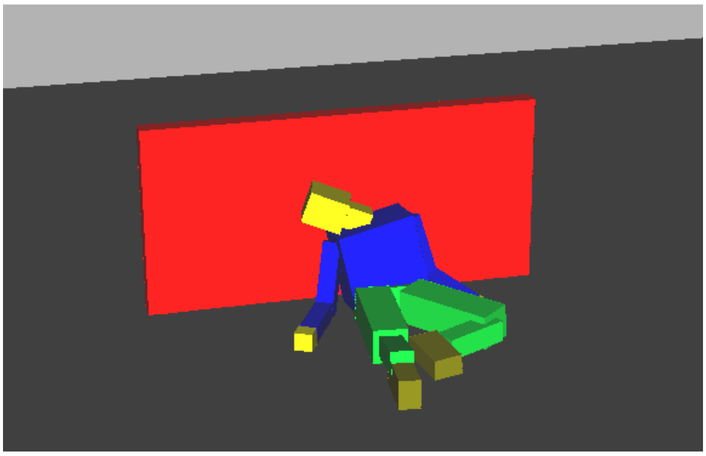
Comparison of ragdoll methods
-
Physics-based animation

Stefan Glimberg - glimberg@diku.dk
Morten Engel - engel@diku.dk
Department of Computer Science, University of Copenhagen

23rd January 2007

Abstract

Believability is the keyword in most of today's computer games, whether it is about gameplay, game history, game sound or the visual presentation. In this paper we concentrate on the visual part, namely the simulation of ragdolls used to animate dead bodies. A ragdoll is a procedural animation of a 'dead' articulated body, typically used to make falling bodies look real. The hardware in modern computers makes it possible to do physics-based computations during simulation, but how believable is this feature, and how many resources is acceptable to use? In this paper we analyze, implement and compare two different ragdoll methods. The first method is based on analytic correct mathematics, using constraints to animate the body joints. The second method is based on particles bound by distance constraints to make up rigid bodies. The joints are then composed by sharing particles among the body parts. The methods will be referred to as the constraint-based (multibody) method and the particle-based method. Many parts of the constraint-based method was implemented in the used physics engine, therefore the particle-based method has been in focus in this project. We discuss how good these methods are with respect to visual result (believability), resource use and usability in modern game engines. We have come up with efficient techniques to connect the body parts using stick constraints, such that the angular movement can be limited. We also present a method to minimize the computational requirement when solving multiple contact point collisions between boxes. This paper should give the reader a good understanding of the problems and requirements that might occur when developing a particle-based ragdoll simulator. Both implemented methods ends up as being compatible in a modern game engine environment. Unfortunately the particle-based method sometimes behaves unrealistic in its present state. Unwanted energy seems to capture the movement when articulated figures are simulated. However, we believe that it is close to a usable state. All ragdoll source files are available in the appendix and the rest of the physics engine can freely be checked out from the internet.



Contents

1 Introduction **5**

1.1 Using OpenTissue as framework . . . 5

1.2 Results . . . 6

1.3 Related work . . . 6

1.4 Reading guide . . . 8

2 Introduction to dynamic theory **9**

2.1 Particle dynamics . . . 9

2.2 Rigid body dynamics . . . 10

2.3 Particles vs rigid bodies . . . 14

2.4 Constraint-based multibody dynamics . . . 15

2.5 Numeric integration . . . 16

3 Analysis of particle based ragdolls **18**

3.1 Ragdoll bones . . . 18

3.1.1 Particles and constraints . . . 18

3.1.2 Visualize the bones . . . 18

3.2 Joints . . . 19

3.3 Joint limits . . . 20

3.3.1 Hinge joint limits . . . 20

3.3.2 Ball joint limits . . . 23

3.4 Collision Detection . . . 25

3.5 Collision Handling . . . 25

3.5.1 Multiple contact points . . . 26

3.5.2 Box masses . . . 26

3.5.3 Rotation . . . 27

3.6 Friction . . . 28

4 Implementation notes **29**

4.1 Integrating with OpenTissue . . . 29

5 Comparison and results **31**

5.1 Visual results . . . 31

5.1.1 Test executables . . . 31

5.1.2 Test 1. Basic collision handling . . . 32

5.1.3 Test 2. Basic ball joints . . . 32

5.1.4 Test 3. Basic hinge joints . . . 34

5.1.5 Test 4. Box masses . . . 35

5.1.6 Test 5. Limited ball joints . . . 36

5.1.7 Test 6. Limited hinge joints . . . 37

5.1.8 Test 7. Articulated figures . . . 38

5.1.9 Test 8. Human ragdoll . . . 39

5.2 Performance results . . . 40

5.2.1 Test 9. Memory usage . . . 40

5.2.2 Test 10. Timestep . . . 42

5.2.3 Test 11. Number of bones 43
5.2.4 Test 12. Iterations 44
5.3 Conclusion 45
5.4 Future work and nice to haves 46

A Human proportions 48
A.1 Modeling the bones 48
A.2 Body joints 49
A.3 Modeling the constraints 49

B OpenTissue implementation issues 51
B.1 Collision detection 51
B.2 Particles stored in dynamic array 51
B.3 Wrong satisfy calculation 51

C Program code 52
C.1 psys_ragdoll.h 53
C.2 psys_ragdoll_bone.h 57
C.3 psys_ragdoll_human.h 60
C.4 psys_joint.h 64
C.5 psys_hinge_joint.h 65
C.6 psys_ball_joint.h 67
C.7 retro_ragdoll.h 69
C.8 retro_ragdoll_human.h 72

1 Introduction

The computer scientific focus in this project is computer graphics and physics-based animation. Most of the theory used is based on physics theorems developed over the course of several centuries, even before computers were available. The two simulation methods we seek to implement are based on different approaches, but they are theoretically and visually comparable. We discuss the visual comparing further in the result section 5.
Since we use computers with numeric precision to reflect real world behavior, we must expect some loss of believability. In addition to this drawback, the ragdoll methods are based on analytical theories that tend to model only 'perfect' worlds. Ie. in the real world, moving objects are affected by a virtually unlimited number of external factors, including shifting winds, air humidity, different materials and attraction forces from every object. Modeling a completely realistic physical world taking all these factors into consideration would be impossible to handle. This leads to the creation of simpler physical theorems, that model a more perfect world with less outside factors and changing states. Luckily the final result is usually still good. Figure 1 illustrates the simulating process from the real world to the screen.

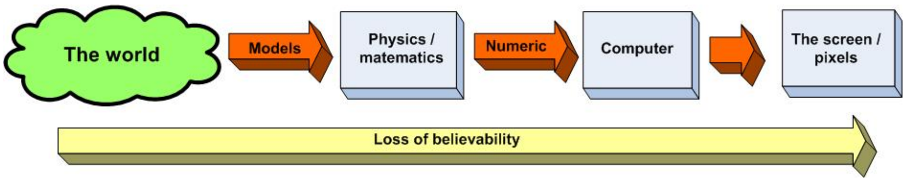


Figure 1: The process of modeling the real world using computer hardware. Each step introduces some loss of believability.

One problem we have sought to solve, is to keep the lack of believability at a minimum and still achieve an acceptable performance.
To our knowledge, not much work has been presented describing particle-based rigid body modeling and especially not particle joint modeling. Thomas Jakobsen [7] gives an introduction on how to model tetrahedra using particles and Jeroen Wagenaar presents the joints possible to make with particle models in his thesis [12]. However, they do not describe all the techniques needed for a particle-based ragdoll implementation.

Our motivation for this project is to analyze the necessities for implementing a particle-based rigid body system, and implementing not only this system, but also the more conventional constraint-based multibody method. Leading to a qualitative judgement of the pros and cons of both methods, and an in depth analysis of the sparsely documented particle system method.

1.1 Using OpenTissue as framework

The implementation of the two ragdoll methods uses the physics engine OpenTissue [9], which is mainly developed at the Department of Computer Science at the University of Copenhagen. The engine is developed with special reference

to physics-based simulations, this is both a drawback and an advantage for our purpose. Game engines often use 'tricks and hacks' to give better performance, sometimes even on behalf of visual result. Since games often run fast, the inaccuracy will be minimal with respect to visual result, and thus the believability is retained. The advantage is that constraint-based multibody methods already are an implemented toolkit, leaving us with a limited amount of work and testing to implement this part, giving us more time to focus on the particle based system. Furthermore OpenTissue has a large toolbox available for collision detection, particle systems, mathematical functionalities and simulators.
Since a lot of work has been put into the implementation of the particle-based method, we will describe the analysis and results of the problems we encountered. During preparation of this project we encountered quite many interesting difficulties that did not explicitly fit into the project description. Still they had to be solved to continue. These difficulties, that required some kind of effort and time, is briefly described in the appendixes.

1.2 Results

The results are presented in a visual oriented part and through a number of performance tests. Both methods give nice visual results when simple test scenes are simulated. The particle-based method unfortunately reveals some drawbacks when more complicated figures are used. Nevertheless, we believe that some tweaking, could make the method at least as good as the constraint-based method, because the method in its non-optimized state gives some nice performance results.
Figure 2 illustrates the ball joint limits, implemented with the particle-based method. Both ball and hinge joints have been limited using new methods presented in this paper. The solution has the advantage that all connected joints are effected by the projection, making them imitate Newton's third law of motion. The discussion on how to limit the joints can be found in the analysis, section 3. Figure 3 illustrates various screenshots taken when testing the constraint-based ragdoll.

1.3 Related work

Physics-based animation is a widely studied area in computer science. It is mainly based on classical mechanics, also known as Newtonian mechanics, after Isaac Newton and his laws of motion. Especially the dynamics and kinematics subcategories of classical mechanics and particle physics are used in physics-based animation and will also be used in our project. A solid background for this area can therefore be learned from regular physic books and articles.
[4] gives a well described introduction from the physical perspective into the use in physics-based animation. For this project, chapter 5, 6, 7 and 22 will give a good base to understand the underlaying theory. When implementing the two ragdoll methods, we will especially use the principles described in [4] and the article by Thomas Jakobsen [7]. The latter article actually describes the principles of a method that have been used in a modern computer game,

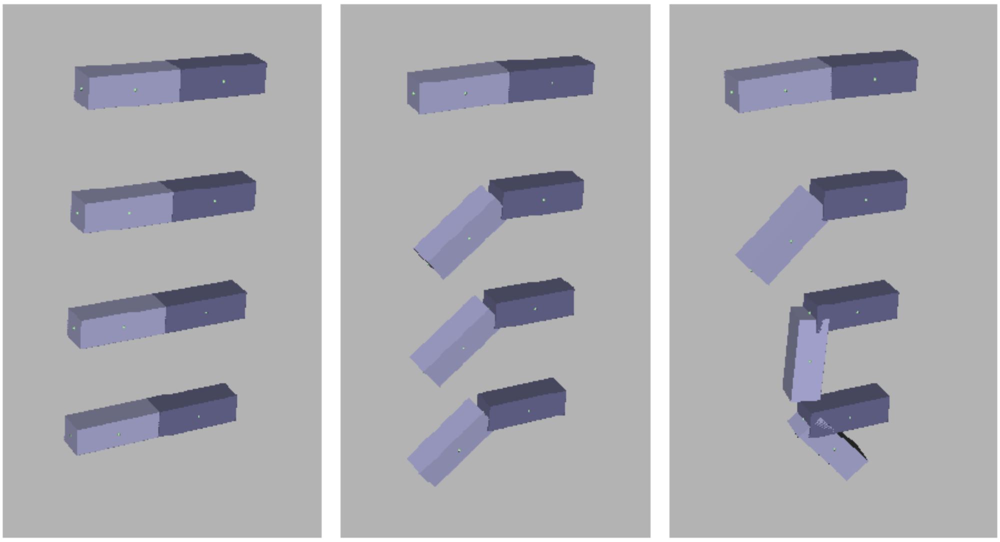


Figure 2: Limited ball joints. The limitation method is analyzed in

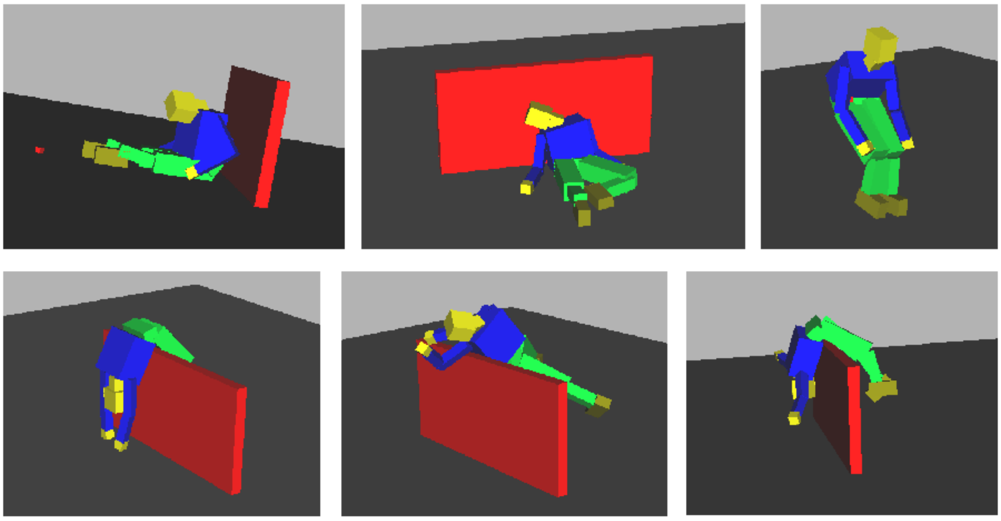


Figure 3: Various screenshots of the constraint-based ragdoll implementation. The doll is tossed over an obstacle and shot with small red boxes.

Hitman [6].
A variety of articles describing branches of physics-based animation and alternative algorithms exists. [10] describes a way the animator can manipulate a rigid body during simulation, letting the computer make the necessary adjustment to position, velocity and so on. The articles [12, 1, 5, 2] give good descriptions of the theories used in both particle systems and for rigid body dynamics and can be used if more details on the theory is wanted. [2] treats a paradigm for simulating rigid bodies and also deformable- and liquid bodies using particle systems. [12] models rigid bodies using particles and constraints, in a similar

way to the one we present. The thesis also handles the particulate problem that occur when using particles to represent rigid bodies, we discus particulating in section 2.3.

1.4 Reading guide

To understand some of the more advanced details in this project we recommend that the reader is familiar with basic mathematics and animation techniques. Read the previous section for some recommendations. The project is mainly dedicated readers of computer graphic interest, at the level of at least a second year computer science student.
We have sought to write an intuitive order of sections as follows:

Section 1 Gives an introduction to the whole project and the area of computer science.

Section 2 This section describes the physical theories and the simulation principles used through the project. An introduction to particle and rigid body dynamics is given along with a description of numeric integration principles.

Section 3 In this section we give an analytic discussion of the problems we encountered before and while implementing the ragdoll methods. The main emphasis has been laid on the particle based method. Many figures are used to ease the understanding of the analysis.

Section 4 A short introduction to the implementation techniques used.

Section 5 Results and conclusions are given in this section. The sections contains a visual part, with many screenshots of the visual accomplishments we have achieved, and a performance part. The conclusion of the whole project is given in the end along with some future work ideas.

Appendix A Here we have placed all the research done before we were able to model an anatomic correct human. We describe the human proportions and the angular limits between joints.

Appendix B A description of different problems encountered, not explicit related to the project, are placed in this appendix section.

Appendix C All the code made specifically to implement the ragdoll methods, are placed in this appendix section. The whole engine would be to big to place in the appendix.

Enclosed CD This paper has been handed in to censor and supervisor with an enclosed CD. On the CD you will find this paper in pdf format, all used OpenTissue code, screen captures, profiling files and executable files to start the test demos.

2 Introduction to dynamic theory

The ragdoll animations in this project are based on math and physic theories. This section will introduce the basic theories. The two methods we seek to implement are based on essentially the same physics. The particle method can be used to derive much of the theory used for rigid bodies. In the following two subsections we treat both methods. The presentation given is quite similar to the theory used in [1, 5, 4, 12]. Vectors will be denoted with an arrow over the symbol, matrices with **bold** and scalar values with normal typeface letters.

2.1 Particle dynamics

Start by observing a single particle. A particle is a non-rotating object with mass m and a location in space. The position vector is defined in a cartesian world coordinate system (WCS). All particles can be described relative to the WCS. At time t the position is given by the function $r(t)$.

$$\vec{r}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \tag{1}$$

Where x, y and z are the coordinates in the 3 dimensional space. The average change in velocity between two positions $\Delta\vec{r}$, in time Δt is

$$\vec{v}_{average} = \frac{\Delta\vec{r}}{\Delta t} \tag{2}$$

The instantaneous velocity is found by letting the time interval Δt go to 0.

$$\vec{v}(t) = \dot{\vec{r}}(t) = \lim_{\Delta t \to 0} \frac{\Delta\vec{r}}{\Delta t} = \frac{d}{dt}\vec{r}(t) \tag{3}$$

The dot notation $\dot{\vec{r}}(t)$, is used to describe the derivation. For a particle we now define the *state vector*[1] also known as the *phase state*[4] as

$$\vec{Y}(t) = \begin{pmatrix} \vec{r}(t) \\ \vec{v}(t) \end{pmatrix} = \begin{pmatrix} \vec{r}(t) \\ \dot{\vec{r}}(t) \end{pmatrix} \tag{4}$$

A state vector is a first-order differential equation and describes the instantaneous position and velocity of a particle. For a system of n particles it will be

$$\vec{Y}(t) = \begin{pmatrix} \vec{r}_1(t) \\ \vec{v}_1(t) \\ \vdots \\ \vec{r}_n(t) \\ \vec{v}_n(t) \end{pmatrix} = \begin{pmatrix} \vec{r}_1(t) \\ \dot{\vec{r}}_1(t) \\ \vdots \\ \vec{r}_n(t) \\ \dot{\vec{r}}_n(t) \end{pmatrix} \tag{5}$$

This is the kinematic part, but to simulate some sort of change in position and velocity we introduce forces acting on the particle at time t as $\vec{F}(t)$. $\vec{F}(t)$ is the sum of all external forces acting on the particle. The force is known from

Newton's second law of motion, stating that $\vec{F} = m\vec{a}$, where $\vec{a}$ is the acceleration and m is the mass. The mass of a particle is considered to be constant. Since acceleration is the change in velocity over a period of time, we can define it similarly to the velocity in (3)

$$\vec{a}(t) = \ddot{\vec{r}}(t) = \lim_{\Delta t \to 0} \frac{\Delta \vec{v}}{\Delta t} = \frac{d}{dt}\vec{v}(t) = \vec{F}(t)/m \tag{6}$$

This means that if the force $\vec{F}(t)$ change at time t, it will have influence on the acceleration and therefore change the direction and/or the speed of the particle.
We can now define the change in state as the derivative of the state vector from (4)

$$\dot{\vec{Y}}(t) = \frac{d}{dt}\begin{pmatrix} \vec{r}(t) \\ \dot{\vec{r}}(t) \end{pmatrix} = \begin{pmatrix} \dot{\vec{r}}(t) \\ \ddot{\vec{r}}(t) \end{pmatrix} = \begin{pmatrix} \dot{\vec{r}}(t) \\ \vec{F}(t)/m \end{pmatrix} \tag{7}$$

A numeric integrator can now be used to update the state of the particle or even the whole particle system. We use a verlet integrator as described by Thomas Jakobsen in [7], we give a description of the integrator in section 2.5.

2.2 Rigid body dynamics

The Newtonian principles described for a particle system can be applied to rigid bodies if we consider a rigid object as an infinite collection of particles. Rather than a single particle, we get a rigid object with volume, known as a rigid body. When referring to a particle in this section we mean one of the infinite many particles the rigid body can be considered composed of. The center of mass can be used as the reference point of the rigid body and can be found as a mass weighted average of all particle positions

$$\vec{r}_{cm} = \lim_{\Delta n \to \infty} \frac{1}{M} \sum_{i}^{N} (m_i \vec{r}_i) \tag{8}$$

Where M is the total mass and m_i is the mass of the *i'th* particle. If the mass density is distributed equally among the body, it can be written as an integral

$$\vec{r}_{cm} = \frac{1}{M} \int \vec{r}\ dm = \frac{1}{M} \int \rho(\vec{r})\vec{r}\ dV \tag{9}$$

Where ρ is the density and V is the volume. Since a rigid body has volume it can translate as well as rotate in 3 dimensional space. To describe the rotational movement a new coordinate system called the body frame (BF) is introduced[4, 1]. It has origin at the center of mass of the rigid body and follows the body such that all internal particles in the body is fixed with respect to the body frame. A 3×3 matrix, $\mathbf{R}$, is used to describe the orientation of the BF coordinate system. Each column in $\mathbf{R}$ correspond to the x, y and z coordinate axis of the BF given in world coordinates. A particle p in BF coordinates is then transformed into world coordinates by first rotating it about the origin and then translating it by $\vec{r}_{cm}(t)$

$$\vec{r}_p(t) = \mathbf{R}(t)\vec{p} + \vec{r}_{cm}(t) \tag{10}$$

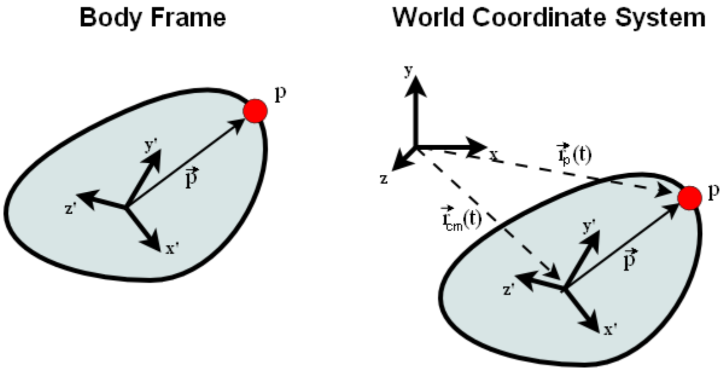


Figure 4: A particle p on a rigid body in the corresponding body frame and in the world coordinate system.

Figure 4 shows the situation.

The angular change in position $\Delta\phi$ of a particle, with respect to the center of mass, is measured in revolutions per second and is illustrated in figure 5. The angular change over time for the rigid body, is called the angular velocity, $\vec{\omega}$, and is derived by letting the timestep converge to 0, similar to the linear velocity from (3).

$$\vec{\omega}(t) = \lim_{\Delta t \to 0} \frac{\Delta\phi}{\Delta t} = \frac{d}{dt}\phi(t) \qquad (11)$$

The angular velocity $\vec{\omega} \in \mathbf{R}^3$, is a vector and defines the axis the body is rotating/spinning around. The magnitude of $\vec{\omega}$ tells how fast the body is rotating in counterclockwise direction, see figure 6

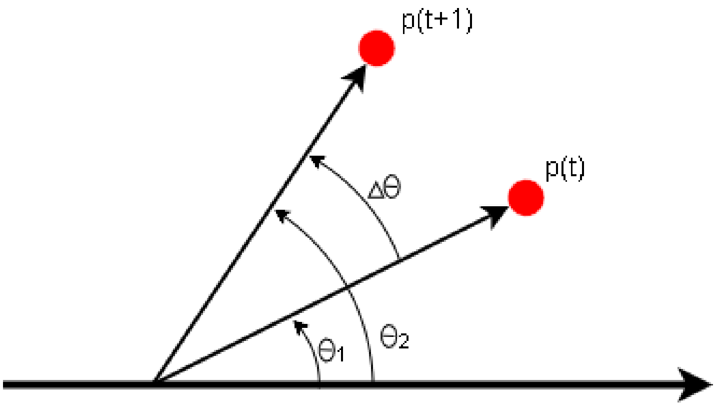


Figure 5: The angular difference of a particle at time t and $t+1$. The angular velocity is the change in angle over time.

