

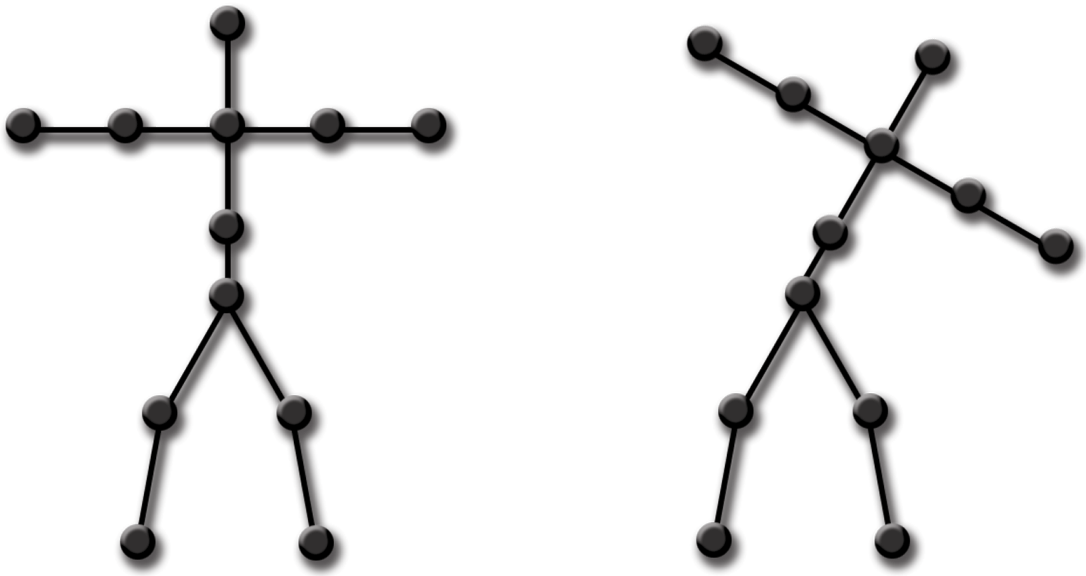


Figure 1: The effect on the upper body of a single rotation at the waist node. Note that only the waist node has been altered.

Typically the matrices will be such that traversing the tree and at each node combining (multiplying in the case of standard rotation matrices) the current node's local orientation with the parent node's global orientation will yield the current node's global orientation[1]: this is its location in the game's world space.

For heavily detailed animation targeted to a relatively small area, such as facial animation, a different technique called Morphing is often used (Evangelista 2008), but that is not the focus of this project.

Individually, the matrices' scaling and translation can be represented as 3D vectors (or 4D in homogeneous coordinates, but the 4th element is simply set to the value '1'). Rotational components are most obviously represented by a set of Euler angles representing rotations in each respective axis. However, instead, rotational data is often encoded as a unit quaternion, i.e. a four-dimensional number [w, x, y, z], representing a co-ordinate in a real dimension (w) and three imaginary dimensions (x, y, z) (Wesstein “Quaternion” 2009). Quaternions are an extension to complex numbers and so inherit some arithmetic properties from standard two-dimensional complex arithmetic, meaning that there are fewer necessary operations in computing successive rotations by a sequence of quaternion multiplications than by the equivalent matrix multiplications.

Euler angles are affected by Gimbal Lock, a loss of a degree of freedom resulting from a rotation in which a rotation in one axis places two axes pointing in the same direction (Fjeld 2006), however, as quaternions exist in four dimensions they do not

1 This is self evident, as multiplying successive transformation matrices will make the transformation effect stack up

suffer from this problem. Quaternions exhibit smooth interpolations, contrary to Euler angles which often do not yield perfectly smooth results when interpolated (Martin 1999).

Animation data exists in two important forms, pre-defined and procedural.

2.2 Pre-defined Animation

Pre-defined animation is currently the most common method for handling animation. The animation data for any particular animation is encoded as a sequence of transformation properties (scaling, rotation, translation) for each node in a sequence of time-steps. This works similarly to a recording, as it is then played back. The application then determines for any animation being played back the correct (time) index of sequence at each frame (a relative time that describes the current position in the animation sequence. When the end of the sequence is reached, the relative time will usually 'wrap-around', i.e. begin again at zero). The relative time will often lie between two indices in the sequence, so the transformation at that particular time will need to be calculated rather than purely read. The appropriate transformation is determined by interpolation between the previous and the next index's transformation properties (Adams 2003). This results in frame rate independent playback speed. This is a space/time trade-off, saving storage and memory requirements by forcing the CPU to approximate the in-between values. Vector components are usually interpolated with simple linear interpolation (i.e. the path from the first vector to the second is represented as a parametrised line such that when the distance parameter is iteratively increased the points along the line are traced out incrementally),

$$V_t = V_0 + t(V_1 - V_0), \quad 0 \leq t \leq 1$$

Interpolation between rotations is done using slightly different methods, one such method is Spherical Linear Interpolation (SLERP), a method in which the quantity being interpolated is moved along the surface of a sphere so that its distance from the origin remains constant (Shoemake 1985).

SLERP is somewhat controversial, having spawned its own discussion on “when not to use it” (Blow 2004). The argument Blow makes is that SLERP may be too slow for use in real time game systems, but he does not provide any analysis that supports his supposition that a 'real' system would spend so much time calling a SLERP routine that it would make a measurable difference. A common alternative to SLERP is Normalised Quaternion Linear Interpolation (NLERP). A comparison is given in (Tremethick 2006), in which it is shown that NLERP is slightly less accurate but not noticeably so. It is also shown that to be faster to compute, but whether it is measurably faster is not addressed.

It is possible to blend together multiple animations to combine them and play them back together. For example, blending a 'running', animation with a 'shooting' animation would make the character appear to be performing both actions simultaneously. This is achieved by averaging the transformations derived from multiple animation sources (Adams 2003). This works very well when applied to relatively similar animations, but it is self evident that blending a “lying down” animation with a “run” animation obviously would not give pleasing results (the character would appear at a 45 degree angle).

Predefined animation can be created either by an artist using a modelling package, or by motion capture; the origin of the data is not important in terms of playing it back.

Procedural animation (and procedural creation of other game assets, too) is a relatively new concept compared to pre-defined animation. The idea concerns itself with algorithmically generating animation rather than relying only on pre-defined sequences. An important area of procedural animation, on which this project focusses, is ragdoll physics.

2.3 Ragdoll Physics

In this section the word 'body' typically refers to a rigid body (typically a single node in a skeleton), and the terms “articulated body”, “skeleton”, and “model” are roughly interchangeable, referring to something that visually represents a character.

Implementing physics simulation (or instead creating illusions of physics simulation) within games is not a new area, having been described in depth from a very practical point of view at least as early as 1996 (Hecker 1996), and similar ideas were explored from a more academic point of view by Hahn (1988). The articles describe the mathematical simulation of Newtonian physics within real-time applications, using simple numerical methods to solve the sets of differential equations that arise from Newtonian dynamics, so as to model the movement of a body as a force is applied to it. The articles also describe areas such as angular effects and collision detection of bodies. At the time these were written, the authors (so the writing implies) intended their methods to be applied to lifeless objects within the game world to create an environment that to some degree reacts to in-world events. An interesting development of recent years is the application of these methods to character animation to allow for dynamically generated death animations (more commonly known as 'ragdoll physics').

Ragdoll physics tends to fall into two categories: rigid body dynamics (as given in Hecker (1996)) and particle dynamics. These are not dissimilar, and in fact the particle dynamics approach uses its particles to model rigid bodies; but it is convenient to draw a distinction between particle and non-particle approaches as the internal details of the two systems are conceptually separate.

Hennix et al (2003) describe a rigid body approach to ragdoll physics using many of the ideas (Hecker 1996) introduces to create an articulated body. The idea presented is the modelling of a ragdoll as a set of linked boxes (each acting as a rigid body, representing a limb/node). The bodies are free to move, but exert a force upon each other at their joints, so by moving any body, the bodies to which it is joined are pulled as a result. The methods described by Hecker (1996) and Hahn (1988) are employed on each body and their joints to create a set of linked bodies that behave like a limp human. This system shows a physically accurate approach to simulating ragdoll physics. It likely produces more accurate results than less rigorous physics approaches, however, the computational cost of evaluating the ragdoll and its joints is likely higher as a result.

Within this system a non-trivial joint constraint system is employed; the exact system is described by Smith (2004); a constraint is expressed as a linear system of equations where individual elements can easily be scaled or 'zeroed' out.

Hennix et all (2003) also describe collision detection, which is a necessary component of a ragdoll physics simulation. Objects within computer graphics cannot literally 'collide' so the application has to be able to determine when two objects intersect, and make them react accordingly. Hennix et al (2003) outline collision detection as two processes; a 'broad' and 'narrow' phase. The first phase seeks to determine which objects are close enough that they may potentially collide with each other, and the second determines whether they actually do. Collision detection is a difficult problem to solve computationally as there are potentially many objects and a computer cannot determine which are intersecting without running some test(s) on each of them with respect to every other object, which results in super-polynomial complexity growth in terms of the number of objects being tested. The 'broad' phase quickly excludes the majority of other objects from having to be subjected to more expensive collision detection (i.e. explicit intersection tests). Reacting to collisions is another area covered by Hecker (1996) although in the context of ragdoll physics, reaction needs only to be minimal as characters do not usually need to visibly bounce as a result of a collision.

In contrast to the physically rigorous approach is that presented by Jakobson (2001), which solves the problem in a simplified and more imaginative manner.
Jakobson describes a particle system which is used to construct an articulated body. From a high level, this method consists of a set of particles held together by distance constraints. Each node within a skeleton is described by two particles (points in space), forming a line. At each time-step, forces are applied individually to each node, causing them to move independently of one another. To retain its articulated structure the particle system is subject to a number of constraints which are enforced as much as possible at the end of each iteration of the simulation. The constraints are solely distance constraints (i.e. two points must be the a set distance from each other) and come in three forms (although the last two are equivalent in that within a skeleton, for each example of 2) there exists an example of 3)):

1. Distance of the start of a node to the end of the node
2. Distance of the start of a node to the end of its parent
3. Distance of the end of a node to the start of its child

Thus, upon moving any point, an process is started whereby the points to which it is attached are affected and pulled by the movement in order to satisfy the constraint. This is repeated along the 'chain' of linked nodes.

A convenient side effect of this method is that rotations of nodes do not need to be handled explicitly by the physics simulation, because they arise implicitly. Applying a force equally to all particles within a node will result in a translation, but for realism a force's effect on the particles would be weighted to reflect the exact position of the force's application. In this case, the translation of each particle will be different and the node therefore rotates.

The process of applying angular movement is far more involved than linear movement, from both an implementation and computation point of view, so the avoidance of explicit rotation handling puts this system in a very favourable light. It is an obvious supposition that since the angular movement in this system is a side effect of applying the forces, that it is probably not as accurate or realistic as handling it explicitly as in Hecker (1996) or Hahn (1988). Conversely, it is worth considering that human joints are not 'smooth', they have damping forces from friction and more importantly from basic anatomical restrictions. Simulating this in full from a strict physics viewpoint would be a difficult problem and would require a lot of artificial tweaking to perfect; consequently the accuracy benefit of the strict physics approach may not be as great as it first appears.

Furthermore, the only calculations necessary to affect the system itself are translations in space (on each particle), which are simply three additions and therefore can be evaluated very fast. The speed of execution is evidenced by Brown (2009) who details a recent successful creation of a ragdoll physics engine using Jakobson's approach, the performance of which was suitable for a portable console, the Nintendo DS.

The advantages of this system lie in its simplicity to implement and its speed resulting from this simplicity.

The disadvantages however are significant: Jakobson describes a method for collision detection based on making sure nodes do not come too close to each other, and making sure legs do not cross, and similar. This is a very naive approach and, to quote Rosen (2007), it "addresses the symptoms rather than the problems". However, the system as is given does not provide much support for anything more robust.
The second problem is that by representing each node as a line, it is impossible for the system itself to provide any meaningful data about the node's rotation about the

axis going through itself from top to bottom (for example, it cannot represent turning one's head from left to right). Whether this really is a problem or not is unclear: with the exception of the neck/head, a body rotating about itself is a very subtle transformation and would probably not be missed. In the case of the head, this rotation could probably be applied afterwards, as a special case, outside of the particle system.

An important but seemingly extremely overlooked contribution is presented by Rosen (2007). Rosen builds directly on Jakobson's method, adding some extra considerations that address the inherent problems. Rosen argues that Jakobson's approach can be extended in a conceptually simple manner to give each node in the system easily derivable orientation data by building nodes out of triangles instead of lines; from a triangle an orthonormal basis that describes the node's rotation can be derived.

The constraints between the shapes that make up a node are enforced as distance constraints; each point in the triangle, for example, has two constraints to enforce the shape of the triangle, as well as any constraints connecting the particle to other nodes (i.e. the particle will have extra constraints if it is forming a joint between multiple nodes).

Rosen extends this by using multiple points of a node to create different joint types, an idea mentioned by Jakobson. By themselves, these joints can act as either hinges (2 particles per joint) or ball joints (one particle) dependent upon the number of points on each node which have a connection to the other node. This goes some way to address angular constraints between nodes. The orientation between two nodes is derivable from comparison between their individual orientations (accessible from the orthonormal basis), and he posits that this provides the possibility to apply more specific constraints, as Euler angles.

In the 'further work' section of his paper, Rosen suggests that this system could be extended to handle balance by approximating the centre of balance of the character from its foot positions, and comparing that to a centre of mass (which could be approximated automatically but would make more sense to be arbitrarily defined as being located somewhere in the upper body of a character), and then when the model becomes unbalanced to try to return it to the angular joint rotations of some known balanced position.

This system provides a number of improvements over Jakobson's while retaining many of the advantages of simplicity. The disadvantages are that firstly some of the performance benefit of Jakobson is lost in evaluating a much greater number of constraints arising from the node shapes, and secondly, by treating an articulated body as a set of linked two dimensional shapes it still by itself does not provide a good basis for a collision detection system; a model lying on the floor for example would have its skeleton lying directly on the floor plane, but half of its actual rendered

vertices would likely clip below this plane. Rosen notes that the collision detection system could be improved by wrapping hit-boxes[1] around the nodes.

In augmentation to a ragdoll system, there is also a growing area of active research in trying to combine the pre-defined animation with dynamic response provided by a ragdoll simulation.

An attempt at merging traditional pre-defined animation data and real time ragdoll physics simulation was undertaken by (Park 2008). Unfortunately the project was a failure in the sense that it did not yield any direct success and does not provide much information of its overall approach, so it is difficult to ascertain from it exactly what did not work, however, it is still worth some consideration as it states the conclusion that any combination of these two data sources is a very involved process requiring a deceptively complicated system capable of scheduling events and representing a lot of data about the whole body. The lesson it teaches is that such a problem should not be approached lightly; rigorous physical simulation is necessary, and so is a strong underlying software system.

A more rigorous and more successful attempt at combination of the two data sources was given by Zordan et al (2005), who used the idea of having the simulation 'follow' data from the pre-defined animation source to smoothly move between different animations. Each node in the skeleton is given a motor controller which has knowledge of the node's current position and of a target position (each node having the usual 6 degrees of freedom); the motor is then responsible for trying to move the node into the target position. No type of joint constraints are built into the controller they describe but such constraints could be built in elsewhere in the overall system and evaluated and enforced after each time-step of a simulation. Between an initial and target position that are of short (translational and rotational) distance, Zordon et al's approach yields visually pleasing results, however, to handle movement between contrasting poses such as standing and sitting it proves insufficient and results in unrealistic artefacts.

A similar approach to that described by Zordon et al (2005) is presented by Wroteck et al (2006) in the Dynamo system, which uses the same concepts to create dynamic animation response to environmental events. They assert that most similar methods have not been overly successful due to attempting to apply the necessary combinations of pre-defined and simulation data to the transformations of each node's local space. By using world space instead, they claim to have had more success, and have reduced a lot of the implausibilities that otherwise arise. They provide little information on the implementation of their underlying ragdoll system but mention that it is a rigid body stiff spring system, so is presumably roughly equivalent

1 A box that encloses a node on a character model. The box is used for collision detection, in that it serves as a structure which which it is relatively easy to test for intersection – it is much easier to test if something has intersected with a box, than with all of the polygons that make up the node in the model.

to that described by Hennix et al (2003). From a high level, each node is 'powered' and its resistance to simulated change is inversely proportional to its power.

They combine data from the simulation with data from the keyframe/motion capture data to try to allow realistic responses to dynamic events that occur to the characters. The combination is not a simple blending; when the character is undergoing dynamic processing, a rotational force is applied to the character's nodes to try to make it gravitate towards the target orientation (given by the motion capture data). The speed at which the limbs move towards their targets is controlled by a motor, similarly to Zordon et al's approach (2005), the force exerted by which can be altered easily and can be instantly reduced and then quickly increased to its full value in order to simulate effects such as being stunned. This (the stunning effect) appears to be a method by which their system controls the precedence between ragdoll and pre-defined animation data, so that the resultant animation can focus more on one than the other when necessary.

It also includes a balance system. The balance is determined by the root node's 'power', which basically represents a resistance to the character becoming limp and falling over. Although it is not detailed how the Dynamo system computes balance, it can be inferred that such a system would have to consider the positions of the feet of a character relative to its centre of mass.

3 Problem Description and Requirements

The aim of this project is to design and implement a generic approach to including ragdoll physics into real-time applications, which will also contain the ability to combine ragdoll physics with a pre-defined animation source. The main deliverables of this project are a specification of a method to achieve this, and a software implementation of the method.

Of importance to the method is generality: the method should be able to act on a wide variety of character models: some restrictions may be necessary, but a model that is unsuitable should be a special case with some unusual property, rather than requiring that a suitable model is a special case of character model in that it is specifically tailored for the method. Furthermore, the method should be as automated as possible although some user intervention will inevitably be necessary.

The blending method (between a ragdoll physics source and pre-defined animation source) should be such that the character can dynamically react to impacts, and then revert to its pre-defined animation cycle in a believable fashion.

The software should be a useful system in its own right, by which is meant, the software itself should be suitable for inclusion into applications in which ragdoll effects are desired. Therefore, the software shall be in the form of a fully functional animation library.

The library should implement both ragdoll physics and pre-defined animation playback.

There is no obvious reason that an animation library should have special software dependencies, and so should be written using portable code such that it is suitable for use on a wide number of environments (operating systems/platforms). This is especially of interest because the library's most likely user is a developer of a low-budget application (a more commercially oriented development team would likely prefer stronger supported solutions), for example an Open Source game (which are usually written by hobbyists who do not have the same affinity towards Microsoft Windows that is displayed by commercial games).

The library should be suitable for inclusion into a project via dynamic linking, meaning that it should provide in its public API all the necessary methods to for an external application to use the library to perform the generic ragdoll method. Any reasonable usage of the library (i.e. usage defined within its requirements) which requires direct access to, or modification of, the source code would make it an unacceptable solution.

The library's target users are real time 3D graphics applications (primarily games) developers. To make the library a usable system, a full API documentation is required as well as example code illustrating usage of the library's basic constructs.

The project will also require a simple demonstration program to be written to show that the library is working.

4 Approach

4.1 Analysis of requirements

The requirements specify that a generic, portable library shall be written. This leads to an important implementation decision of choosing the most suitable programming language for the task.

The subset of sensible language choices for most application and library programming can safely be restricted to C, C++, C# and Java.

	Advantages	Disadvantages	Portability	Familiarity
C	Strong, high performance language	Not object oriented very small standard library Memory leaks possible due to no automatic garbage-collection	Very high if written conforming to standards	Very High
C++	Inherits C's advantages, Supports OO Much larger standard library	Memory leaks possible due to no automatic garbage-collection	Very high if written conforming to standards	None
C#	Supports OO Huge library support	Tends to be slower than C/C++	Microsoft Only (with weak unix-support from MONO (mono-project.com 2009)	Low
Java	Supports OO Huge library support Very cross-platform	Difficult to compile native library code; would be hard to use from a non-Java program (and Java is not a common choice in games programming)	High across PCs (Windows/Mac/ Unix-like)	Moderate

Performance comparison data provided by Bruckschlegel (2005)

Although it loses dramatically with respect to prior familiarity, C++ was chosen as the best tool for implementing this project. C++ is a super-set of C, thus making familiarity less of a consideration. C++ is a well established standard in games programming, making it the ideal choice as it means other C++ applications will easily be able to use its interface. The performance gains from the fact C++ is compiled to native code are also significant. C++'s standard library is not as extensive as C#'s or Java's but is sufficient. C++ provides cross-platform access, the GCC compiler alone can target systems as diverse as Windows/Linux through to Sony's Playstation (playstation2-linux.com 2009).

