Ragdoll Physics

Gabe Mulley, Matt Bittarelli

April 25th, 2007

Abstract

The goal of this project was to create a real-time, interactive, and above all, stable, ragdoll physics simulation. This simulation will allow the user to apply arbitrary impulse forces to the ragdoll and observe its reactions. Finally the system will be designed in such a way that arbitrary dynamic systems can be simulated by extending this framework.

1 Introduction

This paper presents an implementation of a constrained dynamics solution which is heavily based on the work of Jakobsen [1]. The goal of solving the constrained dynamics problem is to be able to define relationships between particles in a system, and to ensure that those relationships are never violated. These relationships are called constraints. The simplest example of a constraint is a point that is fixed in space, the constraint specifies that the point must remain at a given position. Given that there may be many constraints associated with a particle, it follows that finding a particle configuration that does not violate any constraints is non-trivial. Using a system capable of solving this problem, realistic animations can be generated for various dynamic systems such as articulated systems, cloth, and even rigid bodies [1]. Specifically, this paper focuses on the ragdoll dynamics problem which attempts to model a limp human body using a system of particles and constraints. This type of model is useful for animating the effects of external forces on a human body which does not actively resist the action of the force.

2 Background

2.1 Previous Work

Advanced Character Physics [1] was the primary resource for the implementation of this project. This paper was written by a lead developer at IO Interactive Studios, makers of the “Hitman” game series. Their implementation included a constraint solver using a relaxation method. Given system of particles being used to model a dynamic system, each constraint specified in this system would be repeatedly satisfied locally allowing the entire system to converge to an approximate solution. Also, when colliding geometry was detected using this implementation, the offending geometrys position would simply be moved (or projected) out of the intruded area. This was the ideal paper to model our solution after because it had already been implemented by a well-respected game studio, and was described to be a stable, computationally inexpensive, and easy to implement. Although this paper was the optimal choice for us to emulate, the main flaw was that while the math was based on papers which provided a more accurate solution, the solution described in [1] is merely an approximation.

Witkin [2] presents a mathematical model and a solution method for the constrained dynamics problem. A brief summary of the method presented in his paper follows. First consider a vector $\mathbf{q}$ of length $3n$ where n is the number of particles in the system. It contains all of the $3D$ positions of each particle and is called the state vector. The implementation of this method requires the definition of a vector function, $\mathbf{C}(\mathbf{q})$, which accept a vector of length $3n$ and return a vector of length m, where m is the number of constraints in the system. Each row of the resulting vector is the result of passing q to a single constraint function. $\dot{\mathbf{C}}(\mathbf{q})$ is defined as the first time-derivative of $\mathbf{C}(\mathbf{q})$. To implement this system functions must be written to evaluate $\mathbf{C}(\mathbf{q})$, and $\dot{\mathbf{C}}(\mathbf{q})$ for every constraint in the system. These functions should be implicit functions for which $\mathbf{C}(\mathbf{q}) = 0$, $\dot{\mathbf{C}}(\mathbf{q}) = 0$, and $\ddot{\mathbf{C}}(\mathbf{q}) = 0$ if the constraint is satisfied, and nonzero otherwise. Also functions must be written to evaluate the Jacobian, $\mathbf{J}$, of $\mathbf{C}(\mathbf{q})$ with respect to $\mathbf{q}$, and $\dot{\mathbf{J}}$. This Jacobian matrix will have n rows and m columns, and allow the evaluation of the partial derivative of any constraint function with respect to any member of the state vector.

$$\mathbf{J} = \begin{pmatrix} \frac{\partial C_1}{\partial q_1} & \frac{\partial C_2}{\partial q_1} & \cdots & \frac{\partial C_m}{\partial q_1} \\ \frac{\partial C_1}{\partial q_2} & \frac{\partial C_2}{\partial q_2} & \cdots & \frac{\partial C_m}{\partial q_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial C_1}{\partial q_n} & \frac{\partial C_2}{\partial q_n} & \cdots & \frac{\partial C_m}{\partial q_n} \end{pmatrix}$$