To derive the velocity of any particle p in the rigid body, we use an approach similar to [12]. A more in depth derivation can be found in section 22.1.5 in [4]. We introduce two new vectors, $\vec{a}$ and $\vec{b}$. The vectors are illustrated in figure 7, $\vec{a}$ is parallel to $\vec{\omega}(t)$ and $\vec{b}$ is perpendicular to $\vec{a}$ and $\vec{\omega}(t)$. The position of p with respect to BF can be expressed as $\Delta\vec{r}_p = \vec{a} + \vec{b}$. At time t, the instantaneous

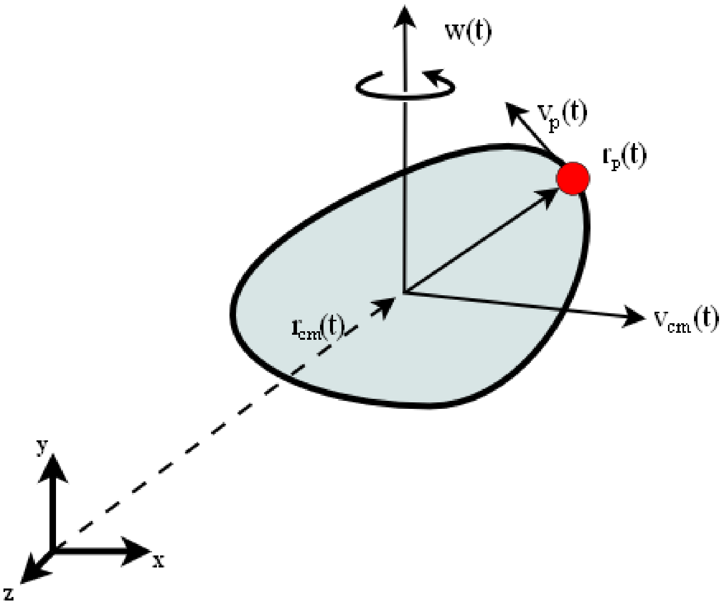


Figure 6: A rotating rigid body. The angular velocity vector defines the rotation axis. The particle at location $\vec{r}_p(t)$ does not have the same velocity as the center of mass, because the center of mass has no rotating velocity.

velocity $\Delta\dot{\vec{r}}_p$, has direction perpendicular to both $\vec{b}$ and $\vec{\omega}(t)$. Since it is rotating around $\vec{\omega}(t)$ at $\|\vec{\omega}\|$ revolutions per second, the magnitude of the velocity is $\|\vec{\omega}\| \cdot \|\vec{b}\|$. Since $\vec{b}$ and $\vec{\omega}(t)$ are perpendicular we also have that

$$\|\vec{\omega}(t) \times \vec{b}\| = \|\vec{\omega}(t)\|\|\vec{b}\| \tag{12}$$

which can be written as

$$\Delta\dot{\vec{r}}_p = \vec{\omega}(t) \times \vec{b} \tag{13}$$

finally, since $\vec{a}$ and $\vec{\omega}(t)$ are parallel

$$\Delta\dot{\vec{r}}_p = \vec{\omega}(t) \times \Delta\vec{r}_p \tag{14}$$

Using the orientation matrix R together with (14) to express the angular velocity of the whole body gives

$$\dot{\mathbf{R}}(t) = \left(\vec{\omega}(t) \times \begin{pmatrix} x_{xx} \\ x_{xy} \\ x_{xz} \end{pmatrix} \quad \vec{\omega}(t) \times \begin{pmatrix} y_{yx} \\ y_{yy} \\ y_{yz} \end{pmatrix} \quad \vec{\omega}(t) \times \begin{pmatrix} z_{zx} \\ z_{zy} \\ z_{zz} \end{pmatrix} \right) = \vec{\omega}(t) \star \mathbf{R}(t) \tag{15}$$

We now introduce forces to the rigid body. A force $\vec{F_i}$ acting on a particle p_i has the effect of moving/rotating the body. The force will effect the body with a torque defined as

$$\vec{\tau_i} = \vec{r_i} \times \vec{F_i} \tag{16}$$

By definition of the cross product, we see that $\vec{\tau_i}$ is perpendicular to $\vec{r_i}$ and $\vec{F_i}$. If $\vec{F_i}$ is the only force acting on the body, $\vec{\tau_i}$ will be the direction of the rotation

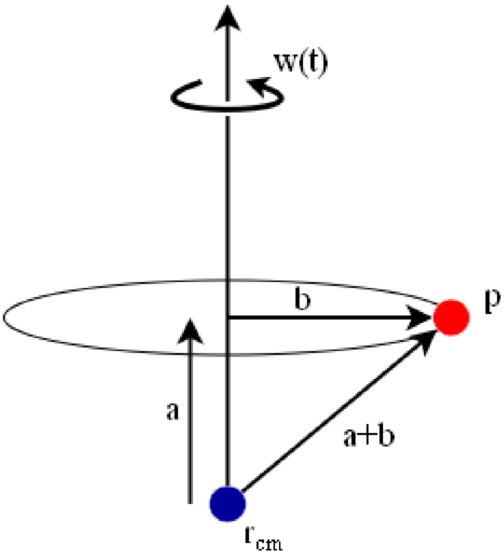


Figure 7: A particle p defined using two new vectors a and b. The instantaneous velocity of the particle has magnitude $\|\omega\|\|b\|$. r_{cm} is the center of mass.

axis. The torque for the whole body is $\vec{\tau} = \vec{r}_{cm} \times \vec{F}$. The linear momentum $\vec{p}_i$ and the angular momentum $\vec{L}_i$ of a particle i are defined as

$$\vec{p}_i = m_i \cdot \vec{v}_i \tag{17}$$

$$\vec{L}_i = \vec{r}_i \times \vec{p}_i \tag{18}$$

To find the change in angular momentum we differentiate with respect to time

$$\begin{aligned} \frac{d}{dt}\vec{L}_i &= \frac{d}{dt}\vec{r}_i \times \vec{p}_i && (19)\\ &= \frac{d\vec{r}_i}{dt} \times \vec{p}_i + \vec{r}_i \times \frac{d\vec{p}_i}{dt} && (20)\\ &= \underbrace{\vec{v}_i \times \vec{p}_i}_{0} + \vec{r}_i \times \vec{F}_i && (21)\\ &= \vec{\tau}_i && (22) \end{aligned}$$

The cross product $\vec{v}_i \times \vec{p}_i$ equals 0 because the vectors are parallel. The relation between angular momentum and torque is known as Euler's equation. For the whole rigid body the angular momentum can also be derived as [4, 12]

$$\vec{L}(t) = \mathbf{I}(t)\vec{\omega}(t) \tag{23}$$

$\mathbf{I} \in \mathbf{R}^{3\times3}$ is the inertia tensor and it is related to the mass distribution and therefore the amount of force needed to rotate a specific object. The inertia tensor has the form

$$\mathbf{I} = \begin{pmatrix} \sum_i m_i(y_i^2 + z_i^2) & -\sum_i m_i(x_i y_i) & -\sum_i m_i(x_i z_i) \\ -\sum_i m_i(x_i y_i) & \sum_i m_i(x_i^2 + z_i^2) & -\sum_i m_i(y_i z_i) \\ -\sum_i m_i(x_i z_i) & -\sum_i m_i(y_i z_i) & \sum_i m_i(x_i^2 + y_i^2) \end{pmatrix} \tag{24}$$

The diagonal part of the inertia tensor is called the products of inertia and the off-diagonal part are called moments of inertia. The inertia tenser is a function of the object orientation and must therefore be recomputed each time the object orientation change. However another formulation of the inertia tensor exist that only depend on the orientation matrix **R** and a fixed inertia tensor with respect to the body frame

$$\mathbf{I} = \mathbf{R}\mathbf{I}_{body}\mathbf{R} \tag{25}$$

The values in the inertia tensor $\mathbf{I}_{body}$, depends on the shape and size of the of the rigid body.

We have now derived enough information for the rigid body to make a state vector similar to the state vector for a single particle. The state is given by

$$\vec{Y}(t) = \begin{pmatrix} \vec{r}(t) \\ \mathbf{R}(t) \\ \vec{p}(t) \\ \vec{L}(t) \end{pmatrix} \tag{26}$$

To simulate the change in state we get the derivative

$$\dot{\vec{Y}}(t) = \frac{d}{dt}\begin{pmatrix} \vec{r}(t) \\ \mathbf{R}(t) \\ \vec{p}(t) \\ \vec{L}(t) \end{pmatrix} = \begin{pmatrix} \dot{\vec{r}}(t) \\ \vec{\omega}(t) \star \mathbf{R}(t) \\ \vec{F}(t) \\ \vec{\tau}(t) \end{pmatrix} \tag{27}$$

2.3 Particles vs rigid bodies

The theory just presented, provided us with state vectors for a particle and for a rigid body. It is easy to see, that the rigid body state vector is more complicated than the particle state vector. It should therefore be both faster and easier to compute the simulation of a singe particle than a rigid body. However, the particle-based method we seek to implement, uses four particles for every one rigid body. The final results will reveal which method is the most efficient. The state vector derived in (27) is an ordinary differential equation known as the Hamiltonian formulation, sometimes a Lagrangian formulation is used instead. For both equation (7) and (27) the motion can be found if initial conditions are known, using a numeric integrator. In section 2.5 we describe the numeric integration used in this project.

To perform the integration, the contribution of external forces and the torque has to be known at any given time. This is often the tricky part, and various methods have been developed to provide the integrator with some feasible values. [4] describes three different methods; penalty-based, impulse-based and constraint-based. The method used in this project is based on the constraint-based approach. The whole theory for multibody dynamics is very complex and too big a task for this paper. A short introduction is though given in the next section.

The particle-based method tries to simulate a rigid body by only using four particles. If the particle object should behave as the corresponding rigid body it tries to simulate, their properties should be equal. Finding the correct masses and positions of such particles are called particulating[12]. The first properties needed to be equal are the total mass and the position of the center of mass

$$M = \sum_{i=1}^{4} m_i \tag{28}$$

$$\vec{r}_{cm} = \frac{1}{M}\sum_{i=1}^{4} m_i \vec{r}_i \tag{29}$$

Hereafter the four particles should also have to the same inertia tensor as the rigid body. Since the inertia tensor is a symmetric 3×3 matrix, only the diagonal and three of the off-diagonal values need to be resolved

$$I_{xx} = \sum_{i=0}^{4} m_i(y_i^2 + z_i^2) \qquad I_{yy} = \sum_{i=0}^{4} m_i(x_i^2 + z_i^2) \qquad I_{zz} = \sum_{i=0}^{4} m_i(x_i^2 + y_i^2) \tag{30}$$

$$I_{xy} = -\sum_{i=0}^{4} m_i(x_i y_i) \qquad I_{xz} = -\sum_{i=0}^{4} m_i(x_i z_i) \qquad I_{yz} = -\sum_{i=0}^{4} m_i(y_i z_i) \tag{31}$$

This ends up with a system of 10 equations and 16 unknowns, meaning that more than one solution exists. We have limited ourselves from finding such a solution, because it would be a too time consuming task for this project. Wagenaar[12] implements a particulating program using a QuasiNewton method that seeks to minimize the square of the difference between the wanted and the target values. A nice feature in his program is the ability to add extra constraints, for example if particle masses or positions are known in advance.

Because we are not particulating, the visual result might not be exactly as wanted. What we need to do, is to place the particles such that they almost imitate the corresponding rigid body. Since particulating distributes the mass equally in the the object, we must try to place our particles such that they cover equally sized areas, to achieve a good result. In section A we discuss how the particles are placed with respect to the rigid body and in section 5.1 we discuss the visual impact of not using exact particulating.

2.4 Constraint-based multibody dynamics

Multibody is a short term for multiple bodies. It covers the representation and simulation of multiple rigid bodies, possibly connected by joints, also known as articulated bodies.

Our constraint-based method uses analytic mathematics to simulate the ragdolls. A thorough explanation of multibody dynamics can be found in [4], the following description is merely a shallow introduction.

The constraint-based method uses physics to calculate the contact forces, which is precise, but can be quite expensive. The representation of all bodies are concatenated into generalized position and orientation matrices. Likewise are velocities, forces, torques and contact conditions represented in matrices, making it possible to manipulate the rigid bodies by using matrix operations.

Constraints are used for modeling joints between rigid bodies. When talking about joints, the expression *degrees of freedom* (DOF) is often used. A single rigid body without limitations has 6 degrees of freedom, one for each translation direction (x, y, z) and one for each rotational axis. Two rigid bodies have 12 degrees of freedom and so on. When connecting rigid bodies with joints, some of the total degrees of freedom are removed. The constraint-based method uses holonomic constraints, defined as a function of spatial position, to remove each wanted degree of freedom. The spatial position vector $\vec{s}$ is written as

$$\vec{s} = [\vec{r}_i, \vec{q}_i, \vec{r}_j, \vec{q}_j]^T \tag{32}$$

Where $\vec{r}_i$ and $\vec{r}_j$ are the positions of the respective bodies and $\vec{q}_i$ and $\vec{q}_j$ are the orientations given in quaternions. The holonomic constraints are then defined as

$$\begin{aligned}
\phi_1(\vec{s}) &= 0, && (33)\\
\phi_2(\vec{s}) &= 0, && (34)\\
&\vdots && (35)\\
\phi_m(\vec{s}) &= 0 && (36)\\
& && (37)
\end{aligned}$$

Where m is the number of each removed degree of freedom. Contact conditions are expressed in an almost similar way, using non-holonomic constraints. Joint limits and are modeled by using a kinematic constraint formulation of jacobian matrices, derived by differentiation of non-holonomic constraints. Different jacobian matrices are then derived for each joint type.

2.5 Numeric integration

Recall from (7) that the change in state for a particle is expressed using the differentiated state vector

$$\dot{Y}(t) = \frac{d}{dt}\begin{pmatrix} \vec{r}(t) \\ \dot{\vec{r}}(t) \end{pmatrix} = \begin{pmatrix} \dot{\vec{r}}(t) \\ \ddot{\vec{r}}(t) \end{pmatrix} = \begin{pmatrix} \dot{\vec{r}}(t) \\ \vec{F}(t)/m \end{pmatrix} \tag{38}$$

This means, that if initializing conditions are given, the position and velocity of a particle can be updated using the scheme

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \dot{\vec{r}}(t) \tag{39}$$

$$\dot{\vec{r}}(t + \Delta t) = \dot{\vec{r}}(t) + \ddot{\vec{r}}(t) \tag{40}$$

Where Δt is the timestep. This is the classical way of updating the position and velocity. $\ddot{\vec{r}}(t)$ is found by applying all external forces acting on the particle, using Newton's second law of motion $\vec{F} = m\vec{a}$. However, we will not use this common approach, but instead a Verlet integration scheme as described by Thomas Jakobsen in [7]. The Verlet integration is a bit untraditional, because it does not store the velocity explicitly. Instead it uses the current position of the particle along with the old position to approximate the next position. The Verlet integration has the form

$$\vec{r}(t+\Delta t) = 2\vec{r}(t) - \vec{r}(t-\Delta t) + \ddot{\vec{r}}(t)\Delta t^2 \tag{41}$$

Hereafter the current position becomes the old position and is saved for use in the next iteration. If we look at the unit of the last part of the scheme $\ddot{\vec{r}}(t) \cdot \Delta t^2$, we have $(\text{length}/\text{time}^2) \cdot \text{time}^2$ which reduces to length. This is the contribution from the acceleration to the new position of the particle. The first part of the right hand side can be rewritten as

$$2\vec{r}(t) - \vec{r}(t-\Delta t) = \vec{r}(t) + (\vec{r}(t) - \vec{r}(t-\Delta t)) \tag{42}$$

$\vec{r}(t) - \vec{r}(t-\Delta t)$ is the difference between two succeeding positions, which is a fairly good approximation to the velocity at time t. The complete scheme therefore gives a good position update and since it is velocity-less, no drifting between position and velocity will occur.

3 Analysis of particle based ragdolls

A ragdoll is basically a body composed of jointed bones which react on collision with each other and the world around them. In this section we will describe our considerations behind an implementation of such a system in a particle-based system, starting with the bones and joints, and lastly how to handle collisions.

3.1 Ragdoll bones

The bones are the building blocks of our ragdoll. Each bone is basically independent in terms of movement, size and position. Setting bones up correctly, and putting joints between them will make our ragdoll come alive. In this section we will describe what is needed to define a bone.

3.1.1 Particles and constraints

The heart of each bone is a tetrahedron, which is composed of four particles connected by 6 stick constraints. A stick constraint is defined between two particles along with a rest length. The stick constraint then pulls and pushes the particles in positions such that the distance between them equal the rest length. This will make up a solid structure that can model the movement, size and orientation of our bones. The particles will not have individual masses, as the task of calculating the right masses is far from trivial. In section 2.3 we discussed the formulas needed for such a particulating computation. This limitation will unfortunately cause our final ragdoll to seem less realistic.

A nice feature of using constraints together with a Verlet integrator, is that the between each verlet step, readjustments to the particles can be done. Two types of adjustment is used, relaxation and projection. Relaxation means that all constraints are satisfied and projection means that all particles are projected out of possible collision penetrations. Since satisfying constraints and projecting particles, might lead to unsatisfied constraints and new collisions, the adjustment is done several times. The positions will then converge to more correct placements. The number of iterations of relaxation and projection can be adjusted, few satisfy calls will make the bone seem less rigid, but it will give a good performance. Oppositely, many calls will give worse performance but a better composition of the rigid body. When modeling human bodies, perfect rigid bodies are not necessarily a good thing, since the human body is not rigid. The number of relaxation and projection can therefore be relatively low for our purpose.

3.1.2 Visualize the bones

The particles and constraints together model the movement of the bone, and also provides us with a fast and easy way of making joints. However, they do not give a usable visual result of what we are trying to model. We have limited ourselves from skinning the ragdolls, instead we will display every bone as a box.

We need to define the center, size and orientation of each box. The center has to be defined in accordance to the bone (the tetrahedron), so the box will correctly follow the bone when it moves. This is done by defining a new coordinate system locally in the bone, much like a body frame, except that it does not needs to be placed at the center of mass. The local coordinate system will be computed using the particle positions of the tetrahedron. In this way we can always compute the coordinate system such that it follows the orientation of the tetrahedron. The center and orientation of the box can then be saved in local coordinates and converted to world coordinates when visualized on the screen.

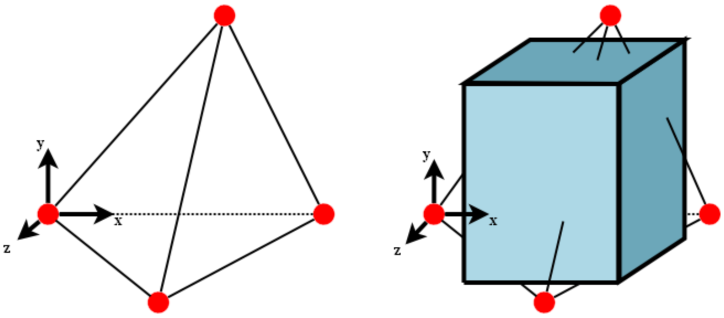


Figure 8: A tetrahedron and box defined using a local coordinate system. The local coordinate systems makes it possible to place the box independently on the particles positions.

3.2 Joints

Once the bones are constructed, we can attach them with joints. To model a joint, the theory suggests that two bones share one or more particles. Sharing one particle would model a ball joint, two particles a hinge and three would make them stick together completely, see figure 9.

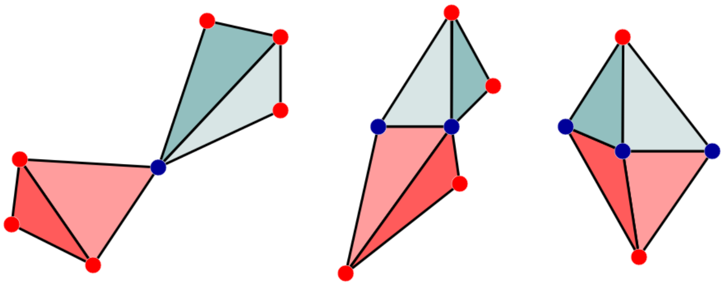


Figure 9: Three ways to model joints using particles. The first is a ball joint where the two boxes share one particle. Second is a hinge joint where two particles are shared. The last shows three shared particles which yield one rigid object.

There are two alternatives for jointing the bones. Either they share one or more particles, thereby sticking together. Or they have each their own set of particles, connected by a zero length stick constraint.
By sharing particles we need a minimum number of particles and constraints to model our ragdoll, which also means less system resources. Furthermore the two

bones are sure to be completely stuck to each other, and less drifting will occur in the joints. On the other hand, connecting two particles on top of each other makes it easy to assign different masses to each particle, to correctly model the total mass and inertia of the individual bones. On top of that it also opens for interesting features seen from a computer game perspective, where an arm could easily be severed from the rest of the body, by merely removing the constraint between the particles.
We have chosen to share particles, as it is the most efficient, and as we limited ourselves from using particle masses.

3.3 Joint limits

The jointed bones can portray a seemingly correct body. But without joint limits the simulation would look very odd, for instance the knee could bend forward.
For each joint limit there is three separate issues, namely defining the limits, detecting if they are breached, and handling the breaches. The two bones connected by a joint are denoted bone A and B. As long as the denotations are consistent, it does not matter which bone is A or B. We will handle limits for ball joints and hinge joints in separate ways, described in the next two subsections.

3.3.1 Hinge joint limits

A hinge joint is also known as a rotational joint, but we use the term hinge joint. The hinge joint is quite simple, as it has only one rotational degree of freedom, namely the rotation around the axis between the two shared particles. When limiting a hinge rotation, a positive and negative rotation angle has to be defined. The angles should be given as parameters when creating the limit, so that different hinge type functionalities can be modeled. If we denote the two particles of the joint as p_1 and p_2, the positive rotation direction is given as a right hand rule:

Definition 1 *Grab with your right hand with the thump pointing from particle p_1 to particle p_2, a positive rotation is then going in the direction of the rest of your fingers.*

Figure 10 illustrates how the limit can be thought of as the area between two half planes intersecting at the joint axis.

To express the relative position between bone A and B, we will exclusively use bone A's local coordinate system, given the positive and negative angle limits, and a reference particle in bone B. We can define two half planes with respect to bone A that confine the allowed movement of the reference particle in B. See figure 11.

To find out whether a limit is breached, we can calculate the signed distance between the reference particle and the two half planes. If the signed distance is negative, we must satisfy the angular constraint.

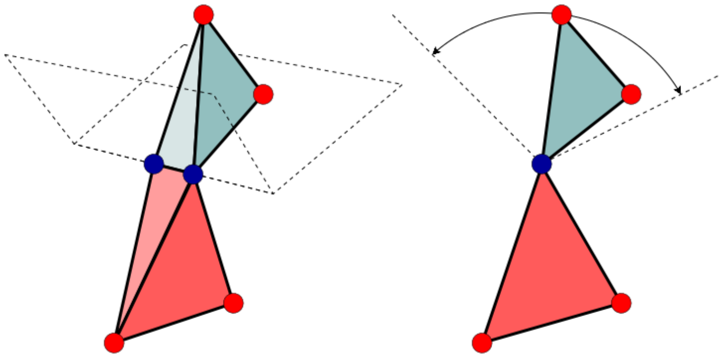


Figure 10: A limited hinge joint. The first figure shows the half planes in a 3D view. The second picture shows the same scene in a 2D view.

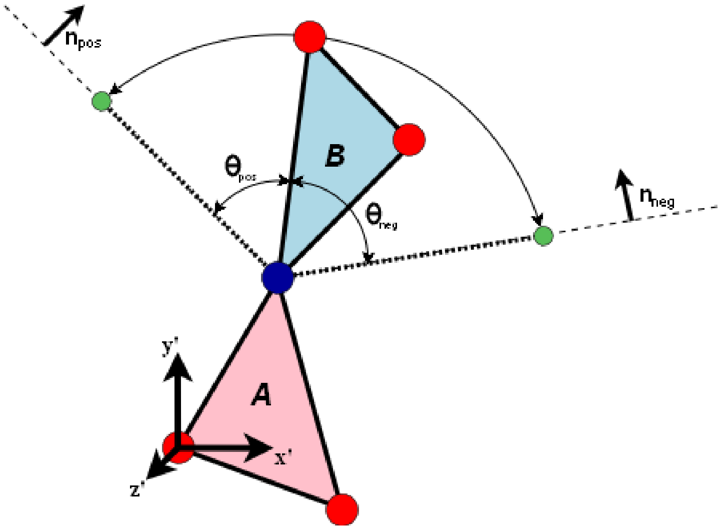


Figure 11: The confinement planes can be calculated by rotating the reference particle in B by the given angle limits.

We have considered two ways to correct a violated hinge limit. The simplest method would be to compute the world coordinates of the precalculated local limit point (the green dots in figure 11) and then simply project the reference particle in bone B to this position. This only calls for one coordinate system transformation, but it unfortunately has a drawback. If only the position of bone B is corrected it could result in an unnatural visual animation. Consider two hinge connected bones, A and B. If bone B falls to the ground with bone A on top of it and the angular limit then reaches, there is no other possibilities than to push bone B along the ground to satisfy the angular constraint. If they swap place it would look alright. Figure 12 illustrates the situation.

We have chosen to use a method that gives a more correct visual result. When the limit is initially set up, we define two stick constraints between a reference particle in A, and the reference particle in B. Each constraint is ini-

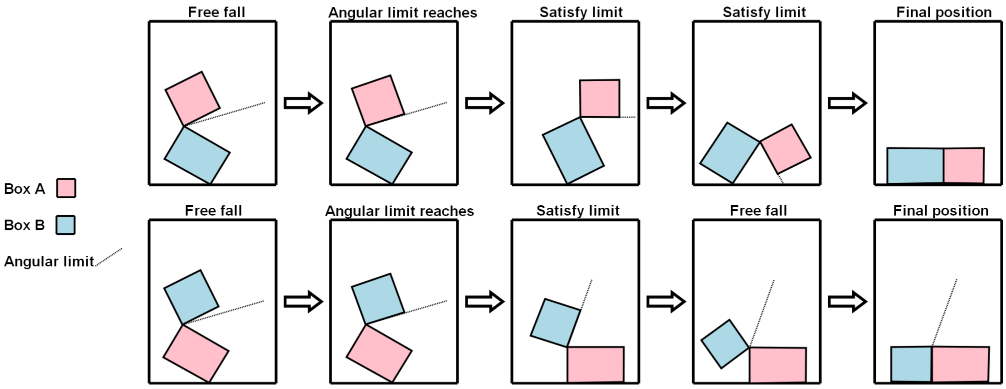


Figure 12: The same initial positions, but the outcome is different depending on which bone is A and B. If only the position of bone B is updated when the angular limit is breached, it might look unnatural, especially if bone B seems bigger and heavier than bone A (top illustration). The bottom illustration seems more realistic because bone B is on top of bone A. The figure shows the bone boxes and not the tetrahedra.

tialized with a rest length equal to the length from the reference particle in A, to each of the limit points of the reference particle in B (green dots in figure 11). Whenever a breach is detected, we satisfy the belonging stick constraint, and subsequently A and B are both affected by the constraint. Satisfying the stick constraint does not necessarily leave the bones correctly positioned, but using enough satisfy iterations, it should end up with a nice visual result. Figure 13 illustrates how the constraints are set up.

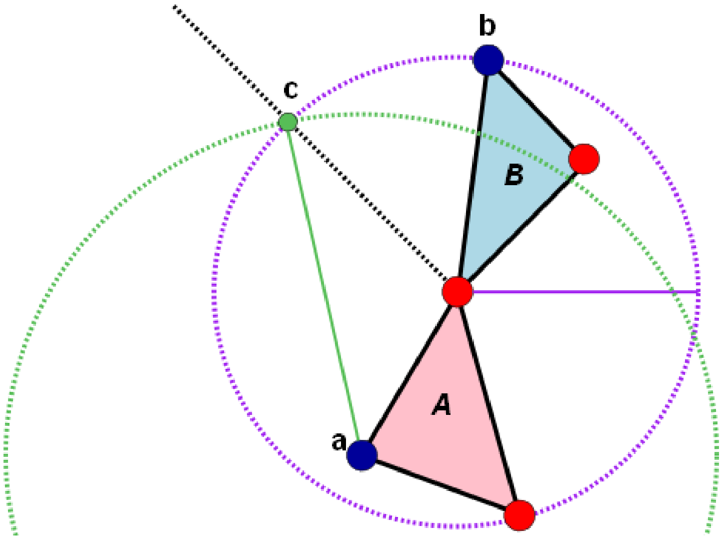


Figure 13: The blue particles, a and b, are selected as reference particles. The hinge axis limits the movement of particle b to the purple circle. The green circle indicates the impact of the angular stick constraint, found by using the length from a to the limit point c (green dot). If a breach occurs, all stick constraints will be satisfied, which converges to the only legal position of b, namely the intersection between the two circles c.