GCC (G++) was used as the main compiler throughout. Valgrind and the GNU Profiler were also used throughout the project. Valgrind provides memory safety analysis and memory leak detection (valgrind.org 2009), and gprof provides performance profiling, identifying bottlenecks[1] (Osier 2009).

The decision was made to avoid any third party dependencies to try to retain portability, with the exception of the Open Asset Import Library (Assimp.sourceforge.net 2009).

4.2 Open Asset Import Library

The Open Asset Import Library (ASSIMP) is a generic model importer. It is provided freely under a BSD license (which is very permissive and if the situation ever dictated would legally allow any needed code to be copied and pasted directly into this project, and secondly, a compiled version could be redistributed freely). ASSIMP requires one C++ dependency, Boost, but claims to be very cross platform, in which case ASSIMP's dependency does not undermine the portability goal of this library. ASSIMP provides two things of importance:

1) A generalised model importer that serves as an abstraction layer to model file formats and the data they contain (Figure 2). This removes from the library the task of parsing file formats, and gives data obtained from models of a number of different file formats in such a way that it can always be relied upon to be in the same form. A standard in-memory model format is a necessity of a generic ragdoll approach.

2) A number of linear algebra primitive data structures, such as vectors, matrices, and some of their associated mathematical operators and functionality.

1 When bottlenecks are referred to within this report, their existence or position is not the result of speculation but instead a result of profiling data given by gprof.

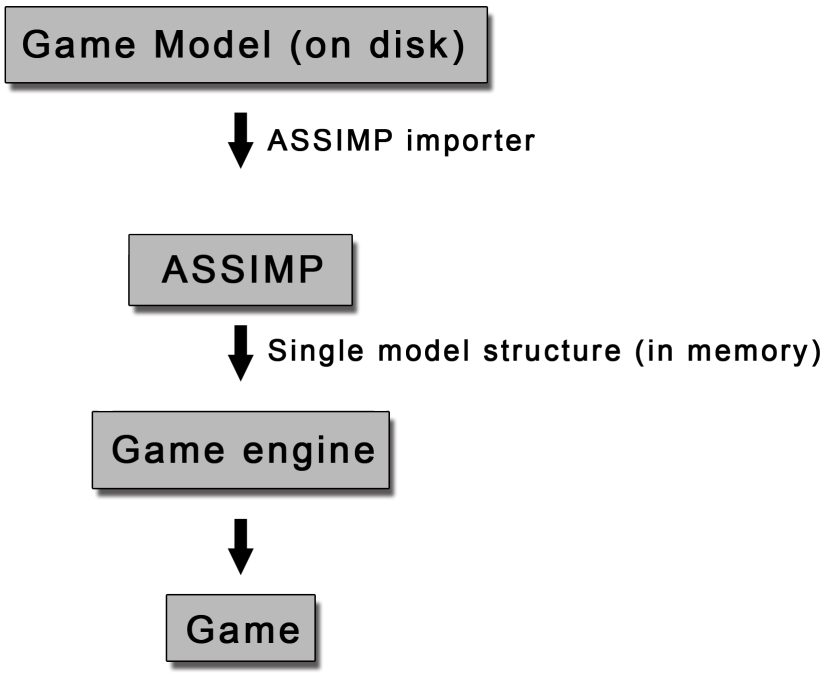


Figure 2: Workflow for using the Open Asset Import Library. It acts as an abstraction interface to the model formats as encoded on disk. An animation library would typically be encapsulated within the Game Engine.

The caveat of using ASSIMP is that ASSIMP is still in beta and contains some instabilities, however it is assumed these will be fixed in future. During early usage, all of the instabilities that were encountered were related to model loading, and could be avoided by avoiding certain file formats. In short, ASSIMP is worth the risk of using because it provides functionality useful to solving the problem. Of this functionality, a non-trivial subset would need to be implemented from scratch in ASSIMP's (or an equivalent library's) absence.

4.3 Project, Time Management, And Methodology

The project lifetime was approximately three and a half months, lasting from 15/06/09 to 23/09/09 (although the deadline was extended from 11/09/09), or 14 full weeks. The first four weeks were spent engaging purely in research. The final four weeks were allocated to bringing together the whole project, and to writing this report.

This left around 6 to 7 weeks for the design and implementation phase of the project.

Due to having no prior familiarity with computer animation, the initial research phase initially covered animation from very basic principles. This then progressed to learning to use the Open Asset Library (and in doing so, becoming familiar with C++), and then into researching into physics/rigid body simulation. Within this period, some time was also spent practically experimenting using short programs written using Python[1].

The 6 to 7 weeks design and implementation time is quite short for a project of this size. Furthermore, given no previous familiarity with the whole topic area there was the considerable risk of making large oversights in the design process, or spending too much time pursuing dead-end approaches. This is to say that there was a risk of losing time as a result of making mistakes that a more experienced animation programmer might regard as being obvious.

For this reason, a prototyping based methodology, evolutionary prototyping, was used for the software side of the project. Prototyping is often used to quickly identify misunderstandings between developers and end users (Sommerville 2004), but the fast results it gives also provide the possibility to quickly highlight developers' misunderstandings of their own approach and algorithms. Evolutionary prototyping makes continual refinements to the software rather than (as in traditional throw-away prototyping) discarding and rewriting it when a working prototype is found; which is the optimal use of time in a relatively short project. The major risk of such an approach is that code is likely to become bloated and unmaintainable and might need rewriting once a working prototype is found, but the fact that flaws are revealed quickly and in such a fashion that they are anticipated to exist makes this a useful methodology for this project given the circumstances.

The software was designed from a high level with appropriate abstraction interfaces, so that any problems that were revealed would hopefully be isolated to their individual modules (classes), but lower level details were not considered during design as they would be best seen as from the point of view of implementation as they arose. A low level design risked predicting details with results that may be a product of accumulative oversights elsewhere in the design. The design is outlined below, in 5 – Software Design.

1 Python chosen because
1) It is a scripting language with high level constructs: it is very fast to write code for, focussing more on the solving the problem than on implementing the solution.
2) It has extensive mathematics library support in NumPy (numpy.scipy.org 2009).
3) Its functional-style list (array) operations make using vectors and matrices easy.

5 Software Design

This section describes briefly from a high level the overall software infrastructure that was employed to solve the problem. This section is intended only to give an understanding of how the software is structured; exact details on the functionality and behaviour given here is provided in the next sections (6 – Keyframe System, 7 – Ragdoll System and 7.2.6 – Inverse Kinematics). Some data structures (those prefixed by ai) are provided by the Open Asset Library, these include the skeleton nodes (given in a hierarchical tree-like linked list), matrices and vectors, and animation data. Because of time constraints these are used indiscriminately within the library with no wrapping or abstraction around them. The Open Asset library uses a right handed co-ordinate system, and quaternions to represent pure orientations. These are also used throughout the animation library.

The software is divided into three major sections, the Animator, KeyframeController (KFC) and RagdollController (RDC) class; these are abstractions and make the sections agnostic to the implementation of each other. Later in implementation was also introduced the InverseKinematicsSolver (IKS) class, which can be thought of as roughly equivalent[1] to KFC and RDC .

The Animator class contains an instance of each of the above classes, and is responsible for switching one on and the other off at the appropriate times, and storing the overall transformations matrices of each node. The RDC and KFC classes also hold a set of transformation matrices, these are specific to their own calculations and are expected to be combined as appropriate by Animator. Appendix 2 – UML Class Diagrams contains a high level class relationship.

The main user-interface to the library, for simple functionality[2], are the methods,

```
void Animator::Update(float dt)

void Animator::StartAnimation(const char *name)

void Animator::StopAnimation(const char *name)

void RagdollController::RegisterForce(const aiNode *node, const aiVector3D
&force, float time)
```

The Animator's update method is responsible for advancing the state of the entire animation library (including keyframe, ragdoll and IK) and is intended to be called each frame with the time elapsed in seconds since the last frame.

1 But only superficially! The RDC and KFC classes contain their own data (and represent Objects in a literal physical sense), whereas the IKS is a wrapper around the RDC to provide an interface that acts on the underlying ragdoll skeleton. These are explored more in the coming chapters.

2 Inverse Kinematics functionality is not simple, and is detailed in its own section, 7.2.6 – Inverse Kinematics

The RDC class is responsible for containing the necessary components for ragdoll physics simulation. It provides an update method, which advances the simulation by a given time-step. The Animator class is responsible for calling this in its own update cycle. The IKS also works in the same way, with respect to updating.

The KFC class is memory-less between frames. It is responsible only for taking an animation and time (or a set of animations and times), and calculating resultant transformation matrices. Which animations should be played, and at what times in their cycles, is determined by the Animator class.

Separation of the two main systems (ragdoll and keyframe) gives a robust software design and allows the implementation, and behaviour, of one system to be independent of the other, meaning that problems in one will not affect the other, and the exact inner workings of one may be changed without disruption to the other. The disadvantage this design brings is a redundancy in data (and calculation); for example, there are no less than three places in which a full set of node-transform matrices are stored. However, the superior software design is worth the increased memory requirements.

Pseudo-code for the general work flow of the animation cycle follows:

```
Animator::Update(float dt)
{
  if (keyframe_active)
  {
    keyframe->NewFrame();
    for each animation in active_animations
    {
      keyframe->PlayAnimation(animation, time, weighting);  // time is
                                                 specific to the animation
      keyframe->FinalizeFrame();
    }
  }
  if (ragdoll_active)
    ragdoll->Update(dt);
  if (ik_active)
    iksolver->Update(dt);
  CalculateOrientations();
}

Animator::CalculateOrientations()
{
  orientations.clear();
  for each node in model
  {
    orientations[node] = CombineData(
      ragdoll->GetOrientation(node),
      keyframe->GetOrienation(node)
      );
  }
}
```

6 Keyframe System

As mentioned in the design overview, the keyframe functionality is split between the KeyframeController and Animator class. The Animator class is intended to be slightly 'omniscient' compared to the KF class, and keeps an indexed database of known animations (which are registered by the library caller, and are given in the form provided by ASSIMP). The internal index used is simply an integer (size_t) which uniquely describes the animation, but so as to avoid the caller needing to keep a lookup table of which animations' ID is which, there also exists functionality to define an animation by a text name (char*). For example, the caller can register an animation under the name “walk” or “run” and refer to it by that name thereafter. All related methods are overloaded to provide access by either index or name. The caller can thus pass the current state of the character (i.e. “walking”, “running”, etc.) as determined elsewhere in the program by an English word.

6.1 Keyframe Blending

Blending is an important part of an animation library as it provides a simple (but limited) way to combine animation from different sources, with the possibility of favouring one over another, to give a much richer library of animations available to the application.

Blending is an effective technique for similar animations but for dissimilar ones it quickly exhibits artefacts. Suitability of blending is not in any way considered by the library and it is the caller's responsibility to make sure the character is not told to lie down while running. The blend system works by taking the orientation quaternions, the translation vectors and scaling vectors for each animation source, and averaging each property (scaling, translation, rotation) before combining then. Vectors are easy to average; for a set of n vectors, V and their (normalised) weighting factors, w, the average is given by,

$$V_{avg} = \sum_{i=0}^{n} w_i V_i$$

Quaternions are less simple as they are not a simple linear quantity, but repeatedly applying an interpolation process can be used to give a weighted average (Figure 3) of an arbitrary number of quaternions.

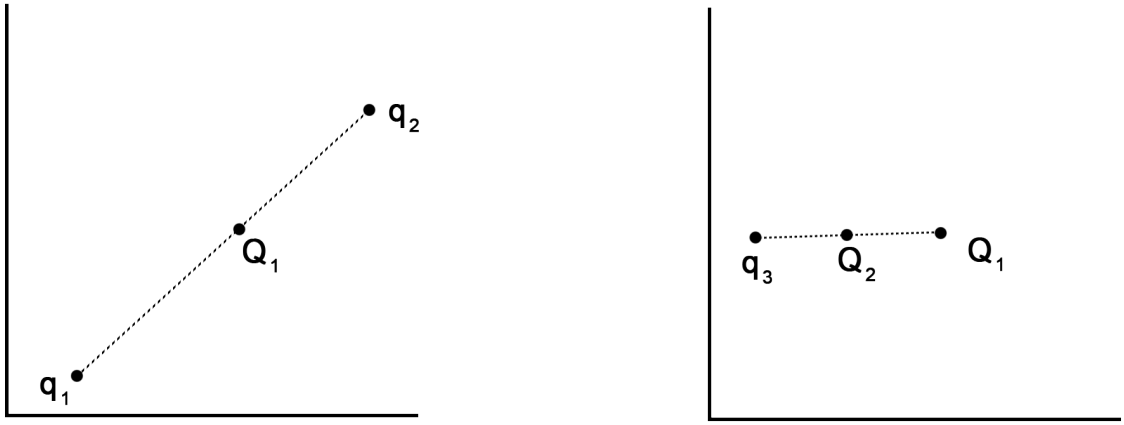


Figure 3: Averaging three elements (q_1, q_2, q_3) using two iterations of an interpolation process. Q_1 represents an average between q_1 and q_2, Q_2 then represents an average between Q_1 (i.e. q_1 and q_2) and q_3. This example is not weighted, but weightings would be worked into the process in the interpolation distance.

Formally, given a set of n quaternions, q, with weightings, w, an average, Q_{n-1}, can be calculated by the recursive process,

$$Q_0 = q_0$$

$$\Omega_0 = w_0$$

$$\Omega_{i+1} = \Omega_i + w_{i+1}$$

$$Q_{i+1} = slerp(Q_i, q_{i+1}, \frac{w_{i+1}}{\Omega_{i+1}})$$

or, less formally, the result of a blend between two quaternions is an interpolation (this library uses spherical linear interpolation, or SLERP (Shoemake 1985) between them both, with appropriate weighting factor. After the first two quaternions have been blended, any further quaternions are blended in successively, using the output of the previous blend as one part of the SLERP input, and the weighting factor is the current weighting divided by the sum of the current and all previous weightings. The result is a weighted average.

The end scaling, translation and rotation are then combined to form a transformation matrix.

A problem with naive blending is that blends will often change the speed at which a character appears to move. For example, in a blend between a running and shooting animation, if the legs inherit 50% of the movement from the shooting animation (in

which they are likely stationary), they will move at 50% of the speed of the running animation. This will desynchronise the movement speed of the character from the speed at which the legs move, thereby making its feet appear to slide.

To circumvent this the library allows weighting to occur on a node by node basis; so the waist downwards may be weighted 1.0/0.0 in favour of the running animation, and the waist upwards may be 0.0/1.0 in favour of the shooting animation (Figure 4).

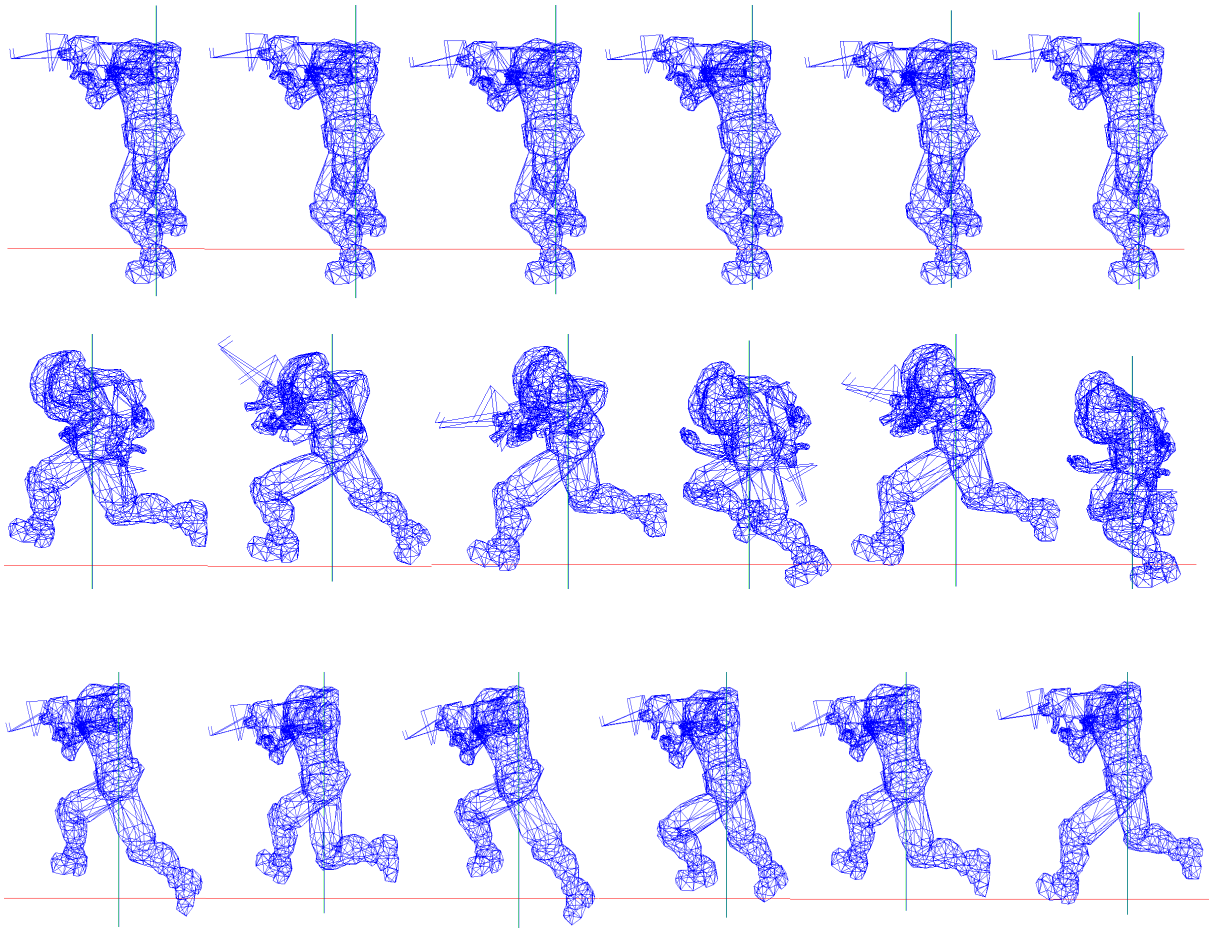


Figure 4: A (headless) Doom3 model demonstrating blending; the first two sequences show (random) frames from the shooting and running animation respectively, and the final sequence shows the two blended together. The blend is as described in the text; the upper half of the body strongly favours the shooting animation while the lower half favours the running animation. The aggressive variation in the leg of the runing animation can be seen to be present (in contrast to the stationary pose of the shooting animation), but retaining the relative constant direction in which the gun is pointed (in contrast to the running animation where the gun's direction rotates through about 180 degrees (compare running frame 2-4).

The library also provides a pair of methods,

```
void Animator::Transition(size_t from,
                          size_t to,
                          float time = 1.0f)

void Animator::Transition(const char *from,
                          const char *to,
                          float time = 1.0f)
```

to transition between two animations over a given time period. When this method is invoked the library applies a blend factor to both the 'from' and 'to' animations, and over the given time period linearly changes the blend factors from 1.0/0.0 to 0.0/1.0. This could be used to give a smoother transition between a run and walk, or a movement to an idle animation (or vice versa).

The Animator class also provides time scaling functionality, as the entire library expects to treat times as being seconds seconds, but an animation might have its time indexes encoded differently (for example, Tiny.x was found to use milliseconds and Doom3's animations used tenths of seconds), and the Animator needs to be able to scale accordingly. Similarly, individual animations can be time-scaled, which has limited usefulness[1] but a scaling factor of -1 would make the animation run backwards which may in some situations be useful.