Since most constraints will only actually make use of a very small number of elements of the state vector, the resulting $\mathbf{J}$ and $\dot{\mathbf{J}}$ matrices will be very sparse. For this reason, Witkin recommends representing them as sparse matrices. Once all of this has been established some simple matrix arithmetic will produce a matrix that can be inverted to solve for the Lagrange multiplier vector, λ, which can, in turn, be used to determine the constraint forces which must be applied to the particle in addition to the external forces to result in all of the constraints being satisfied. The advantage of this method is that the solution it provides is physically accurate. The disadvantage, however, is that it requires a great deal of computation to compute the $\mathbf{J}$, $\dot{\mathbf{J}}$ matrices and finally to invert the sparse matrix to solve for λ. Disregarding the efficiency of the sparse matrix inversion method, this solution method is $O(2m + 2mn)$, compared to the solution presented by Jakobsen which runs in approximately $O(lm)$ where l is the number of relaxation iterations as discussed in the following sections. It is already obvious that it is not nearly as fast as the Jakobsen method, which does not even depend on the number of particles in the system, at least as far as the constraint solver is concerned.

Although the Jakobsen and Witkin methods both solve the same problem, they do so in two very different ways, and end up producing different solutions. The focus of the Jakobsen method is on producing a stable, believable solution, while the Witkin method attempts to produce a physically accurate solution. Witkins solution ensures that all constraints are satisfied after every timestep, while Jakobsens makes no such guarantee. Jakobsen is willing to allow some constraints to remain unsatisfied as long as they are close to being satisfied. This results in a significant loss of accuracy, but still results in believable motion. In addition, Jakobsen's method also garantees stability at the cost of accuracy. A situation that would normally cause instability due to the timestep being too large will instead simply cause further inaccuracy, which is acceptable for a video game type application.

2.2 Practical Use

Ragdoll physics are becoming more and more widely used in commercial as well as proprietary video game engines to simulate a wide variety of behaviors. These can include rigid bodies, cloth, and of course, biologically based characters which have gone limp. As mentioned previously, the developers of the Hitman game series used this simulation to model dead or dying characters in their games. However, other developers have extended this behavior in games such as Ico where the avatar is leading a non-player character by the hand. This is an interesting situation because in this case, predefined animations are spliced with the procedural rag doll animations to produce a life-like animation sequence of a person being pulled through an environment by their hand.

3 Implementation Details

The goal concerning the class and data structure organization was to make an expandable system which would allow for easy code portability, and encapsulation for extensions on our implementation. This section will include an overview of the code structure we used for our ragdoll system (not including environment and user input code), and our constraint algorithms.

Our base DynamicSystem class is responsible for providing any behaviors that all kinds of dynamic systems (which includes rag dolls, cloth, and rigid bodies) could possibly share. This mainly included parsing an input file to read in joints and constraints. The DynamicSystem class is also responsible for storing an array of Constraint objects, and Joint objects. Joint objects can be thought of as particles which store a position, whose job is to approximate an object, whether it is a 2D grid for a piece of cloth, or a more complex shape to represent a humanoid skeleton.

The Constraint class is merely a pure virtual class used to define a few functions which all inheriting Constraint objects will be expected to implement given each of their unique configurations. With this object oriented approach, all inheriting Constraint objects are able to satisfy their own local constraints without affecting the global outcome. This is optimal to produce a converging solution for the entire system.

For the specific case we modeled with our implementation, we created a Skeleton class which derived from the DynamicSystem class. The Skeleton object has all the basic constraint information that every other Dynamic System could have, except all Skeleton objects had a few additional features. In order to visualize the entire skeleton model, each pair of Joints is assigned to a Bone object which joins them. A Bone object is used specifically to visualize the skeleton hierarchy and the physical relationship between each joint and they are stored in an array used by the Skeleton class. The Bone objects are also used in procedurally generating a bounding box to encapsulate each Bone and giving the Skeleton some visual mass. The Bounding Box objects are also used for interaction between the user and the simulation for the user-exerted impulse forces.

3.1 Solving the Constraint System

This paper makes use of the Jakobsen [1] method to solve the constraint system. Basically this method makes the following assumption which Jakobsen claims is justifiable but does not justify it explicitly in his publication. The assumption is that iterating through all of the constraints in the system multiple times will converge to a solution which approximately satisfies all of the constraints in the system. That is, repeatedly satisfying each of the constraints locally will result in a globally satisfied system given the correct conditions. The number of iterations is a parameter to our system, and affects both the accuracy and speed of the model. Once a constraint solution has been acquired, the system can then perform verlet integration to advance the particles in preparation for the next timestep.