The method just presented limits the total rotation angle to 180 degrees, because the limit planes would swap over each other from the opposite sides. To solve this, a third plane is placed along the hinge axis and through the initial position of the measure particle in bone B. When checking for breaches, the signed distance to this plane is used to tell us on which side the particle is on.

3.3.2 Ball joint limits

A ball joint can also be referred to as a spherical joint. The ball joint has all three rotational degrees of freedom. Limiting the ball joint is a bit more tricky than limiting a hinge joint. Because the ball joint has more degrees of freedom, no point can be found on a limit border. Different approaches has been developed to simulate ball joint limits. A interesting discussion is found in [13]. The authors create a reach cone defined by multiple direction vectors, which makes it possible to create a non convex reach cone. This is a nice feature if the movement of the jointed objects should bend more in some directions than others. For our purpose we find it sufficient to use a circular convex cone as the limitation border. The ball joints we need for the human body, are for the shoulder, hip and neck. By observing ourselves we have found it to be a fair restriction. Using a circular cone restriction enables us to propose a simpler method.

As with the hinge joint, an angular limit parameter should be passed along when the constraint is created. But the angular limit is not enough, since only one angular limit is given and the particles of the bones does not necessarily start in a centered position with respect to the joint. Instead a plane normal will be used to define a plane going through the joint position. The reach cone is then defined by the angle going outwards from the plane normal starting at the joint article. Figure 14 illustrates the cone definition.

We have analyzed different methods to keep the limit satisfied. An intuitive way would be to use bone B's center of mass as a reference point, and then detect when the point is outside the cone. It should give a good visual result, but using a reference point that is not a particle, leaves us with the problem of reflecting the breached limit back to the particles, which is not a trivial task. Another option is to use a random particle or even all particles of bone B, and make sure they are inside the cone. This has the drawback, that the positions of the particles might not be placed such that it results as visually intended. Figure 15 illustrates how two visually alike bones results in different movement, when the particles have different positions.

A good reference particle to chose would be a particle that is near the center line of the reach cone in the initial position, because it will give the bone an equally sized rotational freedom in all directions. In appendix A we discuss how our human model is composed. It turns out that for all the ball joints needed, we have such a particle laying exactly on the center line of the cone. Therefore we use this method and pass along the reference particle when creating the ball joint.

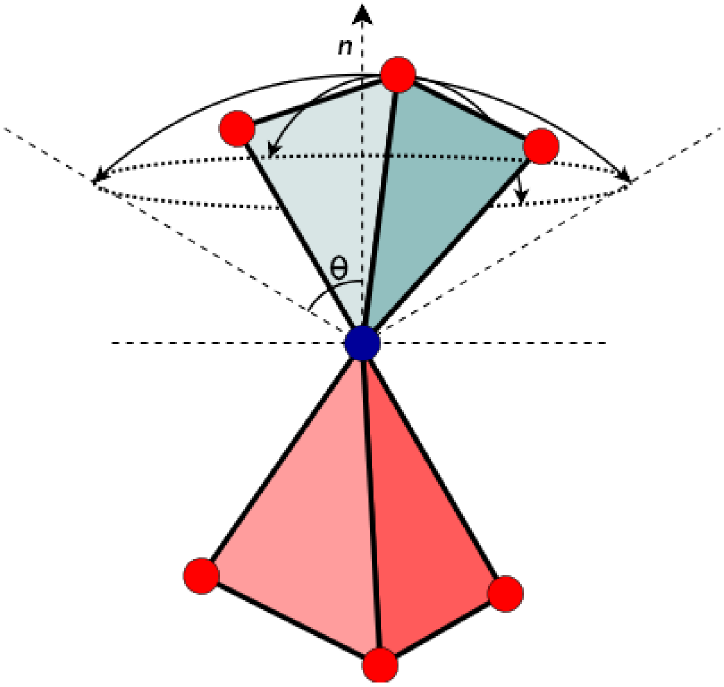


Figure 14: A ball joint and a limiting reach cone defined using a limit angle, θ, and a plane normal n. If a reference particle is chosen it will trace out a circle on the cone (dotted line).

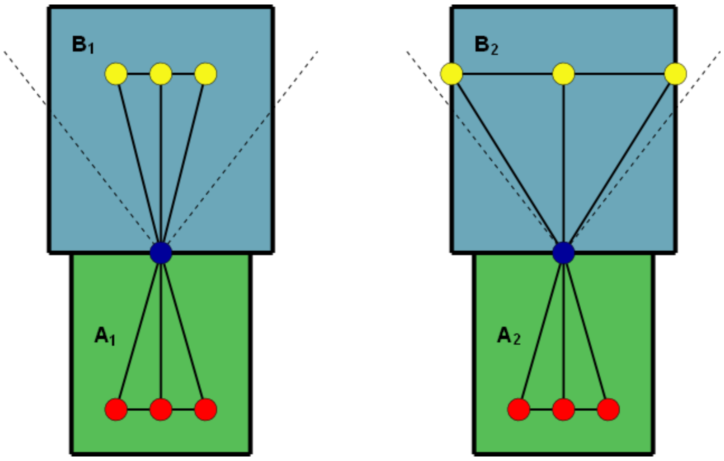


Figure 15: A ball joint between two bones A and B. If all the yellow particles of bone B is chosen as reference particles and projected back when breaching the limit cone, then the relative positions of the particles will have great impact on the movement and hence the visual result. Bone B_1 will be able to move much more than bone B_2 even though they have the same box sizes.

We have now found the particle to test against the cone, but how do we detect and solve a breach? As seen in figure 14, the reference particle will trace out a circle on the intersection with the cone. The distance c_{dist}, from this circle to the plane does not change as the simulation starts, which means it can be precomputed by using trigonometry, when the joint is created. Figure 16 shows the computation needed to find c_{dist}.

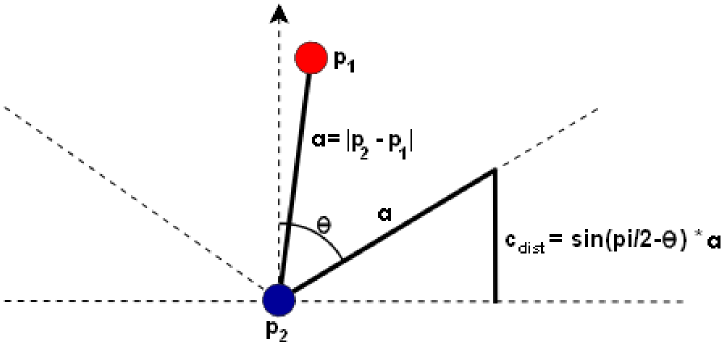


Figure 16: Using the length a from the ball joint particle p_2 to the reference particle p_1 along with the cone angle θ, the length from the plane to the circle, c_{dist} can be computed.

To test whether the limit is breached, the signed distance from the reference particle to the plane, p_{dist}, is computed and compared to the circle distance c_{dist}. To project the bones back in place, we could use trigonometry to calculate the exceeded angle and project all particles back by this angle. This is a bit complicated and not very efficient, because it uses sine and cosine functions, along with matrix operations. Instead we do as with the hinge joint and place a stick constraint between the reference particle in B and any particle in A that is not the ball joint particle. The restitution length of the stick is not known in advance, since the breach point is not unique (it can be anywhere on the circle). To solve this, we set the restitution length to be the length between the stick particles plus the size of the breach, that is $c_{dist} - p_{dist}$. The stick constraint is then satisfied and both bones are projected back.

3.4 Collision Detection

OpenTissue already has tools for detecting collisions, these return collision points and depths between two objects, if any collision exists. We need to run the collision detection between any two bones in our ragdoll, including the ground, with one exception; bones that are tied together by a joint. Where as real human bodies are soft and flexible, the ragdolls are composed of solid boxes, which might just barely touch in their resting position, and overlap somewhat in all other positions. Therefore it makes no sense to perform collision detection on them, as it would render the ragdoll perfectly stiff.

3.5 Collision Handling

When a collision has been detected, we need to handle it to reflect the impact on the two colliding objects. This task is not trivial, as the collision has been detected on the bounding box, but the effects has to be handled by the underlying rigid body made of particles.

This opens up for 2 different approaches, either we calculate the effect on the

bounding box, and then translate it to the rigid body or we translate the collision directly on to the rigid body.
We have chosen to go by the second approach, as it is the most direct, even though the translation of a collision point outside the tetrahedron does seem counter-intuitive. Below we will discuss how we handle a collision.

3.5.1 Multiple contact points

In the real world, collision depth is non-existent, since the reaction on the colliding objects will happen the instant they meet. However, our physical model is highly dependant on contact depth in order to calculate the reaction.
Depending on the size of our timesteps, detection of multiple collisions of varying depths is a relatively frequent occurrence.
The physical model we use suggests that all collisions, regardless of depth is to be handled. With an infinitely small timestep however, the collisions of lesser depth would never have occurred, since the deepest collision(s) would have caused a reaction that would prevent the other collisions from happening, much like in a real world scenario.
We will attempt to model an approximation of the real world physics by only handling the first collisions. This will be done by finding the deepest collision(s). If there is exactly one, we handle this collision exclusively. If there are more with the same depth, we assume that they had an impact at the same time, and handle them together.
Handling several collision points with the same depth seperatly, would give the same result as calculating one point that's exactly the average of the other points, and then handling one collision based on that point.
This leads us to a situation, where we first find exactly one collision point, which is either the deepest or a combination of multiple deepest points, and then handle just one collision. This is more efficient than handling up to 8 points seperatly and arguably also a more correct approach.
One might argue that this approximation has too many flaws, such as the contact point we use is most likely wrong due to rotation and penetration angle - it might not even have been the original contact point. These observations are correct, but were also factors if we had handled all contact points. Even though our model will create a slightly different result in most cases of multiple contact points, we do not believe it to be less correct.

3.5.2 Box masses

In reality, the mass of a given box is not necessarily linear to its size. However we have chosen to make this simplification, since it comes close to reality, as all the boxes we are modeling consist of skin, flesh and bones.
Two bones of the same size (and thereby mass) should be translated equally far in separate directions upon collision. Where as a smaller box should be translated further than a bigger box when they collide with each other.

This results in the following scaling of the effect on the colliding bones:

$$\Delta_a = \frac{M_b}{M_a + M_b} \cdot (\overrightarrow{n} \cdot d)$$

$$\Delta_b = \frac{M_a}{M_a + M_b} \cdot (\overrightarrow{n} \cdot d)$$

Where $\overrightarrow{n}$ is the collision normal, and d is the collision depth. Box masses would have been unnecessary if we instead included particle masses in our implementation.

3.5.3 Rotation

Now that we know the translation direction and distance of the two bones, we must calculate the effect on each particle individually. If all particles were affected evenly, it would eliminate all rotation of bones upon collision handling, which for instance would result in a situation similar to figure 17.

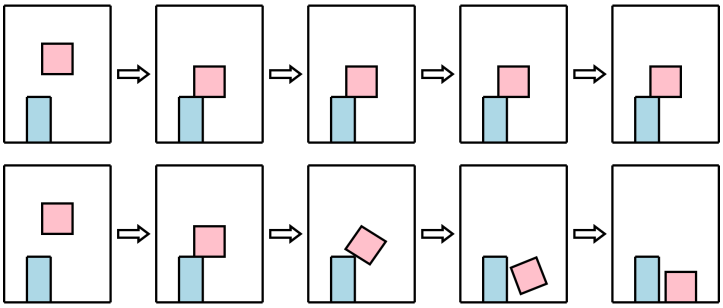


Figure 17: A falling red box that collides with another blue box. The collision normal $\vec{n}$ points upwards. The top situation shows what will happen if the particle translations are not scaled and the lower shows the same situation using scales.

We will implement the individual particle translations as Thomas Jakobsen suggests in [7]. The particle scalars c_1 to c_4 needs to be defined such, that the closer a particle is to the collision point, the larger the scalar should be. Furthermore all four must be between zero and one, summing to a total of one. Jakobsen suggests determining the scalars by linear combination. This scheme would however only work if the collision point was on or inside the tetrahedron. In our case, the collision occurs on the bounding box, which will result in a linear combination of large positive and negative scalars, which in turn, would result in a very small λ (see definition below) and end up in projections being too small and both positive and negative. Instead we will use the following formulas:

$$c_1 = \left(1 - \frac{\|\vec{P} - \vec{p}_1\|}{\|\vec{P} - \vec{p}_1 + \vec{P} - \vec{p}_2 + \vec{P} - \vec{p}_3 + \vec{P} - \vec{p}_4\|}\right)/3$$

$$c_2 = \left(1 - \frac{\|\vec{P} - \vec{p}_1\|}{\|\vec{P} - \vec{p}_1 + \vec{P} - \vec{p}_2 + \vec{P} - \vec{p}_3 + \vec{P} - \vec{p}_4\|}\right)/3$$

$$c_3 = \left(1 - \frac{\|\vec{P} - \vec{p}_1\|}{\|\vec{P} - \vec{p}_1 + \vec{P} - \vec{p}_2 + \vec{P} - \vec{p}_3 + \vec{P} - \vec{p}_4\|}\right)/3$$

$$c_4 = \left(1 - \frac{\|\vec{P} - \vec{p}_1\|}{\|\vec{P} - \vec{p}_1 + \vec{P} - \vec{p}_2 + \vec{P} - \vec{p}_3 + \vec{P} - \vec{p}_4\|}\right)/3$$

Where $\vec{P}$ is the collision point and $\vec{p}_1$ to $\vec{p}_4$ are the particle positions.
Having these values we can use the rest of Jakobsen's formulas:

$$\lambda = \frac{1}{c_1^2 + c_2^2 + c_3^2 + c_4^2}$$

$$p_1' = p_1 + c_1 \cdot \lambda \cdot \Delta$$

$$p_2' = p_2 + c_2 \cdot \lambda \cdot \Delta$$

$$p_3' = p_3 + c_3 \cdot \lambda \cdot \Delta$$

$$p_4' = p_4 + c_4 \cdot \lambda \cdot \Delta$$

with Δ as explained in 3.5.2. This results in our four particle positions being replaced with the four new particle positions p_1' to p_4' and the collision has been handled.

3.6 Friction

The collision handling described previously has a flaw when it comes to believability, namely there is no loss of movement energy at any point. In the real world, friction between objects as well as air friction, transforms energy from movement into heat. This does not occur in our system.

When two objects with different velocities touch, friction arises. This slows down both objects by a certain amount. We will limit ourselves from going into much detail in this. Instead we will make a very simple version, where every time two objects collide, they automatically loose a small fixed amount of their velocity, regardless of the circumstances.

The way we will implement this, is every time collision handling is invoked, the old position of all particles in both bones will be moved closer to the current position by a fixed amount. Thereby slowing the implicit velocity for the next verlet integration step .

This is not a physically correct implementation, but a very efficient estimation that should give a satisfying visual result.

4 Implementation notes

In this section we discus the structure of OpenTissue and the modifications we did while implementing the ragdoll methods. OpenTissue is implemented using intense C++ template style. We therefore recommend the book [11] describing the aspects of template programming in C++.
The OpenTissue revision 3144 has been used throughout the implementation. All ragdoll code files can be found in the appendix. Instead of giving a fully description of the classes and their functionalities in this section, we recommend interested readers to read the comments in the files. All classes and functions are equipped with useful comments.

4.1 Integrating with OpenTissue

OpenTissue is already abundant with useful physics and math libraries for simulation purposes. We will utilize these libraries as much as possible, there is no need to reinvent the wheel.
Our two ragdoll methods fit nicely into two of the sublibraries in OpenTissue, namely `multibody` and `psys`. Both of them are part of the `dynamics` library. To model the geometry objects needed, we use functionalities from the geometric folder. Finally we will use the collision detection functions and a huge amount of arithmetic operations. A simple overview over the used folders is shown in figure 18.

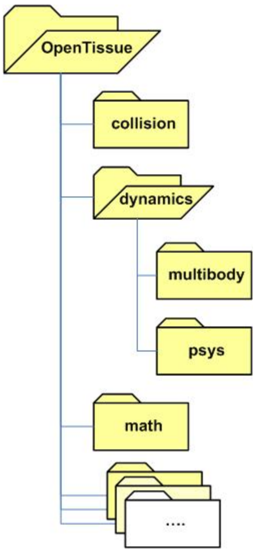


Figure 18: Overview of the used folders in OpenTissue

Taking a closer look at `psys` we see that it contains functionalities to handle and manipulate multiple particles. The class `PSYSSystem` works as a container for all particles. `PSYSMassSpringSystem` is an extension from that class and makes it possible to add constraints and forces to the system. `psys` contains the folder `util` where we will place our ragdoll class. Other useful classes needed,

such as joint constraints, will go into the respective folders.
Since the ragdoll is made of particles, we find it natural to let the ragdoll class itself extend `PSYSMassSpringSystem`. That will give us the needed functionalities to manipulate the particles. It is also straightforward to conceive the ragdoll as a particle system. To make different articulated bodies, we make yet another class that will extend the ragdoll class. In this class the composition of the body is made. Using such an extending class will enable flexibility to make different ragdolls. For this project we wish to make a human ragdoll, but a ragdoll could just as well be of a dog, a horse or something entirely different.
The ragdoll is composed of bones, each consisting of four particles. A ragdoll bone class will be created with pointers to each of the four particles. The bone class will take care of the continuous update of the local coordinate system and useful setup functionalities.
Since all bones are visual represented using boxes, only the `collision_box_box_improved` function will be used to detect collisions. The collision detection will be called directly from the ragdoll class, even though it is not a very intuitive way to do it, it is easy and not less effective, so we have limited ourselves from implementing more advanced techniques.
For the analytic approach we mainly use the `multibody` also known as `retro` sublibrary. To make the two implementations alike, we again make a human ragdoll class and a ragdoll class to extend from. The parts to make a ragdoll in `multibody` are already available, so our task is limited to modeling the ragdoll classes with the necessary utility functions and then build the human body. Collision detection methods are already implemented in other multibody demos, so we just reuse the techniques.

5 Comparison and results

The goal of our efforts has been to make a comparison between the two methods. This section will evaluate them against each other both on visual results, and on their performance.
Starting this section marked the end of our implementation and tweaking. We are well aware that our program is not entirely free of bugs, but this section will describe them as they occur, and describe possible fixes, they will however not be implemented and retested.
After the tests, we will give a conclusion on the two ragdoll methods compared to each other, followed by a list of ideas to improve and expand the current implementation.

5.1 Visual results

In this section we will produce a series of test scenes, designed to show the visual result of each part of our implementation. We will start by testing simple cases in our implementation, and then work our way up to more advanced parts of the implementation, ending with the ragdoll which utilizes all parts of the implementation. All of the simulations contain the same scene, with the two different methods. The left part of the screen image has a blue scene which is the particle-based simulation, the right part has a red scene which is the constraint-based multibody simulation.
This test is not designed to be exhaustive, as that is an impossible assignment for visual results. It is, however, designed to check that all parts of the program works individually as well as together.

5.1.1 Test executables

The enclosed CD contains all of the test simulations as individual executable files, named `test1.exe` to `test8.exe`. We highly recommend to run the files, because the following screenshots are too small to reveal details and to give a good impression of motion. To use these executables, run them and use the below controls to interact with the application:

- **Right Mouse Button** gives a menu with all the below options.
- **Left Mouse Button** enables rotation of the point of view.
- **Space** starts the animation.
- **T** runs a single timestep of the animation.
- **S** stops/starts the simulation, started by default.
- **ESC or Q** ends the program.
- **1** Resets the simulation to start.
- **2** starts/resets other implementations of this simulation (only available in tests 2, 3, 4).

- **3** starts/resets other implementations of this simulation (only available in test 3).

The simulations must be executed from an x86 compatible architecture.

5.1.2 Test 1. Basic collision handling

The test scene consists of two boxes falling onto the ground hitting each other. This setup will test the most basic parts of our implementation, including setting up the particles, sticking the bounding boxes to them, detecting collisions between bones, handling the collisions and applying friction. Figure 19 displays a series of screenshots from the simulation.

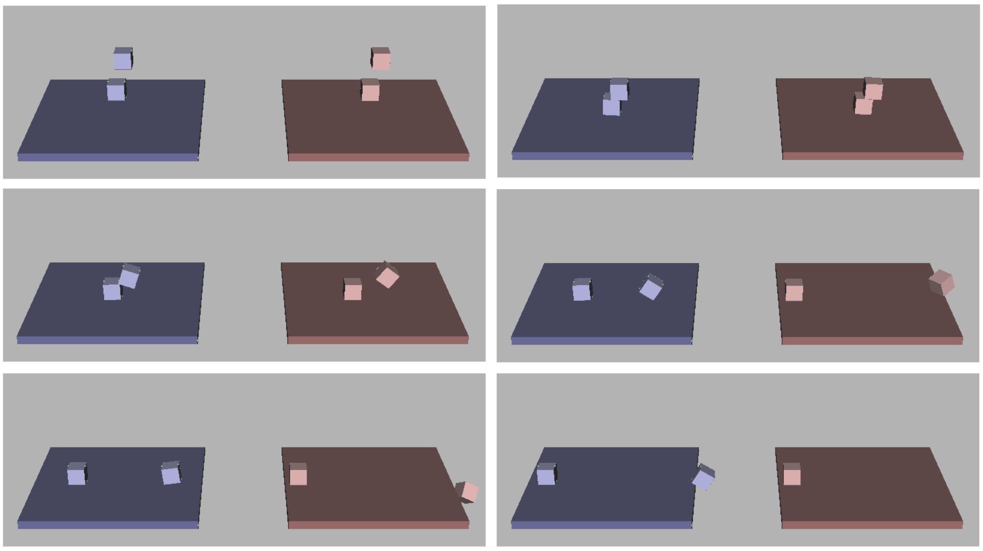


Figure 19: Screenshots of the test 1 simulation, the screenshots are ordered chronologically, going in conventional reading direction. Each screenshot contains both simulation methods, the left (blue) is the particle system, and the right (red) is the multibody version.

The two simulations look much alike, and both look realistic. We are satisfied with the result.

5.1.3 Test 2. Basic ball joints

The test scene consists of 6 boxes attached as a string with ball joints, the top box is fixed, forming an imitation of a pendulum. The 5 non-fixed boxes start in an elevated position, giving the initial position a lot of potential energy. A seventh box will fall and hit the pendulum, to introduce some liveliness to the setup.

This setup will test non-limited ball joints, and further show collision detection and handling. Figure 20 displays the visual result of this test.

The initial test setup showed an obvious error in how the scene had been generated. The setup of the particle system part (the left, blue part of the

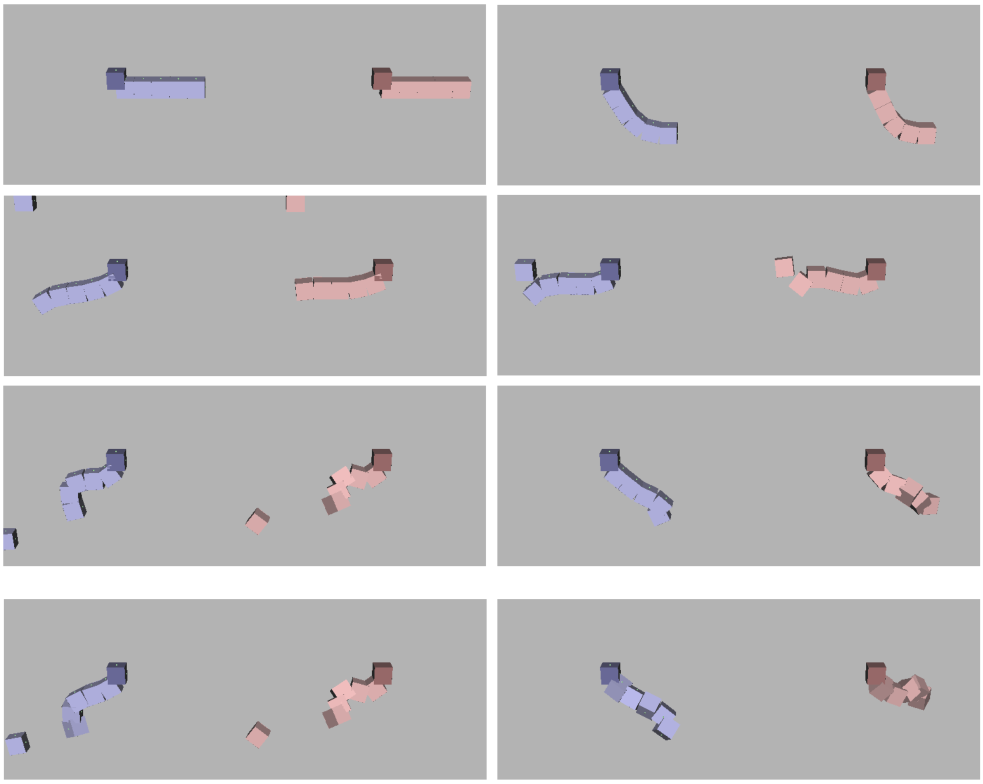


Figure 20: Screenshots of the test 2 simulation, the screenshots below the dotted line are additional screenshots after the method was corrected.

screenshots in figure 20) did not gain rotation on the pendulum as it should have, it acted much like hinge joints would have done.
This error is due to how the particles was positioned within the boxes. All particles were aligned at the same Y-coordinate (the depth coordinate) A strictly vertical collision on such a setup, will result in a projection of all particles that follow the vertical collision vector, and regardless of weights, they will stay aligned on the Y coordinate.
To achieve the expected effect of box rotation, we redefined the particles of one of the boxes in the collision. The row of images below the dotted line in 20 shows this better result.
The executable `test2.exe` on the enclosed CD contains both the original and corrected simulation. press `1` for the original setup and `2` for the corrected version (`1` is the default choice at start up).
The corrected simulation runs much like expected, and both methods provide a realistic visual results, but it seems obvious that the energy disappears much faster in the particle-based method. We believe this occur because the stick constraints pull back falling particles, and hence some of the velocity are removed from the particles.
We find the results satisfying.