Another feature provided by the library is the ability to prevent the model from translating from its initial position as a result of the animation. In some animation files (exhibited by those in Doom3), the character often translates from its initial position in a walk or run animation. This might work within some animation libraries, but here, it presents a problem if one wishes to blend animations whose models are in different locations in space. This works simply by fixing the translation property of the root node (or optionally, some other node). This is set via a public member attribute of KeyframeController.

For performance and consistency the Keyframe library includes the possibility of setting a minimum time-step (i.e. it will update a maximum of so many times per second – this defaults to 40 per second). Setting this too low loses a smooth appearance, but allowing it to run unrestrained may use a lot of CPU time better spent elsewhere in redundantly animating the model more minutely than the human eye can appreciate.

1 Animations from different sources might have different time scales, but one would not expect animations from different sources to have been created for a single model.

7 Ragdoll System

The ragdoll system was a large source of challenges throughout this project. Two approaches were pursued over the course of the project as it slowly became evident that the first approach was inadequate. The second approach was an augmentation to address the failings of the first method, so the first method is discussed here.

It is ambiguous in this section whether the words “skeleton” and “model” refer to a rigged model exported from an animation package, or to the internal ragdoll structure. To clarify, model is used only for something that would be created in an 3D-modelling/animation package, and a skeleton always refers to the dynamically created ragdoll skeleton that exists only at the library/program's run time.

7.1 Approach 1

Ragdoll systems are often modelled as articulated rigid body using spring constraints between them; this ragdoll system instead uses a particle system (Jakobson 2001), which is similar in the way forces are modelled but diverges slightly in the structure of the skeleton. The skeleton is a set of linked particles with constraints holding each particle a set distance from at least one other particle, like a stiff-spring. All particles are in world space and the particle system has no need for the concept of local space.

The advantages of this over an articulated rigid body approach are that it is a lot simpler to understand and implement, rotations appear completely implicitly (figure 5), and due to the simple nature of the arithmetic needed on each particle (purely vector addition), it is very fast to process computationally.

The disadvantage lies in the simplicity; the lack of explicit rotations means that rotational data (for the purposes of building the overall model's transformations) needs to be derived from the skeleton or its movements.

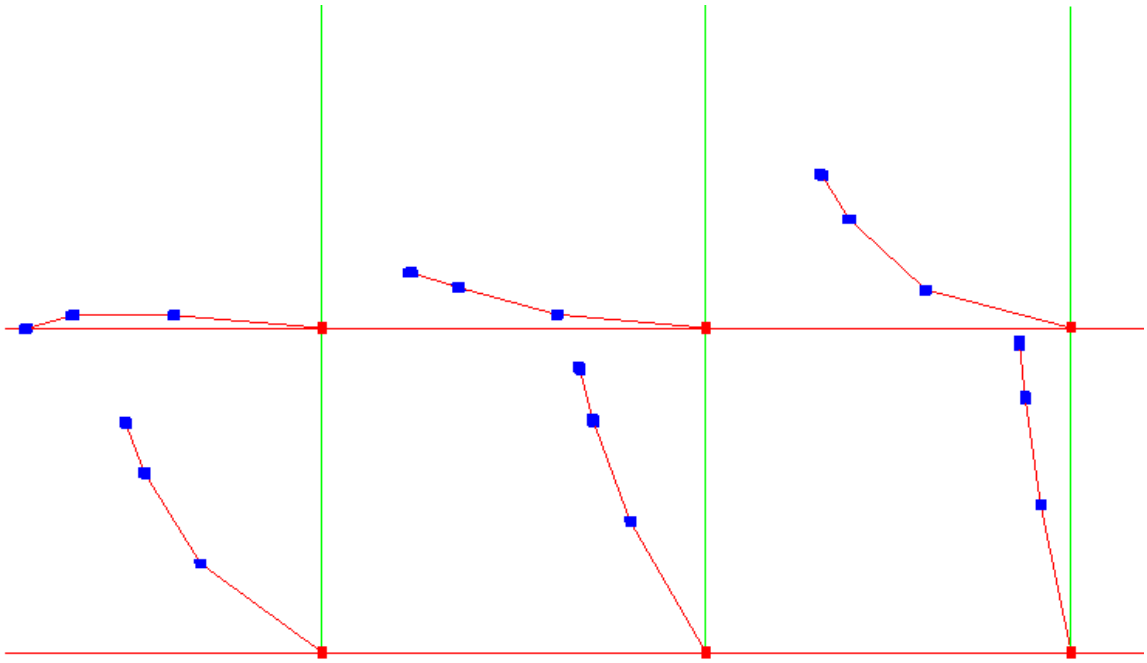


Figure 5: a simple particle system and its behaviour as a force is applied to the end node; the first particle is fixed at the origin, and an upward force is applied to the end-most particle resulting in the rest of the particles being pulled up behind it. This acts much like a metal chain, using a higher number of particles one could model a string or rope. The particle system is surprisingly versatile, it could potentially be extended to model cloth by building a lattice of particles, and complex structures of particles could potentially model deformable 3 dimensional bodies.

7.1.1 Initial Creation Of Skeleton

Initially, the skeleton has to be constructed from the input model. This is done by wrapping boxes (which double as hit-boxes) around each nodes' vertices. Originally it was tried using the middle-points of the top and bottom planes of the box to create a pair of particles (or a line, defined between the particles) which acted as the node. This lead to joints between particles being of non-zero length. This is unacceptable as it leads to artefacts: for example, two particles a = (0,0) and b = (10, 0) have a separation distance of 10 along the X axis. Suppose these particles represent a joint in the skeleton (a connection between two nodes). Suppose the particles are then altered, such that a = (0, 0) and b = (0, 10). Both pairs have a separation distance of 10, but in the case of the first pair the separation is along the X axis and in the case of the second, it is along the Y axis. Whilst a distance constraint[1] specifying a separation distance of 10 would satisfy both pairs of particles, a transformation from

1 The skeleton is held together by distance constraints (a scalar, specifying the required distance between two particles). These are explained in greater detail later, in 7.1.3 - Distance Constraints

the first pair to the second pair would give the joint a stretch of 10 units along the Y axis, and a squash of 10 units along the X axis.

To make the system consider constraints as direction as well as distance (i.e. a vector) would involve a more difficult (and fragile) implementation than altering the node structure so that particles forming a joint are always a zero length from each other. This was achieved by 'stretching' the node definition such that each node's top most particle (parent-connector) is set equal to the parent node's bottom most particle (child-connector).

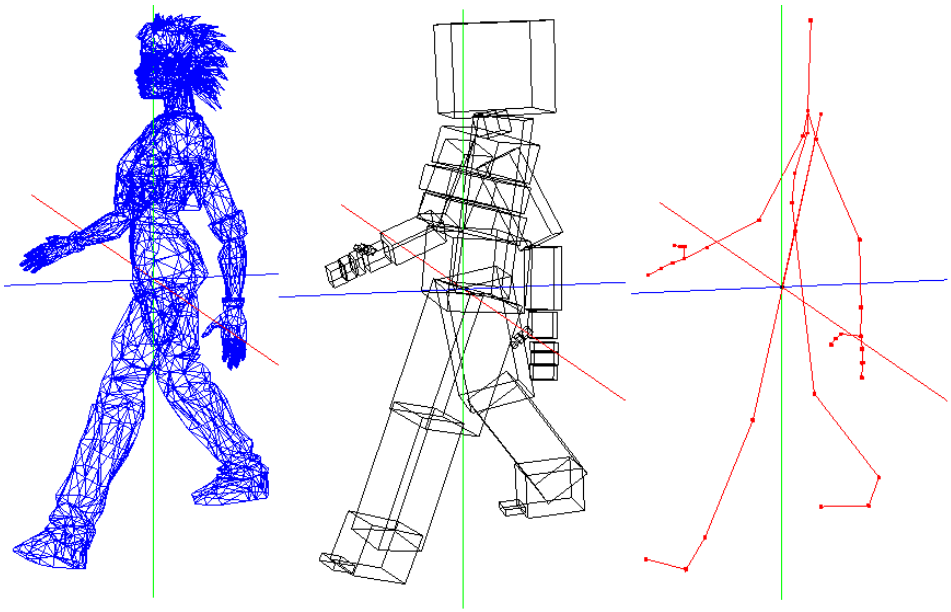


Figure 6: The model, and resultant hit-boxes, and ragdoll skeleton for Tiny.x

7.1.2 Forces

Forces are registered with the skeleton, and bound to a node. Each node keeps a list of the forces acting upon it, and this list is iterated over at each step in the simulation, and its effect calculated. Forces are given a time for which they are active; impacts rarely exert their full force as a one off instantaneous effect but instead as a short push and the library attempts to model such an effect accurately.

A force is modelled in the physical, Newtonian sense of a force, and the visual effect it has upon the particle is a change in position. Newton's second law of motion leads to a relationship between force and displacement, resulting in a chain of differential equations:

$$\vec{F} = m\vec{a}$$

$$\vec{a} = \frac{d\vec{v}}{dt}$$

$$\vec{v} = \frac{d\vec{x}}{dt}$$

Using the symbol set: force F, mass m, acceleration a, velocity v, displacement x, time t.

As a result of the animation library's design, the ragdoll system expects to be called in discrete time-steps, which is congruous with numerical methods for simulating models expressed as differential equations. The most obvious way to solve these equations is to simply calculate the acceleration from the force, then to multiply this by the change in time since the last time-step, add this product to a velocity accumulator, and so forth; this is called the Euler Forward Method[1] for integration, which has the general recursive formula:

$$f(t + \Delta t) = f(t) + \Delta t f'(t)$$

Euler integration is often unpopular within game development communities because although it is very fast it is seen as not particularly accurate or stable . This is not necessarily a correct assumption (Weisstein 2009 “Euler Forward Method”), but with respect to accuracy and stability, there do exist more preferable methods.

Within molecular dynamics two such methods, Verlet integration and Beeman's algorithm, are commonly used because they are quite accurate, stable and fast to compute. This project implements a variation of Verlet integration, common within computer games, known as Velocity-Verlet integration. This is a slight upgrade on the Verlet method as (contrary to standard Verlet) it explicitly considers velocity. Its

1 Also: The Forward Euler Method, and Euler's Forward Method, and simply Euler Integration

derivation is not as simple as the Euler method and is omitted here, but its formula and thus implementation is still relatively simple:

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \Delta t\vec{v}(t) + \tfrac{1}{2}(\Delta t)^2\vec{a}(t)$$
$$\vec{v}(t+\Delta t) = \vec{v}(t) + \tfrac{1}{2}\Delta t(\vec{a}(t) + \vec{a}(t+\Delta t))$$

As can be seen, the two formulas separately calculate x (displacement) and v (velocity). Within both, acceleration is misleadingly referred to as a function of time. For the purposes of implementation, acceleration is calculated from the resultant force acting on the node at that time-step, using Newton's second law.

7.1.3 Distance Constraints

Enforcing distance constraints between particles is very simple: if the distance between the two particles being evaluated is different to what it should be, both particles are moved either closer together or further apart as necessary. Suppose the particles represent a line, the line before and the line after (applying a constraint) are parallel. The following (adapted from Jakobson (2001)) describes an algorithm to apply a constraint between two particles (p0 and p1). The algorithm takes a pair of particles, denoted by n, and calculates a corresponding pair, denoted by n+1, that satisfies the constraints. The pair denoted by the subscript 0 are the particles' initial positions, and are assumed to define the constraint between them.

$$A_0 = p1_0 - p0_0$$
$$A_n = p1_n - p0_n$$
$$\Delta = \frac{||A_n|| - ||A_0||}{||A_n||}$$
$$p0_{n+1} = p0_n + \tfrac{1}{2}\Delta(p1_n - p0_n)$$
$$p1_{n+1} = p1_n - \tfrac{1}{2}\Delta(p1_n - p0_n)$$

The growth/shrinking effect is equal at each end of the line. Consequently, solving one constraint will dislocate both particles, and therefore break other constraints. This is expected, and the constraints are applied to one node after another, then repeated several times; each successive iteration will result in each constraint, on average, becoming closer to being satisfied.

Small errors will be unnoticeable to a human observer; furthermore any particular constraint left unsolved at the end of a frame will (probably) not remain unsolved for more than a small number of frames. For optimisation an acceptable error threshold was set, and during testing a threshold of ±2% gave no visibly wrong results.

7.1.4 Results And Discussion

The results of this approach were initially promising, but as development progressed the approach was showing itself to be unsuitable to be further developed into a full ragdoll system. As well as the areas already listed in this chapter, a collision detection system was half-implemented before the approach was accepted to be inadequate.

On the positive side, the particle system reacted very well in dragging its constituent parts around and for small effects the results were not unconvincing (Figure 7).

For larger effects however the results certainly were unconvincing as the body essentially came 'undone' in that it lost its resemblance to a believable human skeleton. The reasons for this fall into quirks in the hierarchical layout of the models, and problems in synchronising the model with the ragdoll skeleton.

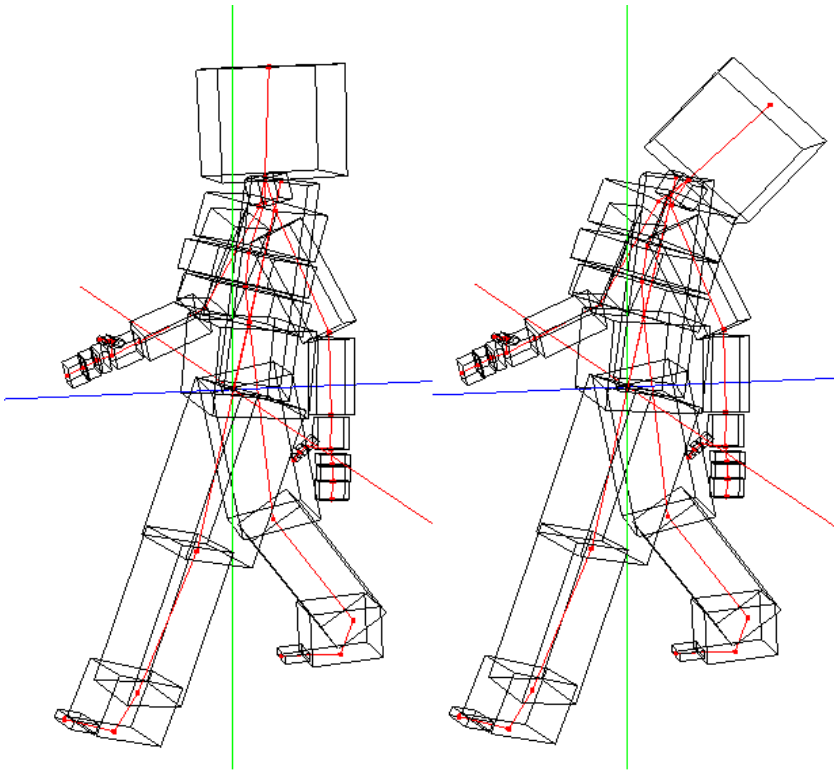


Figure 7: The effect of a small push on the head node. Careful observation will show that the upper body's boxes have been pulled slightly from their original position as a result.

7.1.4.1 Model Quirks

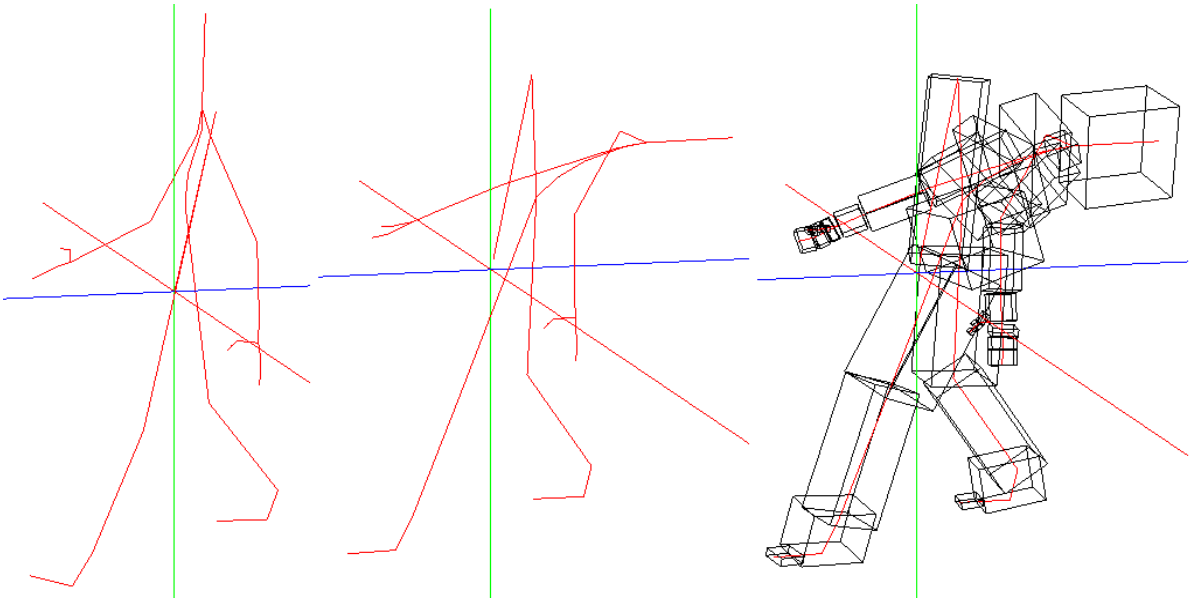


Figure 8: Result of a strong force applied to Tiny's head. The final pane shows the hit-boxes for the same skeleton orientation as the second pane. The skeleton's resemblance to a human being has begun to fail.

During implementation it was found that models were often not arranged in completely intuitive, or logical, ways. Construction of the skeleton for Tiny resulted in a false 'spine' (for lack of a better word) appearing, which was a result of parts (vertices) of her hair being logically a part of her 'pelvis' node; the vertices' bounding hit-box and thus the start and end points (particles) of the skeleton node were in an unnatural, nonsensical position. Figure 8 shows how the unnaturally structured skeleton effectively comes apart when it is manipulated freely; although the skeleton has preserved a roughly human shape, the false 'spine' has not been moved in a natural way. Appendix 4 – Ragdoll Approach 1 Further Resultscontains a screenshot showing the effect of a much bigger force, showing that the skeleton begins to lose its resemblance to a human body as the ragdoll is allowed to handle bigger effects.

To (attempt to) circumvent this the system provided the ability to register artificial distance constraints between arbitrary points but these proved insufficient; analogously to the problem detailed earlier (in 7.1.3 – Distance Constraints) of distance constraints of non-touching particles, the resultant behaviour was unconvincing and often resulted in a 'popping' effect where the spine would 'pop' from behind to in front of the skeleton. This effect was exhibited with a constraint between the neck node and the and the top of the 'spine'. If time had been spent defining a complex lattice of such constraints all across the body, it might have been possible to make the skeleton more rigid and less prone to unraveling, but it is a

fragile solution and it is possible the increased dependency between particles might have made the popping effect more frequent.

Relating to the model structure is collision detection: the 'spine' is inherently inside all the body nodes and therefore colliding with them. Furthermore, as can be seen in the screenshots, the sheer number of nodes makes a lot of collisions very likely to occur after very small movements. To recognise these as actual collisions would give unrealistic results, so the ability to exclude some nodes from colliding with each other was implemented. However, this was not without a cost: the retrieval of node data in general was a performance bottleneck[1], and when the system began looking up which nodes may collide with each other, the bottleneck became a problem and the overall process greatly increased computation time to unacceptable levels.