3.2 Verlet Integration

Constraints as defined in the Jakobsen paper enforce position relationships, they are unconcerned with the velocity and acceleration of the particles they manipulate. Because of this fact, particles are being moved around quite frequently without any concern for the present direction and magnitude of their velocity vectors. Therefore, when attempting to integrate, a method must be used to derive the particles present velocity. To do so, the previous position, x_0, and the present position, x is stored on every particle. Thus we can approximate the particles present velocity by (1).

$$v = \frac{(x - x_0)}{dt} \tag{1}$$

Given Euler integration:

$$\begin{align} v &= v + a{\cdot}dt \tag{2} \\ x &= x + v{\cdot}dt \tag{3} \\ x &= x + v{\cdot}dt + a{\cdot}dt^2 \tag{4} \end{align}$$

Now substituting (1) into (4)

$$x = 2x - x_0 + a{\cdot}dt^2 \tag{5}$$

Verlet integration is defined as (5). Therefore Verlet integration is simply Euler integration with a velocity approximation derived from the last position of the particle. This integration method is crucial to the constraint solution method used by this paper, as it allows particles to be moved arbitrarily by the constraints. The Verlet integration itself does not appear to improve the stability or speed of the constrained dynamics solver, it does, however, enable the use of a faster and more stable Jakobsen solver. Due to the inherent dependency of the solver on Verlet integration it is impossible to directly compare its performance with another integration technique.

3.3 Constraints

3.3.1 Fixed

The fixed constraint is created by specifying a given joint. The fixed constraint stores the initial joint position as specified in the input file. The algorithm used to enforce this constraint, is to check if the joint's position is ever different from its initial position. If the position values were not equal, the fixed constraint class would simply override the joint object's position value to its original value.

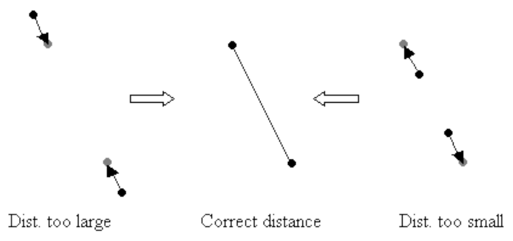


Figure 1: Length Constraint [1]

3.3.2 Length

The length constraint was derived by specifying two joints, $\mathbf{j_1}$ and $\mathbf{j_2}$, in the input file. Our method assumed that the "natural length", which the constraint would look to enforce, would simply be: $|\mathbf{j_1} - \mathbf{j_2}|$ from their starting positions specified in the input file. In the future, it would be possible to extend this to specify a specific length in the input file. If $|\mathbf{j_1} - \mathbf{j_2}|$ ever violates the length value derived from the "natural length", the two joints will calculate their new positions depending on if their current length is greater or less than the natural length. Figure 1 illustrates this process.

3.3.3 Angular

The angular constraint takes three joints ($\mathbf{j_1}$, $\mathbf{j_2}$, $\mathbf{j_3}$), a minimum angle (θ_{min}), a maximum angle (θ_{max}) as parameters. It requires that the angle between the vectors $\mathbf{j_1}$, $\mathbf{j_2}$ and $\mathbf{j_3}$, $\mathbf{j_2}$ is greater than θ_{min} and less than θ_{max}. This constraint is enforced with conditional length constraints. Given a θ_{min} and a θ_{max} one can derive a D_{min} and D_{max} using the law of cosines. D_{min}, D_{max} represent the minimum and maximum distance between $\mathbf{j_3}$ and $\mathbf{j_1}$), and can be used construct two length constraints, MinLengthConstraint and MaxLengthConstraint. If $|\mathbf{j_3} - \mathbf{j_1}| < D_{min}$ then the rule for MinLengthConstraint is used and if $|\mathbf{j_3} - \mathbf{j_1}| > D_{max}$ then the rule for MaxLengthConstraint is used to ensure this constraint is satisfied. It does not have any concept of positive and negative angles, this limitation is described in more detail later in this paper.