5.1.4 Test 3. Basic hinge joints

The test scene is almost identical to the previous scene and models an imitation of a pendulum, the difference is that this setup uses hinge joints rather than ball joints.
This setup will test non-limited hinge joints as well as collision detection and handling. Furthermore we expect the particle-based part of the setup to reveal that the hinge joints are not perfectly confined to one degree of freedom, which is a result of the method relying on multiple iterations to converge to the correct result, as discussed in 3.1.1. Figure 21 shows screenshots of the animated scene.

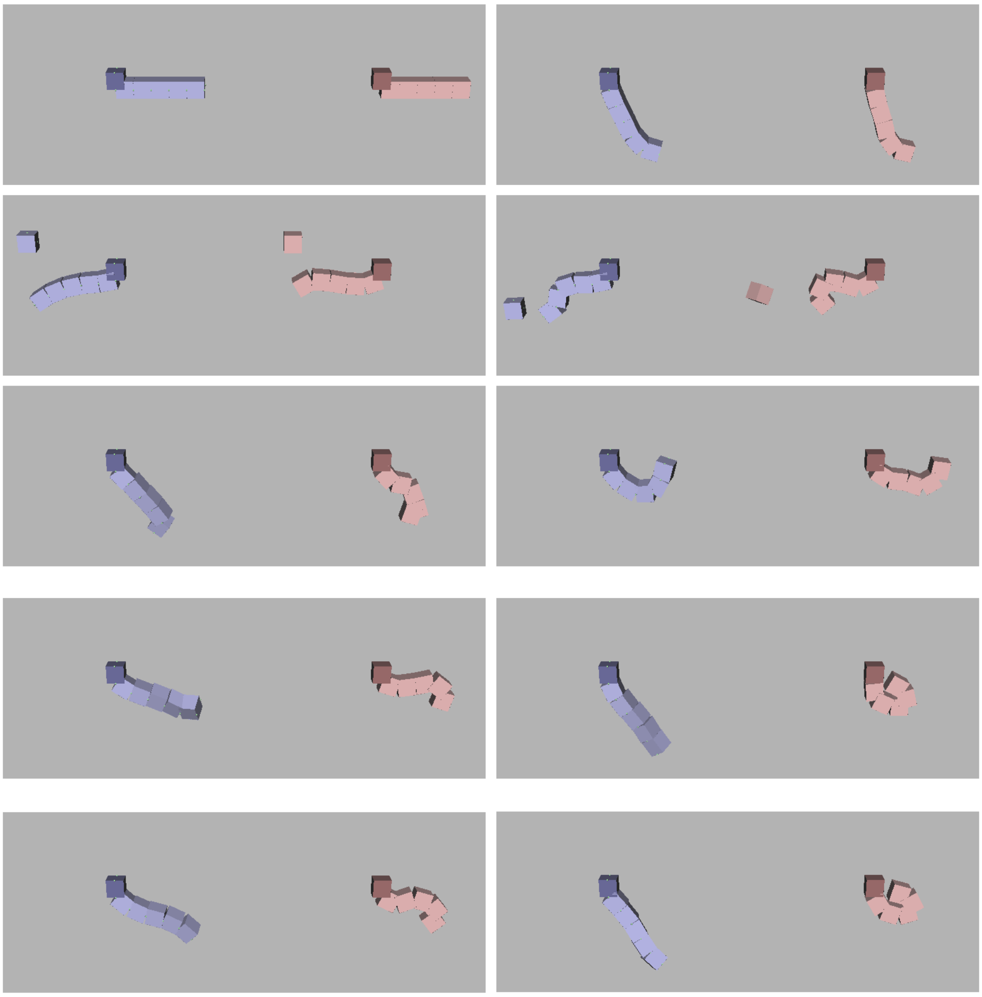


Figure 21: Screenshots of the test 3 simulation, the top 6 screenshots illustrate 5 iterations per timestep. Below the first dotted line is a simulation using only 2 iterations per timestep, and below the lowest dotted line is a simulation using 20 iterations per timestep.

The original test runs much like expected, and gives a satisfying visual result on both methods. It is also very obvious to see that the number of iterations has a impact on the rigidity of the hinge joints in the particle-based method. We recommend running `test3.exe` on the enclosed CD, pressing key `1` will show

our standard (5) iterations, key `2` is 2 iterations per timestep and key `3` is 20 iterations per timestep. The visual differences between the 5 and 20 iterations are not very significant. Hence, if the wanted result is found using few iterations, there is no need to go higher.
It is worth noting, that a major reason to the lack of rigidity being so obvious in these tests, is due to the bones consisting of boxes. If they had been cylinders, or perhaps skinned with human features, the lack of rigidity would have been less apparent - if not invisible. We find the results of the test satisfying.

5.1.5 Test 4. Box masses

The test scene is two boxes falling towards each other and colliding. One of the boxes is eight times larger than the other, and subsequently has a mass that is eight times as high. Upon their collision we expect the big box to dominate, and push the small box away loosing only little speed.
This setup will test the box masses, along with collision detection, collision handling and friction. Figure 22 shows screenshots of the simulation.

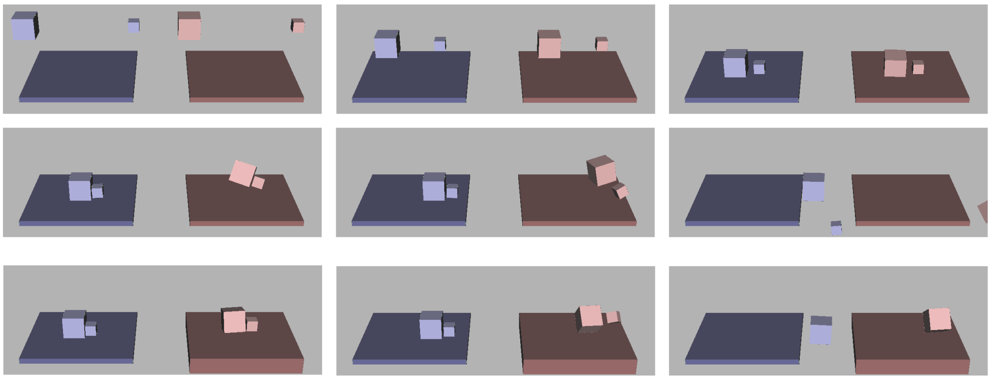


Figure 22: Screenshots of the test 4 simulation, the top screenshots above the dotted line shows the initial setup. Below the dotted line is a simulation with the two attempts to fix the odd behavior of the constraint-based (red, right) method.

The particle-based part of the initial setup ran much like expected, where as the constraint-based moved slightly through the floor before getting handled, which resulted in the box being shot up from the floor looking very unnatural. The masses on the constraint-based method was set manually in our test setup, and as a possible fix, we tried setting the masses 10 times lower than our initial setup (which leads to the same mass ratio between boxes, but less mass), also we made the floor in the constraint-based method 3 times as thick. The result was pretty much the same, but the velocity deteriorated much faster, see figure 22. Additionally, both tests can be seen by running `test4.exe` on the enclosed CD, pressing `1` will show the initial set up, and `2` will show our attempt to correct it. We find the result for the particle-based method satisfying. The multibody method was not visually realistic in either of the two tests.

5.1.6 Test 5. Limited ball joints

The test scene consists of four fixed boxes, each joined by another unfixed box by a limited ball joint. The top box has a 0 degree limit, the next a 45 degree limit the third a 90 degree limit and the bottom joint has a 135 degree limit. This setup will test ball joints and limits. Figure 23 shows screenshots of the simulation.

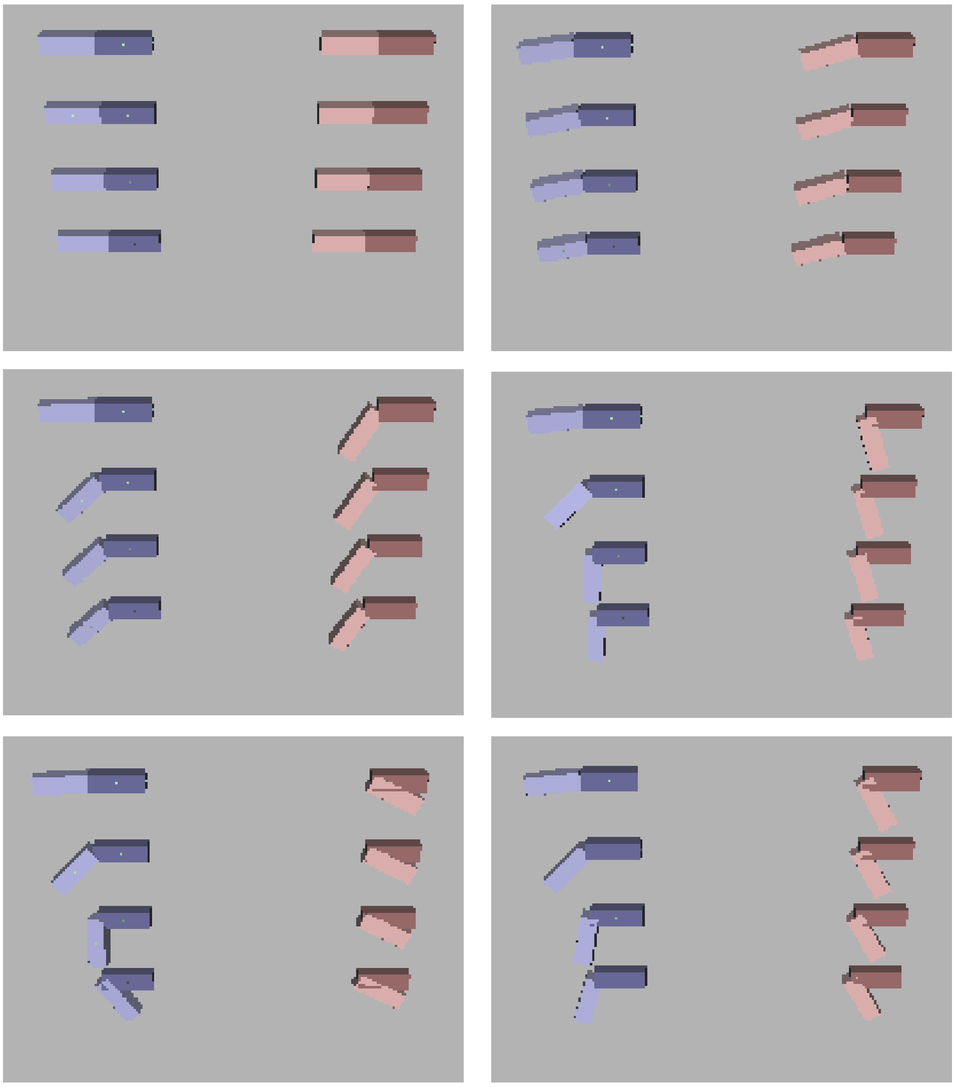


Figure 23: Screenshots of the test 5 simulation

The multibody ball limits has been dropped since they caused the program to shut down. However it is obvious that the particle-based model is far from perfectly rigid. Especially the 0 degree particle-based illustration is obviously not fixed in the initial position as it ideally should be. This is however one of the most obvious displays of this lack of rigidity, and in a moving simulation it will rarely become this obvious.

5.1.7 Test 6. Limited hinge joints

Much like the scene in the previous test, this scene consists of four fixed boxes joined to four other boxes, again with the top joint having a 0 degree limit, and 45, 90 and 135 degrees for the others. These boxes are however joined by limited hinge joints.
This setup tests hinge joints and limits. Figure 24 shows screenshots of the simulation.

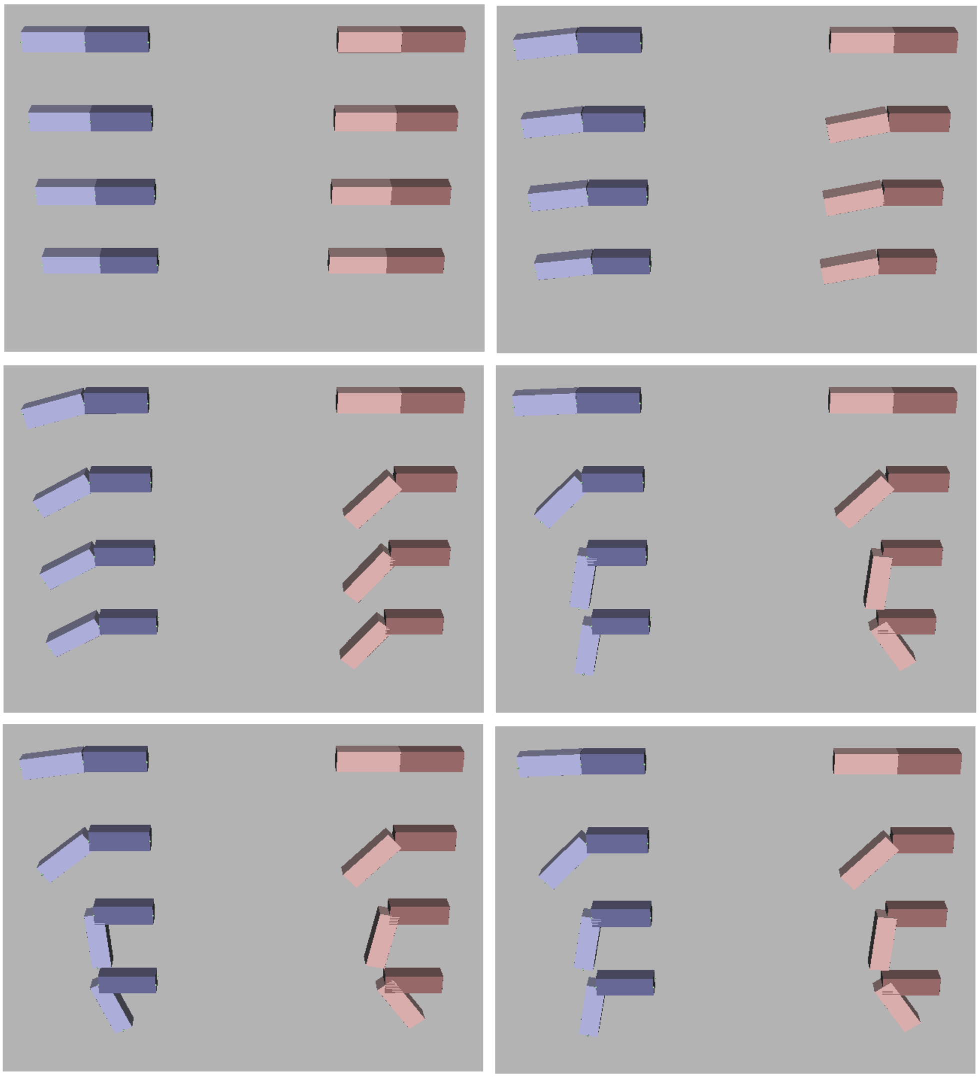


Figure 24: Screenshots of the test 6 simulation

As with the limited ball joints in test 5, it is obvious that the particle-based model is not perfectly rigid, seen here next to a multibody version of the same

setup. Multibody provides a much more rigid joint.
One other problem that shines through, is the 135 degree particle system joint, that after a few swings, gets stuck at the constraint. This is due to the way we have implemented the hinge joint limits, where a stick constraint is satisfied between a pair of non-jointed particles of the two boxes whenever a limit is breached. Here however, one of the boxes is fixed, and the particles will thus not be moved far enough from each other, resulting in a continuous breach of the constraint. A trivial fix to this problem, would be to check if either bone was fixed, and then apply all of the projection to the unfixed bone, we will however not implement this at this point, as the ragdolls we intend to model does not contain fixed objects.
We find that both methods give a satisfying result.

5.1.8 Test 7. Articulated figures

The scene consists of a 4 legged spider, modeled by a cube in the center, with ball joints on each side to 4 upper legs, which are hinge jointed to 4 lower legs. There are no joint limits.
This setup tests collision detection and handling, ball and hinge joints and lastly friction, it is also the first articulated, falling figure we test. Figure 25 shows screenshots of the simulation.

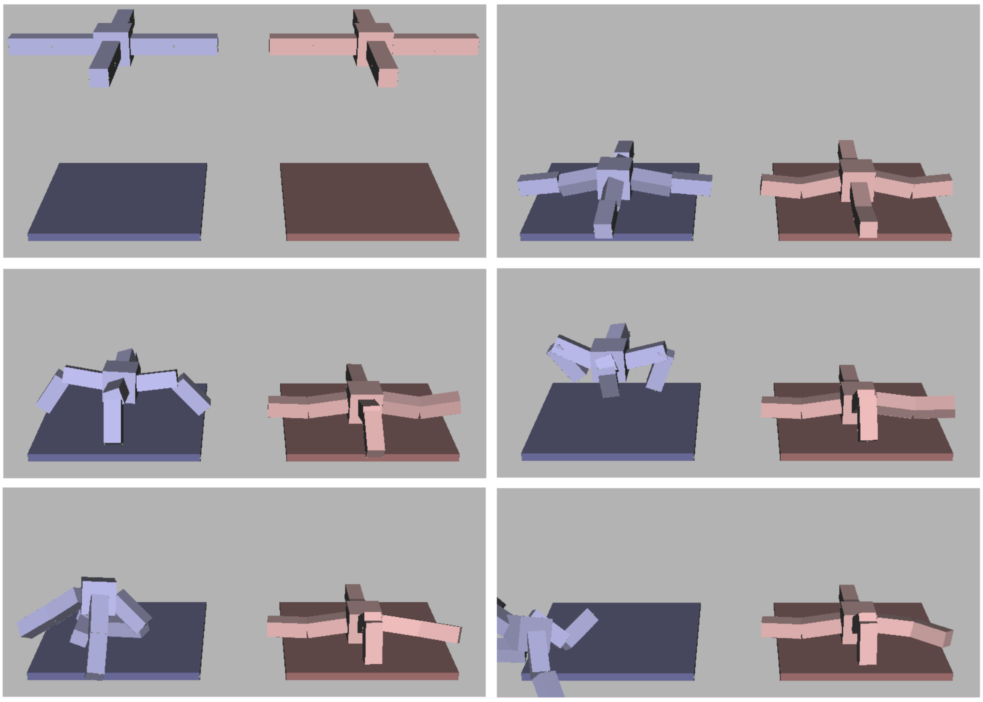


Figure 25: Screenshots of the test 7 simulation

In a perfect world, each of the figures would end up in a perfectly symmetric position, seeing as they start symmetrically, and are at no point affected asymmetrically. However due to the models used to implement these methods, events that in the perfect world would occur at the same instant, now occurs in an ordered row and hence affect each other. Apart from this, the multibody version looks fairly realistic, whereas the particle-based method seems to gain too much energy of the separate bones pushing and pulling each other around. Looking at the collisions and joints individually the result is satisfying, but as a whole the particle system provides with an unconvincing simulation. This will be further discussed in test 8.

5.1.9 Test 8. Human ragdoll

The last test scene consists of the human ragdoll, composed as described in appendix A. This setup was the goal of our paper, and utilizes all areas of our code, ranging from collisions, to box masses, friction and joint limits. Figure 26 shows screenshots of the simulation.

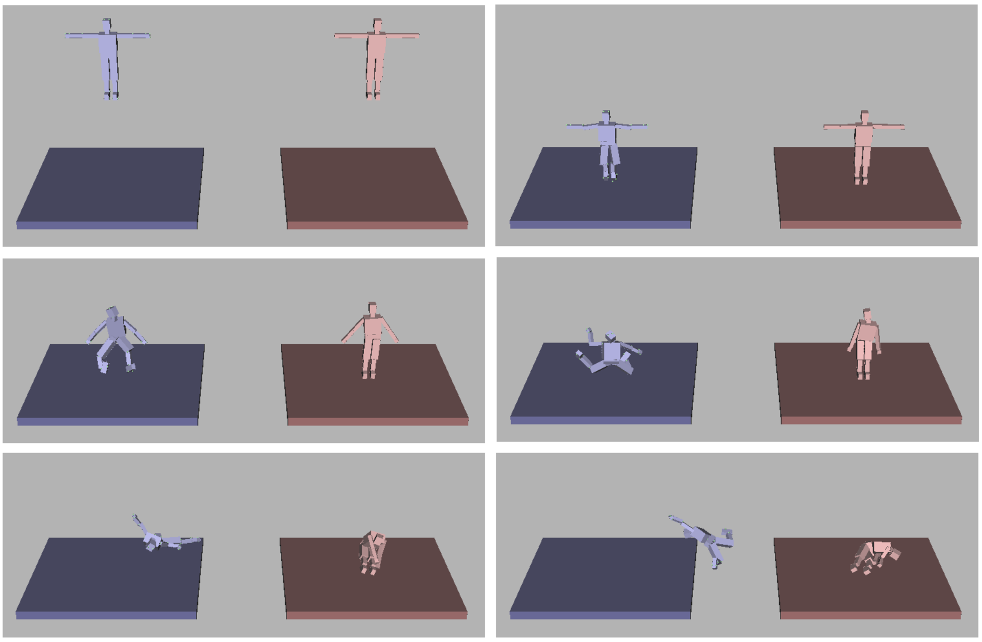


Figure 26: Screenshots of the human ragdoll simulation

Much like test 7, an asymmetric reaction was to be expected, and the constraint-based simulation gave a nice visual result. But once again the particle system model gains too much energy from its internal collisions and starts pushing and pulling. This is not a satisfaction result, and we tried to re-implement and tweak on almost anything in order to gain a better result, among our attempts has been:

- Giving all bones equal mass.
- Using a different scheme for calculating the particle weights used for rotation, as well as defining all weights to 0.25, thereby canceling out rotation

gained from collisions.

- Removing friction.
- Removing joint limits.
- Simulating only the upper body or the lower body.
- Redefining the unjointed particles so each body part was perfectly "balanced" mass wise.
- Redefining the internal coordinate system of the bones.
- Implementing a system, that sorted the list of bones at every iteration, such that colliding bones was first to have their constraints satisfied, and then rippling out relaxation to the adjoining body parts.
- Implementing a constraint loop, that kept on satisfying the constraints until the corrections was below a certain percentage of the rest length.

All of these attempts have been tried, and yielded very varying visual results, but all contained the extra energy and unpredictable behavior of our articulated figures and ragdolls. We are unsure of where the problem lies, but we have some thoughts which are mentioned in 5.4.

5.2 Performance results

Visual results and believability have been the key words in this project. The performance is however not an unimportant parameter. If any of the two models are to be used in a modern computer engine, the requirements should be as low as possible. Potentially, hundreds of events could all take place at the same timestep of a simulation, and the physics calculation are only given a small percentage of the total memory and processor use. According to [5] The lead programmer of IO-Interactive, Thomas Jakobsen states that about 5-10% of each frame is reserved for physics calculations.
The performance results are not intended to be a thorough analysis, but rather a hint of the actual usability. The performance has neither been an important issue when implementing the methods, so several improvements are likely to speed up the performance. We used the profiler program AQTime[3] to measure the time usage and OpenTissue functions to get the physical memory usage. The tests where executed on a Zepto 4200 laptop with an Intel Pentium 1.7 GHz mobile processor, 512 MB RAM and a ATI mobility Radeon 9700 graphic card.

5.2.1 Test 9. Memory usage

The test scene contains small bones, placed above each other with space in between them. The purpose is to measure how the memory usage increase as the number of bones increase. The results are the average of four test runs and are illustrated in figure 27 and table 1.

Boxes	Memory, particle-based	Memory, constraint-based
2	3.5 KB	2 KB
4	12 KB	4 KB
8	24 KB	4 KB
16	28 KB	16 KB
32	76 KB	32 KB
64	95 KB	60 KB
128	176 KB	128 KB
256	386 KB	264 KB
512	886 KB	512 KB
1024	1308 KB	1012 KB
2048	3000 KB	2032 KB
4096	6000 KB	4016 KB

Table 1: Memory usage as a function of the number of bones simulated.

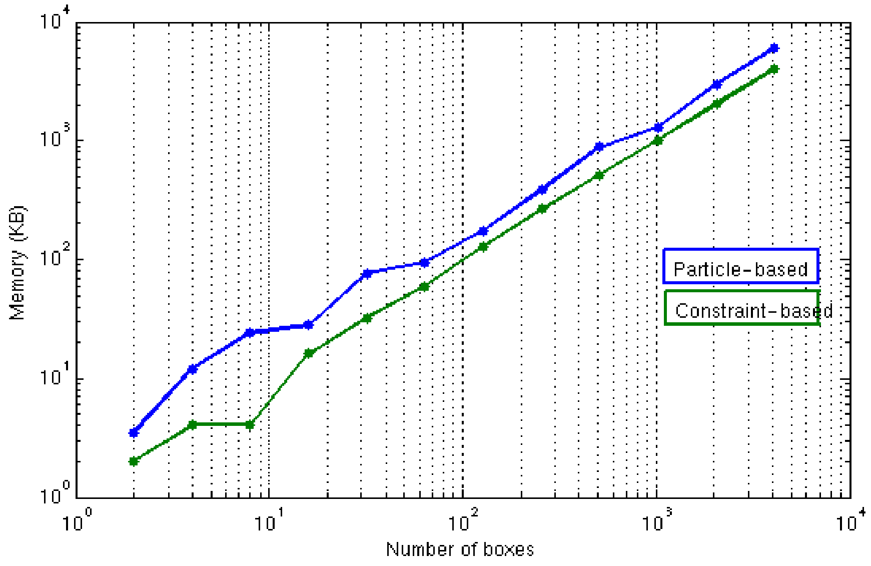


Figure 27: The memory usage as a function of the number of bones simulated. The memory increase linear as the number of boxes increase. Notice that both axes are logarithmic.