7.1.4.2 Synchronisation of the model and skeleton

It was known when choosing this approach that the particle system did not provide an easy way to derive node orientations. To calculate them, when performing a translation of a particle, the system observed the 'before' and 'after' vectors between the particles. These two vectors were normalised to ***u*** and ***v***, and the change in orientation was calculated as being an axis-angle pair, ***θ*** and ***A***,

$$\theta = cos^{-1}\left(\frac{u \cdot v}{||u||||v||}\right)$$

$$A = u \times v$$

However, in using this, the implementation failed to keep the model and skeleton synchronised. The resultant rotations on the model did reflect the rough rotation of the ragdoll (evidenced by the hit-boxes shown in the screenshots throughout this section: they are oriented using the data this process derives.), they accumulated 'error' quickly such that anything but very small movement noticeably diverged the model from the ragdoll skeleton. The implementation was checked carefully, so it seems likely that either some part of the method was overlooked, or that because the orientation changes were very small for any individual change, floating point and rounding errors caused rapid accumulation of error. The ASSIMP library provides a method,

```
aiMatrix4x4 &
FromToMatrix(const&aiVector3D&, const aiVector3D&, aiMatrix4x4 &)
```

which creates a rotation matrix that transforms one vector into another through an algorithm described by Hughes (1999). This was also tried on the normalised 'before' and 'after' vectors, but was no more effective.

1 Not speculation: this was found as a resulting of profiling using gprof.

A correction to, or suitable replacement of, the method was not found.

Brown (2009) posits that deriving orientation directly (without using such fragile methods) is possible given some assumption that the model's rigged layout has some geometric properties, but for any arbitrary model this assumption cannot be made.

Similarly, the fact that the particle system could not represent a rotational degree of freedom of a node about its own axis was a problem that was never solved, and this resulted in arbitrary and uncontrollable rotations about this axis (visible by careful observation of the screenshots in this section); no method was found for cancelling out this rotation.

7.1.4.3 Performance

Performance was a problem towards the end of implementation. Although manipulation of the skeleton is relatively easy, the sheer number of nodes greatly increased the CPU time necessary to evaluate the skeleton. This was especially problematic with regard to self collision detection. A number of optimisations were employed to deal with this, the first concerns the data structures in which the ragdoll skeleton is stored, and is detailed below (in chapter 8). The second was to simply set the ragdoll system to use a fixed time-step, which means that the ragdoll system is not evaluated at every frame (an update frequency of 40 times per second is the default value in the library). These two alterations made a significant difference and are universally applicable to any ragdoll model.

7.1.5 Conclusions of Approach 1

The results highlight two important lessons:

1. The 1-dimensional particle system is inadequate for a general approach because it cannot, in a general sense, be relied upon to provide a method for deriving rotational information from the particle system.

2. A complex skeleton is not well suited to being pulled around like a ragdoll for reasons of both realism and performance. Performance can be optimised, likely to within acceptable levels, but to try to patch it to produce more natural looking behaviour is a difficult problem.

The first lesson can be worked around by changing the skeleton system, but the second presents a barrier when trying to find a general approach to ragdoll physics. The best way to approach the problem is to remove it: a ragdoll does not need the (for example) 47 nodes that Tiny.x defines. Retargeting animation data between

skeletons is an established area of computer animation and the overall idea is applicable here. By retargeting the complex model to a much simpler ragdoll, the effects of ragdoll physics could be applied to the complex model while avoiding most of the model's complexities and quirks.

This is explored further in the remainder of the project.

7.2 Approach 2

After finding failure in the first system and evaluating why those failures came about, a different approach became evident. Coinciding with the acceptance of the first system's inadequacy, the discovery of Rosen's (2007) paper was made, which makes similar observations about a 1-dimensional particle system as well as presenting similar ideas on how the problems could be fixed. At this point roughly 8 weeks remained of the project and this was felt to be long enough to build another system, but several short-cuts were taken. These are mentioned where relevant in the following chapter.

The second system is an augmentation of the first and builds on its shortcomings. The first addition was to represent nodes as two-dimensional shapes instead of lines. A 3D approach could have been employed, but the added complexity of doing so was a deterrent: this would have meant increased CPU load, increased difficulty of implementation, and more difficulty in deciding on node-to-node connection points. A 2- (or 3-) dimensional approach allows rotational information to be derived from the skeleton. The second major change is that the skeleton is no longer built directly from the input model. The exact sizes of the nodes are dynamically determined from the input model, but its architecture is static containing only upper/lower body, upper/lower arms, upper/lower legs and a head; it is intended to represent a biped but there is no reason a similar quadruped such as a dog could not be modelled with the same set of nodes. Any other nodes are regarded as superfluous for a ragdoll simulation, and by keeping the skeleton simple, the risk of unnatural behaviour is reduced because there are fewer ways in which it can occur. The input model is retargeted to this fixed skeleton, the simulation is run on the fixed skeleton, and at each time-step the transformation data of the skeleton is retargeted back to the model.

From a software point of view, the existing Ragdoll class now acts as an interface to a new class RagdollSkeleton. It is a container and handles miscellaneous node data that doesn't logically belong in the skeleton, and more importantly, it is responsible for performing the retargeting process. Within RagdollSkeleton objects are a set of RagdollSkeletonNodes objects. The original RagdollNode is still used to hold some data, the reason for this is that completely rewriting the architecture from scratch would have taken too much time.

7.2.1 Derivation Of Orientation

Using two- instead of one-dimensional shapes to represent a node results in its orientation being derivable. Rosen (2007) partly outlines the steps to do this, but provides no argument towards its correctness, which is not immediately obvious. The process, for a shape with three points, (p_0, p_1, p_2), is:

$$A = \frac{p_1 - p_0}{||p_1 - p_0||}$$

$$B_{temp} = p_2 - p_0$$

$$B = \frac{A \times B_{temp}}{||A \times B_{temp}||}$$

$$C = \frac{A \times B}{||A \times B||}$$

$$M = \begin{bmatrix} A_x & B_x & C_x \\ A_y & B_y & C_y \\ A_z & B_z & C_z \end{bmatrix}$$

From this three axes are derived; the first is the line between two points on the shape. A temporary second is constructed by taking a different line between between two points. A real second is calculated by the cross product of those two (giving a line which is orthogonal to axis1), and then a third is calculated by the cross product of those (giving a line that is orthogonal to both previous axes).

The resulting normalised vectors form an orthonormal basis that describes a rotation from the standard basis, {$\mathbf{e}_x$, $\mathbf{e}_y$, $\mathbf{e}_z$} to a co-ordinate system defined by the node orientation. These are put into the columns of a 3x3 matrix to give a rotation matrix that represents the node's orientation (although the system actually uses this as an intermediate step to calculating an equivalent quaternion).

The absolute orientation of a node is not of particular interest, only the orientation that transforms the node from its initial position (at the time the ragdoll was invoked) to its current position. This value represents the rotational transformation that the node has undergone since the start of the simulation. This value can then be sent back into the rest of the animation system as it will specify the transformation that the model node should undergo.

This is equivalent to solving:

$$O_1.O = O_2$$

As quaternion multiplication is associative[1], by inspection,

$$O = O_1^{-1}.O_2$$

All rotations are unit quaternions, and a unit quaternion's inverse is equal to its conjugate (i.e. negation of the signs of the imaginary units' coefficients)

7.2.2 Nodes

A node in the skeleton is represented as either a 3 or 4 sided polygon. The different numbers of points are implemented as separate classes as they sometimes behave slightly differently but both are derived from the abstract class RagdollSkeletonNode and make use of polymorphism.

As in the first approach, the node itself is responsible for providing functionality to know which forces have been applied to it and to react to the forces. The node is also responsible for calculating its own orientation data (see 7.2.1).

The shape of each node is enforced at each time-step by distance constraints (as in 7.1.3 – Distance Constraints) defined between each point.

Node to node connections are implemented differently[2]; nodes store pointers to vectors (particles) and therefore two nodes may literally share a particle (by storing pointers to the same location). Joints are therefore enforced implicitly, reducing computation time.

Having the possibility of two particles per joint means that there can be formed 'hinge' joints. These are particularly useful at the elbows and knees of biped character models. The distance constraints implicitly dictate that the knee/elbow may only rotate through the axis represented by the hinge. Further, from a quaternion can easily be determined an axis-angle rotation, so in the case of the lower-leg/forearm the angle can easily be used in a comparison to enforce that elbows and knees cannot rotate to unnatural extremes. It is more difficult to enforce angular constraints on ball joints as quaternions do not provide an intuitive basis for enforcing limits on rotation, and converting from quaternions to Euler angles is not an easy process as

1 The Ragdoll system uses mostly quaternions but associative multiplication is also a property of matrices. i.e. ABC = (AB)C = A(BC).

2 In approach 1, a particle was only ever a member of one node (although two particles might share the same spatial position) and joints were implemented as distance constraints between two particles.

there exist some possible singularities in the conversion. As a result angular constraints on ball joints were not addressed in this project.

The library implements some constraints on the angles; these are enforced by rotating in the direction of nearest limit if the node is outside of the allowed limits. It was found that soft-limits provided more pleasing behaviour than hard-limits. A hard-limit simply limits orientation so that it cannot go past a certain limit. A soft-limit is one that can go beyond its limit, but it will try to rotate back to the limit. The library allows the user to set the soft-constraint factor (i.e. a speed of rotation) under the preferences API.

7.2.3 Retargeting

One of the two major improvements of this second approach over the first is the constant skeleton. The idea of the retargeting is to create an abstraction layer between the input model and ragdoll skeleton so that the resultant skeletons are all equivalent, thus avoiding many of the unique quirks of models that caused problems in the first approach.

The whole process is conceptually quite simple but the implementation details are less so and warrant their own explanation.

The retargeting process is divided into two parts, forwards retargeting (retargeting/binding the model to the ragdoll), and backwards retargeting (using the ragdoll to apply the appropriate transformations to the model).

7.2.3.1 Forwards Retargeting

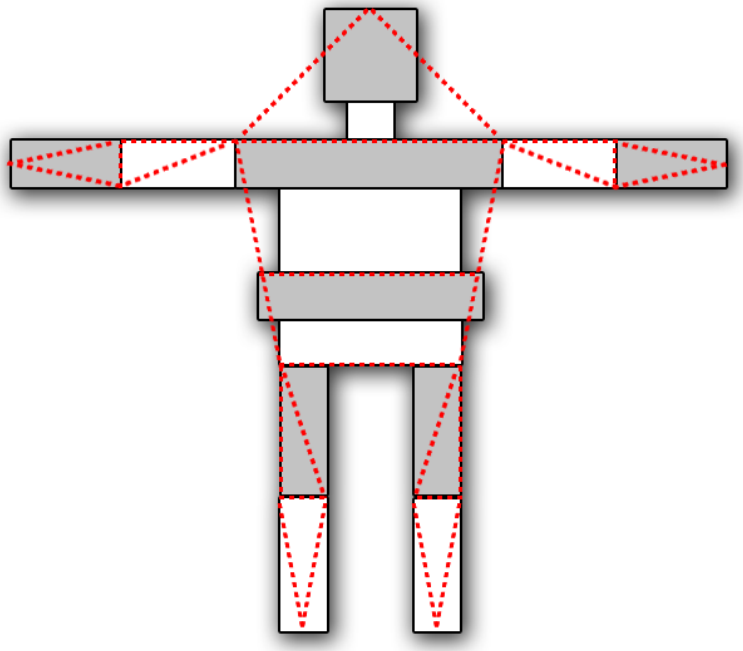


Figure 9: Example construction of a skeleton from a model. Whilst the skeleton's arms and legs each correspond to a single model node, the skeleton's nodes in the head and torso enclose several model nodes, thereby simplifying the skeleton.

The system must be able to retarget from the model skeleton to the ragdoll skeleton and vice versa in an automated manner. The library trades off some automation in favour of simplicity (a fully automated retargeting system is a pattern recognition/AI problem and would be far beyond the scope of this project), and places some responsibility upon the library's caller to provide the library with data on how the retargeting should occur. An API is supplied which the caller should use to 'bind' model nodes to skeleton nodes, and a further property is supplied for flexibility, as explained in this section.
The rough approach the library uses to create a skeleton out of a model is to create bounding boxes around the model's vertices, and to use the extrema in these boxes as the vertices in the particle system. RagdollNode is a class which is leftover from the first approach and contains boxes wrapping the limbs of the model, initially intended to be hit-boxes, which is the data source used in the forward retargeting process. The positions and dimensions of these boxes are employed to create a

skeleton. The resulting skeleton is made of a set of polygons which approximate the input model, as shown in Figure 9.

The skeleton is made out of a set of 3 or 4 sided polygons. Each vertex of each polygon is a 'particle' within the particle system, and in the implementation, the adjacent polygons (as determined by the model's node-tree – a parent is adjacent to its child) quite literally share particles (by having a pointer to the same location), thus resulting in the bottom of the head's polygon starting from the top of the model's shoulders rather than the model's neck. The excess space covered on the neck area is obviously very large which might lead to unrealistic collision detection but this tends not to be a problem because most impacts will be performed at a level much higher than the ragdoll level: they are expected to be performed on the model (from an application's perspective), then the library (ragdoll system) translates an impact on a model node into an impact on the corresponding skeleton node.

Determining the correct vertex of the box to use as a skeleton node vertex is not a trivial task, however. In Figure 9 the model is stood with its arms stretched out the vertices can be determined by their position on the X and Y axis, but an input model may conceivably be in any position. Even if one can assume that arms and legs always point down the Y axis, the X and Z axes are still arbitrarily interchangeable. Getting this wrong would result in the wrong vertices being chosen from the boxes, and thus the skeleton being deformed: if the lower-body node had its top and bottom points switched, due to the constraints holding the particles at the joints together, the joint connecting the upper leg to lower body would occur at around the waist. There is no obvious way to normalise the box such that its points are in predictable positions.

The system *could* attempt to estimate which vertices are which using a comparison between parent and child nodes: by taking the pair of points (one per box) which are closest to each other the system might obtain the correct location for a ball-joint. By adding in additional comparisons to sibling nodes (at least comparing left and right legs, left and right arms), the system might also be able to obtain the points needed for a hinge joint. But this method is both fragile and complex, and in future a model with a slightly unusual layout might not conform to these assumptions. The alternate approach is to force the programmer to specify the operations needed to transform the box such that the vertices are where the library expects them to be. This approach also is not ideal, because it places a burden upon the library's caller, but it has the advantage of a less complex implementation and one that is reliable.

In fact, the best (from the view of the end user of the library) approach would be to combine the two so that the library could estimate the correct skeleton, then allow the programmer to alter anything that is incorrect, but there was not the time to implement this. Thus, the 'rotation' of the box must be specified by the programmer.

The rotation is not literally a rotation: a box is a graph of 8 nodes, each connected to three other nodes. A rotation would then be defined as a permutation of the tuple that

preserves the first-order relations in the graph. This mathematically rigorous explanation however is not fully intuitive but is simply clarified with a diagram (figure 10).

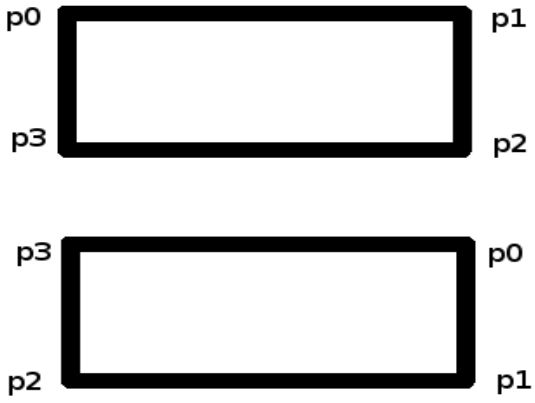


Figure 10: Permutation of (p0, p1, p2, p3) to (p1, p2, p3, p0). The result is a 'rotation' of the node labels. It does not affect the shape nor to which nodes each node is connected, but it is intuitive to think of it as being a clockwise rotation by 90 degrees.

In figure 10 there is only one axis about which the box can be rotated, but in in three dimensions, there are two. The API (optionally) takes a list (std::vector) of BoxRotation structs[1] so that this rotation can be performed before binding.

Determining the exact values required for this is inconvenient, but it is likely that each node for any given model would need the same rotation, and the process would only have to be performed once, furthermore this is the only way by which the library allows a truly generic binding process that does not require alteration of the animation library's source code for it to be suitable for versatile use.

Bringing this together, the library's interface to this is a single method,

```
RagdollController::AddBinding(const char*,
                              const RagdollSkeletonNode *,
                              std::vector<BoxRotation>)
```

that the programmer should call to specify the model node's attachment to the RagdollSkeleton. This should be done for each node in the skeleton. For any skeleton node, the order in which model nodes are bound is important: the first and last are used as the extrema from which the ragdoll's nodes' dimensions are derived. The intermediate nodes are recorded (for the purposes of translating an impact on the model to an impact on the ragdoll skeleton) but they have no geometric effect.

Each call to AddBinding() should be written within a single function, although the application itself should never need to call it (this will be done by the library). A pointer to this function is then given to the library. The prototype is defined as:

```
(void)(RagdollController *ragdoll)
```

1 struct BoxRotation holds a pair of members:
int axis ∈ [0, 1]
int amount: the number of rotations to perform (modulo 4, i.e. setting this to 4*a + b with a, b ∈ ℕ is equivalent to setting it just to b)

It is necessary for the library to hold a function pointer because the library creates multiple skeletons for the purposes of blending ragdoll animation with animation from pre-defined sources (explained in 7.2.5 – Ragdoll/Keyframe Blending), each of which needs binding, so the library needs to be able to invoke the binding process on its own.

A short sample of such an implementation might look like:

```
void Bind(Animator::RagdollController *ragdoll)
{
  std::vector<Animator::BoxRotation> v;
  BoxRotation b(0, 1), b1(1, 1);
  v.push_back(b);
  v.push_back(b1);

  ragdoll->AddBinding("Bip01_Spine", &ragdoll->skeleton->lower_body, v);
  ragdoll->AddBinding("Bip01_Spine1", &ragdoll->skeleton->lower_body, v);
  //etc
}
```

This would tell the library that the lower_body node is composed of the model's nodes Bip01_Spine, and Bip01_Spine1, and before they are bound, they should both be be rotated once in the first axis, and then once in the second axis.

This results in a considerable, but one-off inconvenience for the programmer, but it provides a powerful interface to the library allowing skeletons to be created from diverse models without having to tailor the library's source code.

Three examples of models and their generated skeletons are shown in Appendix 3 – Model To Ragdoll Skeleton Retargeting Examples.

7.2.3.2 Backwards Retargeting

Backwards retargeting is the process of building the actual model from the results of the ragdoll simulation. To do so, the model's global transformations are built recursively from the root node using local transformations.

A node in the skeleton may potentially enclose several nodes in the model (as explained in 7.2.3.1), so for any skeleton node. Each node in the ragdoll skeleton has a set, {S}, of model nodes which are bound to it. Henceforth, the member of S with the shortest distance (in the node hierarchy) from the model's root node will be referred to as the 'top' node.

In general, for every skeleton node, only the top most node will need to consider the skeleton node's orientation, as the child nodes will inherit this[1].

At each time-step the ragdoll provides functionality to retrieve the 'change in rotation' (section 7.2.1) and 'change in translation' since the ragdoll was engaged. Each node in the model has a rotation and translation at the time the ragdoll was engaged (this will usually be derived purely from the keyframe stream); an initial value. Thus, for a 'top' node, the local rotation (quaternion or matrix), R, is the original rotation to which is then applied the ragdoll's change in rotation:

$$R = \begin{cases} R_0 * \Delta R_{ragdoll} : node = top \\ R_0 : node \neq top \end{cases}$$

The translation property is slightly different. The root node's translation is inherited from the ragdoll simulation (this will translate the model to the same point in the world as the ragdoll skeleton), but for all other nodes it is hoped that the rotation is sufficient[2]. This approach is used because deriving translation information perfectly from the ragdoll skeleton is not trivial and is prone to small amounts of error (due to the fact that some nodes, those with ball-joints, don't have a well defined axis of rotation, so one cannot simply look at an arbitrary point on the rotation axis and take the difference between its initial and current position as being a translation), which would accumulate when applied down the node hierarchy. The local translation (vector), T, is given as:

$$T = \begin{cases} T_0 + \Delta T_{ragdoll} : node = root \\ T_0 : node \neq root \end{cases}$$

Scaling is not allowed at the ragdoll level and therefore is always set to its initial value.

1 It is also possible in rare situations that two sibling nodes represent the 'top' level in which case this system will not work; this is a strange situation that would probably never occur in sensible usage so this has not been addressed.

2 The ragdoll rigidly enforces that its nodes may not become detached from one another, and therefore the local position of a node is just a rotation from the end-point of its parent node.