3.3.4 Minimum Distance

The minimum distance constraint requires two joints ($\mathbf{j_1}$, $\mathbf{j_2}$) and a minimum distance (D_{min}). It can be thought of as a conditional length constraint. It is satisfied if and only if the distance between the two joints is greater than the minimum distance. If it is violated it simply uses the same rule as the length constraint to ensure it is satisfied. It ensures that $|\mathbf{j_2} - \mathbf{j_1}| > D_{min}$.

3.4 Bounding Box Generation

Bounding boxes are procedurally generated given a bone and the two joints associated with each bone. Bounding box objects are stored using two arrays of length four. Each array represents four vertices of an extruded square which encapsulates the entire bone visualization.

To generate the position of each end of the bounding box, a vector from $\mathbf{j_1}$ to $\mathbf{j_2}$ must be specified by the following operations: Let $\mathbf{u} = \mathbf{j_2} - \mathbf{j_1}$. Secondly, we must find another vector $\mathbf{v}$ $90°$ from $\mathbf{u}$ by solving:

$$\mathbf{u} \bullet \mathbf{v} = 0 \tag{6}$$

$$u_x{\cdot}v_x + u_y{\cdot}v_y + u_z{\cdot}v_z = 0 \tag{7}$$

$$v_z = \frac{-u_x{\cdot}v_x - u_y{\cdot}v_y}{u_z} \tag{8}$$

Since we are already given the vector $\mathbf{u}$, we can substitute arbitrary values for v_x and v_y and solve for v_z. Once vector $\mathbf{v}$ is solved for, the vector must be normalized. Now $\mathbf{w}$ can be determined.

$$\mathbf{w} = \mathbf{u} \otimes \mathbf{v} \tag{9}$$

Having vectors $\mathbf{v}$ and $\mathbf{w}$, they can be scaled by an arbitrary size variable which indicates how far apart the four vertices of a bounding box end will be. These four vertices can be calculated by: $\mathbf{v}+\mathbf{w}$, $\mathbf{v}-\mathbf{w}$, $-\mathbf{v}+\mathbf{w}$, and $-\mathbf{v}-\mathbf{w}$. These operations can be repeated a second time to calculate the other end of the bounding box with $\mathbf{u} = \mathbf{j_1} - \mathbf{j_2}$. These operations also must be done every frame to update the orientation and position of each bounding box in the system. These operations would not be necessary for a production system as bounding boxes would be derived from the actual mesh.

3.5 Impulse Forces

In our implementation, the user is allowed to exert impulse forces on the rag doll system by clicking the right mouse button while the cursor is positioned over a bounding box in the ragdoll system. The first obstacle is to calculate which bounding box the user is clicking on. This can be achieved by creating a vector from the eye of the viewer (or in this case the Camera objects position) to the users mouse cursor in $3D$ world space and checking to see if it intersected with any of the bounding boxes in the system. The position of the users mouse cursor was translated from $2D$ screen space to $3D$ world space using the gluUnProject function.

Once a valid intersection was found with the impulse force vector and a bounding box, a linear interpolation between the Joints associated with the bounding boxs bone, will provide the appropriate force acceleration to apply to each joint. Assuming the point at which the user clicked in $3D$ world space was called vector $\mathbf{u}$ and the impulse force is $\mathbf{F}$, then

$$\mathbf{a_1} = \mathbf{a_1} + \frac{|\mathbf{j_1} - \mathbf{u}|}{|\mathbf{j_1} - \mathbf{u}| + |\mathbf{j_2} - \mathbf{u}|} \cdot \mathbf{F} \tag{10}$$

$$\mathbf{a_2} = \mathbf{a_2} + \frac{|\mathbf{j_2} - \mathbf{u}|}{|\mathbf{j_1} - \mathbf{u}| + |\mathbf{j_2} - \mathbf{u}|} \cdot \mathbf{F} \tag{11}$$

where $\mathbf{a_1}, \mathbf{a_2}$ are the new accelerations of joints one and two. Once each joints acceleration values are modified, the simulation will be sent to the Verlet integration function to calculate the new positions of each joint.