Not surprisingly, the graph shows a linear relation between the memory usage and the number of simulated bones. The jumping curve for the particle-based method is presumably caused by the reallocation in the container classes. The human ragdoll model is composed of 16 bones, which for both methods requires around 15-20 KB. The memory requirement are therefore not a problem if used with modern hardware, where at least 512 MB are available. If 5% are preserved physic calculations, the required memory for the ragdoll methods are still very low.

		Particle-based method		Constraint-based method	
Timestep	t_{real}	t_{sim}	$100\% \cdot t_{real}/t_{sim}$	t_{sim}	$100\% \cdot t_{real}/t_{sim}$
0.001	10	3.3	303%	2.6	385%
0.005	10	17.5	57%	12.9	78%
0.01	10	33.0	30%	26.5	38%
0.05	10	166.2	6%	132.3	8%
0.1	10	317.2	6%	267.7	4%

Table 2: The simulation time/realtime relation as a function of the timestep size. All time measurements are in seconds.

5.2.2 Test 10. Timestep

When the test programs are executed, the processor calculates at full speed, meaning that the simulated time not necessarily follows the realtime. To find out whether a test runs in realtime, we calculate the relation between simulated time t_{sim} and the actual time of simulation, t_{real}. If the result is 100% the test executed exactly in realtime, less than 100% means faster than realtime and over 100% is slower than realtime.

Several factors interfere with the simulation time, such as the size of the timestep, the number of bones and the number of relaxation and projection iterations. This first test will examine the relation between the timestep and the realtime usability. The scene consists of three bones, connected with one hinge joint and one ball joint. The results are seen in table 2 and the graph in figure 28.

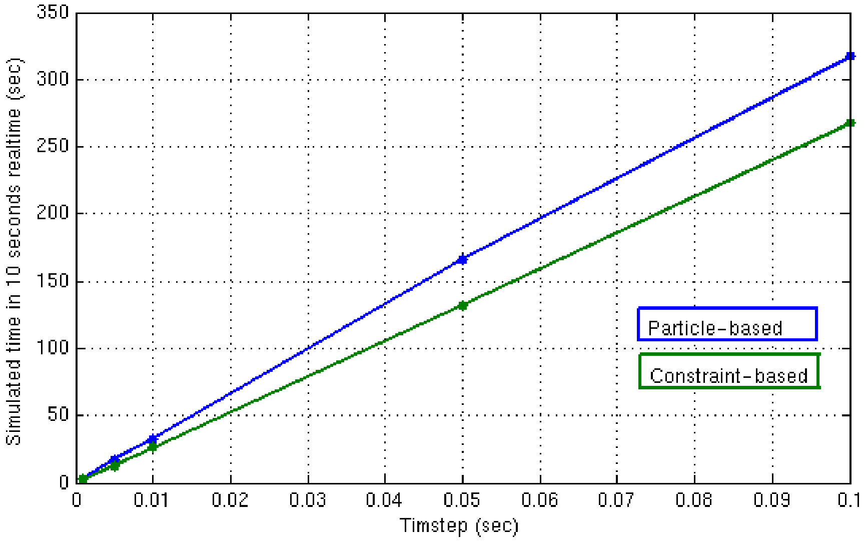


Figure 28: The simulation time as a function of the timestep size

Realtime simulation is achieved when the timestep is 0.005 seconds and higher. If the requirements to only use 5-10% of the system resources must be respected, a timestep around 0.05 should be used. The simulated time for both the particle-based method and the constraint-based multibody method increase linear as the timestep increase. The particle-based method has a better performance in this test scene. The next test will show which of the two methods that are fastest when the number of bones increase.

5.2.3 Test 11. Number of bones

Like the previous test, the relation between the simulated time t_{sim} and the realtime t_{real} for both methods are measured. This time the number of bones and joints are varied and the timestep is set to 0.01 seconds and iterations to 5. There will be two joints for every three bones. The results are shown in table 3 and the graph in figure 29.

		Particle-based method		**Constraint-based method**	
Bones	t_{real}	t_{sim}	$100\% \cdot t_{real}/t_{sim}$	t_{sim}	$100\% \cdot t_{real}/t_{sim}$
3	10	33.0	30%	25.2	40%
6	10	31.8	31%	25.1	40%
15	10	14.1	71%	25.0	40%
30	10	4.5	222%	25.0	40%
60	10	1.3	769%	23.1	43%
120	10	0.3	3333%	16.6	60%

Table 3: The simulation time/realtime relation as a function of the number of bones and joints. All time measurements are in seconds.

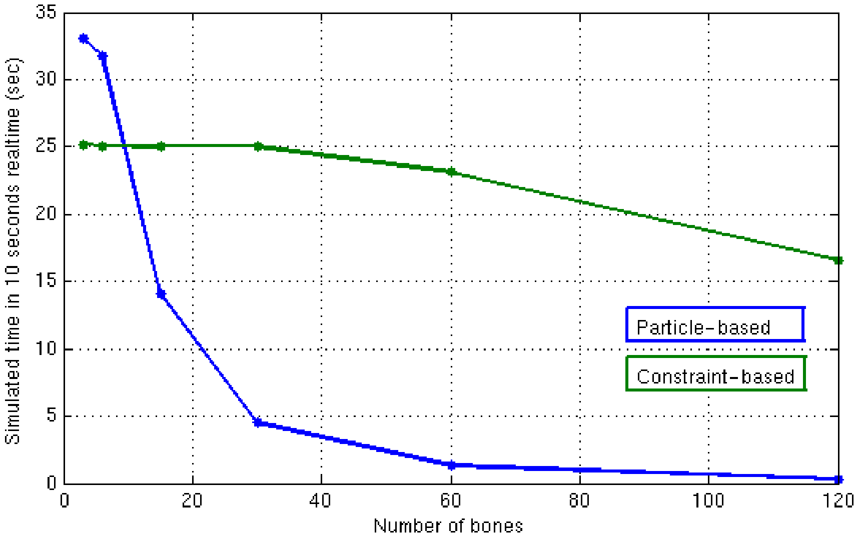


Figure 29: The simulation time as a function of the number of bones.

This test reveals an interesting progress. Even though the particle-based method is the fastest when the number of bones are low, t_{sim} falls drastically compared to the constraint-based method when the number is increased. To analyze the issue, we run the profiling program AQTime, during the test simulation. AQTime can tell us the time spent in each class and each function. This information can then be used to optimize the critical operations in the program. This has however not been an important issue in this project, mainly because there has been no time for such optimizations.

When AQTime runs in the background, the performance of the tests falls, so running a program for 10 seconds in realtime, does not result in the same simulation time as before. The purpose of using AQTime is however not to measure the relation between realtime and simulation time. To achieve a better comparison, the two methods are profiled when running at the same time, simulating 15 bones for 45 seconds.

The call graph shows that the `collision_box_box_improved` function used to detect collisions, is part of the critical path. It is also the second most time consuming function of all, using more than 15% of the total time. If we run the two methods separately, we only get `collision_box_box_improved` as the fourth most time consuming function for the constraint-based method, but still the second most for the particle-based method and it uses 3% and 17% of the total time respectively. Observing the test where both methods were executed together, we found that the `collision_box_box_improved` function is called a total of 126,871 times, in which only 2,021 come from the constraint-based method. This tells us that the collision detection method used for the particle-based method might not be very effective. The particle-based method uses `collision_box_box_improved` to detect collisions between every pair of boxes that is not connected, every time an iteration of projection is done. Doing collision detection is unavoidable, but it is possible to implement significant optimizations. The constraint-based method uses a broad and a narrow phase detection method, which makes sure that cheap detection algorithms are used when boxes are far from each other and more precise, such as `collision_box_box_improved` when they are close. This is the primary reason that the constraint-based method is faster. The number of collision detection calls increase exponentially when the number of boxes on the scene increase, lowering the simulated time, indicated by the graph in figure 29. We have not focused on optimal performance and therefore we have not implemented a better collision detection method.

The rest of the AQTime information, is not very surprising. The functions specific for both methods use a very small amount of the total time. The `satisfy` function for example, which is called six times for every bone, in every iteration loop, only uses 0.17% of the total time. The AQTime project files are saved in the AQTime folder on the enclosed CD, but they require AQTime to be opened.

5.2.4 Test 12. Iterations

Again the relation between the simulated time t_{sim} and the real time t_{real} are measured. This time for a varying number of projection and relaxation itera-

Iterations	t_{real}	t_{sim}	$100\% \cdot t_{real}/t_{sim}$
1	10	30.7	33%
2	10	30.4	33%
5	10	25.4	39%
10	10	17.5	57%
20	10	10.6	94%
40	10	5.9	169%

Table 4: The simulation time/real time relation as a function of the projection and relaxation iterations. All time measurements are in seconds.

tions per timestep for the particle-based method. We have not tested for the constraint-based method because changing the iterations for the time stepper did not have any impact on the simulation. The timestep is set to 0.01 seconds and the number of boxes are 9. The results are shown in table 4 and the graph in figure 30.

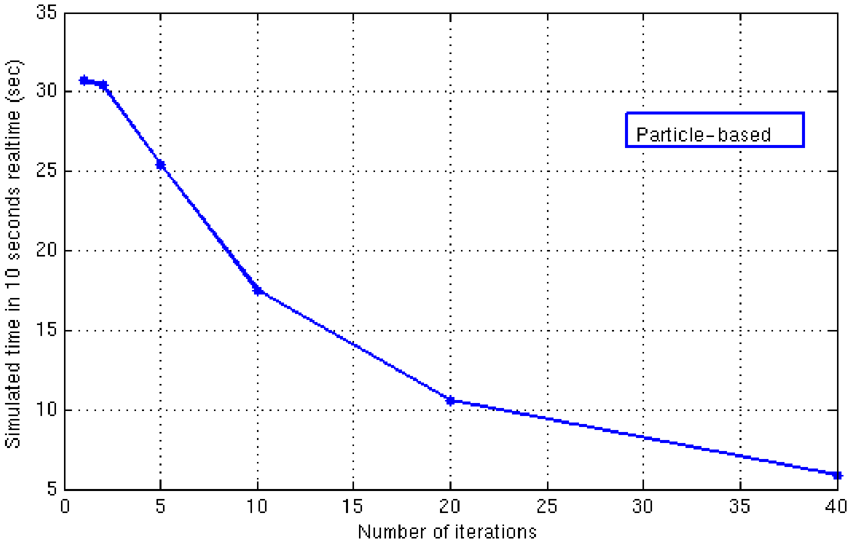


Figure 30: The simulation time as a function of the number of iterations.

The results are pretty much as expected. The simulation time falls as the number of iterations increase. As described in test 3, the visual result is not getting much better, even though the iterations are set to more than 5. Fortunately the simulation time for 5 iterations are almost as good as the time for 1 iteration.

5.3 Conclusion

Even though the visual tests of the particle system could not provide proper results for articulated bodies, we have been able to produce a satisfying visual

result with all the minor visual tests. We believe that the particle system in the current state is close to being a visually acceptable simulator, we even believe that the particle system approach provides better visual results for the purpose of modeling ragdolls, because of the smoothness caused by the less rigid movements.
This, together with the fact that the particle based implementation provided a competitive performance, especially for small simulations even in its current state without optimizations, leads us to believe that the particle based implementation, with some extra effort, would be more suited for running realtime game animations than the constraint-based methods.
There are some drawbacks however. In its current state, the particle system is considerably more tedious to model with, and basically requires a mathematician to set up even the simplest of scenes. Furthermore to be of any use, the poor scaling of scene sizes would need to be addressed, mainly referring to the collision detection scheme currently used.
Given the needed adjustments, we expect the particle based approach to be more efficient, with little to no loss of believability in a fast paced realtime game simulation. However, the constraint based multibody method did not require too much resources to be used in in realtime simulations. Both methods must be said to be usable.

5.4 Future work and nice to haves

If we had more time at our disposal to continue our work, several functionalities would have been nice to have in a game simulator using ragdolls. In this section we give a brief introduction to such future work.

Better visual results. The first step would obviously be to make the particle-based method more visually believable. As described in test 8, the particle-based method suffer from some sort of instability, making it twitch around. In test 8 we discussed some of the redesigns and tweakings we have already attempted, but given more time to pursue the problem, we would have turned our attention to subjects such as:

- Particulating (which we have limited ourselves from).
- TODO: DET DER FRA BOGEN! omkring gennemsnit af partikler og hvor meget de flytter sig, til brug af udregning af koordinatsystem... omkring side 475 eller noget? .. numeric stability gejl.
- More than four particles per bone, enabling a more dynamic bone structure and particle placement.

Collision detection The implementation we have made only uses a box-box collision detection algorithm. The idea to use a simple geometry for the ragdoll is good, a spherical cylinder might have been even better to use. If the ragdoll should react on different shaped object however, the collision detection procedure should be changed to take care of such other shapes. The efficiency could also be improved if a better narrow-broad phase collision detection were used as mentioned in test 11.

Robustness. More tests...?

Skinning. The box representation of the ragdoll bones is good to have when doing collision detection and simulation tests. It is though not very good for practice use without some character skin around it. A necessary feature to have, would be the possibility to skin the ragdolls with character models saved in a standard format. Animators or game artist often make their models in 3D programs such as Maya or 3D Studio Max. The ragdoll class should be able to read such a character file format and wrap it to the ragdoll.

Start position. When a ragdoll is used in a computer game, the ragdoll usually takes over from the animated character when the character dies. In a first person shooter game for example, the dying character would often be in a defending or shooting posture when he is shot. The ragdoll should therefore start in different positions depending on the character posture. A little trick could be to predefine a number of start positions and then use a character animation to move into one of the predefined postures before the ragdoll takes over.

Final position. Not only the start position is interesting, but the final position is as well. If we let the ragdoll slide into one of many predefined positions, all that is needed to save in the memory when the ragdoll is not displayed, is a pointer such a predefined position. If the ragdoll should be displayed again, the position is loaded and the ragdoll is again movable.

TODO: Some references to work done in the areas

Body part	**Height**	**Width**	**Depth**
Head	20.0	15.7	20.0
Neck	12.3	10.9	10.9
Chest	40.9	39.2	25.0
Hip	14.5	35.8	14.3
Upper arms	8.8	36.6	8.8
Forearms	8.5	28.8	8.5
Fists	8.9	8.9	8.9
Thighs	51.2	16.9	16.9
Calfs	30.5	10.6	10.6
Feet	13.9	9.9	27.3

Table 5: Measurements for all the body parts in centimeters.

A Human proportions

In this appendix we discuss the parts that we wish to "build" our human ragdolls from.

This is needed for several reasons. First and foremost, it is important that we model the exact same doll with both of the methods in question, as well as model them fairly anatomically correct to make sure we can make a valid qualitative judgement between them. Furthermore the findings we make here, might affect on how we wish to implement the solution to OpenTissue.

The modeling consists of three subdivisions, the modeling of body parts the jointing, and making constraints on the joints to make sure that the body will act much like an anatomically correct human body. We will discuss each of these three parts separately below.

A.1 Modeling the bones

There are several different approaches to model the human body. The most correct one would arguably be to model an exact skeleton with hundreds of bones and joints, later on adding every internal organ and lastly skin it. However, our goal is to create an effective simulator, and the above solution would be too demanding to use at runtime in a computer game setting.

Therefore it is needed to come up with a simplification, but one that does not sacrifice too much visual quality. We will do this by representing each major part of the body by one or more boxes, based on an evaluation on the rigidity of each body part (the less rigid, the more boxes will be used).

Table 5 shows the body parts we have chosen for our subdivision, as well as their measurements. Their mass will be derived directly from their size. The proportions of this has been taken from [8], but a few of them were undocumented or missing. To fill in the blanks we have made calculations on related measurements, such as circumference. Only the height of the neck has been an undocumented estimate.

Joint	Type
Head to neck	Hinge
Neck to chest	Ball
Chest to hip	Ball
Chest to upper arms	Ball
Upper arms to forearms	Hinge
Forearms to hands	Hinge
Hip to thighs	Ball
Thighs to calves	Hinge
Calves to feet	Hinge

Table 6: The joint types between all body parts.

Joint	Forward movement	Backward movement
Head to neck	45°	45°
Upper arms to forearms	135°	0°
Forearms to hands	90°	90°
Thighs to calves	0°	135°
Calves to feet	0°	90°

Table 7: The angular limits for all the hinge joints.

A.2 Body joints

The joints are the glue between our body parts. For all connected body parts, we need to have a placement and a type of joint. In the human body we model, only two types of joints occur, namely hinge and ball joints. Table 6 shows the joint types used for our human ragdoll.

All joints between adjacent body parts has been centered with respect to the minor body part.

For the particle system, we furthermore have to decide on the placement of all particles. As a rule of thumb they have all been placed inside their respective body part, as far away from each other as possible.

Figure 31 shows the initial set up of our ragdoll, and the particle positions.

A.3 Modeling the constraints

Having a body of correct proportions is only the first step of having creating a realistic ragdoll. Adding constraints to all joints is at least as important for it to act realistically. Many of these joint constraints can also be found at [8], but as a simplification we have decided to model the constraints by 45 degree increments based on our own bodies. The hinge joints are only limited in the two directions they can turn. Table 7 shows the angular limitations.

The ball joints are more complex. As explained in section 3.3 they will be defined using a vector that representing the center of their limit cones and an

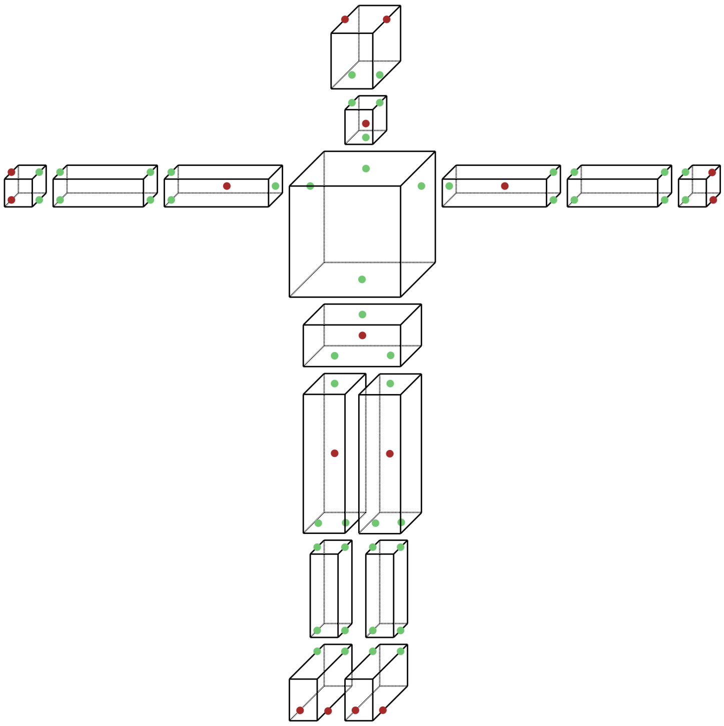


Figure 31: Diagram of the ragdoll body parts, including particle positions. The green particles are shared among adjacent bones and the reds are internal particles.

Joint	**Movement**
Neck to chest	45°
Chest to hip	45°
Chest to upper arms	90°
Hip to thighs	45°

Table 8: The angular limits for the ball joints.

angle that defines the cone width. Table 8 shows the maximal movement angles for the ball joints.

B OpenTissue implementation issues

B.1 Collision detection

During testing of `Collision_Box_Box_Improved` we encountered a bug in the existing code: The bug occurs if a small box A, collides with a larger box B, in a face-face collision, such that box A has 4 corners in box B, and box B has no corners in box A. The "separation plane" for the minimum overlap is chosen for box A, and A is also the reference box. Thus the penetration depth will always be calculated to 0. A quick fix for this, is to make a check whether A has 4 points in B and B has 0 points in A, and then force the "separation plane" to be defined in accordance to B. This is now corrected and reported to the OpenTissue forum.

B.2 Particles stored in dynamic array

The particles created for the particle system are stored in a vector array of the type `std::vector<particle_type>`. Because the vector class might reallocate the array after inserting particles, all particle pointers will be lost. The constraints already implemented in `psys` and the bone classes we have made, all use pointers to such particles. This restrict us to create all particles before being able to manipulate them. This will make it difficult to insert more ragdolls after creating the first.
A fix could be to use index numbers instead of pointers or to use a static array. This, however, would require a lot redefinition of existing code, which we have limited our selves from.

B.3 Wrong satisfy calculation

The stick constraint has three different satisfy functions. Two optimized and one non-optimized. During tests, we tried all three of them, and it then turned out that the non-optimized (`satisfy_type1()`) had an error. The particles where projected in the wrong signed direction. This is now corrected and reported to the OpenTissue forum.

C Program code

We have chosen to include program files that we have made from scratch. This is just a selection of all the code used to make the ragdoll simulations. The rest of the code can be found on the enclosed CD or checked out online from the OpenTissue webpage[9].

C.1 psys_ragdoll.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_H
#define OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_H
////////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
//
// This file is part of OpenTissue.
//
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
//
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
//
//    Att: Kenny Erleben and Jon Sporring
//    Department of Computing Science, University of Copenhagen
//    Universitetsparken 1
//    DK-2100 Copenhagen
//    Denmark
//
////////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/utility/GL/gl_util.h>
#include <OpenTissue/collision/collision_box_box_improved.h>
#include <OpenTissue/dynamics/psys/mass_spring_system/psys_mass_spring_system.h>
#include <OpenTissue/dynamics/psys/integrators/psys_verlet_integrator.h>


namespace OpenTissue
{
  namespace psys
  {
  /**
   * This class represents a ragdoll
   * The class containts the functionalities to create
   * bones composed of particles, join them and add
   * joint limits.
   * The class also has a 'hard-coded' collision resolving,
   * should be moved out later..
   */
template<typename types>
class PSYSRagdoll
  : public PSYSMassSpringSystem< types, PSYSVerletIntegrator >
{
public:
  typedef PSYSMassSpringSystem< types, PSYSVerletIntegrator > base_class;
  typedef typename types::real_type                       real_type;
  typedef typename types::vector3_type                    vector3_type;
  typedef typename types::matrix3x3_type                  matrix3x3_type;
  typedef typename types::ragdoll_bone_type               ragdoll_bone_type;
  typedef          std::vector<ragdoll_bone_type>         ragdoll_bone_container;
  typedef typename ragdoll_bone_container::iterator       ragdoll_bone_iterator;
  typedef typename types::stick_constraint_type           stick_constraint_type;
  typedef typename types::hinge_joint_type                hinge_joint_type;
  typedef typename types::ball_joint_type                 ball_joint_type;
  typedef typename types::gravity_type                    gravity_type;

protected:
  ragdoll_bone_container   m_ragdoll_bones;   ///< Container of bones
  real_type                m_friction;        ///< Coefficient of friction, 0<
       m_friction <1.

protected:
  ragdoll_bone_iterator ragdoll_bones_begin()
  {
    return ragdoll_bone_iterator(m_ragdoll_bones.begin());
  }

  ragdoll_bone_iterator ragdoll_bones_end()
  {
    return ragdoll_bone_iterator(m_ragdoll_bones.end());
  }

  void add_ragdoll_bone(ragdoll_bone_type & B)
  {
    B.connect(*this);
    m_ragdoll_bones.push_back(B);
  }

  void remove_ragdoll_bone(ragdoll_bone_type const * B)
  {
    m_ragdoll_bones.remove(B);
    B->disconnect();
  }

public:
  PSYSRagdoll()
```

```
      : m_friction(0.00025)
    {}

    ~PSYSRagdoll()
    {
      base_class::clear();
      m_ragdoll_bones.clear();
    }

    void clear()
    {
      base_class::clear();
      m_ragdoll_bones.clear();
    }

    /**
    * Draws all ragdoll bones.
    **/
    void draw(unsigned int mode)
    {
      glColorPicker(.6,.9,.6);
      particle_iterator p = particle_begin();
      particle_iterator end = particle_end();
      for(;p!=end;++p)
      {
         glDrawPoint(p->position());
      }

      ragdoll_bone_iterator b = ragdoll_bones_begin();
      ragdoll_bone_iterator b_end = ragdoll_bones_end();
      for(;b!=b_end;++b)
      {
        b->draw(mode);
      }
    }

    /**
    * Computes new posistions for every particle and bone.
    **/
    void run(real_type timestep)
    {
      base_class::run(timestep);

      // Do relaxation
      if(!m_relaxation)
        return;

      for(unsigned int i=0;i<m_iterations;++i)
      {
        // Update bones
        ragdoll_bone_iterator b = ragdoll_bones_begin();
        ragdoll_bone_iterator endb = ragdoll_bones_end();

        for(;b!=endb;++b)
          b->update_coordsys();

        // detect collsions and handle them
        collision_detection();

        // satisfy each constraint
        constraint_iterator c   = constraint_begin();
        constraint_iterator end = constraint_end();
        for(;c!=end;++c)
          c->satisfy();
      }
    }

  protected:

    void collision_detection()
    {
      unsigned int collisions = 0;
      vector3_type collision_points[16];
      vector3_type n;
      real_type distance[16];

      ragdoll_bone_iterator b1 = ragdoll_bones_begin();
      ragdoll_bone_iterator b2 = b1;
      ragdoll_bone_iterator endb = ragdoll_bones_end();

      // Check all bones against eachother once
      for(;b1!=endb;++b1)
      {
        b2=b1+1;
        for(;b2!=endb;++b2)
        {
          // If bones are connected, no collision detection is done
          if(b1->is_connected(&(*b2)))
            continue;

          //Check for collisions
          collisions = collision_box_box_improved(
                          b1->get_obb_in_WCS().center()
                        , b1->get_obb_in_WCS().orientation()
                        , b1->get_obb_in_WCS().ext()
                        , b2->get_obb_in_WCS().center()
                        , b2->get_obb_in_WCS().orientation()
                        , b2->get_obb_in_WCS().ext()
                        , 0.00 // envelope
                        , collision_points
                        , n
                        , distance
                        );

          // Run collision handling if any collisions are found
          if(collisions>0)
        {
          collision_handling((*b1),(*b2),collisions,collision_points,n,distance
               );
          b1->update_coordsys();
          b2->update_coordsys();
        }
      }
    }
  }