7.2.4 Collision detection

7.2.4.1 Self Collision

The ragdoll system provides simple collision detection with itself. Each node has an associated scalar which is equal to the greatest distance from the node's barycentre to any of its points (referred to in the source code as a 'poking potential'). This is used to determine whether nodes are in range of touching each other in order to narrow the search space before checking for intersections between objects (Hennix et all 2003). The nodes that are found to be in range are then checked by using each vertex of one node as a ray which is tested for intersection with the other node using ray/triangle intersection methods (Ericson 2005). This method is fast to execute and provides the point of intersection; the collision response is to set the endpoint of the vertex to the intersection point so it no longer intersects, but touches instead (and its length is later enforced by constraints).

7.2.4.2 Environment Collision

The library itself does not provide collision detection with the environment. It is assumed that in a full application the collision detection would be handled by a dedicated subsystem, and it would be far more appropriate to handle this there than to build a collision detection system into the animation library. Further, collision detection is a considerable problem in its own right and there is simply not enough time to implement it sufficiently well to be a universally adequate system.

Instead, the library expects collision detection to be implemented externally and a set of function pointers should be given; the corresponding functions are expected to provide various collision detection related functionality: Each ragdoll instance expects to register (on creation) and unregister (on destruction) its nodes with the collision detection system, and it also expects to be able to determine whether a certain point is on the floor (or other gravity resistant surface).

The functions expected to be implemented are:

Register:
```
(void) (RagdollSkeletonNode*)
```

Unregister:
```
(void) (RagdollSkeletonNode*)
```

Determine whether a point is on the floor (the error argument is an 'acceptable error'):
```
(bool) (aiVector3D *point, float error);
```

How the collision detection system is implemented internally is entirely irrelevant, but it is expected that it will keep references to the nodes for as long as they exist. Thus the library guarantees that no element inside a RagdollSkeletonNode will be destroyed prior to the unregister function being called. However, after the unregister function returns any access to those references will be unsafe.

A simple collision detection and response system is provided in the demonstration program, but implementing one that fully provides realistic response was beyond the scope of this project.

7.2.5 Ragdoll/Keyframe Blending

The most obvious solution to merging two sources of animation data is by simple animation blending; where the data from the sources is essentially averaged (described in section 6.1 – Keyframe Blending). Wrotek et all (2006) asserted this to be insufficient and experimental results arising from implementing this during the course of this project support their arguments: using this method introduces many unrealistic artefacts for anything but very subtle changes between the two data sources. Furthermore it means that after averaging the sources, the model's position in space is likely not the same as the ragdoll skeleton's (weighting 50/50 between the ragdoll and keyframe will yield a result translated exactly 50% of the way from the keyframe's centre and the ragdoll's position), which means that collision detection (on the RagdollSkeleton) will appear to react far too early.

Weighting on a node-by-node basis (described in 6.1 – Keyframe Blending) between ragdoll and keyframe goes some way to address this but the body still distorts (stretches at joints were noticeable) and overall this method of blending is simply insufficient. The overall problem is that such operations are a blunt use of mathematics, which do not effectively consider the 'human' context of the character's skeleton.

A better way is to build at each time-step another, different ragdoll skeleton from the combined data of the previous frame, and use that as a 'target': that is to say, to have the ragdoll skeleton interpolate towards that skeleton. This is essentially what the Dynamo system describes (Wrotek et al 2006) but it results in a much simpler concept in this system because the interpolation is between vectors only; the skeleton is given a 'target' skeleton and its particles simply need to interpolate towards their corresponding targets.

The usual angular and length constraints are applied throughout so the skeleton remains human looking, although whether its motion follows a realistic human pattern is harder to enforce (and has not been addressed). The danger of unrealistic motion paths is alleviated somewhat as typical paths between the model and its target will always be quite short due to the fact that when the skeleton is a large distance from its target it is likely unbalanced and in pure ragdoll mode (and

therefore not trying to interpolate to a target), but it does mean that once a character falls, it will be unable to rise again realistically.

The interpolation is a relatively strong effect which directly counters the application of forces to the skeleton. This must be considered, because it could result in the skeleton being immovable with respect to impacts. In this system, a 'stun factor' is introduced, represented as a real number $\in [0,1]$. This is a scaling factor on the speed of interpolation; at 0 the model is fully stunned and makes no attempt to follow its targets, and at 1 the skeleton interpolates towards its targets at some arbitrarily chosen maximum speed. The library has to keep track of multiple stun effects so they can stack up and so they can be treated individually (so that one stun effect coming to an end doesn't automatically end all other effects), and therefore needs to keep a list (whose elements are examined to calculate the overall stun factor).

7.2.6 Inverse Kinematics

Inverse kinematics (IK) is an area of animation borrowed from robotics. Suppose in a sequence of linked nodes it is desired that an end-point (effector) of a node should touch some defined point in space. Loosely, the role of an IK solver is to determine the necessary positions and orientations of each node in the chain for that goal to be achievable.

IK was not initially intended to be a part of this project as the library was complex enough already, but the particle system skeleton naturally models a simple IK solver and without a lot of work a layer was added on top of it to provide an API to IK-like functionality.

IK is a problem that is not always best solved instantly because (in computer simulation especially) the skeleton should iteratively approach the solution rather than instantly assume it. Elias (2000) describes a visually intuitive approach to IK involving pulling the limbs in an articulated chain towards a target. IK is a deep field of research and much more robust methods are known than this, but Elias's method fits perfectly into the existing ragdoll system. The IK method used is not an exact replication of Elias's, but is visually similar: the motion is powered by a pulling force solely on the effector, in the direction of the target relative to the effector's current position.

For a position, P, and a target, T, the direction, D, is a unit vector calculated simply by:

$$D = \frac{\vec{PT}}{||\vec{PT}||},$$

to which scaling coefficients (derived from the speed) are applied to calculate the distance the effector moves on any given iteration. Moving the effector breaks the distance constraints on the skeleton and therefore forces the attached nodes to be pulled along. Therefore, whilst the effector's node is an active node with powered movement, the nodes to which it is attached are 'passive' and being moved only as a result of the skeleton constraints. This is not at all analogous to real human muscle behaviour, but provides an effective illusion.

As well as active and passive nodes, there also exist inactive nodes. In the context of the full ragdoll system, the effected nodes will also be attempting to interpolate towards their keyframe (or bind) positions. To this end the IK system automatically cancels the interpolation processing on the affected node, and allows the caller to specify how far up the node hierarchy tree the effect reaches (so if an arm is modelled by two nodes then calling it with '1' would make the upper and lower arm movable, but the shoulder stiff as far as the IK solver is concerned).

The interface to the IK system works in a similar manner to the interface to the keyframe animation system: the caller registers an IK target with a limb, and in return receives an ID uniquely identifying the target, which may later be used to modify the target.

IK itself is often only half the problem, however. As well as making a character reach out and touch something, the successful execution of the action will generally mark an 'event' the application is likely to wish to handle. Post-behaviour can be specified in two ways to the library. The first is an optional callback function to be executed when the IK target is reached (or the effector is as close to its target as possible); the callback receives a pointer to the Animator object, the ID of the target, and whether it has successfully reached its target[1]. This allows advanced behaviour to be specified by the caller, for example the callback could trigger modification of the game environment (e.g. moving an object), or it could invoke a reaction in another character.

The second post-arrival behaviour method is simply a set of common instructions (defined in the IKBehaviour enum) to tell the node after reaching its target (or getting as close as possible), to:

1. Go limp (cancel the IK solver's effect)
2. Stick to that point (keep the IK solver fully active)
3. Retain the resultant orientation but make no attempt to stick to that absolute point in space[2]

1 If this is 'false', it means the effector got as close as possible, but was restricted by something (most likely the target was out of reach)

2 e.g. if an IK target made a character point forward, the character would still point ahead of themselves when they turned around 180 degrees. Contrast this to option 2. where turning around would make the character's arm also rotate to try to point behind themselves.

4. Forwarding to another target, i.e. cancel this target and activate another target[2]

With the forwarding behaviour (defined in the IKBehaviour enum as FORWARD_AT_TARGET) , paths and cycles can be registered. Using two points in a cycle one can synthesise simple articulations such as a hand waving gesture. It was found that repeated cycles looked quite mechanical; to alleviate this the IK solver allows a randomness length to be set (the resulting target is then uniformly distributed within the set of points less than or equal to that distance from the 'real' target). Secondly, the speed at which the limb moves as a result of the IK solver can be set to be modelled by a cubic (or lower order) polynomial over the domain [0, 1] thereby (for example) allowing it to slow down at the end of its path. The variable and its domain [0, 1] corresponds to the distance of the effector from its target over its initial distance. Effects from other parts of the animation system might therefore pull it beyond 1, which is allowed by the library for lack of any better solution – clipping it to 1 could make it appear to warp as the polynomial effectively stretches. The library does not make it possible to define a polynomial that is defined over [0,1] but undefined anywhere outside of that domain, so this is a minor inconvenience rather than a major problem.

An example on using the cycle functionality of the library is given in Appendix 7.3.

7.2.7 Other Considerations

Ragdoll 'jitter' is commonly a problem of ragdoll systems; i.e. the body does not come to rest at the exact right time, and appears to continue shaking as it falls through the floor and the collision detection system moves it back up. One obvious way to try to determine when to switch off the gravitational effect on a body is to determine how far it has moved since the last time-step. The problem with this method is that it is quite arbitrary in deciding the cut off point and this could lead to problems in having bodies stop moving unnaturally abruptly, or, in setting the threshold too low and having the body shake for a noticeable time. Instead a more physically accurate solution is employed: the “is a point on the ground?” collision detection check is used to cease applying the effect of gravity. This is not perfect either, because the effects of gravity remain in the differential equations for a number of time-steps before a damping effect overcomes them and they disappear, but this was not found to be noticeable.

2 This is equivalent to, but more convenient than, using a callback to set another IK target with the same effector. However, the callback approach allows more advanced behaviour to be specified, such as setting an IK target for a *different* node, or having a time-delay.

8 Miscellaneous Library Details

As well as the software structures outlined in the previous chapters, the library also provides a Preferences API and a wrapper class to a hash table. The former provides an extendible method to store arbitrary choices (many constants used in calculations are quite arbitrary and/or are related to the scale of the model. Gravity is the most obvious example of these) that can be accessed and altered by the calling application at run-time if necessary.

The hash table wrapper (AnimHashTable) exists as a performance optimisation. By default, C++ type, the std::map, is used frequently as it is often convenient to store a node(name)-to-matrix (or other structure) lookup table. The map is usually implemented as a self-balancing binary search tree, which has an average lookup time of O(log n). On top of this, some models use a common prefix in front of their nodes' names: for example Tiny.x uses “Bip01_”. In this case, a minimum of 7 wholly unnecessary comparisons need to be made at each point in the tree.

A 'real' hashtable provides more scalable lookups at O(1)[1]. Testing revealed that in the scale of usage this library employs (i.e. about ~40-80 elements in a lookup table) the lookups were approximately twice as fast using a hashtable. Standard C++ does not (yet) provide a hashtable, although some compilers do. A simple wrapper (AnimHashTable) is created which compiles using a hashtable if it is available, or an std::map otherwise, thereby making the code standards compliant.

1 However, this does not provide the full story in itself: big-oh notation gives a relative computation time to the size of its data; it measures computation speed with respect to growth rather than absolute speed. While the the lookup time might be constant with respect to the number of elements in the hashtable, the hashing function's speed is an additional overhead, and this is dependent on the amount of data being hashed (i.e. the length of the key). It *could* have worked out that the hashing time was longer than the average case lookup of the map. Although, in this case, it did not, but it is an important consideration when comparing two algorithms using big-oh notation.

9 Conclusions and Evaluation

This project has covered a number of areas which shall be analysed separately. Each section discusses results, from which follows shortcomings and future work. For clarity a single brief list of the (most important) areas of future work outlined in each section is again listed in section 9.3 - Future Work as well as some miscellaneous points that did not fit into any of the previous sections.

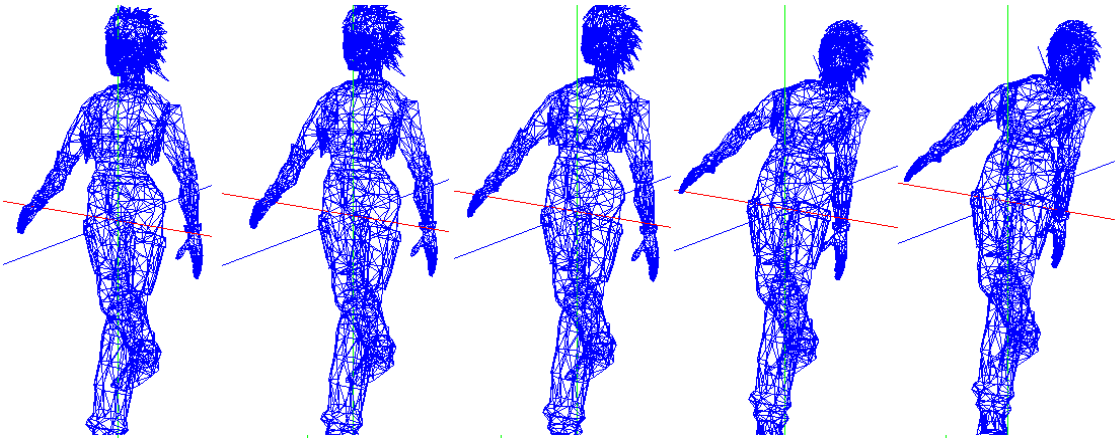
9.1 Discussion Of The Keyframe System

The library as presented handles keyframe animations acceptably, and the controller and keyframe remain a well organised and extendible codebase. The keyframe system provides a full animation library in its own right and could be used as such by an application whether the application also required ragdoll physics or not.

Further work in the keyframing system would best be focussed on a more robust blending system. For example, when fading from one animation to another the library could attempt to time the blend such that the 'swap over' occurs in the period of time in each of their animation cycles where the nodes' translational and rotational differences from each other (i.e. from one animation to the other) is the smallest. A similar idea could be employed in normal blending; to try to synchronise the animations playing so that they appear to align with each other as best as possible.

9.2 Discussion of the Ragdoll System

9.2.1 Particle System



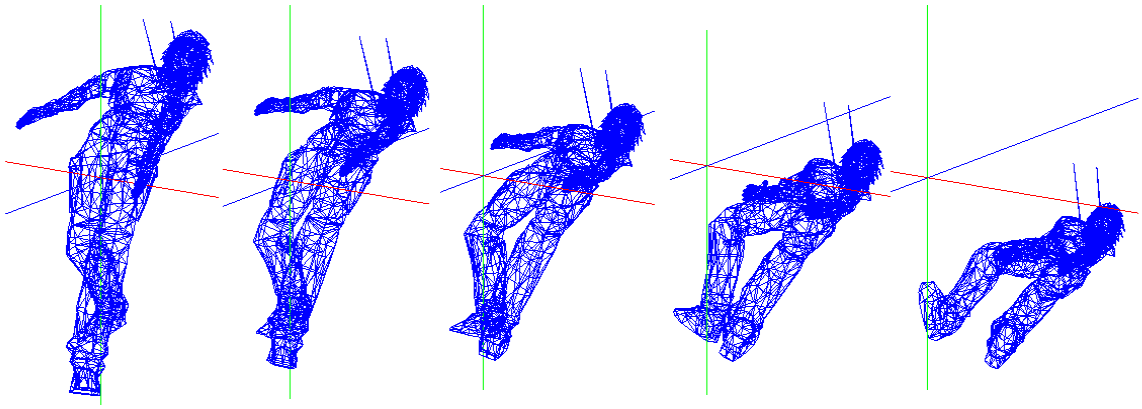


Figure 11: Tiny being pushed gently until she falls backward

On the positive side, the 2D particle system was a vast improvement over the 1D system and gives more flexible and believable human behaviour. Secondly, retargeting to a standard skeleton also proved an effective solution to problems encountered earlier in the project. It does not fix *all* problems; the issue that part of Tiny.x's hair is attached to her pelvis (detailed in section whatever) still causes artefacts, some of which are visible in the screenshots (Figure 11) throughout this report. However, the library's inability to perfectly handle such illogical details has to be accepted as a reasonable limitation.

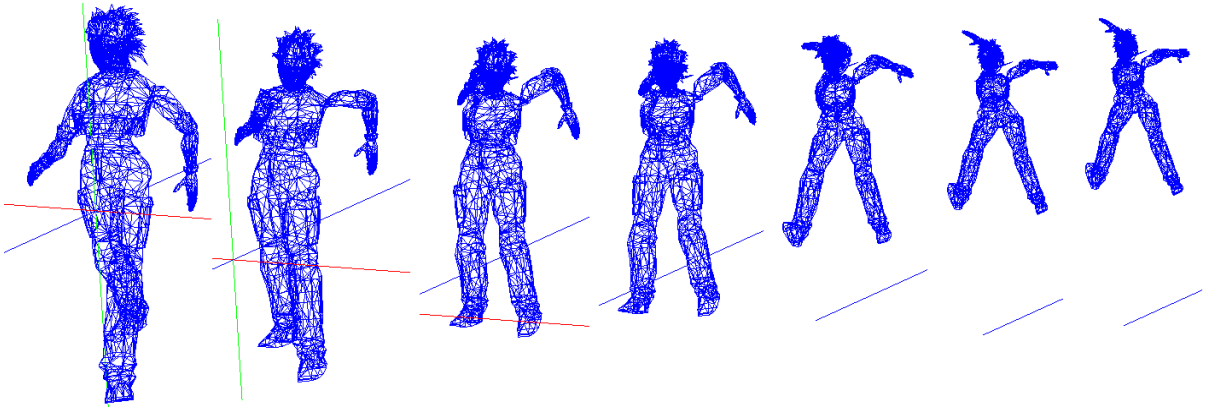


Figure 12: Tiny being sent flying from an explosion effect

In terms of ragdoll simulation, the results are very mixed. The system responds well to explosions by sending the body flying in a believable fashion (Figure 12), and it also works well for pushing bodies around (Figure 11), but fails to display such believable behaviour for less extreme ragdoll effects like a character simply going limp and falling (straight) down. The reasons for this are debatable: The overall ragdoll system is a complex arrangement of a lot of different algorithms many of

which are intertwined. Within some there are a number of arbitrary choices (damping coefficients on velocity/acceleration, relative effects of various constraints, etc), and a lot of them have direct consequences on other parts of the ragdoll system. Unfortunately this makes it difficult to discern whether more advanced behaviour is an inherent limit of the straightforward particle system approach, or whether the software just needs reorganising before the different parts can be tweaked independently of each other.

The fact that the system does provide satisfactory behaviour when a model is sent flying through the world, and that it handles small push effects very well, indicates that there is potential within the method. Furthermore, the fact that it allows such simplicity in the IK and keyframe/ragdoll blending areas, and that performance is so high, mean that it is a very valuable approach to ragdoll physics, assuming that it *could* be made to work better in other areas. Therefore, further work on this approach to determine whether it could be improved in its handling of some effects would make a very worthwhile area of investigation.

Any further work on this system, or in implementing the same functionality in another system, would best be started with a restructuring of the ragdoll subsystem's codebase. A basic proposition of such a layout is given in Figure 13, and had time not become scarce towards the end of the project, the ragdoll system would have been rewritten in this manner.

The idea of a particle system could also be generalised and separated from the idea of a skeleton such that this animation library would be suitable for modelling other particle animation effects, like cloth simulation.