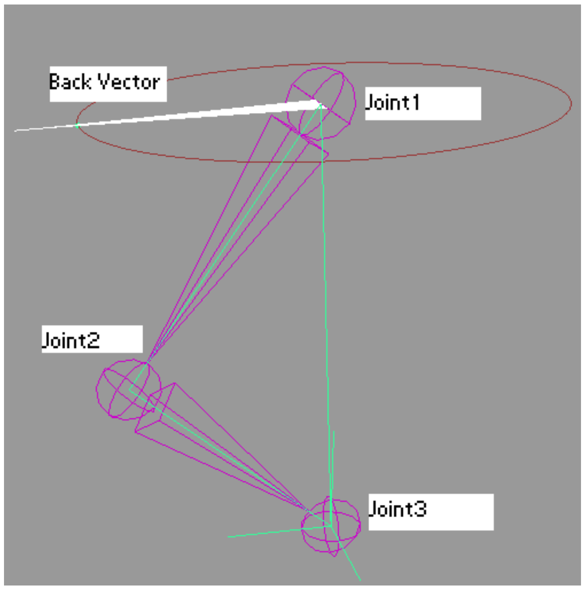


Figure 2: Back Vector Illustration from Maya

4 Limitations

In our implementation, one of our major limitations can be found in enforcing the angular constraint in a given system. This solution is time-step and iteration sensitive, so the lower the time-step and the more iterations given to the Verlet integration function, the more accurate the results will potentially be. A common problem in this simulation is given a joint configuration with a large maximum angular constraint; the joint will be visually moving too fast and violate its angle constraint. With this implementation, we do not differentiate between an angle of 170° and 190° (or -170°). To enforce this constraint in a more accurate way, a back vector would always need to be calculated (such as done with the animation software Maya). As shown in Figure 2, the back vector should always point in the direction of Joint2 to ensure that the angle between Joint1, Joint2, and Joint3 can never be greater than 180° . Calculating this back vector proved to be challenging and will require a further investigation.

Another limitation worth noting is that the system described in this paper has no constraint to handle axial rotation. Axial rotation is defined as rotation of a bone about the axis that is collinear with the vector that defines the bone. Rotation about this axis causes the bounding box for the bone to spin unnaturally, it also allows the ragdolls head to spin a complete 360° around its neck. As the system stands presently each joint is represented by a single point, to be able to limit this type of motion the joint would require additional information to be stored. In addition, any additional vectors stored on each joint would have to be modified and handled correctly by all other constraints in the system.

5 Results

The simplest scene used to test this system was a double pendulum (See Figure 3). When gravity is allowed to act on the system, then it is expected to behave like a double pendulum. It demonstrates working length constraints, fixed constraints, and correct advancement of the physical simulation from one timestep to the next.

The second demo (Figure 4 introduces an angular constraint on the double pendulum example, preventing the angle between the two pieces of the double pendulum to exceed 90° . This was used to test angular constraints.

The third demo (Figure 5) shows the same double pendulum with another angular constraint for which θ_{max} is 170° . When simulating this scenario, the angle between the bones does in fact exceed 170° , in fact it exceeds 180° , then when the pendulum reverses direction and the angle approaches 170° again, this time it behaves correctly. This failure demonstrates two limitations of our system. The first is timestep sensitivity. Although the systems stability is not sensitive to the timestep, its accuracy is, so given high velocity motion the timestep must be sufficiently small to prevent potentially significant losses in accuracy. The second is the angular constraints lack of knowledge about positive and negative angles. If the desired behavior for this configuration was for the angle to vary between 0 and 170° and the assumption is that clockwise angles are positive, then in this example the constraint would actually have been enforced at -170° or 190° . This is because the angular constraint is really just measuring the angle between two vectors, and that angle can never be greater than 180° .

The fourth demonstration (Figure 6) consists of two bones which are connected and centrally fixed. When their motion is simulated they should hit one another and bounce off. They do not, and this scenario was chosen specifically because it highlights this lack of collision detection.

The fifth demonstration (Figure 7) is a very similar setup as the fourth test, however a minimum distance constraint is put between the two joints that are in motion. This simulates some very crude collision detection and prevention.