  /**
  * Handles a collision
  * Including particle scaling, bone mass scaling and friction
  *
  * @param A        Bone A
  * @param B        Bone B
  * @param collisions   Number of collisions
  * @param p        List of collision points
  * @param n        Collision normal
  * @param distance   List of collision depths
  **/
  void collision_handling(
    ragdoll_bone_type & A
    , ragdoll_bone_type & B
    , unsigned int collisions
    , vector3_type * p
    , vector3_type n
    , real_type * distance
    )
  {
    real_type c_a, c_b, c_c, c_d;
    real_type a_p_length, b_p_length, c_p_length, d_p_length, total_length;
    real_type total_mass, lambda;
    vector3_type total_trans, A_trans, B_trans;

    // Find the max distance, and the collision point(s) for that.
    int max_dist = 0; int max_cnt = 1;

    for(unsigned int i=1; i<collisions; i++)
    {
      if(distance[i] <= distance[max_dist])
      {
        max_cnt = (distance[i] == distance[max_dist]) ? max_cnt+1 : 1;
        max_dist=i;
      }
    }

    // If the max collision distance is too small, we ignore the collision.
    if(distance[max_dist]>-0.00001)
      return;

    // set the contact point
    vector3_type ref_point = p[max_dist];

    // If there is more than one max dist, a new average contact point is used
    // check if more max distances is found
    if(max_cnt>1)
    {
      vector3_type tmp = vector3_type(0,0,0);
      for(unsigned int i=0; i<collisions; i++)
      {
        if(distance[i] == distance[max_dist])
          tmp += p[i];
      }
      ref_point = tmp / max_cnt;
    }

    // Calculate the friction per iteration
    real_type friction = m_friction / m_iterations;

    // Calculate the translation ratio of the two boxes based on their mass
    total_trans = distance[max_dist]*n;
    if(A.is_fixed())
      B_trans=total_trans;
    else if(B.is_fixed())
      A_trans=total_trans;
    else {
      total_mass = A.mass() + B.mass();
      A_trans = (total_trans) * (B.mass() / total_mass);
      B_trans = (total_trans) * (A.mass() / total_mass);
    }

    // If A is not fixed, we calculate the effects for each particle, and move
         them
    if(!A.is_fixed())
    {
      a_p_length = length(ref_point - A.particle_A()->position());
      b_p_length = length(ref_point - A.particle_B()->position());
      c_p_length = length(ref_point - A.particle_C()->position());
      d_p_length = length(ref_point - A.particle_D()->position());
      total_length = a_p_length+b_p_length+c_p_length+d_p_length;

      // Calculate the individual effect per particle. c_a to c_d sums to 1
      c_a = (1 - (a_p_length / total_length)) / 3;
      c_b = (1 - (b_p_length / total_length)) / 3;
      c_c = (1 - (c_p_length / total_length)) / 3;
      c_d = (1 - (d_p_length / total_length)) / 3;

      lambda = 1/(c_a * c_a + c_b * c_b + c_c * c_c + c_d * c_d);

      // Push each particle in box 1
      A.particle_A()->position() += c_a * lambda * A_trans ;
      A.particle_B()->position() += c_b * lambda * A_trans ;
      A.particle_C()->position() += c_c * lambda * A_trans ;
      A.particle_D()->position() += c_d * lambda * A_trans ;
    // We implement friction here, since we'd like to use the particle
         weights
    A.particle_A()->old_position() += normalize(A.particle_A()->position() -
         A.particle_A()->old_position()) * friction * c_a;
    A.particle_B()->old_position() += normalize(A.particle_B()->position() -
         A.particle_B()->old_position()) * friction * c_b;
    A.particle_C()->old_position() += normalize(A.particle_C()->position() -
         A.particle_C()->old_position()) * friction * c_c;
    A.particle_D()->old_position() += normalize(A.particle_D()->position() -
         A.particle_D()->old_position()) * friction * c_d;
  }

  // If b2 is not fixed, we calculate the effects for each particle, and move
       them
  if(!B.is_fixed())
  {
    a_p_length = length(ref_point - B.particle_A()->position());
    b_p_length = length(ref_point - B.particle_B()->position());
    c_p_length = length(ref_point - B.particle_C()->position());
    d_p_length = length(ref_point - B.particle_D()->position());
    total_length = a_p_length+b_p_length+c_p_length+d_p_length;

    // Calculate the individual effect per particle. c_a to c_d sums to 1
    c_a = (1 - (a_p_length / total_length)) / 3;
    c_b = (1 - (b_p_length / total_length)) / 3;
    c_c = (1 - (c_p_length / total_length)) / 3;
    c_d = (1 - (d_p_length / total_length)) / 3;

    lambda = 1/(c_a * c_a + c_b * c_b + c_c * c_c + c_d * c_d);

    // Push each particle in box 2
    B.particle_A()->position() -= c_a * lambda * B_trans ;
    B.particle_B()->position() -= c_b * lambda * B_trans ;
    B.particle_C()->position() -= c_c * lambda * B_trans ;
    B.particle_D()->position() -= c_d * lambda * B_trans ;

    // We implement friction here, since we'd like to use the particle
         weights
    B.particle_A()->old_position() += normalize(B.particle_A()->position() -
         B.particle_A()->old_position()) * friction * c_a;
    B.particle_B()->old_position() += normalize(B.particle_B()->position() -
         B.particle_B()->old_position()) * friction * c_b;
    B.particle_C()->old_position() += normalize(B.particle_C()->position() -
         B.particle_C()->old_position()) * friction * c_c;
    B.particle_D()->old_position() += normalize(B.particle_D()->position() -
         B.particle_D()->old_position()) * friction * c_d;
  }
}


/**
* This function creates a ragdoll bone, using the 4 particles
*
* Returns a iterator pointing at the newly created bone.
**/
ragdoll_bone_iterator create_bone(
  particle_iterator & p_A
  , particle_iterator & p_B
  , particle_iterator & p_C
  , particle_iterator & p_D
  )
{
  p_A->old_position() = p_A->position();
  p_B->old_position() = p_B->position();
  p_C->old_position() = p_C->position();
  p_D->old_position() = p_D->position();

  ragdoll_bone_type * bone = new ragdoll_bone_type();
  bone->init(this, &(*p_A), &(*p_B), &(*p_C), &(*p_D));
  add_ragdoll_bone(*bone);

  ragdoll_bone_iterator bone_it = m_ragdoll_bones.end()-1;
  return bone_it;
}

/**
* Connects the two bones as a joint.
* Will make sure that they do not collide
* Use this if no joint limits are wanted.
**/
void link_bones_joint(ragdoll_bone_type & A, ragdoll_bone_type & B)
{
  A.connect(B);
  B.connect(A);
}

/**
* Connects the two bones as a ball joint.
* Will make sure that they do not collide
*
* @param A         Bone A
* @param B         Bone B
* @param p         The shared ball particle
* @param p_ref       The reference particle in B used to detect and correct
     breaches
* @param angle       The angle from the plane to the cone. Defines the width
     of the reach cone
* @param plane_normal Plane normal defining the direction of the cone
**/
void link_bones_ball_joint(
  ragdoll_bone_type & A
  , ragdoll_bone_type & B
  , particle_type * p
  , particle_type * p_ref
  , real_type angle
```

```
      , vector3_type plane_normal
      )
    {
      ball_joint_type * joint = new ball_joint_type();
      joint->init(&A, &B, p, p_ref, angle , plane_normal);
      add_constraint(joint);

      A.connect(B);
      B.connect(A);
    }

    /**
    * Connects the two bones as a hinge joint.
    * Will make sure that they do not collide
    *
    * @param A        Bone A
    * @param B        Bone B
    * @param p1       The first shared hinge particle
    * @param p2       The second shared hinge particle
    * @param pos_angle    The positive maximum angle from initial position
    * @param neg_angle    The negative maximum angle from initial position
    **/
    void link_bones_hinge_joint(
      ragdoll_bone_type & A
      , ragdoll_bone_type & B
      , particle_type * p1
      , particle_type * p2
      , real_type pos_angle
      , real_type neg_angle
      )
    {
      hinge_joint_type * joint = new hinge_joint_type();
      joint->init( &A, &B, p1, p2, pos_angle, neg_angle );
      add_constraint( joint );

      A.connect(B);
      B.connect(A);
    }

  };

  } // namespace psys
} // namespace OpenTissue

//OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_H
#endif
```

C.2 psys_ragdoll_bone.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_BONE_H
#define OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_BONE_H
////////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
//
// This file is part of OpenTissue.
//
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
//
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
//
//    Att: Kenny Erleben and Jon Sporring
//    Department of Computing Science, University of Copenhagen
//    Universitetsparken 1
//    DK-2100 Copenhagen
//    Denmark
//
////////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/geometry/obb.h>
#include <OpenTissue/math/coordsys.h>


namespace OpenTissue
{
  namespace psys
  {
  /**
   *
   *
   */
  template<typename types>
  class PSYSRagdollBone
  {
  public:
    typedef typename types::real_type                   real_type;
    typedef typename types::vector3_type              vector3_type;
    typedef typename types::matrix3x3_type            matrix3x3_type;
    typedef typename types::ragdoll_type              ragdoll_type;
    typedef typename types::coordsys_type             coordsys_type;
    typedef typename types::particle_type             particle_type;
    typedef typename types::stick_constraint_type     stick_constraint_type;
    typedef      OBB<real_type>                   obb_type;
    typedef typename types::ragdoll_bone_type         ragdoll_bone_type;
    typedef          std::vector<ragdoll_bone_type * >  ragdoll_bone_container;
    typedef typename ragdoll_bone_container::iterator   ragdoll_bone_iterator;

  protected:
    // particle A, B, C and D makes up the tetrahedron
    particle_type         * m_A;              ///< pointer to particle A in
          mass_spring_sys
    particle_type         * m_B;              ///< pointer to particle B in
          mass_spring_sys
    particle_type         * m_C;              ///< pointer to particle C in
          mass_spring_sys
    particle_type         * m_D;              ///< pointer to particle D in
          mass_spring_sys
    ragdoll_type          * m_owner;          ///< Pointer to the ragdoll that
          this bone is a part of
    stick_constraint_type   m_stick[6];       ///< The 6 constraints making up
          the tetrahedron
    vector3_type            m_coord_T;        ///< The position of the local
          coordinate sys in WCS
    matrix3x3_type          m_coord_R;        ///< The orientation of the local
           coordinate sys in WCS
    coordsys_type           m_coords_wsc_to_bf; ///< Coordinate system to go from
           WSC to BF
    obb_type                m_obb;            ///< The oreiented bounding box
    vector3_type            m_center;         ///< The center of the
          tetrahedron, will be the obb center as default
    vector3_type            m_color;          ///< The color of the bone
    ragdoll_bone_container  m_ragdoll_bones;  ///< Bones that are connected to
          this one
    real_type               m_mass;           ///< The mass of the bone (based
          on the OBB dimensions)
    bool                    m_fixed;          ///< Is the bone fixed?
    std::string             m_name;           ///< Give a bone a name, used for
           testing / printing purposes

  public:

    particle_type *     particle_A()    { return m_A;     }
    particle_type *     particle_B()    { return m_B;     }
    particle_type *     particle_C()    { return m_C;     }
    particle_type *   particle_D()  { return m_D;     }

    vector3_type    coord_T()   { return m_coord_T; }
    matrix3x3_type    coord_R()   { return m_coord_R; }
    coordsys_type   coordsys()    { return m_coords_wsc_to_bf; }

    // Connect and disconnect the ragdoll that contains the bone
    void connect(ragdoll_type & owner)  { m_owner = &owner; }
    void disconnect()         { m_owner = 0;    }

    obb_type  obb()     { return m_obb;   }
    obb_type  get_obb_in_WCS(){ obb_type tmp = m_obb; tmp.xform(m_coord_T,
         m_coord_R); return tmp;  }

    vector3_type  center()  { return m_center;  }

    void connect(ragdoll_bone_type & B)     { m_ragdoll_bones.push_back(&B);  }
    void disconect(ragdoll_bone_type const * B) { m_ragdoll_bones.remove(B);    }

    ragdoll_bone_iterator ragdoll_bones_begin()   { return ragdoll_bone_iterator(
         m_ragdoll_bones.begin());}
    ragdoll_bone_iterator   ragdoll_bones_end()   { return ragdoll_bone_iterator(
         m_ragdoll_bones.end());  }

    bool      is_fixed()  { return m_fixed; }
    real_type   mass()    { return m_mass;  }
    std::string   name()    { return m_name;  }

    void set_color(vector3_type color)  { this->m_color=color; }

  public:
    /**
    * Constructor
    **/
    PSYSRagdollBone()
      : m_A(0)
      , m_B(0)
      , m_C(0)
      , m_D(0)
      , m_owner(0)
      , m_color(vector3_type(0.5,0.5,0.5))
      , m_fixed(false)
      , m_mass(1)
    {}

    ~PSYSRagdollBone(){}

    /**
    * Init must be called before the bone is functional
    * Sets up the bone and obb.
    **/
    void init(
      ragdoll_type * m_owner
      , particle_type * A
      , particle_type * B
      , particle_type * C
      , particle_type * D
      )
    {
      this->m_owner=m_owner;
      this->m_A=A;
      this->m_B=B;
      this->m_C=C;
      this->m_D=D;

      //init stick constraints and add them to the particle system
      m_stick[0].init( A, B );
      m_stick[1].init( A, C );
      m_stick[2].init( A, D );
      m_stick[3].init( B, C );
      m_stick[4].init( B, D );
      m_stick[5].init( C, D );
      for(unsigned int i=0;i<6;i++)
        m_owner->add_constraint( &m_stick[i] );

      //Make the bone coordinate system
      update_coordsys();

      // Place the obb
      // The default is a box centered at the tetrahedron mass midtpoint
      vector3_type A_to_B = B->position() - A->position();
      vector3_type A_to_C = C->position() - A->position();
      vector3_type A_to_D = D->position() - A->position();

      set_obb_size(length(A_to_B),length(A_to_C),length(A_to_D));

      m_center = (A->position() + B->position()
            + C->position() + D->position()) / 4;
      m_coords_wsc_to_bf.xform_point(m_center);

      matrix3x3_type r = diag(1.0);
      r = ortonormalize(r);
      m_obb.place(m_center,r);
    }

    /**
    * Updates the tetrahedron coordinate system. Particle A's posistion is the
         origin.
    */
    void update_coordsys()
    {
      vector3_type X = normalize(vector3_type(m_B->position() - m_A->position())
           ;
      vector3_type Z = normalize(X % vector3_type(m_D->position() - m_A->position
           ()));
      vector3_type Y = (Z % X);
      m_coord_R.set_column(0,X);
      m_coord_R.set_column(1,Y);
      m_coord_R.set_column(2,Z);
      m_coord_T = m_A->position();
      m_coords_wsc_to_bf.set(m_coord_T,m_coord_R);
      // Inverse it to go from WCS to BF
      m_coords_wsc_to_bf = inverse(m_coords_wsc_to_bf);
    }

    void draw(unsigned int mode)
    {
      if(m_fixed)
        glColorPicker(.3,.3,.5);
      else
        glColorPicker(.6,.6,.8);

      obb_type tmp = m_obb;
      tmp.xform(m_coord_T, m_coord_R);
      tmp.draw( mode ); //GL_POLYGON or GL_LINE_LOOP

      glDrawPoint(tmp.center());
    }

    void set_obb_size(real_type const & width,real_type const & height,real_type
         const & depth)
    {
      m_obb.init(width, height, depth);
      m_mass = width * height * depth;
    }

    void set_obb_center_wcs(vector3_type const & center)
    {
      vector3_type c(center);
      m_coords_wsc_to_bf.xform_point(c);
      m_obb.center() = c;
    }

    void set_obb_orientation_wcs(matrix3x3_type const & ori)
    {
      matrix3x3_type r(ori);
      m_coords_wsc_to_bf.xform_matrix(r);
      m_obb.orientation() = r;
    }

    void set_obb_wcs(vector3_type const & center, matrix3x3_type const & ori)
    {
      set_obb_center_wcs(center);
      set_obb_orientation_wcs(ori);
    }

    void set_mass(real_type const & m)  { m_mass=m;  }
    void set_name(std::string const & n){ m_name=n;  }

    void set_fixed()
    {
      m_fixed = true;

      types::pin_constraint_type * c1 = new types::pin_constraint_type();
      types::pin_constraint_type * c2 = new types::pin_constraint_type();
      types::pin_constraint_type * c3 = new types::pin_constraint_type();
      types::pin_constraint_type * c4 = new types::pin_constraint_type();
      c1->init(&(*(m_A)));
      c2->init(&(*(m_B)));
      c3->init(&(*(m_C)));
      c4->init(&(*(m_D)));
      m_owner->add_constraint(c1);
      m_owner->add_constraint(c2);
      m_owner->add_constraint(c3);
      m_owner->add_constraint(c4);
    }

    void set_velocity(vector3_type const & v)
    {
      m_A->old_position() = m_A->position() - v*0.1;
      m_B->old_position() = m_B->position() - v*0.1;
      m_C->old_position() = m_C->position() - v*0.1;
      m_D->old_position() = m_D->position() - v*0.1;
    }

    void set_velocity_A(vector3_type const & v){m_A->old_position() = m_A->
         position()-v*0.1;  }
    void set_velocity_B(vector3_type const & v){m_B->old_position() = m_B->
         position()-v*0.1;  }
    void set_velocity_C(vector3_type const & v){m_C->old_position() = m_C->
         position()-v*0.1;  }
    void set_velocity_D(vector3_type const & v){m_D->old_position() = m_D->
         position()-v*0.1;  }

    bool is_connected(ragdoll_bone_type * r)
    {
      ragdoll_bone_iterator b = ragdoll_bones_begin();
      ragdoll_bone_iterator end = ragdoll_bones_end();
      for(;b!=end;++b)
      {
        if(*b == r)
          return true;
      }
      return false;
    }

    void scale(real_type S)
    {
      m_obb.center() *= S;
      m_obb.ext() *= S;
    }

  };

  } // namespace psys
} // namespace OpenTissue

//OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_BONE_H
#endif
```

C.3 psys_ragdoll_human.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_HUMAN_H
#define OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_HUMAN_H
//////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2003 Department of Computer Science, University of Copenhagen
//
//////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/math/boost_matrix_solvers.h>

namespace OpenTissue
{
  namespace psys
  {
  /**
   * This is a human ragdoll, build using particles
   *
   */
  template<typename types>
  class PSYSRagdollHuman : public PSYSRagdoll<types>
  {
  public:
    typedef PSYSRagdoll< types >                base_class;
    typedef typename types::real_type           real_type;
    typedef typename types::vector3_type        vector3_type;

  protected:

  public:

    void init()
    {
      std::cout << "PSYSRagdollHuman::init(): Initialize a human ragdoll"  << std
            ::endl;
      build_human(vector3_type(0,0,0), .1);
    }

    void build_human(vector3_type const & position, real_type const & size)
    {

```

```
// All particles must be added to the mass_spring system before they can be
       used to make bones
particle_type A,B,C,D;

//Ground
A.position() = vector3_type( 100,-100,  -400) * size + position;  //p 0
B.position() = vector3_type( 100, 100,  -400) * size + position;  //p 1
C.position() = vector3_type(-100,-100,  -401) * size + position;  //p 2
D.position() = vector3_type(-100, 100,  -400) * size + position;  //p 3
create_particle( A ); create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for head
A.position() = vector3_type( -5.45,   0,  -10.0) * size + position;  //p 4
B.position() = vector3_type(  5.45,   0,  -10.0) * size + position;  //p 5
C.position() = vector3_type( -5.45,   0,   10.0) * size + position;  //p 6
D.position() = vector3_type(  5.45,   0,   10.0) * size + position;  //p 7
create_particle( A ); create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for neck
//A.position() = vector3_type( -5.45,   0,  -10.0) * size + position;
//B.position() = vector3_type(  5.45,   0,  -10.0) * size + position;
C.position() = vector3_type(     0,   0,  -22.3) * size + position;  //p 8
D.position() = vector3_type(     0,   0,  -15.0) * size + position;  //p 9
/*create_particle( A ); create_particle( B );*/ create_particle( C );
      create_particle( D );

// Particles for chest
//A.position() = vector3_type(     0,   0,  -22.3) * size + position;
B.position() = vector3_type(  19.6,   0,  -26.7) * size + position;  //p 10
C.position() = vector3_type( -19.6,   0,  -26.7) * size + position;  //p 11
D.position() = vector3_type(     0,   0,  -61.5) * size + position;  //p 12
/*create_particle( A );*/ create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for hip
//A.position() = vector3_type(     0,   0,  -61.5) * size + position;
B.position() = vector3_type(     0,   0,  -70.4) * size + position;  //p 13
C.position() = vector3_type(  9.45,   0,  -77.7) * size + position;  //p 14
D.position() = vector3_type( -9.45,   0,  -77.7) * size + position;  //p 15
/*create_particle( A );*/ create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for left overarm
//A.position() = vector3_type(  19.6,   0,  -26.7) * size + position;
B.position() = vector3_type(  37.9,   0,  -26.7) * size + position;  //p 16
C.position() = vector3_type(  56.2,   0, -30.95) * size + position;  //p 17
D.position() = vector3_type(  56.2,   0, -22.45) * size + position;  //p 18
/*create_particle( A );*/ create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for right overarm
//A.position() = vector3_type( -19.6,   0,  -26.7) * size + position;
B.position() = vector3_type( -37.9,   0,  -26.7) * size + position;  //p 19
C.position() = vector3_type( -56.2,   0, -30.95) * size + position;  //p 20
D.position() = vector3_type( -56.2,   0, -22.45) * size + position;  //p 21
/*create_particle( A );*/ create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for left underarm
A.position() = vector3_type(  85   ,   0, -30.95) * size + position;  //p 22
B.position() = vector3_type(  85   ,   0, -22.45) * size + position;  //p 23
//C.position() = vector3_type(  56.2,   0, -30.95) * size + position;
//D.position() = vector3_type(  56.2,   0, -22.45) * size + position;
create_particle( A ); create_particle( B ); /*create_particle( C );
      create_particle( D );*/

// Particles for right underarm
A.position() = vector3_type( -85   ,   0, -30.95) * size + position;  //p 24
B.position() = vector3_type( -85   ,   0, -22.45) * size + position;  //p 25
//C.position() = vector3_type( -56.2,   0, -30.95) * size + position;
//D.position() = vector3_type( -56.2,   0, -22.45) * size + position;
create_particle( A ); create_particle( B ); /*create_particle( C );
      create_particle( D );*/

// Particles for left hand
//A.position() = vector3_type(  85   ,   0, -30.95) * size + position;
//B.position() = vector3_type(  85   ,   0, -22.45) * size + position;
C.position() = vector3_type(  93.9  ,   0, -30.95) * size + position;  //p
      26
D.position() = vector3_type(  93.9  ,   0, -22.45) * size + position;  //p
      27
/*create_particle( A ); create_particle( B );*/ create_particle( C );
      create_particle( D );

// Particles for right hand
//A.position() = vector3_type( -85   ,   0, -30.95) * size + position;
//B.position() = vector3_type( -85   ,   0, -22.45) * size + position;
C.position() = vector3_type( -93.9,   0, -30.95) * size + position;  //p 28
D.position() = vector3_type( -93.9,   0, -22.45) * size + position;  //p 29
/*create_particle( A ); create_particle( B );*/ create_particle( C );
      create_particle( D );

// Particles for left thigh
//A.position() = vector3_type(  9.45,  0,   -77.7) * size + position;
B.position() = vector3_type(  9.45,  0,  -100   ) * size + position;  //p 30
C.position() = vector3_type(  4.15,  0,  -128.9) * size + position;  //p 31
D.position() = vector3_type( 14.75,  0,  -128.9) * size + position;  //p 32
/*create_particle( A );*/ create_particle( B ); create_particle( C );
      create_particle( D );

// Particles for right thigh
//A.position() = vector3_type( -9.45,  0,   -77.7) * size + position;
B.position() = vector3_type( -9.45,  0,  -100   ) * size + position;  //p 33
C.position() = vector3_type( -4.15,  0,  -128.9) * size + position;  //p 34
```

```
    D.position() = vector3_type(-14.75,  0,  -128.9) * size + position;  //p 35
    /*create_particle( A );*/ create_particle( B ); create_particle( C );
         create_particle( D );

    // Particles for left calf
    A.position() = vector3_type(  4.15,  0,  -159.4) * size + position;  //p 36
    B.position() = vector3_type( 14.75,  0,  -159.4) * size + position;  //p 37
    //C.position() = vector3_type(  4.15,  0,  -128.9) * size + position;
    //D.position() = vector3_type( 14.75,  0,  -128.9) * size + position;
    create_particle( A ); create_particle( B ); /*create_particle( C );
         create_particle( D );*/

    // Particles for right calf
    A.position() = vector3_type( -4.15,  0,  -159.4) * size + position;  //p 38
    B.position() = vector3_type(-14.75,  0,  -159.4) * size + position;  //p 39
    //C.position() = vector3_type( -4.15,  0,  -128.9) * size + position;
    //D.position() = vector3_type(-14.75,  0,  -128.9) * size + position;
    create_particle( A ); create_particle( B ); /*create_particle( C );
         create_particle( D );*/

    // Particles for left foot
    //A.position() = vector3_type(  4.15,  0,  -159.4) * size + position;
    //B.position() = vector3_type( 14.75,  0,  -159.4) * size + position;
    C.position() = vector3_type(  4.15,-10,  -173.3) * size + position;  //p 40
    D.position() = vector3_type( 14.75,-10,  -173.3) * size + position;  //p 41
    /* create_particle( A );  create_particle( B );*/ create_particle( C );
         create_particle( D );