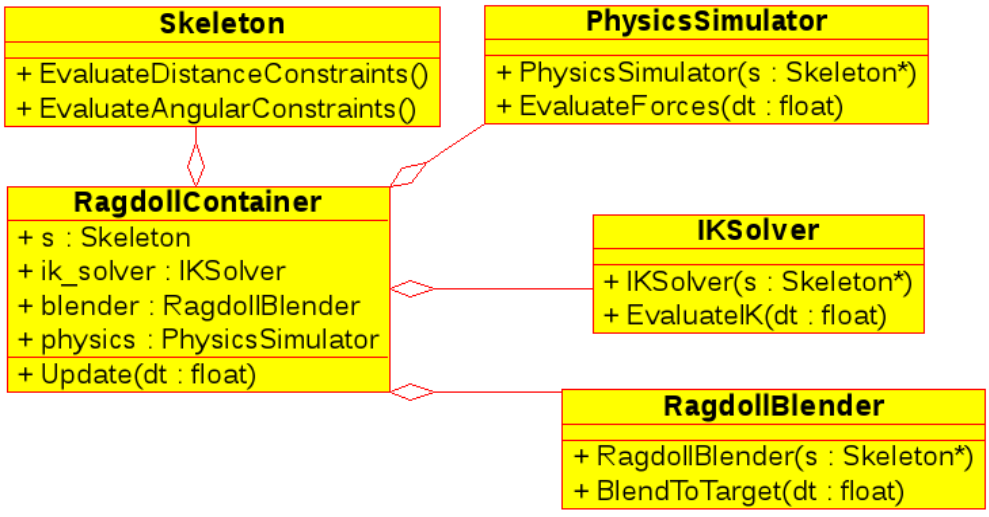


Figure 13: A strong layout for a ragdoll system, focussing on separation of the individual parts. RagdollContainer is the overall controller class, and provides most of the external interface to the rest of the system.

Another related improvement that would be worth investigating is better collision detection and response. The first ragdoll method employed 3D hit-boxes. The second method initially used only the two-dimensional skeleton for collision detection and was intended to be expanded to make use of hit-boxes similar to those of the first approach, but due to time constraints these were not implemented in full. More accurate/realistic collision detection would definitely have a positive effect on the overall quality of the simulation.

9.2.2 Inverse Kinematics

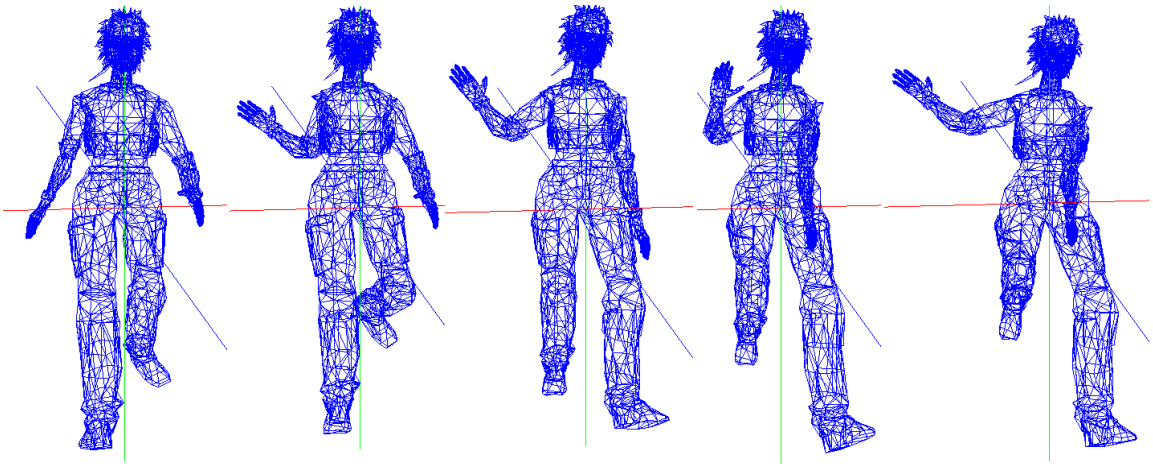


Figure 14: Tiny's walk cycle combined with IK to to form a hand-waving gesture

The particle system worked very well for a simple IK system, but is limited by the fact the backwards retargeting has some error; the limbs (after they have been backwards-retargeted) will always have the correct orientation but the translational property of their position is sometimes a noticeable distance from the skeleton (the right upper-arm in Figure 15, frames 5-7 show this). This is a problem when a specific point needs to be touched precisely. However, it is not a problem specific to the IK system, the IK system in itself provided very pleasing results.

The induced error could be tackled in two ways:

1. Currently, the translational part of all limbs when backwards retargeting is ignored, except for that of the root node. It was hoped that rotation alone would be sufficient; evidently it is not. Trying to factor in translation is not trivial (hence it was ignored) because the ragdoll nodes do not always have a defined axis of rotation, therefore making it difficult to take an arbitrary point

and compare its current versus initial translation; it might be that it was a result of translation, rotation or some combination thereof.

2. Resetting the ragdoll skeleton to the model every few frames would make sure that errors were zeroed often and the two skeletons stayed synchronised. Synchronising incurs an overhead (of the forwards retargeting process) but in light of the pleasing performance data (section whatever), this would be the preferable solution.

IK is not a robust implementation and needs further work before it would be suitable for “production” use; a triangle node with a ball joint has its a joint at its member 'p2', while triangles with hinge joints have the joint at p0 and p1. Therefore, a hand or foot is always at p2. The IK system assumes that p2 is the 'effector' point meaning that upper arms and upper legs cannot (usefully) be assigned IK targets currently.

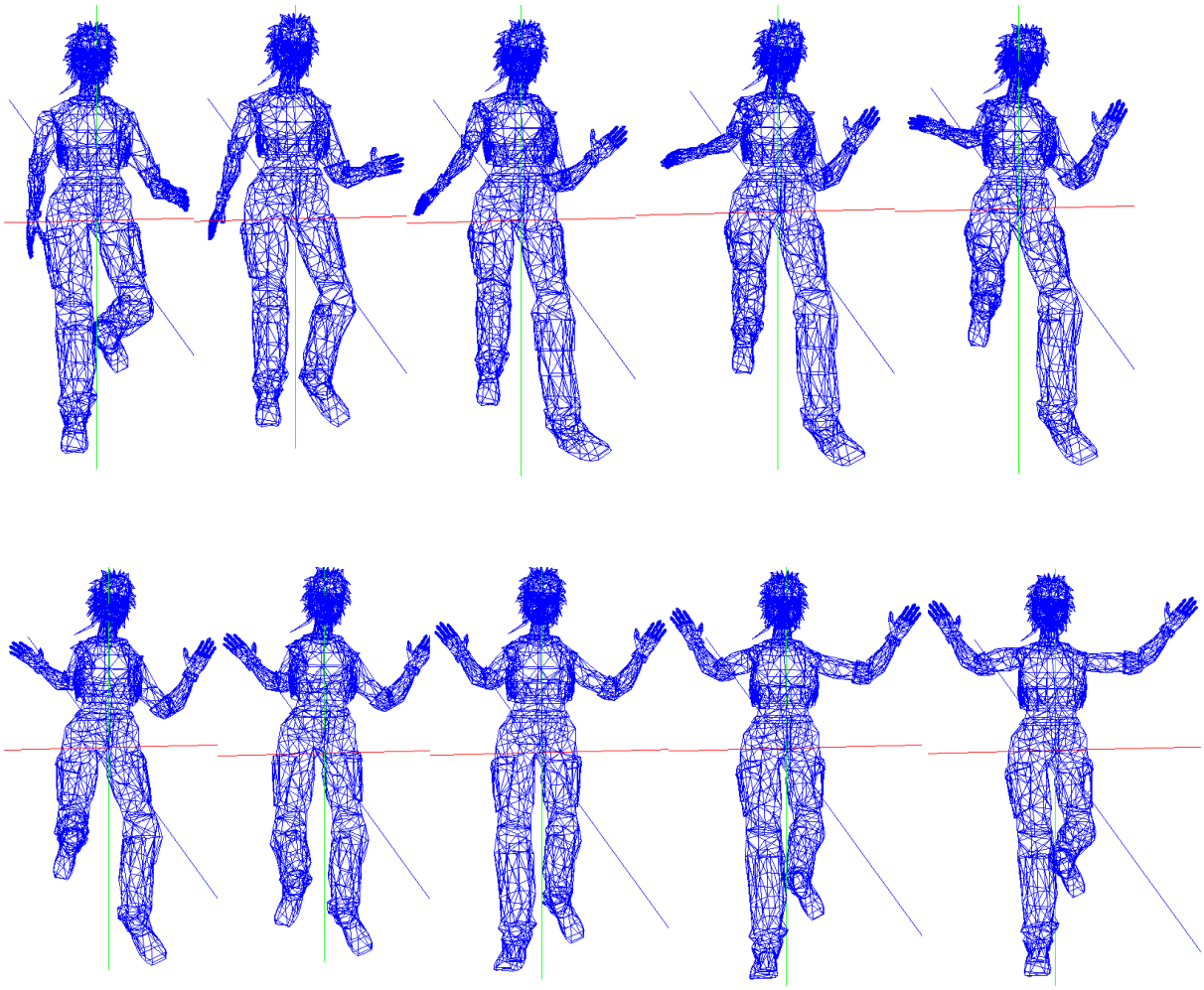


Figure 15: Two IK targets to make Tiny raise both hands while walking

9.2.3 Powered Ragdoll

As an algorithm, the powered ragdoll works very well for blending keyframe and ragdoll motion. In practice, getting the 'right' weighting between the two sources, and allowing the ragdoll to have a free effect for a short period before the keyframe was invoked again proved quite fragile. A 'stun' time was used to try to model this; it gives pleasing results but it is quite arbitrary.

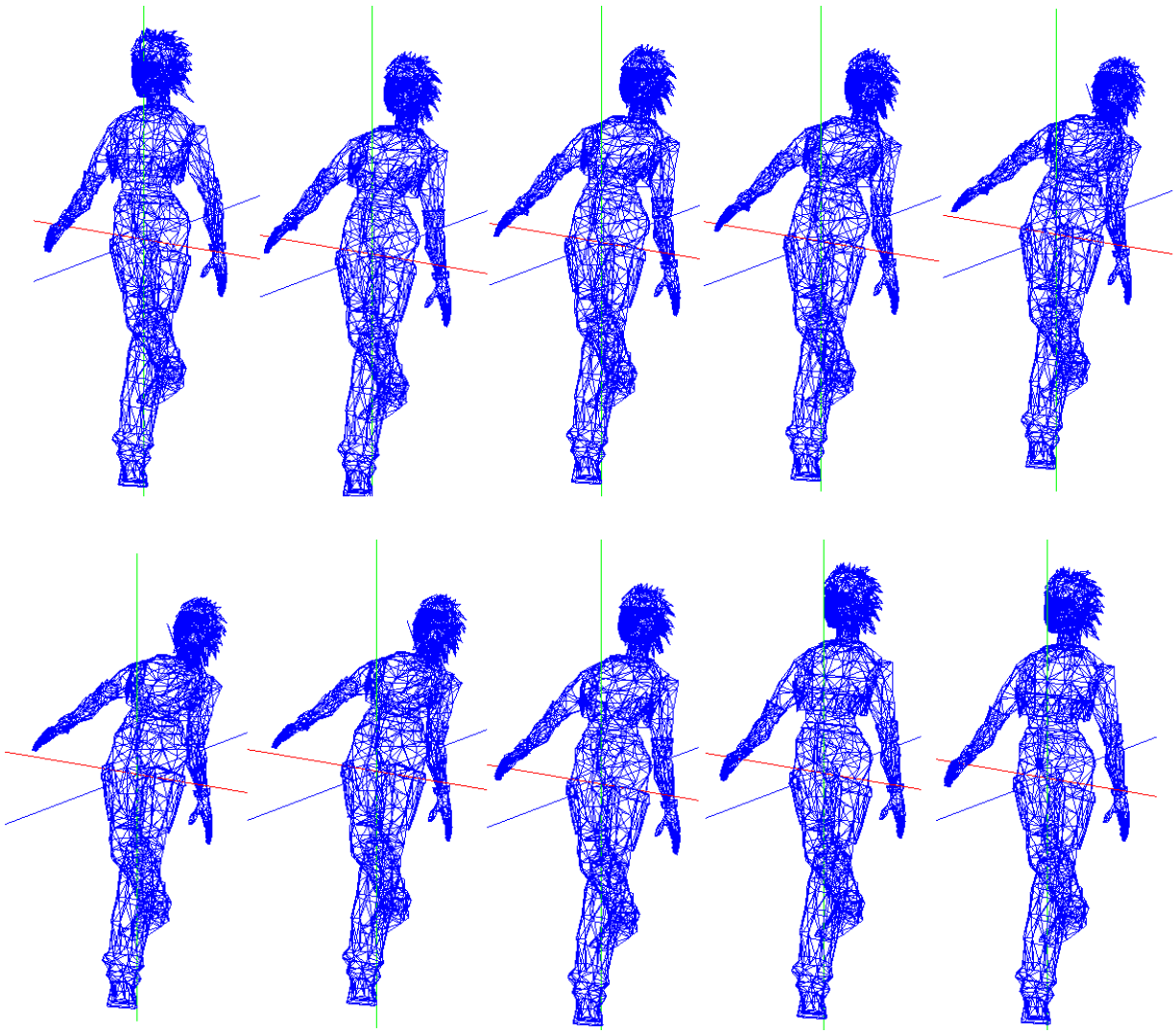


Figure 16: Tiny being pushed backwards, then recovering (starting at frame 6) to her bind pose

The implementation has shortcomings:

1. When the whole body is pushed, one would expect the character to re-position its feet to try to retain its balance. This is non-trivial but would be a useful area of future work and the existing IK API might provide most of the needed functionality.
2. The lack of detail in the Ragdoll skeleton means that extremities are not handled in the resulting animation and therefore the feet *can* appear to drag while walking (although this effect is seemingly random in when it arises). This could be fixed with a more detailed skeleton, but the library does not handle definition of such skeletons in its current state; extending it to do so would be a useful area of work.

As the method uses an abstraction layer to the skeleton and operates on equivalent skeletons, this library could be extended to provide keyframed animation retargeting using the powered ragdoll skeleton as an intermediary between two models.

9.3 Future Work

Many possible areas of future work were outlined in the previous sections. As well as those already mentioned, there are two more of importance:

In terms of actual usability of the library, the ragdoll forwards retargeting process (detailed in 7.2.3.1 - Forwards Retargeting) is a limitation and the library would benefit from a more robust or automated approach. The retargeting system is complicated for a programmer to actually use and the 'rotations' discussed can only be determined by trial and error. A fully automated solution would be preferable but is ultimately very difficult (and since the process is executed quite often one also has to consider that more complicated analysis of the input model would increase CPU load). In section 7.2.3.1 is given a sketch outline of an algorithm for 'guessing' the appropriate skeleton. Implementing this with the possibility of a developer overriding parts of the guesswork would provide a much more accessible solution.

The second implementation concern relates to the use of the Open Asset library. ASSIMP certainly aids the project in that it provides a useful method for generic importation of models and animation data, but the (animation) library's caller has not been shielded from ASSIMP. Thus to use the (animation) library, a developer must include ASSIMP header files, pass some data to the library in ASSIMP data structures and link against the ASSIMP library. This is not necessarily an ideal approach, and the library could feasibly wrap at least most (and probably all) of the ASSIMP functionality/data types making them invisible from the external interface.

In summary, the most interesting areas for future work, in order of importance, are:

1) Refactoring of the ragdoll system's code to decrease interdependency on component parts followed by more rigorous testing of those individual components' shortcomings and then evaluation of whether they can be extended.
2) Implementation of frequent re-synchronisation of ragdoll skeleton to keep the model and ragdoll in better alignment during simulation.
3) Alteration of the IK implementation to be more robust (9.2.2).
4) An easier to use API to the retargeting process.
5) User definable ragdoll skeleton architecture (to allow modelling of characters with unusual anatomies, or more detailed modelling of humans by addition of hands/feet etc).
6) 3D hit-boxes for superior collision detection/response at the ragdoll level.
7) Improved balance response by repositioning of legs/feet.
8) Shielding of the developer from the requirement of also dealing with the Open Asset Import library.

9.4 Satisfaction of Requirements

The overall deliverables for the project were a specification of a method, and an implementation of that method in the form of an animation library, that could handle ragdoll physics in a generic sense, and apply blending between keyframed and ragdoll animation sources. They were divided into a number of points in 3 – Problem Description and Requirements.

Requirement	**Status**
Creation of method	A method was created which allows a generic approach to ragdoll using skeleton retargeting to abstract the ragdoll skeleton. It is detailed throughout this report.
Creation of Library	A library was created, and fully implements the method
Generality of method/library	Some generality of the library was sacrificed (in having a fixed skeleton), but overall generality appears high: The set of models that were tested is small but diverse, and it seems reasonable to extrapolate that (because the skeletons are logically equivalent) the library can handle any human-like biped and some quadrupeds (Appendix 3 – Model To Ragdoll Skeleton Retargeting Examples).
Automation	The automation of the library is not as high as it could be, due to the complex nature of the bind process (section 7.2.3.1). The process is usable, but has been marked as an area of further work. The bind process is a one-off

	consideration, in all other respects the library is very automated and requires very little interaction with each frame.
Keyframe system	A keyframe system was implemented and works very well (9.1 – Discussion Of The Keyframe System).
Ragdoll system	A ragdoll system was implemented and works for some effects but not well for others. It has been marked as an area of future work (9.2 – Discussion of the Ragdoll System).
Ragdoll/Keyframe blending	The blending between keyframe and ragdoll data sources works very well (9.2.3 – Powered Ragdoll).
Full library Interface	The library implements all the functionality it was required to (and more), and provides an interface to each part of it. It should not be necessary for a user wishing to use these functionalities to have to alter any modify of the source code.
Portability Of Library	The library's code is entirely standards compliant and should compile on almost any platform for which there exists a C++ compiler. The demonstration program (which is entirely separate from the library, and whose portability is less important) uses some POSIX extensions and OpenGL. Both the library and demonstration program have been compiled without problems on Linux/GCC4.3 and Windows/GCC3.4(Cygwin)[1].
Documentation of Library	The library's API is documented in full with Doxygen. Sample code for basic operations is provided in Append 7 – Example Code Listings. Using the example code to grasp how the library expects to be called and then using the Doxygen documents to investigate what the example code does not show should provide a perfectly adequate way to learn to use the library.
Demonstration program	A demonstration program was implemented to show the library works.

All requirements are satisfied to some degree, most requirements are satisfied fully. The requirements neglected to make any mention of performance because it is difficult to quantify, and a working system was deemed more important than a fast

1 It was also compiled on Windows/MSVC using Visual Studio, but for some reason only ran within the debugging mode. The program appeared to crash as soon as any code blocks using ASSIMP structures were reached. Having never used Visual Studio/MSVC before, and therefore not understanding its linking method (with .lib objects, opposed to .dll) this was not fixed, but is probably trivial.

one. However, performance is nonetheless an important consideration, and is discussed below (9.5 – Performance).

9.5 Performance

Numbers in this section were obtained by calculating the average length of time over a second spent in the Animator's Update() method (this represents a full cycle). This was recorded for each second over the course of a minute, and then averaged. All numbers are in seconds, and were generated using an Athlon X2 6000+ (using only one core, clocked at 3.0Ghz), compiled with GCC4.3 using -O3 optimisation.

Table 1 shows keyframe data, and Table 2 shows ragdoll data.

	40 updates per second	Update limit unrestrained
1 animation	3.87E-06	2.32E-04
2 animations	4.20E-06	3.93E-04
3 animations	4.71E-06	5.84E-04
4 animations	5.37E-06	8.16E-04
5 animations	6.11E-06	1.07E-03

Table 1: Average time/second spent in the animation library for keyframed animation, blending 1-5 animations, using a restricted update limit of 40 updates/s versus no update limit. The model used was a Doom3 model with 69 nodes.

	40 updates per second	Update limit unrestrained
Pure Ragdoll	7.02E-06	1.27E-03
Blended Ragdoll	1.35E-05	1.42E-03

Table 2: Average time/second spent in the ragdoll system during pure ragdoll animation, and ragdoll animation blended with a single keyframe animation. The model used was Tiny.x with 47 nodes.

Blending of keyframe animations (unsurprisingly) appears to give a linear performance cost. It can be extrapolated that animating multiple models would also stack linearly. In the worst case, 5 blended animations, the update cycle consumed only a single millisecond per second of CPU time. Using the (reasonably set) update limit, the CPU usage is negligible, in the order of microseconds per second. The ragdoll is one order of magnitude slower, but its performance is still very high.