The final and most comprehensive demonstration is that of the entire ragdoll (Figure 8. The skeleton structure is derived directly from the one drawn in [1]. This demonstration allows the user to click on various parts of the ragdoll causing an impulse force to be applied to it at that point. All of the limitations listed above can be seen in this demonstration if the user looks closely enough. In addition, it becomes easy to observe the need for an axial rotation constraint, which is described in more detail below. Most importantly, however, it looks believable, and serves as a proof of concept.

6 Challenges

6.1 Failures

Throughout this project, we had various failed or unattempted features which could be implemented in future work. These limitations of the project includes the angular constraint problem, the axial rotational constraint, and a more accurate Collision detection algorithm between Bounding Boxes in the system.

Our method of solving the collision detection problem was to place minimum distance constraints between two specified joints. While this was satisfactory for an approximation, a more realistic and accurate solution can be derived by creating a hierarchy of bounding boxes encapsulating the rag doll to provide areas of greater or less accuracy, and then testing for an elongated cube intersection. If two elongated cubes happened to collide with each other, the same method that was used in [1] can be used, namely by projecting the offending bounding box out of the other bounding box by a distance proportional to how much they had intersected.

In addition, a more advanced extension of this implementation could provide a realistic character mesh encapsulated by the bounding box approximations. Given the appropriate constraints and not explicitly drawing the bounding boxes, this will provide for a much more realistic ragdoll and simulation. This is similar to techniques used in many game engines.

6.2 Sucesses

During the course of this project, we faced many challenges that we were able to overcome. Specifically the most challenging parts of the code were the definition of the constraints, the generation of the bounding boxes, and the correct handling of user input. The constraints had to be defined carefully, or else the system could produce results that looked very unrealistic, or worse, did not converge. We took great care in defining all of the constraints that we did, with the exception of the angular constraint, which we believe could be handled better but the present solution is functional. The philosophy taken in the constraint implementation was that the constraints should only modify the system in the least possible way to move it to a state that satisfied the constraint. The angular constraint is the only one which does not necessarily follow this philosophy. Instead of its present behavior, which alters the distances between the joints involved, it really should only alter the angle between the vectors connecting the joints.

Bounding box generation was also non-trivial. It required a fair amount of thought as to how to first find a vector that was perpendicular to the vector connecting two joints, and then to find a third vector perpendicular to the other two. The first problem was solved by exploiting the fact that if two vectors are perpendicular their dot product must be equal to 0. The second took advantage of the cross product.

The final significant challenge that the authors had to overcome was that of allowing the user to exert force on the body by clicking with the mouse. This solution to this problem is described in more detail above.

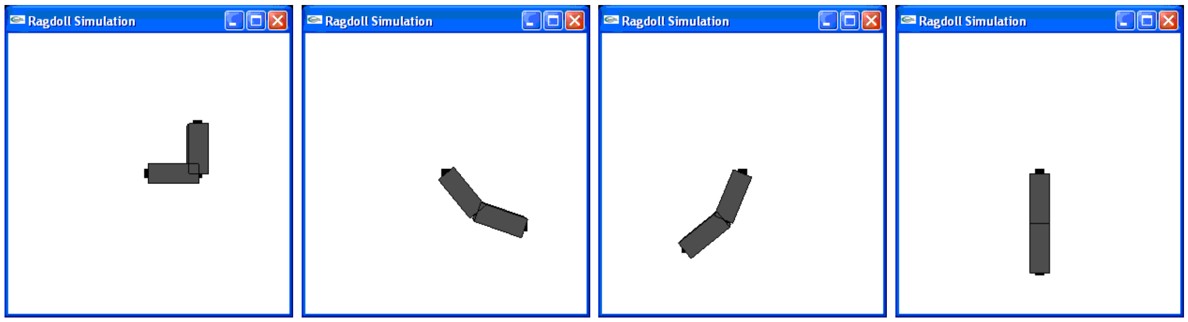
7 Conclusion

We estimate that this project took approximately 70 hours to complete, including design, coding, testing, and finally preparation of presentation materials, including this document. We decided to only use new code or code we had personally written at some time in the past on this project. This enables us to claim full rights for the software, as well as enabling us to use it for portfolios and other miscellaneous purposes. In terms of the division of labor, there are comments interspersed throughout the code which vaguely specifies which functions were written by and contributed to by whom. As a general rule, however, almost all of the core functionality of the system was worked on by both of us at the same time. We worked very closely in defining those problems, generating solutions, and finally implementing them in the code. The only parts of the code that were worked on by only one of us or the other were relating to the setup of the scene and the handling of lights, the camera, and picking.