    // Particles for right foot
    //A.position() = vector3_type( -4.15,  0,  -159.4) * size + position;
    //B.position() = vector3_type(-14.75,  0,  -159.4) * size + position;
    C.position() = vector3_type( -4.15,-10,  -173.3) * size + position;  //p 42
    D.position() = vector3_type(-14.75,-10,  -173.3) * size + position;  //p 43
    /*create_particle( A ); create_particle( B );*/ create_particle( C );
         create_particle( D );

    particle_iterator p = particle_begin();

    create_bone(p,p+1,p+2,p+3);       //  0
    create_bone(p+4,p+5,p+6,p+7);     //  1
    create_bone(p+4,p+5,p+8,p+9);     //  2
    create_bone(p+8,p+10,p+11,p+12);  //  3
    create_bone(p+12,p+13,p+14,p+15); //  4
    create_bone(p+10,p+16,p+17,p+18); //  5
    create_bone(p+11,p+19,p+20,p+21); //  6
    create_bone(p+17,p+18,p+22,p+23); //  7
    create_bone(p+20,p+21,p+24,p+25); //  8
    create_bone(p+22,p+23,p+26,p+27); //  9
    create_bone(p+24,p+25,p+28,p+29); // 10
    create_bone(p+14,p+30,p+31,p+32); // 11
    create_bone(p+15,p+33,p+34,p+35); // 12
    create_bone(p+31,p+32,p+36,p+37); // 13
    create_bone(p+34,p+35,p+38,p+39); // 14
    create_bone(p+36,p+37,p+40,p+41); // 15
    create_bone(p+38,p+39,p+42,p+43); // 16

    matrix3x3_type R = diag(1.0);

    ragdoll_bone_iterator bs = ragdoll_bones_begin();

    //Ground
    (bs+0)->set_obb_wcs(vector3_type(0,0,-400)*size+position,R);
    (bs+0)->set_obb_size( 400*size, 400*size, 20*size );
    (bs+0)->set_fixed();
    (bs+0)->set_name("ground");

    //head
    (bs+1)->set_obb_wcs(vector3_type(0,0,0)*size+position,R);
    (bs+1)->set_obb_size( 15.7*size, 20*size, 20*size );
    (bs+1)->set_name("head");

    //neck
    (bs+2)->set_obb_wcs(vector3_type(0,0,-16.15)*size+position,R);
    (bs+2)->set_obb_size( 10.9*size, 10.9*size, 12.3*size );
    (bs+2)->set_name("neck");

    //chest
    (bs+3)->set_obb_wcs(vector3_type(0,0,-42.75)*size+position,R);
    (bs+3)->set_obb_size( 39.2*size, 25*size, 40.9*size );
    (bs+3)->set_name("chest");

    //hip
    (bs+4)->set_obb_wcs(vector3_type(0,0,-70.45)*size+position,R);
    (bs+4)->set_obb_size( 35.8*size, 14.3*size, 14.5*size );
    (bs+4)->set_name("hip");

    //left overarm
    (bs+5)->set_obb_wcs(vector3_type(37.9,0,-26.7)*size+position,R);
    (bs+5)->set_obb_size( 36.6*size, 8.8*size, 8.8*size );
    (bs+5)->set_name("left overarm");

    //right overarm
    (bs+6)->set_obb_wcs(vector3_type(-37.9,0,-26.7)*size+position,R);
    (bs+6)->set_obb_size( 36.6*size, 8.8*size, 8.8*size );
    (bs+6)->set_name("right overarm");

    //left underarm
    (bs+7)->set_obb_wcs(vector3_type(70.6,0,-26.7)*size+position,R);
    (bs+7)->set_obb_size( 28.8*size, 8.5*size, 8.5*size );
    (bs+7)->set_name("left underarm");

    //right underarm
    (bs+8)->set_obb_wcs(vector3_type(-70.6,0,-26.7)*size+position,R);
    (bs+8)->set_obb_size( 28.8*size, 8.5*size, 8.5*size );
    (bs+8)->set_name("right underarm");
```

```
        //left hand
        (bs+9)->set_obb_wcs(vector3_type(89.45,0,-26.7)*size+position,R);
        (bs+9)->set_obb_size( 8.9*size, 8.9*size, 8.9*size );
        (bs+9)->set_name("left hand");

        //right hand
        (bs+10)->set_obb_wcs(vector3_type(-89.45,0,-26.7)*size+position,R);
        (bs+10)->set_obb_size( 8.9*size, 8.9*size, 8.9*size );
        (bs+10)->set_name("right hand");

        //left thigh
        (bs+11)->set_obb_wcs(vector3_type(9.45,0,-103.3)*size+position,R);
        (bs+11)->set_obb_size( 16.9 *size, 16.9*size, 51.2*size );
        (bs+11)->set_name("left thigh");

        //right thigh
        (bs+12)->set_obb_wcs(vector3_type(-9.45,0,-103.3)*size+position,R);
        (bs+12)->set_obb_size( 16.9*size, 16.9*size, 51.2*size );
        (bs+12)->set_name("right thigh");

        //left calf
        (bs+13)->set_obb_wcs(vector3_type(9.45,0,-144.15)*size+position,R);
        (bs+13)->set_obb_size( 10.6*size, 10.6*size, 30.5*size );
        (bs+13)->set_name("left calf");

        //right calf
        (bs+14)->set_obb_wcs(vector3_type(-9.45,0,-144.15)*size+position,R);
        (bs+14)->set_obb_size( 10.6*size, 10.6*size, 30.5*size );
        (bs+14)->set_name("right calf");

        //left foot
        (bs+15)->set_obb_wcs(vector3_type(9.45,-5.2,-166.45)*size+position,R);
        (bs+15)->set_obb_size( 9.9*size, 27.3*size, 13.9*size );
        (bs+15)->set_name("left foot");

        //right foot
        (bs+16)->set_obb_wcs(vector3_type(-9.45,-5.2,-166.45)*size+position,R);
        (bs+16)->set_obb_size( 9.9*size, 27.3*size, 13.9*size );
        (bs+16)->set_name("right foot");

          //attach head to neck
        link_bones_hinge_joint(*(bs+1),*(bs+2),(bs+2)->particle_A(),(bs+2)->
             particle_B(), OT_M_PI/4, OT_M_PI/4);

          //attach neck to chest
        link_bones_ball_joint(*(bs+3),*(bs+2),(bs+3)->particle_A(), (bs+2)->
             particle_D(), OT_M_PI/4, vector3_type(0,0,1));

          //attach hip to chest
        link_bones_ball_joint(*(bs+3),*(bs+4),(bs+4)->particle_A(), (bs+4)->
             particle_B(), OT_M_PI/4, vector3_type(0,0,-1));

          //attach left overarm to chest
        link_bones_ball_joint(*(bs+3),*(bs+5),(bs+3)->particle_B(), (bs+5)->
             particle_D(), OT_M_PI/2, vector3_type(1,-1,0));

          //attach right overarm to chest
        link_bones_ball_joint(*(bs+6),*(bs+3),(bs+3)->particle_C(), (bs+6)->
             particle_D(), OT_M_PI/2, vector3_type(-1,-1,0));

          //attach left underarm to left overarm
        link_bones_hinge_joint(*(bs+7), *(bs+5),(bs+5)->particle_C(),(bs+5)->
             particle_D(), OT_M_PI*.75, 0);

          //attach right underarm right overarm
        link_bones_hinge_joint(*(bs+8), *(bs+6),(bs+6)->particle_C(),(bs+6)->
             particle_D(), OT_M_PI*.75, 0);

          //attach left hand to left underarm
        link_bones_hinge_joint(*(bs+9), *(bs+7),(bs+7)->particle_A(),(bs+7)->
             particle_B(), OT_M_PI/2, OT_M_PI/2);

          //attach right hand to right underarm
        link_bones_hinge_joint(*(bs+10), *(bs+8),(bs+8)->particle_A(),(bs+8)->
             particle_B(), OT_M_PI/2, OT_M_PI/2);

          //attach left thigh to hip
        link_bones_ball_joint(*(bs+4),*(bs+11),(bs+4)->particle_C(), (bs+11)->
             particle_B(), OT_M_PI/4, vector3_type(0,-1,-1));

          //attach right thigh to hip
        link_bones_ball_joint(*(bs+4),*(bs+12),(bs+4)->particle_D(), (bs+12)->
             particle_B(), OT_M_PI/4, vector3_type(0,-1,-1));

          //attach left calf to left thigh
        link_bones_hinge_joint(*(bs+13), *(bs+11),(bs+11)->particle_C(),(bs+11)->
             particle_D(), 0, OT_M_PI*.75);

          //attach right calf to right thigh
        link_bones_hinge_joint(*(bs+14), *(bs+12),(bs+12)->particle_C(),(bs+12)->
             particle_D(), 0, OT_M_PI*.75);

          //attach left foot to left calf
        link_bones_hinge_joint(*(bs+15), *(bs+13),(bs+15)->particle_A(),(bs+15)->
             particle_B(), 0, OT_M_PI/2);

          //attach right foot to right calf
        link_bones_hinge_joint(*(bs+16), *(bs+14),(bs+16)->particle_A(),(bs+16)->
             particle_B(), 0, OT_M_PI/2);

        }

    };

  } // namespace psys
} // namespace OpenTissue
```

```

//OPENTISSUE_DYNAMICS_PSYS_UTIL_PSYS_RAGDOLL_HUMAN_H
#endif
```

C.4 psys_joint.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_JOINT_H
#define OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_JOINT_H
//////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2003 Department of Computer Science, University of Copenhagen
//
//////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/dynamics/psys/psys_constraint.h>
#include <cassert>

namespace OpenTissue
{
  namespace psys
  {

    template<typename types>
  class PSYSJoint : public PSYSConstraint<types>
    {
    public:

      typedef typename types::real_type              real_type;
      typedef typename types::vector3_type           vector3_type;
    typedef typename types::matrix3x3_type         matrix3x3_type;

    typedef typename types::ragdoll_bone_type      bone_type;
    typedef typename types::coordsys_type          coordsys_type;
    typedef typename types::particle_type          particle_type;

    protected:
    bone_type          * m_A;                 ///< Pointer to bone A
    bone_type          * m_B;                 ///< Pointer to bone B
    coordsys_type        m_coords_wcs_to_bf;  ///< Coordinate system used to go
         from WCS to BF of bone A
    coordsys_type        m_coords_bf_to_wcs;  ///< Coordinate system used to go
         from BF to WCS

    public:
      bone_type       * A()       { return m_A; }
      bone_type       * B()       { return m_B; }
      bone_type const * A() const { return m_A; }
      bone_type const * B() const { return m_B; }

    public:

      PSYSJoint()
        : m_A(0)
        , m_B(0)
      { }

    virtual ~PSYSJoint() {}

    public:

    /**
    * Init Joint.
    *
    * The joint is initialized with:
    *   A     The first bone that should be connected
    *   B     The second bone that should be connected
    *
    */
    void init(bone_type * A, bone_type * B)
    {
      assert(A!=B || !"PSYSJoint::init(): Bone A and B were the same");
      assert(A    || !"PSYSJoint::init(): Bone A was null");
      assert(B    || !"PSYSJoint::init(): Bone B was null");

      this->m_A=A;
      this->m_B=B;

      // Use the BF coordsys from bone A
      m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
      // Inverse it to go from WCS to BF
      m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);
    }

    public:

    /**
    * Satisfy Constraint.
    * Should be implementet in under-classes, extending this class
    **/
    void satisfy()
    {
    }


    };

  } // namespace psys
} // namespace OpenTissue

```

```
// OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_HINGE_JOINT_H
#endif
```

C.5 psys_hinge_joint.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_HINGE_JOINT_H
#define OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_HINGE_JOINT_H
////////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2003 Department of Computer Science, University of Copenhagen
//
////////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/dynamics/psys/joints/psys_joint.h>

namespace OpenTissue
{
  namespace psys
  {

    /**
    * This class takes care of a hinge connection between two bones. Since the
         the pos and neg
    * angles are given to the initializer, the positions to project back to when
         the limits are
    * violated can be precomputed and used in satisfy().
    **/

    template<typename types>
    class PSYSHingeJoint
      : public PSYSJoint< types >
    {
    public:

    typedef PSYSJoint< types >                      base_class;

    typedef typename types::stick_constraint_type   stick_constraint_type;
    typedef plane<real_type>                        plane_type;

    protected:

      particle_type           * m_hinge_1;        ///< Pointer to one of the
           affected particles that is part of the hinge.
      particle_type           * m_hinge_2;        ///< Pointer to one other
           affected particle that is part of the hinge.
      particle_type           * m_pB_1;           ///< Pointer to particle 1 in
           bone B that is not part of the hinge
      particle_type           * m_pB_2;           ///< Pointer to particle 2 in
           bone B that is not part of the hinge
      particle_type           * m_pA_1;           ///< Pointer to particle 1 in
           bone A that is not part of the hinge
      stick_constraint_type     m_stick_pos;      ///< Stick constraint to satisfy
           when positive breach occur
      stick_constraint_type     m_stick_neg;      ///< Stick constraint to satisfy
           when negative breach occur
      vector3_type              m_rotation_axis;  ///< The hinge rotation axis,
           given from m_hinge_1 to m_hinge_2
      plane_type                m_plane_pos;      ///< Plane defining the positive
           border
      plane_type                m_plane_neg;      ///< Plane defining the negative
           border
      plane_type                m_plane_B;        ///< Plane defining the initial
           position

      vector3_type              m_pos_point_1;    ///< The point on m_plane_pos to
           projekt m_pB_1 back to
      vector3_type              m_pos_point_2;    ///< The point on m_plane_pos to
           projekt m_pB_2 back to
      vector3_type              m_neg_point_1;    ///< The point on m_plane_neg to
           projekt m_pB_1 back to
      vector3_type              m_neg_point_2;    ///< The point on m_plane_neg to
           projekt m_pB_2 back to
      unsigned int              m_choice;         ///< Member indicating which
           satisfy type that is used (1,2) default 2

    public:

      particle_type       * hinge_particle1()       { return m_hinge1; }
      particle_type       * hinge_particle2()       { return m_hinge2; }
      particle_type const * hinge_particle1() const { return m_hinge1; }
      particle_type const * hinge_particle2() const { return m_hinge2; }

    public:

      PSYSHingeJoint()
        : m_hinge_1(0)
        , m_hinge_2(0)
    , m_choice(2)
      { }

      virtual ~PSYSHingeJoint()  {  }

    public:

      /**
      * Init Constraint.
      *
    * The constraint is initialized with:
```

```
 *   A      The first bone that should be connected
 *   B      The second bone that should be connected
 *   p1       The first particle that is part of the hinge joint
 *   p2       The second particle that is part of the hinge joint
 *   pos_angle The angle the hinge can bend in the positive direction
 *   neg_angle The angle the hinge can bend in the negative direction
 *
 *   Positive direction is defined as the right hand rule along the rotation
       axis, p1 -> p2
   */
void init(
  bone_type * A
  , bone_type * B
  , particle_type * p1
  , particle_type * p2
  , real_type pos_angle
  , real_type neg_angle
  )
{

  base_class::init(A,B);

  this->m_hinge_1=p1;
  this->m_hinge_2=p2;

  // Find the particles that is not part of the hinge joint.
  // They will be used as messure particles.
  if(B->particle_A() != p1 && B->particle_A() != p2)
  {
    m_pB_1 = B->particle_A();
    m_pB_2 = (B->particle_B() != p1 && B->particle_B() != p2) ? B->particle_B
          () : B->particle_C();
    if(m_pB_2 != B->particle_B())
      m_pB_2 = (B->particle_C() != p1 && B->particle_C() != p2) ? B->
            particle_C() : B->particle_D();
  }
  else if(B->particle_B() != p1 && B->particle_B() != p2)
  {
    m_pB_1 = B->particle_B();
    m_pB_2 = (B->particle_C() != p1 && B->particle_C() != p2) ? B->particle_C
          () : B->particle_D();
  }
  else
  {
    m_pB_1 = B->particle_C();
    m_pB_2 = B->particle_D();
  }

  // Find bone A ref point
  if(A->particle_A() != p1 && A->particle_A() != p2)
    m_pA_1 = A->particle_A();
  else if(A->particle_B() != p1 && A->particle_B() != p2)
    m_pA_1 = A->particle_B();
```

```
else
  m_pA_1 = A->particle_C();

// Rotate around the rotation axis to find the angular limit points for
     both particles
m_rotation_axis = m_hinge_2->position() - m_hinge_1->position();

matrix3x3_type R1 = Ru(pos_angle, m_rotation_axis);
m_pos_point_1 = (R1 * (m_pB_1->position() - m_hinge_1->position())) +
      m_hinge_1->position();
m_pos_point_2 = (R1 * (m_pB_2->position() - m_hinge_1->position())) +
      m_hinge_1->position();

matrix3x3_type R2 = Ru(-1*neg_angle, m_rotation_axis);
m_neg_point_1 = (R2 * (m_pB_1->position() - m_hinge_1->position())) +
      m_hinge_1->position();
m_neg_point_2 = (R2 * (m_pB_2->position() - m_hinge_1->position())) +
      m_hinge_1->position();

//Initialize the stck constraints
m_stick_pos.init( m_pA_1, m_pB_1 );
m_stick_pos.set_rest_length( length(m_pos_point_1 - m_pA_1->position()) );

m_stick_neg.init( m_pA_1, m_pB_1 );
m_stick_neg.set_rest_length( length(m_neg_point_1 - m_pA_1->position()) );

// Find all 3 planes
vector3_type h1 = m_hinge_1->position();
vector3_type h2 = m_hinge_2->position();
vector3_type pB = m_pB_1->position();

// Use the BF coordsys from bone A
m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
// Inverse it to go from WCS to BF
m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);

m_coords_wcs_to_bf.xform_point(m_pos_point_1);  // Now we know m_max_point
      1 in BF coords
m_coords_wcs_to_bf.xform_point(m_pos_point_2);  // Now we know m_max_point
      2 in BF coords
m_coords_wcs_to_bf.xform_point(m_neg_point_1);  // Now we know m_min_point
      1 in BF coords
m_coords_wcs_to_bf.xform_point(m_neg_point_2);  // Now we know m_min_point
      2 in BF coords
m_coords_wcs_to_bf.xform_point(h1);
m_coords_wcs_to_bf.xform_point(h2);
m_coords_wcs_to_bf.xform_point(pB);

m_plane_pos.set(h1,h2,m_pos_point_1);     // Now the plane are in the BF of
       bone A
m_plane_neg.set(h1,h2,m_neg_point_1);     // Now the plane are in the BF of
       bone A
      m_plane_B.set(h1,h2,pB);             // This plane lies in the initial
           position og bone B

    }

    public:

      /**
      * Satisfy Constraint.
      **/
      void satisfy()
      {
    switch(m_choice)
        {
        case 1:       satisfy_type1();       break;
        case 2:       satisfy_type2();       break;
        };
      }

    /**
      * Satisfy Constraint.
    * It pushes just the one bone back in place when the constraint is
           unsatisfied
    * The points to push back to is know in advance, in BF
      **/
      void satisfy_type1()
      {

    m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
    m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);

    vector3_type pB = m_pB_1->position();
    m_coords_wcs_to_bf.xform_point(pB);

    if(m_plane_pos.get_signed_distance(pB)>0 && m_plane_B.get_signed_distance(pB)
         >0){
      // Handle constraint
      vector3_type new_pos_1 = m_pos_point_1;
      vector3_type new_pos_2 = m_pos_point_2;
      m_coords_bf_to_wcs.xform_point(new_pos_1);
      m_coords_bf_to_wcs.xform_point(new_pos_2);
      m_pB_1->position() = new_pos_1;
      m_pB_2->position() = new_pos_2;
    }
    else if(m_plane_neg.get_signed_distance(pB)<0  && m_plane_B.
         get_signed_distance(pB)<0){
      // Handle constraint
      vector3_type new_pos_1 = m_neg_point_1;
      vector3_type new_pos_2 = m_neg_point_2;
      m_coords_bf_to_wcs.xform_point(new_pos_1);
      m_coords_bf_to_wcs.xform_point(new_pos_2);
      m_pB_1->position() = new_pos_1;
      m_pB_2->position() = new_pos_2;
    }
      }

    /**
      * Satisfy Constraint.
    * This methood pushes both bones back in place when the constraint is
         unsatisfied
    * using a stick constraint.
    *
      **/
      void satisfy_type2()
      {
    m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
    m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);

    vector3_type pB = m_pB_1->position();
    m_coords_wcs_to_bf.xform_point(pB);

    if(m_plane_pos.get_signed_distance(pB)>0 && m_plane_B.get_signed_distance(pB)
         >0)
      m_stick_pos.satisfy();
    else if(m_plane_neg.get_signed_distance(pB)<0  && m_plane_B.
         get_signed_distance(pB)<0)
      m_stick_neg.satisfy();
    }

    };

  } // namespace psys
} // namespace OpenTissue

// OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_HINGE_JOINT_H
#endif
```

C.6 psys_ball_joint.h

```
#ifndef OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_BALL_JOINT_H
#define OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_BALL_JOINT_H
///////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2003 Department of Computer Science, University of Copenhagen
//
///////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/dynamics/psys/joints/psys_joint.h>

```

```
#include <cmath>

namespace OpenTissue
{
  namespace psys
  {

    /**
    * This class takes care of a ball connection between two bones.
    **/
    template<typename types>
    class PSYSBallJoint
      : public PSYSJoint< types >
    {
    public:

    typedef   PSYSJoint< types >                    base_class;

    typedef typename types::ragdoll_bone_type       bone_type;
    typedef typename types::stick_constraint_type   stick_constraint_type;
    typedef plane<real_type>                        plane_type;


    protected:

    particle_type          * m_ball_particle;   ///< Pointer to the particles
         that is part of the ball joint.
    particle_type          * m_pA;              ///< Pointer to mass_mid_pontin
         bone A
    particle_type          * m_pB;              ///< Pointer to mass_mid_pontin
         bone B
    particle_type          * m_pA_1;            ///< Pointer to a particle in
         bone A that's not the ball particle;
    particle_type          * m_pB_ref;          ///< Pointer to a reference
         particle in bone B that's not the ball particle;
    real_type                m_angle_limit;     ///< The min allowable angle
         relative to the plane m_plane
    plane_type               m_plane;           ///< The plane used to specify
         the direction of the limit cone
    stick_constraint_type    m_stick_ref;       ///< Stick contraint used to
         project particles back from breaches
    real_type                m_min_length_ref;  ///< The length to measure
         breaches with

    public:

      PSYSBallJoint()
      {}

      virtual ~PSYSBallJoint() {}

    public:

    /**
    * Init Constraint.
    *
    * The constraint is initialized with:
    * @param A        Bone A
    * @param B        Bone B
    * @param p        The shared ball particle
    * @param p_ref      The reference particle used to detect and correct
        breaches
    * @param angle      The angle from the plane to the cone. Defines the width
        of the reach cone
    * @param plane_normal Plane normal defining the direction of the cone
    **/
    void init(
      bone_type * A
      , bone_type * B
      , particle_type * p
      , particle_type * ref_p
      , real_type const & angle
      , vector3_type const & plane_normal
      )
    {
      base_class::init(A,B);

      using::fabs;
      this->m_ball_particle=p;
      this->m_angle_limit=fabs(angle);
      this->m_pB_ref = ref_p;


      // Find a particle in A that is not the ball joint
      m_pA_1 = (A->particle_A() == p) ? A->particle_B() : A->particle_A();
      m_stick_ref.init(m_pA_1, m_pB_ref);

      // Use the BF coordsys from bone A
      m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
      // Inverse it to go from WCS to BF
      m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);

      //Create the cone plane
      vector3_type plane_point_BF = m_ball_particle->position();
      m_coords_wcs_to_bf.xform_point(plane_point_BF);
      vector3_type plane_normal_BF(plane_normal);
      m_coords_wcs_to_bf.xform_vector(plane_normal_BF);

      m_plane.set(plane_normal_BF,plane_point_BF); // The plane is now known in
           BF of bone A

      // Compute the length to detect breaches with
      m_min_length_ref = sin(OT_M_PI/2 - angle) * length(m_pB_ref->position() -
           m_ball_particle->position());
    }
    protected:

      public:

        /**
      * Uses stick constraints to project back particles
      * Effects both bone A and B
      **/
      void satisfy()
      {
        m_coords_bf_to_wcs.set(m_A->coord_T(), m_A->coord_R());
        m_coords_wcs_to_bf = inverse(m_coords_bf_to_wcs);

        vector3_type p = m_pB_ref->position();

        m_coords_wcs_to_bf.xform_point(p);

        real_type dist = m_plane.get_signed_distance(p);
        if( dist < m_min_length_ref )
        {
          m_stick_ref.set_rest_length(length(m_pB_ref->position() - m_pA_1->
               position()) + (m_min_length_ref - dist));
          m_stick_ref.satisfy();
        }
      }
      };

    } // namespace psys
  } // namespace OpenTissue

  // OPENTISSUE_DYNAMICS_PSYS_JOINTS_PSYS_BALL_JOINT_H
  #endif
```

C.7 retro_ragdoll.h

```
#ifndef OPENTISSUE_DYNAMICS_RETRO_UTIL_RETRO_RAGDOLL_H
#define OPENTISSUE_DYNAMICS_RETRO_UTIL_RETRO_RAGDOLL_H
////////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
//
// This file is part of OpenTissue.
//
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
//
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
//
//    Att: Kenny Erleben and Jon Sporring
//    Department of Computing Science, University of Copenhagen
//    Universitetsparken 1
//    DK-2100 Copenhagen
//    Denmark
//
////////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