Memory usage was not evaluated thoroughly, but using Tiny.x (47 nodes), usage increases at roughly ~2-3MB per Animator object (and associated ASSIMP scene

and importer), and a further ~1MB per animation being played. There is a further ~22MB overhead in using the animator and ASSIMP as shared libraries, which appears constant. This is a very reasonable requirement for a desktop PC or modern games console but the 22MB overhead is too expensive for a current generation portable console, the Nintendo DS has only 4MB of RAM which is too unrealistic to satisfy. However, upcoming portable consoles have more RAM (PSP GO: 64MB, Pandora: 256MB (OpenPandora (2009)) and effort spent in reducing the overhead memory usage[1] would likely make the library suitable for use on these platforms.

9.6 Appropriateness of Methods

The overall methodology used worked very well. The strict research phase at the start of the project was apt preparation for the bulk of the work, and allowed the project to branch out into other areas of animation (i.e. skeletal retargeting) when the need to do so arose. The fact this was possible was due to incorporating trial and error and experimentation into the initial research phase, which allowed familiarity of many concepts to be obtained quickly, as well as a good idea of what worked well and was worth pursuing, and what did not.

The development methodology also worked well; the library itself is a substantial piece of software, containing around 10,000 lines of code. Most of its features are 'complete' in the sense they work adequately well, but many are of alpha or beta quality in that they would benefit from improvement ranging from more robust methods, to extra safety/error checks (and of course, heavy testing). Regardless, in the relatively short development time of around 6 to 7 weeks, the library is a success in the sense that something so substantial was created in such a short time scale. The methodology of prototyping is largely responsible for the rapid development. However, prototyping was not without its problems; it was anticipated that such an approach would lead to unmaintainable code if due care was not taken. In the Animator container class, the keyframe class and IK class this was avoided entirely. The ragdoll system, however, is the opposite and has been discussed, and certainly did suffer from the fast development, and the second system would benefit from a stronger design.

Two to three weeks were 'lost' in the sense that they were spent pursuing the first ragdoll approach which ended being discarded due to it being inadequate. This is a difficult situation to evaluate because on one side, had the research phase been stricter, Rosen's (2007) paper may have been found (or the equivalent information), which outlined everything that the two to three weeks development in the first approach revealed, and therefore the project would likely have begun by implementing the second system. On the other hand, the practical experience of

1 As well as alterations to this library, reducing memory usage might involve modification to ASSIMP. This is allowed and perfectly legal under their BSD license. A plugin based importer system where different importers were only loaded at runtime if there was a need for them would lower ASSIMP's memory requirements, for example.

implementing the first system was valuable and much of it translated directly to the second system, making the second system's development quite easy. Clearly, another attempt at the same project would not include the first approach and in fact would likely extend the second approach into a 3D dimensional system, whereby the limbs would be modelled purely by boxes instead of triangles/quadrilaterals. Therefore the ragdoll skeleton would give a full representation of the model with respect to collision detection. The work in such a project would also have to consider implementing realistic joints between boxes, and creating a suitable forwards-retargeting process (which, again, would not be trivial).

Overall, the results obtained are pleasing especially considering the short time frame, and the methods used were very appropriate to the type of research and development that was undertaken over the course of this project.

10 References

Adams, Jim (2003), Advanced Animation With DirectX, Premier Press Game Development

assimp.sourceforge.net (2009), Open Asset Import Library [online],
URL: http://assimp.sourceforge.net [5th July 2009]

Blow, Jonathan (2004), Understanding SLERP, Then Not Using It [online],
URL: http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/index.html [27th June 2009]

Brown, Eric (2009), Ragdoll Physics On The DS [online],
URL: http://www.gamasutra.com/view/feature/3916/ragdoll_physics_on_the_ds.php [2nd July 2009

Bruckschlegel, Thomas (2005) Microbenchmarking C++, C# and Java [online],
URL: http://www.ddj.com/cpp/184401976?pgno=1 [26th June 2009]

Elias, Hugo. (2000) Inverse Kinematics [online],
URL: http://freespace.virgin.net/hugo.elias/models/m_ik.htm [15th August 2009]

Ericson, Christer (2005) Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology), Morgan Kaufmann

Evangelista, Bruno (2008), Playing Nice Animations on XNA [online],
URL: http://www.ziggyware.com/readarticle.php?article_id=190 [27th July 2009]

Fjeld, Paul (2006) Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas [online],
URL: http://history.nasa.gov/alsj/gimbals.html [11th September 2009]

flipcode.com (1998), The Character Animation FAQ [online],
URL: http://www.flipcode.com/documents/charfaq.html [21st June 2009]

Hahn, James (1988), Realistic Animation of Rigid Bodies, CGraphics, Volume 22, Number 4, August 1988

Hecker, Chris (1996) Physics parts 1-4 [online],
URL: http://chrishecker.com/Rigid_Body_Dynamics [2nd July 2009]

Hennix, M., Hugoson, P., Johansson, G., Lombardi, A., Miljevic, T., Nillson, A., Wassborn, M. (2003) Rag doll physics [online],
URL: http://staffwww.itn.liu.se/~gunjo/papers/ragdoll_physics.pdf [3rd July 2009]

Hughes, John F. (1999) Efficiently Building a Matrix to Rotate One Vector to Another, Journal Of Graphics Tools Archive, Volume 4, pp 1-4

IOInteractive (2002) Hitman 2: Silent Assassin, Eidos Interactive

Jakobson, Thomas (2001) Advanced Character Physics [online],
URL: http://teknikus.dk/tj/gdc2001.htm [26th June 2009]

Martin, Brian (1999) Quaternion Interpolation [online],
URL: http://www.theory.org/software/qfa/writeup/node12.html [1st July 2009]

mono-project.com (2009) Mono; a cross platform, open source .NET framework [online],
URL: http://www.mono-project.com/Main_Page [26th June 2009]

numpy.scipy.org (2009), NumPy; Scientific computing for Python [online],
URL: http://numpy.scipy.org/ [1st September 2009]

OpenPandora (2009): PANDORA portable gaming and mobile internet device [online],
URL: http://openpandora.ca [5th September 2009]

Osier, Jeffrey (1993) GNU Gprof [online],
URL: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html [13th September 2009]

Park, Jamie (2008), Motion Capture-driven Dynamic Simulation [online],
URL: http://dynamic-motion.weebly.com/uploads/8/7/1/3/871360/cs275_ai_paper.doc [14th July 2009]

playstation2-linux.com (2009), Setting Up GCC As A Cross-compiler [online],
URL: http://ps2stuff.playstation2-linux.com/gcc_build.html [26th June 2009]

Rosen, David (2007) Starting Point for Physics-Based Character Animation [online],
URL: http://legacy.wolfire.com/rotationconstraintpaper/paper.html [29th July 2009]

Rotenberg, Steve (2005), CSE 169: Computer Animation, Chapter 2: Skeletons [online],
URL: http://graphics.ucsd.edu/courses/cse169_w05/2-Skeleton.htm [21st June 2009]

Shoemake, Ken (1985), Animating Rotation With Quaternion Curves, International Conference on Computer Graphics and Interactive Techniques, pp 245-254 [online],
URL: http://portal.acm.org/citation.cfm?doid=325334.325242

Smith, Russ (2004), Constraints In Rigid Body Dynamics, Game Programming Gems 4, Charles River Media, pp241-256

Sommerville, Ian (2004) Software Engineering, Addison Wesley, 7th Edition

Tremethick, Piran (2006) Real Time Character Animation [online],
URL:
http://ncca.bournemouth.ac.uk/gallery/view/39/Real_Time_Character_Animation
[15th June 2009]

valgrind.org (2009) Valgrind [online],
URL: http://valgrind.org/ [13th September 2009]

Weisstein, Eric W. (2009) "Euler Forward Method."From MathWorld--A Wolfram Web Resource [online],
URL: http://mathworld.wolfram.com/EulerForwardMethod.html [27th July 2009]

Weisstein, Eric W. (2009) "Quaternion." From MathWorld--A Wolfram Web Resource [online],
URL: http://mathworld.wolfram.com/Quaternion.html [12th July 2009]

Wrotek, Pawel; Jenkins, Odest Chadwicke; McGuire, Morgan (2006) Dyanmo: Dynamic, data-driven character control with adjustable balance, Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, pp 61-70

Zordan, Victor B.; Majkowska, Anna; Chiu, Bill, Fast, Mathew (2005) Dynamic Response For Motion Capture Animation, ACM Transactions on Graphics (TOG), Volume 24, pp 697-701

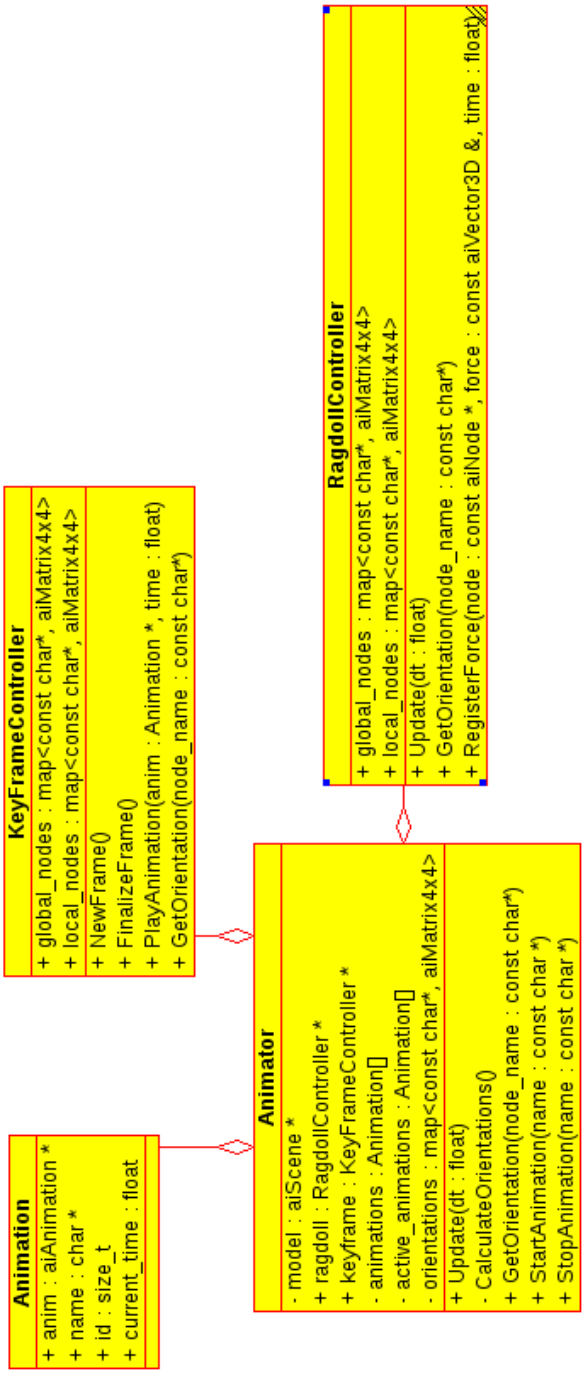
Appendix 1 Software notes

All appropriate notes of the software can be found on the DVD under the root directory. Documentation is provided in Doxygen format, and can be found in docs/html/index.html.

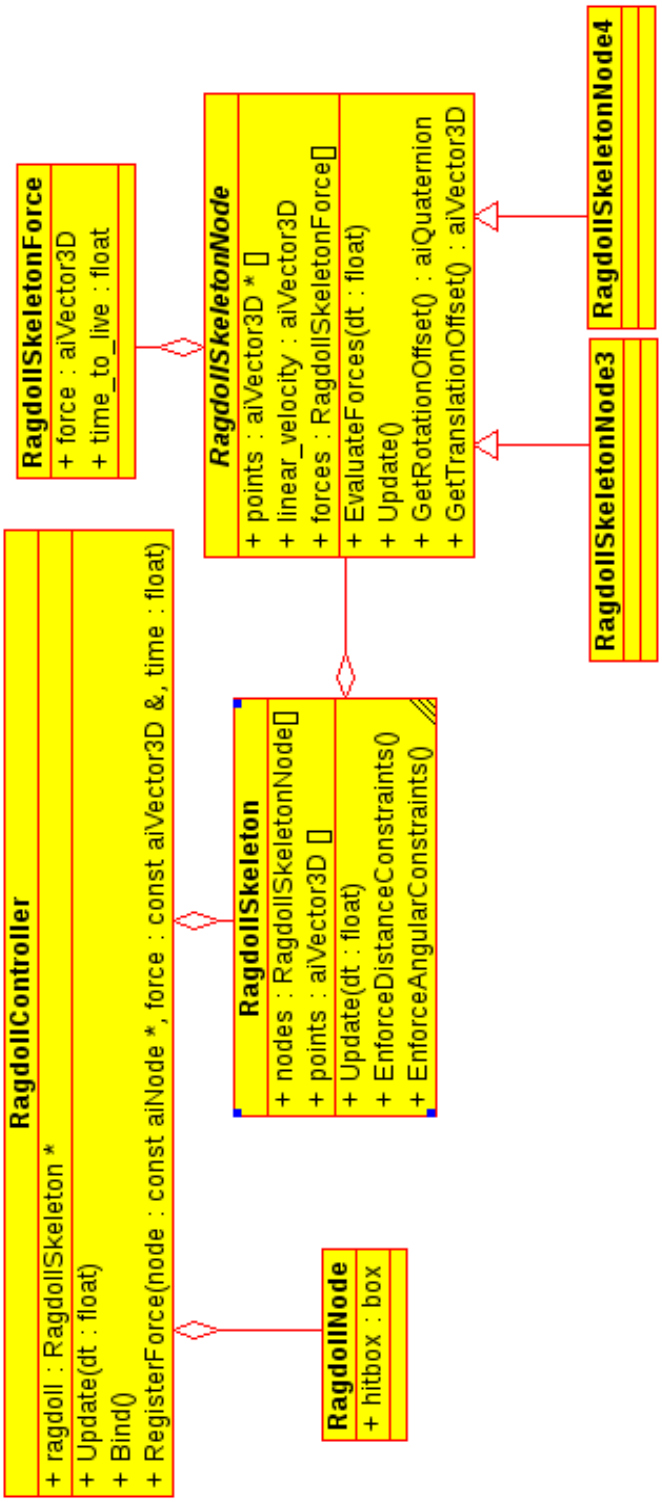
Appendix 2 UML Class Diagrams

Full class listings can be found in the documentation on the DVD. The diagrams provided here are skeletal and represent an overall layout.

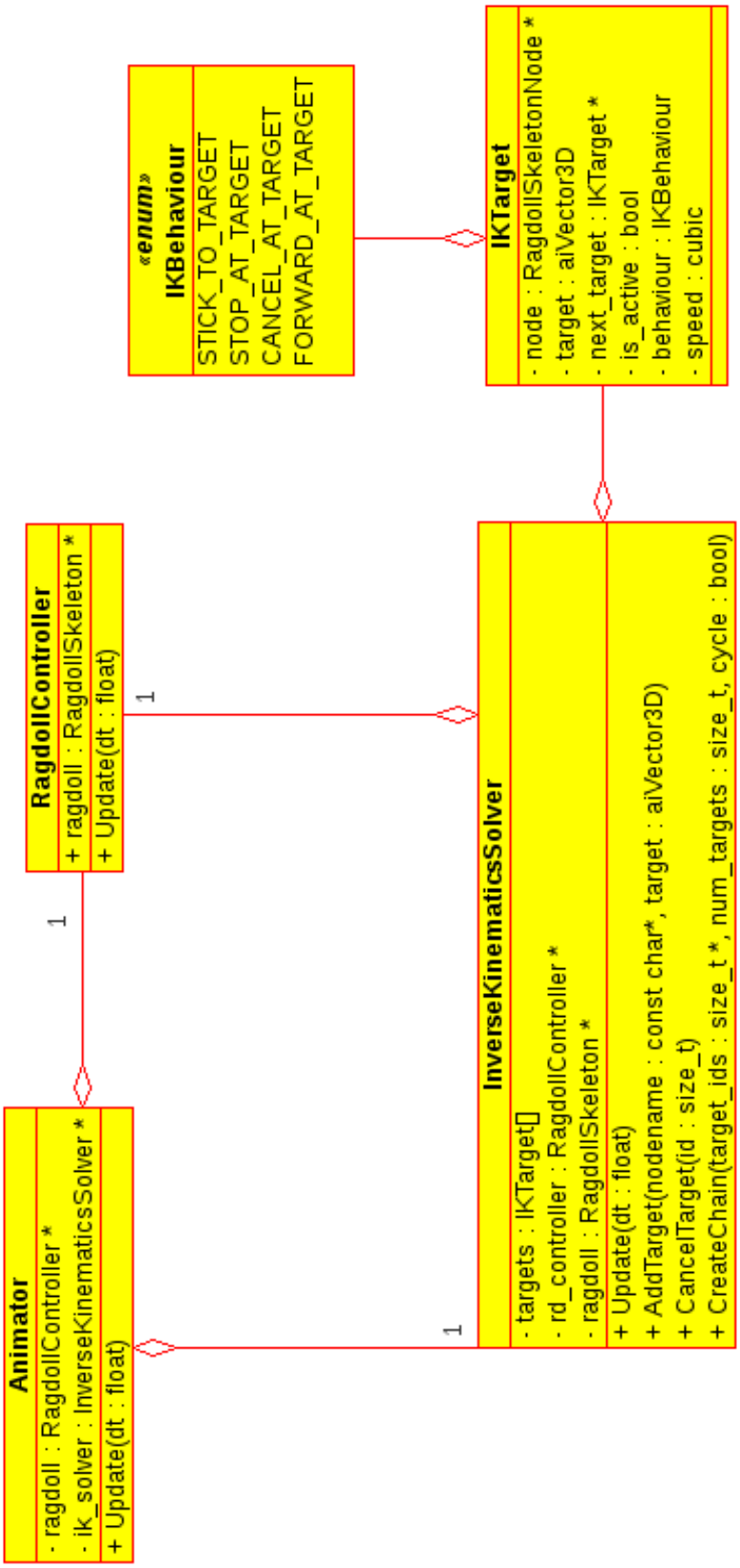
Appendix 2.1 Full Library



Appendix 2.2 Ragdoll System Class Layout



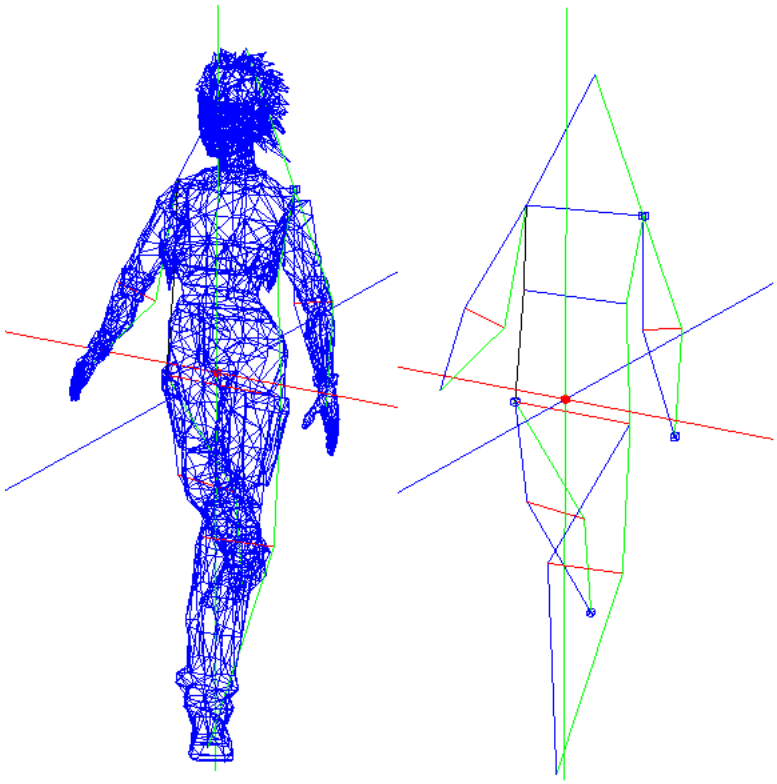
Appendix 2.3 IK System class layout



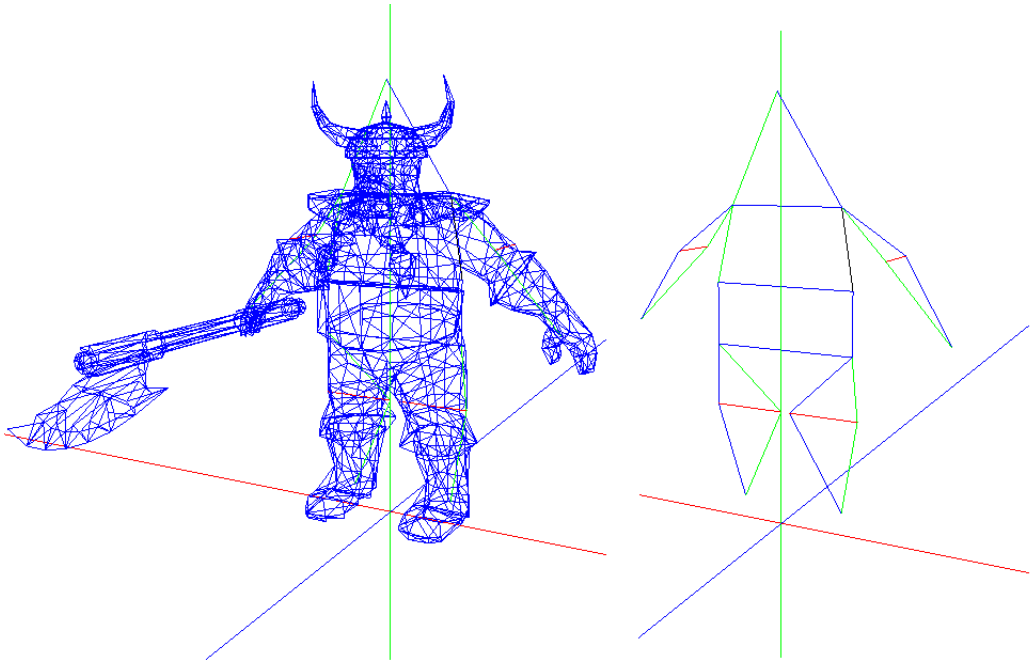
Appendix 3 Model To Ragdoll Skeleton Retargeting Examples