References

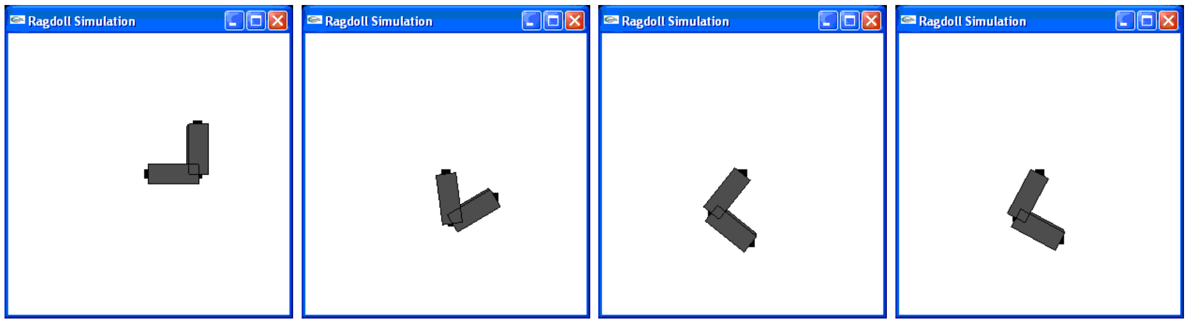
[1] Thomas Jakobsen. Advanced character physics. Game Developers Conference, 2001.

[2] Andrew Witkin. Physically based modeling: Principles and practice. Siggraph 1997 course notes, 1997.



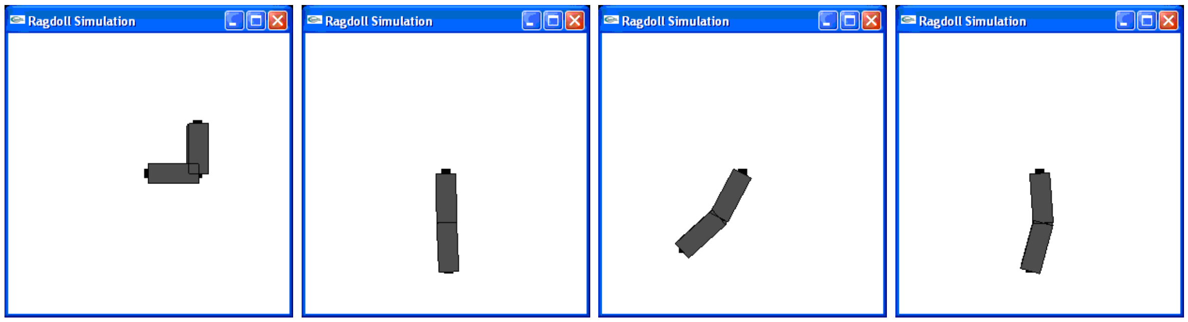
(a) Initial Configuration (b) Down Swing (c) Up Swing (d) Final Configuration

Figure 3: Double Pendulum



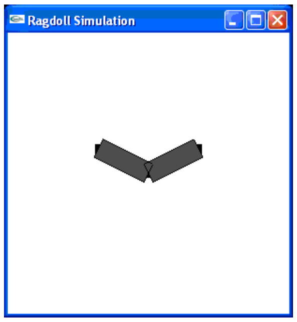
(a) Initial Configuration (b) Down Swing (c) Enforcing 90° Angular Constraint (d) Final Configuration

Figure 4: Double Pendulum with 90° Angular Constraint

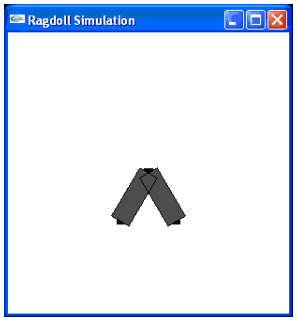


(a) Initial Configuration (b) Violating 170° Angular Constraint (c) Enforcing -170° Angular Constraint (d) Final Configuration