#include <OpenTissue/utility/GL/gl_util.h>

/**
* Draw body functionality
*/
template<typename Types>
struct DrawBody
{
  typedef typename Types::Body                       body_type;
  typedef typename Types::value_type                 real_type;
  typedef typename Types::vector3                    vector3_type;
  typedef typename Types::matrix3x3                  matrix3x3_type;
  typedef typename Types::quaternion                 quaternion_type;
  typedef typename Types::sdf_geometry_type          sdf_geometry_type;
  typedef typename Types::SphereType                 sphere_type;
  typedef typename Types::PlaneType                  plane_type;
  typedef typename Types::BoxType                    box_type;
  typedef typename Types::NodeTraits                 node_traits;

  void operator()(body_type const & body)
  {
    vector3_type color;
  body.getColor(color);

    glColorPicker(color(0), color(1), color(2));

    glPushMatrix();
    vector3_type r;
    quaternion_type Q;
    body.getPosition(r);
    body.getOrientation(Q);
    glTransform(r,Q);
    switch(body.m_gh_geometryType)
    {
    case node_traits::box:
      if(false)
        static_cast<box_type*>(body.m_gh_geometry)->draw(GL_LINE_LOOP);
      else
        static_cast<box_type*>(body.m_gh_geometry)->draw(GL_POLYGON);
      break;
    case node_traits::sphere:
      if(false)
        static_cast<sphere_type*>(body.m_gh_geometry)->draw(GL_LINE_LOOP);
      else
        static_cast<sphere_type*>(body.m_gh_geometry)->draw(GL_POLYGON);
      break;
    case node_traits::plane:
    case node_traits::no_geometry:
    case node_traits::distance_map:
      assert(!"not handled");
      break;
    };//End switch body->m_geometryType
    glPopMatrix();

    vector3_type V,W;

    //body.getSpin(W);
    //glDrawVector(r,W);
    //body.getVelocity(W);
    //glDrawVector(r,V);


  };
};

/**
* Draw joint functionality
*/
template<typename Types>
struct DrawJoint
{
  typedef typename Types::joint_type                   joint_type;
  typedef typename Types::socket_type                  socket_type;
  typedef typename Types::Body                         body_type;
  typedef typename Types::value_type                   real_type;
  typedef typename Types::vector3                      vector3_type;
  typedef typename Types::matrix3x3                    matrix3x3_type;
  typedef typename Types::quaternion                   quaternion_type;

  void operator()(joint_type const & joint)
  {
    vector3_type r;
    quaternion_type Q;

    glPushMatrix();
    joint.get_socket_A()->get_body()->getPosition(r);
    joint.get_socket_A()->get_body()->getOrientation(Q);
    glTransform(r,Q);
    glDrawFrame(joint.get_socket_A()->get_joint_frame());
    glPopMatrix();

    glPushMatrix();
    joint.get_socket_B()->get_body()->getPosition(r);
    joint.get_socket_B()->get_body()->getOrientation(Q);
    glTransform(r,Q);
    glDrawFrame(joint.get_socket_B()->get_joint_frame());
    glPopMatrix();
  };
};

namespace OpenTissue
{

  /**
   * This class represents a ragdoll
   * Uses constrain-based multibody dynamics.
   */
  template<typename MyTypes>
  class RetroRagdoll
  {
  public:
    typedef typename MyTypes::Configuration         configuration;
    typedef typename MyTypes::Simulator             simulator;
    typedef typename MyTypes::Body                  ragdoll_bone_type;
    typedef typename MyTypes::Gravity               gravity_type;
    typedef typename MyTypes::hinge_type            hinge_type;
    typedef typename MyTypes::ball_type             ball_type;
    typedef typename MyTypes::socket_type           socket_type;
    typedef typename MyTypes::BoxType               box_type;
    typedef typename MyTypes::real_type             real_type;
    typedef typename MyTypes::angular_limit_type    angular_limit_type;
    typedef typename MyTypes::reach_cone_type       reach_cone_type;
    typedef typename MyTypes::vector3               vector3_type;
    typedef typename MyTypes::quaternion            quaternion_type;
    typedef typename MyTypes::matrix3x3             matrix3x3_type;
    typedef typename MyTypes::coordsys              coordsys_type;
    typedef typename MyTypes::Material              material_type;
    typedef typename MyTypes::MaterialLibrary       material_library;
    typedef typename MyTypes::angular_limit_type    angular_limit_type;
    typedef typename MyTypes::reach_cone_type       reach_cone_type;

  protected:
    configuration               m_configuration;   ///< Contain setup config
    gravity_type                m_gravity;         ///< Gravity
    ragdoll_bone_type           m_bones[50];       ///< Bone container
    simulator                   m_simulator;       ///< Simulator
    real_type                   m_timestep;        ///< Timestep size
    box_type                    m_boxes[20];        ///< Box dimensions types
    material_library            m_library;          ///< Library


  public:
    RetroRagdoll()
    {}

  public:

    void clear()
    {
      m_configuration.clear();
    }


    void ragdoll_hinge_joint(
      ragdoll_bone_type *A
      , ragdoll_bone_type *B
      , vector3_type const & WCS_point
      , quaternion_type const & hinge_axis
      )
    {

        vector3_type position_A, position_B;
        socket_type * m_socket_A = new socket_type();
        socket_type * m_socket_B = new socket_type();
        hinge_type * m_hinge = new hinge_type;

        A->getPosition(position_A);
        B->getPosition(position_B);

        //Socket_A's quaternion decides which axis the hinge evolves around.
        m_socket_A->init(*A,coordsys_type(WCS_point-position_A,hinge_axis));
        m_socket_B->init(*B,coordsys_type(WCS_point-position_B,hinge_axis));
        m_hinge->connect(*m_socket_A,*m_socket_B);

        m_hinge->set_FPS(1.0/m_timestep);
        m_hinge->set_ERP(0.8);

        m_configuration.add(m_hinge);
    }

    void ragdoll_hinge_joint(
        ragdoll_bone_type *A
      , ragdoll_bone_type *B
      , vector3_type const & WCS_point
      , quaternion_type const & hinge_axis
      , real_type const & min
      , real_type const & max
      )
    {

        vector3_type position_A, position_B;
        socket_type * m_socket_A = new socket_type();
        socket_type * m_socket_B = new socket_type();
        hinge_type * m_hinge = new hinge_type();
        angular_limit_type * m_angular_limit = new angular_limit_type();

        A->getPosition(position_A);
        B->getPosition(position_B);

        //Socket_A's quarternion decides which axis the hinge evolves around.
        m_socket_A->init(*A,coordsys_type(WCS_point-position_A,hinge_axis));
        m_socket_B->init(*B,coordsys_type(WCS_point-position_B,hinge_axis));
        m_hinge->connect(*m_socket_A,*m_socket_B);


        m_angular_limit->set_min_limit(min);
        m_angular_limit->set_max_limit(max);
        //m_angular_limit->set_FPS(1.0/m_timestep);
        //m_angular_limit->set_ERP(0.8);
        m_hinge->set_limit(*m_angular_limit);

        m_hinge->set_FPS(1.0/m_timestep);
        m_hinge->set_ERP(0.8);

        m_configuration.add(m_hinge);
    }

    void ragdoll_ball_joint(ragdoll_bone_type *A, ragdoll_bone_type *B,
         vector3_type const & WCS_point)
    {
        vector3_type position_A, position_B;
        socket_type * m_socket_A = new socket_type();
        socket_type * m_socket_B = new socket_type();
        ball_type * m_ball = new ball_type();

        A->getPosition(position_A);
        B->getPosition(position_B);

        m_socket_A->init(*A,coordsys_type(WCS_point-position_A,quaternion_type())
             );
        m_socket_B->init(*B,coordsys_type(WCS_point-position_B,quaternion_type())
             );
        m_ball->connect(*m_socket_A,*m_socket_B);

        m_ball->set_FPS(1.0/m_timestep);
        m_ball->set_ERP(0.8);

        m_configuration.add(m_ball);
    }

    void ragdoll_ball_joint(
        ragdoll_bone_type *A
      , ragdoll_bone_type *B
      , vectorr3_type const & WCS_point
```

```
      , vector3_type const & WCS_point
      , reach_cone_type * limits)
    {
        vector3_type position_A, position_B;
        socket_type * m_socket_A = new socket_type();
        socket_type * m_socket_B = new socket_type();
        ball_type * m_ball = new ball_type();

        A->getPosition(position_A);
        B->getPosition(position_B);

        m_socket_A->init(*A,coordsys_type(WCS_point-position_A,quaternion_type())
             );
        m_socket_B->init(*B,coordsys_type(WCS_point-position_B,quaternion_type())
             );
        m_ball->connect(*m_socket_A,*m_socket_B);

        m_ball->set_FPS(1.0/m_timestep);
        m_ball->set_ERP(0.8);

        m_ball->set_reach_cone(*limits);

        m_configuration.add(m_ball);
    }


    /**
    * Draws all ragdoll bones
    **/
    void draw(unsigned int mode)
    {
      for_each(m_configuration.body_begin(),m_configuration.body_end(),DrawBody<
           MyTypes>());
    }

    void run(real_type m_timestep)
    {
      m_simulator.run(m_timestep);
    }

  };

} // namespace OpenTissue

//OPENTISSUE_DYNAMICS_RETRO_UTIL_RETRO_RAGDOLL_H
#endif
```

C.8 retro_ragdoll_human.h

```
#ifndef OPENTISSUE_DYNAMICS_RETRO_UTIL_RETRO_RAGDOLL_HUMAN_H
#define OPENTISSUE_DYNAMICS_RETRO_UTIL_RETRO_RAGDOLL_HUMAN_H
////////////////////////////////////////////////////////////////////////////////
//
// OpenTissue, A toolbox for physical based simulation and animation.
// Copyright (C) 2005 Department of Computer Science, University of Copenhagen
//
// This file is part of OpenTissue.
//
// OpenTissue is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation; either version 2.1 of the License, or
// any later version.
//
// OpenTissue is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with OpenTissue; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// Please send remarks, questions and bug reports to OpenTissue@diku.dk,
// or write to:
//
//    Att: Kenny Erleben and Jon Sporring
//    Department of Computing Science, University of Copenhagen
//    Universitetsparken 1
//    DK-2100 Copenhagen
//    Denmark
//
////////////////////////////////////////////////////////////////////////////////
#if (_MSC_VER >= 1200)
# pragma once
# pragma warning(default: 56 61 62 191 263 264 265 287 289 296 347 529 686)
#endif

/**
* This class uses RetroRagdoll to build a human ragdoll
*/
namespace OpenTissue
{
  template<typename MyTypes>
  class RetroRagdollHuman : public RetroRagdoll<MyTypes>
  {

  protected:
    real_type                m_size;      ///< The size of the ragdoll
    vector3_type             m_position;  ///< The position of the ragdoll

  public:

    //Constructor
    RetroRagdollHuman() :
      m_bullet_cnt(0),
      m_size(0),
      m_position(0,0,0)
    {}

    void init(vector3_type const & position, real_type const & size)
    {
        this->m_size = size;
        this->m_position = position;

        m_configuration.clear();

        quaternion_type Q, x, y, z;
        matrix3x3_type R;
        R = diag(1.0);

        x = quaternion_type(1,1,1,1);
        y = quaternion_type(0,0,1,1);
        z = quaternion_type(1,0,0,0);

        m_boxes[0].set(vector3_type(0,0,0),R,vector3_type(1000,1000,10)/2*size);
             //Ground

        //ground
        m_bones[0].setPosition(vector3_type(0,0,-250)*size+position);
        m_bones[0].setOrientation(quaternion_type(1,0,0,0));
        m_bones[0].setVelocity(vector3_type(0,0,0));
        m_bones[0].setSpin(vector3_type(0,0,0));
        m_bones[0].setGeometry(m_boxes[0]);
        m_bones[0].setFixed(true);
        m_bones[0].setColor(vector3_type(0.3,0.3,0.3));
        m_configuration.add(&m_bones[0]);

        m_boxes[1].set(vector3_type(0,0,0),R,vector3_type(15.7,20,20)/2*size); //
              Head
        m_boxes[2].set(vector3_type(0,0,0),R,vector3_type(10.9,10.9,12.3)/2*size)
             ; //Neck
        m_boxes[3].set(vector3_type(0,0,0),R,vector3_type(39.2,25,40.9)/2*size);
             //Chest
        m_boxes[4].set(vector3_type(0,0,0),R,vector3_type(35.8,14.3,14.5)/2*size)
             ; //Hip
        m_boxes[5].set(vector3_type(0,0,0),R,vector3_type(36.6,8.8,8.8)/2*size);
             //Overarm
        m_boxes[6].set(vector3_type(0,0,0),R,vector3_type(28.8,8.5,8.5)/2*size);
             //Underarm
        m_boxes[7].set(vector3_type(0,0,0),R,vector3_type(8.9,8.9,8.9)/2*size);
             //Hand
        m_boxes[8].set(vector3_type(0,0,0),R,vector3_type(16.9,16.9,51.2)/2*size)
             ; //Thigh
        m_boxes[9].set(vector3_type(0,0,0),R,vector3_type(10.6,10.6,30.5)/2*size)
             ; //Calf
        m_boxes[10].set(vector3_type(0,0,0),R,vector3_type(9.9,27.3,13.9)/2*size)
             ; // Foot


        //head
        m_bones[1].attach(&m_gravity);
        m_bones[1].setPosition(vector3_type(0,0,0)*size+position);
        m_bones[1].setOrientation(quaternion_type(1,0,0,0));
        m_bones[1].setVelocity(vector3_type(0,0,0));
        m_bones[1].setSpin(vector3_type(0,0,0));
        m_bones[1].setGeometry(m_boxes[1]);
        m_bones[1].setFixed(false);
        m_bones[1].setColor(vector3_type(1,1,0));
        m_configuration.add(&m_bones[1]);

        //neck
        m_bones[2].attach(&m_gravity);
        m_bones[2].setPosition(vector3_type(0,0,-16.15)*size+position);
        m_bones[2].setOrientation(quaternion_type(1,0,0,0));
        m_bones[2].setVelocity(vector3_type(0,0,0));
        m_bones[2].setSpin(vector3_type(0,0,0));
        m_bones[2].setGeometry(m_boxes[2]);
        m_bones[2].setFixed(false);
        m_bones[2].setColor(vector3_type(1,1,0));
        m_configuration.add(&m_bones[2]);

        //chest
        m_bones[3].attach(&m_gravity);
        m_bones[3].setPosition(vector3_type(0,0,-42.75)*size+position);
        m_bones[3].setOrientation(quaternion_type(1,0,0,0));
        m_bones[3].setVelocity(vector3_type(0,0,0));
        m_bones[3].setSpin(vector3_type(0,0,0));
        m_bones[3].setGeometry(m_boxes[3]);
        m_bones[3].setFixed(false);
        m_bones[3].setColor(vector3_type(0,0,1));
        m_configuration.add(&m_bones[3]);

        //hip
        m_bones[4].attach(&m_gravity);
        m_bones[4].setPosition(vector3_type(0,0,-70.45)*size+position);
        m_bones[4].setOrientation(quaternion_type(1,0,0,0));
        m_bones[4].setVelocity(vector3_type(0,0,0));
        m_bones[4].setSpin(vector3_type(0,0,0));
        m_bones[4].setGeometry(m_boxes[4]);
        m_bones[4].setFixed(false);
        m_bones[4].setColor(vector3_type(0,1,0.2));
        m_configuration.add(&m_bones[4]);

        //left overarm
        m_bones[5].attach(&m_gravity);
        m_bones[5].setPosition(vector3_type(37.9,0,-26.7)*size+position);
        m_bones[5].setOrientation(quaternion_type(1,0,0,0));
        m_bones[5].setVelocity(vector3_type(0,0,0));
        m_bones[5].setSpin(vector3_type(0,0,0));
        m_bones[5].setGeometry(m_boxes[5]);
    m_bones[5].setFixed(false);
    m_bones[5].setColor(vector3_type(0,0,1));
    m_configuration.add(&m_bones[5]);

    //right overarm
    m_bones[6].attach(&m_gravity);
    m_bones[6].setPosition(vector3_type(-37.9,0,-26.7)*size+position);
    m_bones[6].setOrientation(quaternion_type(1,0,0,0));
    m_bones[6].setVelocity(vector3_type(0,0,0));
    m_bones[6].setSpin(vector3_type(0,0,0));
    m_bones[6].setGeometry(m_boxes[5]);
    m_bones[6].setFixed(false);
    m_bones[6].setColor(vector3_type(0,0,1));
    m_configuration.add(&m_bones[6]);

    //left underarm
    m_bones[7].attach(&m_gravity);
    m_bones[7].setPosition(vector3_type(70.6,0,-26.7)*size+position);
    m_bones[7].setOrientation(quaternion_type(1,0,0,0));
    m_bones[7].setVelocity(vector3_type(0,0,0));
    m_bones[7].setSpin(vector3_type(0,0,0));
    m_bones[7].setGeometry(m_boxes[6]);
    m_bones[7].setFixed(false);
    m_bones[7].setColor(vector3_type(0,0,1));
    m_configuration.add(&m_bones[7]);

    //right underarm
    m_bones[8].attach(&m_gravity);
    m_bones[8].setPosition(vector3_type(-70.6,0,-26.7)*size+position);
    m_bones[8].setOrientation(quaternion_type(1,0,0,0));
    m_bones[8].setVelocity(vector3_type(0,0,0));
    m_bones[8].setSpin(vector3_type(0,0,0));
    m_bones[8].setGeometry(m_boxes[6]);
    m_bones[8].setFixed(false);
    m_bones[8].setColor(vector3_type(0,0,1));
    m_configuration.add(&m_bones[8]);

    //left hand
    m_bones[9].attach(&m_gravity);
    m_bones[9].setPosition(vector3_type(89.45,0,-26.7)*size+position);
    m_bones[9].setOrientation(quaternion_type(1,0,0,0));
    m_bones[9].setVelocity(vector3_type(0,0,0));
    m_bones[9].setSpin(vector3_type(0,0,0));
    m_bones[9].setGeometry(m_boxes[7]);
    m_bones[9].setFixed(false);
    m_bones[9].setColor(vector3_type(1,1,0));
    m_configuration.add(&m_bones[9]);

    //right hand
    m_bones[10].attach(&m_gravity);
    m_bones[10].setPosition(vector3_type(-89.45,0,-26.7)*size+position);
    m_bones[10].setOrientation(quaternion_type(1,0,0,0));
    m_bones[10].setVelocity(vector3_type(0,0,0));
```

```
    m_bones[10].setSpin(vector3_type(0,0,0));
    m_bones[10].setGeometry(m_boxes[7]);
    m_bones[10].setFixed(false);
    m_bones[10].setColor(vector3_type(1,1,0));
    m_configuration.add(&m_bones[10]);

    //left thigh
    m_bones[11].attach(&m_gravity);
    m_bones[11].setPosition(vector3_type(9.45,0,-103.3)*size+position);
    m_bones[11].setOrientation(quaternion_type(1,0,0,0));
    m_bones[11].setVelocity(vector3_type(0,0,0));
    m_bones[11].setSpin(vector3_type(0,0,0));
    m_bones[11].setGeometry(m_boxes[8]);
    m_bones[11].setFixed(false);
    m_bones[11].setColor(vector3_type(0,1,0.2));
    m_configuration.add(&m_bones[11]);

    //right thigh
    m_bones[12].attach(&m_gravity);
    m_bones[12].setPosition(vector3_type(-9.45,0,-103.3)*size+position);
    m_bones[12].setOrientation(quaternion_type(1,0,0,0));
    m_bones[12].setVelocity(vector3_type(0,0,0));
    m_bones[12].setSpin(vector3_type(0,0,0));
    m_bones[12].setGeometry(m_boxes[8]);
    m_bones[12].setFixed(false);
    m_bones[12].setColor(vector3_type(0,1,0.2));
    m_configuration.add(&m_bones[12]);

    //left calf
    m_bones[13].attach(&m_gravity);
    m_bones[13].setPosition(vector3_type(9.45,0,-144.15)*size+position);
    m_bones[13].setOrientation(quaternion_type(1,0,0,0));
    m_bones[13].setVelocity(vector3_type(0,0,0));
    m_bones[13].setSpin(vector3_type(0,0,0));
    m_bones[13].setGeometry(m_boxes[9]);
    m_bones[13].setFixed(false);
    m_bones[13].setColor(vector3_type(0,1,0.2));
    m_configuration.add(&m_bones[13]);

    //right calf
    m_bones[14].attach(&m_gravity);
    m_bones[14].setPosition(vector3_type(-9.45,0,-144.15)*size+position);
    m_bones[14].setOrientation(quaternion_type(1,0,0,0));
    m_bones[14].setVelocity(vector3_type(0,0,0));
    m_bones[14].setSpin(vector3_type(0,0,0));
    m_bones[14].setGeometry(m_boxes[9]);
    m_bones[14].setFixed(false);
    m_bones[14].setColor(vector3_type(0,1,0.2));
    m_configuration.add(&m_bones[14]);

    //left foot
    m_bones[15].attach(&m_gravity);
    m_bones[15].setPosition(vector3_type(9.45,-5.2,-166.45)*size+position);
```

```
        m_bones[15].setOrientation(quaternion_type(1,0,0,0));
        m_bones[15].setVelocity(vector3_type(0,0,0));
        m_bones[15].setSpin(vector3_type(0,0,0));
        m_bones[15].setGeometry(m_boxes[10]);
        m_bones[15].setFixed(false);
        m_bones[15].setColor(vector3_type(.5,.5,0));
        m_configuration.add(&m_bones[15]);

        //right foot
        m_bones[16].attach(&m_gravity);
        m_bones[16].setPosition(vector3_type(-9.45,-5.2,-166.45)*size+position);
        m_bones[16].setOrientation(quaternion_type(1,0,0,0));
        m_bones[16].setVelocity(vector3_type(0,0,0));
        m_bones[16].setSpin(vector3_type(0,0,0));
        m_bones[16].setGeometry(m_boxes[10]);
        m_bones[16].setFixed(false);
        m_bones[16].setColor(vector3_type(.5,.5,0));
        m_configuration.add(&m_bones[16]);


        //attach head to neck
        ragdoll_hinge_joint(&m_bones[1], &m_bones[2], vector3_type(0,0,-10)*size+
             position, x, -OT_M_PI/4, OT_M_PI/4);

        //attach neck to chest
        ragdoll_ball_joint(&m_bones[2], &m_bones[3], vector3_type(0,0,-22.3)*size
             +position);

        //attach hip to chest
        ragdoll_ball_joint(&m_bones[4], &m_bones[3], vector3_type(0,0,-61.5)*size
             +position);

        //attach left overarm to chest
        ragdoll_ball_joint(&m_bones[5], &m_bones[3], vector3_type(19.6,0,-26.7)*
             size+position);

        //attach right overarm to chest
        ragdoll_ball_joint(&m_bones[6], &m_bones[3], vector3_type(-19.6,0,-26.7)*
             size+position);

        //attach left underarm to left overarm
        ragdoll_hinge_joint(&m_bones[7], &m_bones[5], vector3_type(52.3,0,-26.7)*
             size+position, z, -OT_M_PI*.75, 0);

        //attach right underarm right overarm
        ragdoll_hinge_joint(&m_bones[8], &m_bones[6], vector3_type(-52.3,0,-26.7)
             *size+position, z, -OT_M_PI*.75, 0);

        //attach left hand to left underarm
        ragdoll_hinge_joint(&m_bones[9], &m_bones[7], vector3_type(81.1,0,-26.7)*
             size+position, x, -OT_M_PI/2, OT_M_PI/2);

        //attach right hand to right underarm
        ragdoll_hinge_joint(&m_bones[10], &m_bones[8], vector3_type
             (-81.1,0,-26.7)*size+position, x, -OT_M_PI/2, OT_M_PI/2);

        //attach left thigh to hip
        ragdoll_ball_joint(&m_bones[11], &m_bones[4], vector3_type(9.45,0,-77.7)*
             size+position);

        //attach right thigh to hip
        ragdoll_ball_joint(&m_bones[12], &m_bones[4], vector3_type(-9.45,0,-77.7)
             *size+position);

        //attach left calf to left thigh
        ragdoll_hinge_joint(&m_bones[13], &m_bones[11], vector3_type
             (9.45,0,-128.9)*size+position, x, -OT_M_PI*.75, 0.0);

        //attach right calf to right thigh
        ragdoll_hinge_joint(&m_bones[14], &m_bones[12], vector3_type
             (-9.45,0,-128.9)*size+position, x, -OT_M_PI*.75, 0.0);

        //attach left foot to left calf
        ragdoll_hinge_joint(&m_bones[15], &m_bones[13], vector3_type
             (9.45,0,-159.4)*size+position, x, -OT_M_PI/2, 0.0);

        //attach right foot to right calf
        ragdoll_hinge_joint(&m_bones[16], &m_bones[14], vector3_type
             (-9.45,0,-159.4)*size+position, x, -OT_M_PI/2, 0.0);


        m_gravity.setAcceleration(vector3_type(0,0,-9.81));
        m_simulator.init(m_configuration);
        material_type * default_material = m_library.getDefault();
        default_material->setFriction(0.25);
        default_material->setNormalRestitution(0.15);

        m_configuration.setMaterialLibrary(m_library);

        m_simulator.m_stepper.set_iterations(10);

    }

  };

}
#endif
```

References

[1] David Baraff. *An Introduction to Physically Based Modeling: Rigid Body Simulation I. Unconstrained Rigid Body Dynamics.* Carnegie Mellon University, 1997.

[2] Niels Boldt. A paradigm for simulating rigid-, deformable- and liquid-bodies. Master's thesis, Computer Science, University of Copenhagem, July 2004.

[3] AutomatedQA Corp. Aqtime v5.10, December 2006. www.automatedqa.com.

[4] Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-based Animation.* Charles River Media, 2005.

[5] Jakob Holck. Dynamisk simulering af artikulerede stive legemer. Master's thesis, University of Copenhagen. Computer science, Marts 2005.

[6] IO Interactive. Hitman, codename 47. Puplished by Eidos, 2001. http://www.ioi.dk/games/hitman1.htm.

[7] Thomas Jakobsen. Advanced character physics. 2001.

[8] NASA. Human proportions.

[9] OpenTissue. Opensource Project, Physical based Animation and Surgery Simulation. http://www.opentissue.org.

[10] Jovan Popovic, Steven M. Seitz, Michael Erdmann, Zoran Popovicy, and Andrew Witkinz. Interactive manipulation of rigid body simulations. 2000.

[11] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley, Boston, MA, USA, 2003.

[12] Jereon M. Wagenaar. *Physically Based Simulation and Visualization.* PhD thesis, The Mærsk Mc-Kinney Møller Institute, DK, 2001.

[13] Jane Wilhelms and Allen Van Gelder. Fast and easy reach-cone joint limits. *J. Graph. Tools*, 6(2):27–41, 2001.