Three different models being retargeted to a standard skeleton. The skeleton shows itself to be surprisingly versatile in that it can handle a dog as well as a human with no changes to its node structure. The skeletons are a very simplified version of the model, suitable for ragdoll simulation. Note the dwarf's axe to be unnecessary for ragdoll simulation

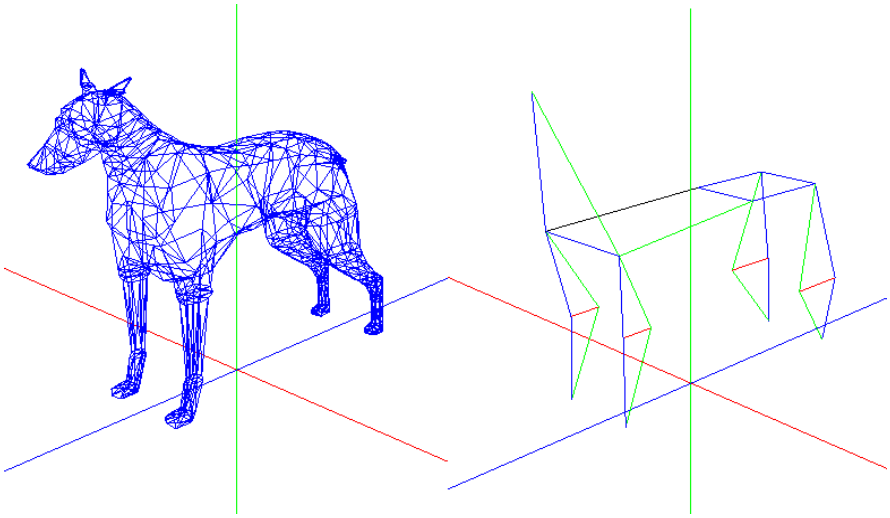
Human:



Dwarf:

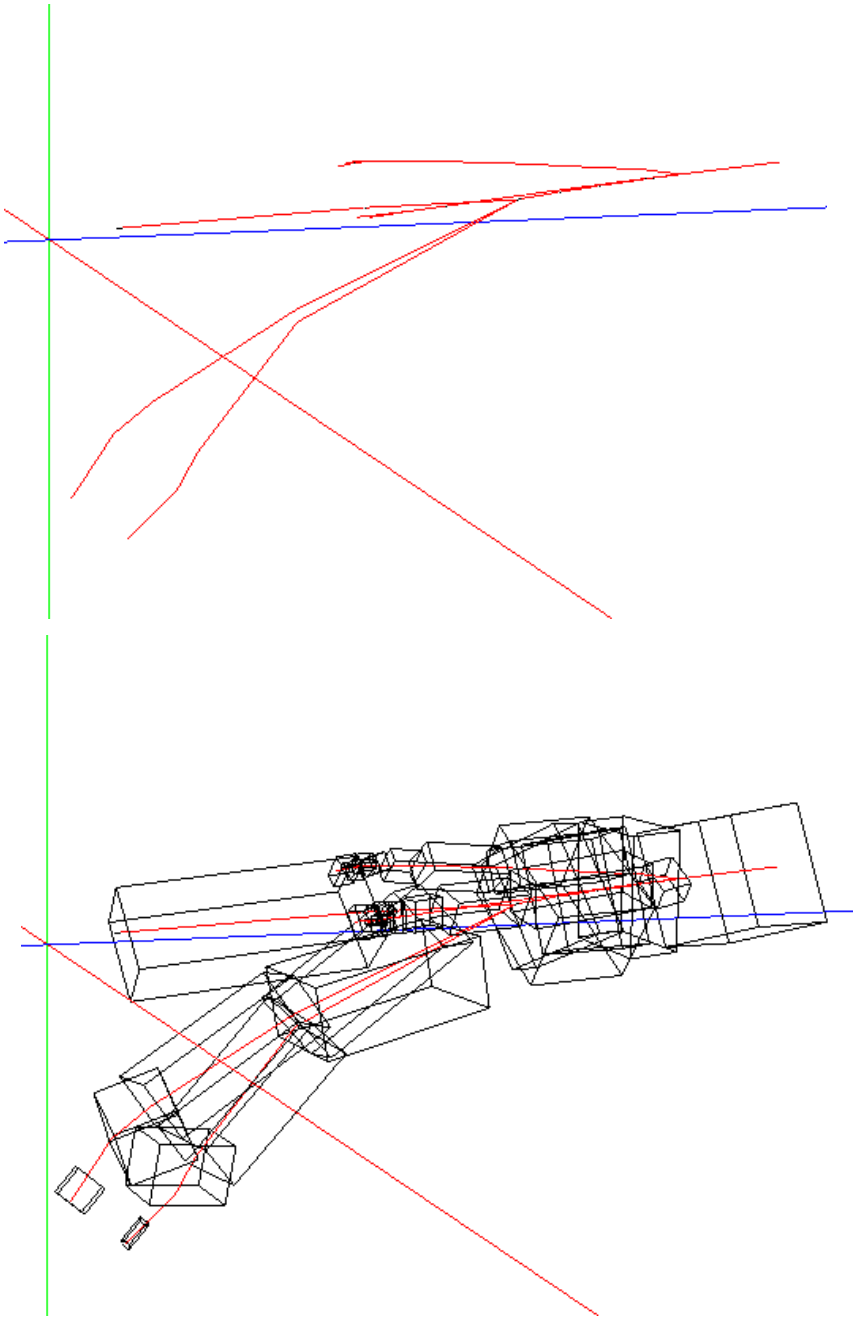


Dog:



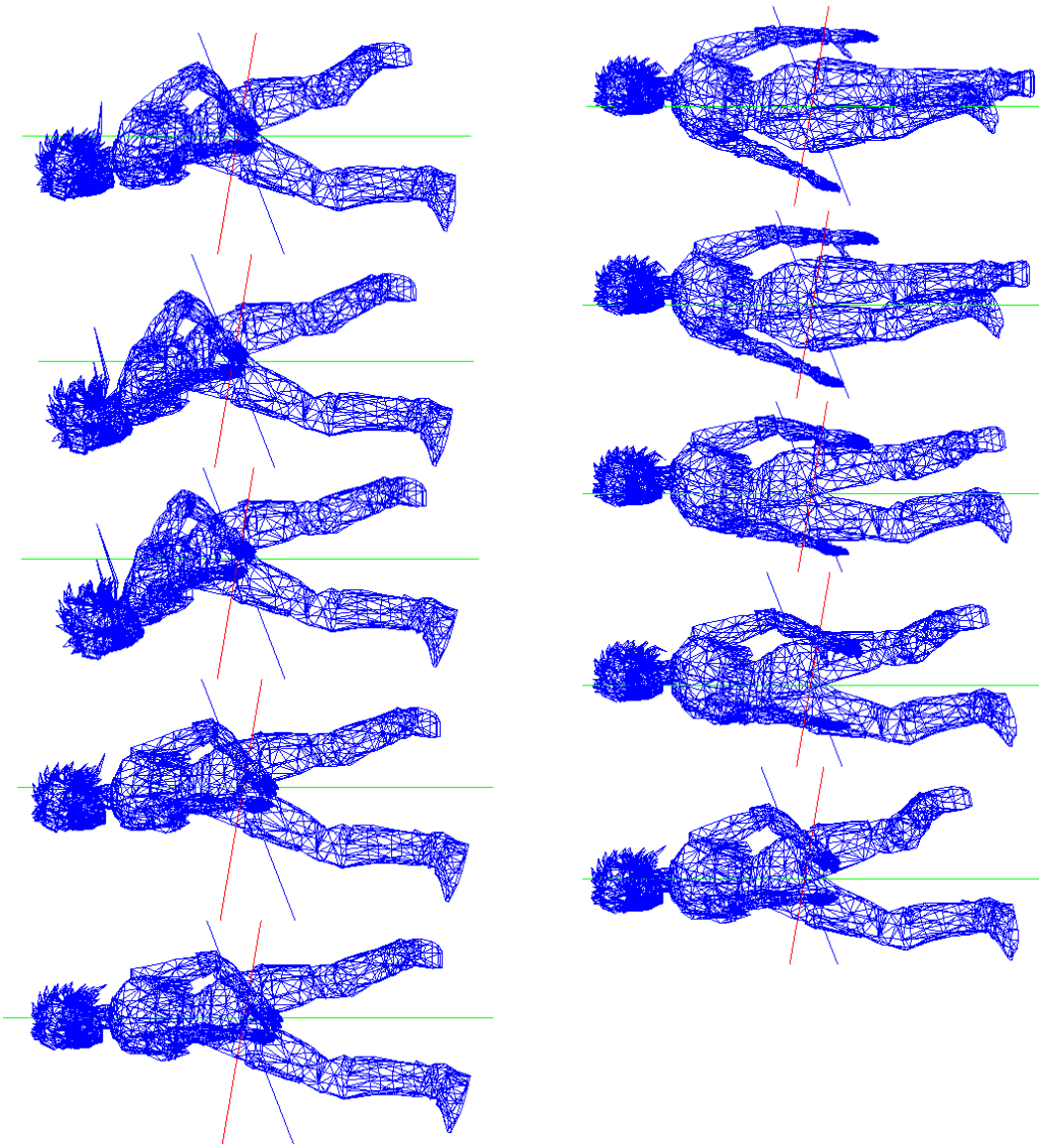
Appendix 4 Ragdoll Approach 1 Further Results

These screenshots are analogous to those found in section 7.1.4.1- Model Quirks. They show the behaviour of the first Ragdoll system when the body is subject to a strong pushing force. It can be seen that the skeleton has almost lost its resemblance to a human skeleton. It can also be seen that some of the boxes's orientations (particularly noticeable from the legs downwards) have come out of synchronisation with the skeleton.

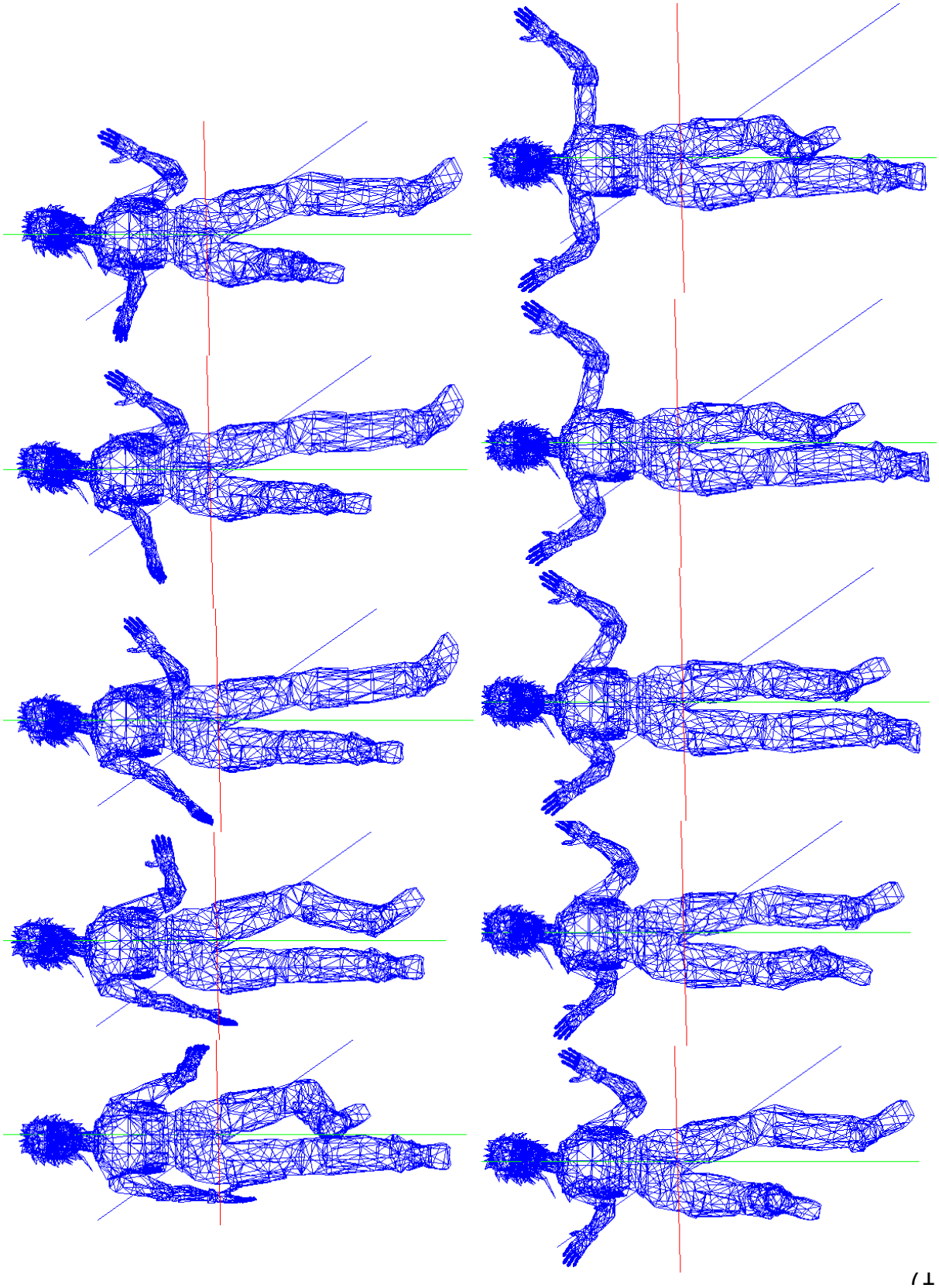


Appendix 5 Ragdoll Blending Examples

Tiny receiving while walking a push from behind, then recovering into her walk cycle.



Appendix 6 IK Examples



Appendix 7 Example Code Listings

A (mostly) full API documentation is provided within the library's source code with Doxygen. An example interaction is given in the demonstration program but due to its complexity from performing more functions than just being an example source code listing (and the fact that it served as an experimental human interface to the project, and has not been cleaned up) it is not always clear. For clarity, some brief examples on calling the library are given here.

Appendix 7.1 Animator Object Creation

Before using the animation library a model needs to be imported by ASSIMP and given to the library. We assume that there exists a model, and two animations, each stored in separate files. For multiple animations per file the mAnimations should be read past index 0. Error checking is omitted, see ASSIMP documentation for details.

```
#include “animator.h”

//ASSIMP includes
#include <assimp.hpp>
#include <aiPostProcess.h>
#include <aiScene.h>
#include <aiAnim.h>

Prefs::SetDefaults(); // set default library preferences. Alter these later
                                                  if needed, see prefs API

Assimp::Importer importer, importer2, importer3;  //ASSIMP  structures
const char *model_file = “/path/to/model.mdl”;
const char *anim1_file = “/path/to/animation.mdl”;
const char *anim2_file = “/path/to/animation2.mdl”;

// import models with ASSIMP
const Assimp::aiScene *mdl = importer.ReadFile(model_file, 0);
const Assimp::aiScene *a1 = importer.ReadFile(anim1_file, 0);
const Assimp::aiScene *a2 = importer.ReadFile(anim2_file, 0);

// Create animator.
Animator::Animator *animator = new Animator::Animator(mdl);

// Register animations with aliases.
animator->RegisterAnimation(a1->mAnimations[0], “walk”);
animator->RegisterAnimation(a2->mAnimations[0], “run”);

// register function pointers
animator->SetBindFunc(&bind);  // see 7.2.3.1
anmator->SetCollisionRegisterFunc(&cd_register); // see 7.2.4.2
animator->SetCollisionUnregisterFunc(&cd_unregister); // see 7.2.4.2
```

```
animator->ragdoll->skeleton->SetCollisionPointIsOnGroundFunc(
                                    &cd_point_on_ground_func
                                    ); // see  7.2.4.2
```

Appendix 7.2 Example frame-by-frame interaction

With a persistently accessible Animator* object can now be demonstrated usage of the library's keyframed functionality on a frame-by-frame basis. We assume that there exists a character object with an internal (movement) state whose state names match animation names. In a 'real' application one would expect the character's object's state change to trigger a call to the animator but that would be less easy to demonstrate so tersely here so instead we assume existence of a function to tell us what state changes the character has undergone this frame.

We also assume existence of a game clock.

```
Animator* animator;  // we assume this has been correctly initialised
                                                              (see 7.1)
float last_frame_time = GameTime::GetTime();

void main_loop()
{
  float t = GameTime::GetTime();
  float dt = t – last_frame_time;
  last_frame_time = t;

  char *from, *to;
  //imaginary function to return a state change of a character
  character->GetStateChangeThisFrame(&from, &to);

  // there are four possibilities: we want to start an animation,
  // we want to stop an animation,
  // we want to transition from one to another,
  // or we just want to continue as we were.

  if (!from && to)
    animator->StartAnimation(to);
  else if (from && !to)
    animator->StopAnimation(from)
  else if (from && to)
    animator->Transition(from, to);

  //if neither to nor from, then no changes are necessary.

  animator->Update(dt);
  draw_character(animator);
}
```

Appendix 7.3 IK Example Code listing

Example code showing how to set up a repeated chain of IK targets in a loop.

Assumes an Animator object pointer, animator, exists and is initialised.

```
bool repeat = true;
const char *nodename = “right_hand”;
aiVector3D ik_target = aiVector3D(-200, 150, 0);
aiVector3D ik_target2 = aiVector3D(-100, 150, 0);

float speed = 2.0f;

size_t t1 = animator->ik_solver->AddTarget(nodename,
                                           ik_target,
                                           Animator::FORWARD_AT_TARGET,
                                           speed);

size_t t2 = animator->ik_solver->AddTarget(nodename,
                                           ik_target2,
                                           Animator::FORWARD_AT_TARGET,
                                           speed);


animator->ik_solver->SetActive(t1, true);

size_t num_targets = 2;
size_t *list = (size_t*)malloc(sizeof(size_t) * num_targets); // it doesn't
                               matter whether this is malloced or new[]ed
list[0] = t1;
list[1] = t2;
animator->ik_solver->CreateChain(list, num_targets, repeat);
free(list);
```

Appendix 8 Turnitin Report