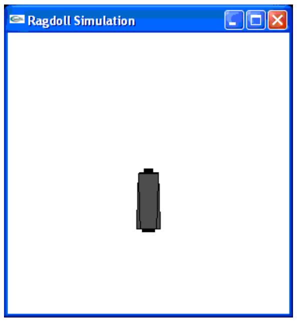
Figure 5: Double Pendulum with 170° Angular Constraint



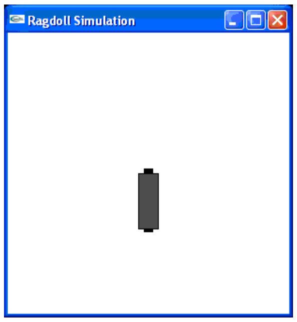
(a) Initial Configuration



(b) Down Swing

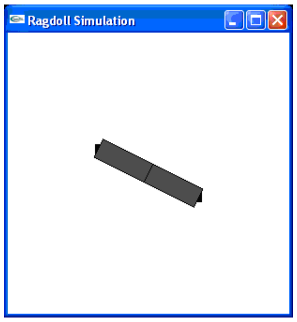


(c) Violated Collision

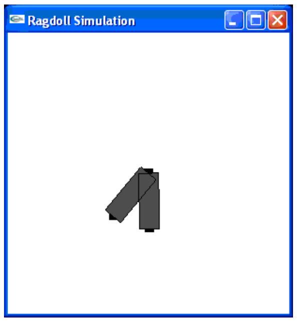


(d) Final Configuration

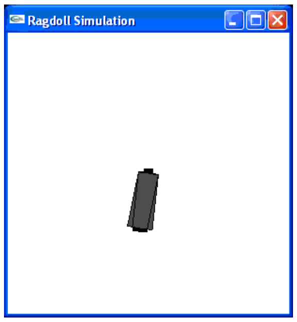
Figure 6: Violated Collision



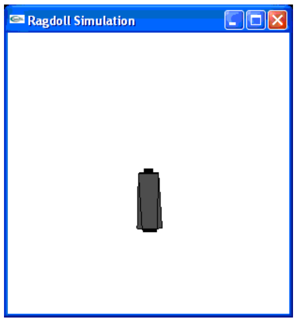
(a) Initial Configuration



(b) Down Swing

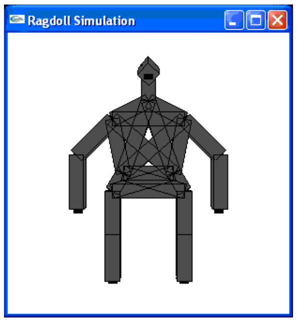


(c) Enforced Minimum Distance Constraint

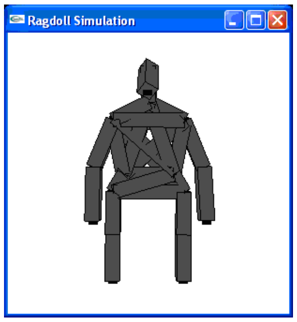


(d) Final Configuration

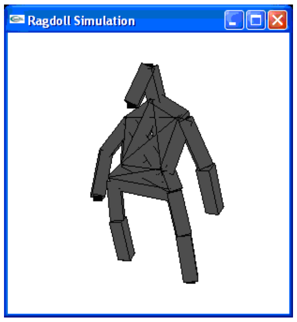
Figure 7: Collision Enforced with Minimum Distance Constraint



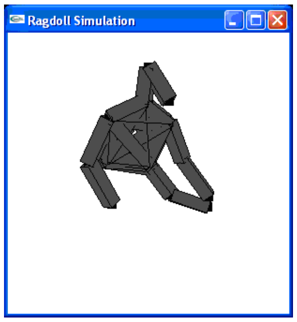
(a) Initial Configuration



(b) Relaxed Rag Doll



(c) Impulse Force Applied to the Dolls Right Shoulder



(d) Second Impulse Force Applied to the Dolls Left Foot

Figure 8: Collision Enforced with Minimum Distance Constraint