A Real-Time Physics Simulator for Jenga™

Matthew South

Supervised by Dr Steve Maddock

Date: 7th May 2003

This report is submitted in partial fulfilment of the requirement for the degree of Master of Science with Honours in Computer Science by Matthew South.

Declaration

Name: Matthew South

Signature:

Date:

Abstract

The project sets out to create a real-time physics simulator for the game of Jenga™ in 3D. It is observed that the game of Jenga™ is physically that of creating a stable stack of blocks, a situation with a high number of contacts indirectly connected to each other at any one time. Analysis of the popular rigid-body method for physics based simulation shows it not to be an ideal solution for such a large number of contacts and an alternative, spring-based method is devised.

The spring-based method developed has the advantage of guaranteed resolution of all collisions after a couple of iterations, an essential feature when dealing with a stack of blocks such as in Jenga™. On the negative side, the blocks can now be deformed, which can only be reduced by taking small time steps. This proves to be quite possible for the game of Jenga™ due to its largely low velocity nature.

The results of the spring-based method are quite encouraging. Though the simulator produced cannot handle the full Jenga™ tower, testing scenarios against a portion of the real-tower produced very convincing results. Deformation of blocks is not noticeable, and physical behaviour is very believable. The simulator achieves real-time speeds for about a third of a full Jenga™ tower.

Acknowledgments

I'd like to thank Steve Maddock, my project supervisor, for letting me do the project in the first place and providing invaluable help and support throughout. I'd also like to thank Brendan Mackrill for many lively debates, not just on this project, but on many topics over the past three years. Finally, I'd like to thank Dale Rushforth for putting up with me when things were going badly and for showing that there's life after Jenga™, difficult though that may seem right now.

Contents

Table of Figures

Chapter 1: Introduction

The aim of this project is to create a real-time physics simulator for the game of Jenga™, a stacking game involving a tower of 52 blocks arranged in 18 rows of 3. Players take it in turns to remove a block from the middle of the tower and place it back on top: the player that causes the tower to collapse is the looser.

Figure 1.1: A Jenga™ tower

The motivation for this project stems from the belief that gameplay wise, games are by and large where they were years ago. Computer games are at a stage where graphics has become the paramount concern for most developers, with each new hardware upgrade used to throw more polygons at the gamer. Whilst it is true that graphics play an extremely important part in representing a realistic environment to the user, it is not the only thing that matters. At the same time graphics have been experiencing huge leaps in technology over the years, notably with the introduction of 3d hardware, it has not been true for gameplay. Many people believe that gameplay is essentially the same as it was years ago; pressing buttons, opening doors, collecting keys, jumping from platform to platform etc. Only the graphics disguise the fact that the gameplay is essentially nothing new.

Physics in games is at a much more primitive state. Games rely heavily on key frames to animate objects rather than model them physically. This reduces, if not eliminates altogether, emergent gameplay as key frames all have to be pre-defined when the game is created. With a physical simulation, behaviour can happen that the developer of the game was not even aware of, creating a much richer experience for the player. Jenga™ is an attempt to create a simple game based entirely on physics; no key frames or pre-defined behaviour whatsoever.

The main aim of this project is to create a physics simulator that can simulate a Jenga™ tower as believably as possible in real-time. The physics simulator must be able to handle a large number of blocks in a stack configuration in real-time. A secondary task, though also crucial to the project, is to develop an input system that will enable a player to interact with the tower much like they do with the real one.

This report will detail the development of a Jenga™ simulation from conception to completion. Chapter 2 of this report will look into the current methods being used to model physics in real-time and will set up the fundamental theory behind physics simulation. Chapter 3 will analyse the specific problem of Jenga™ in detail, mainly its physical requirements but also its gameplay ones. The findings in this chapter will lead to the final design that is used, which will be described in chapter 4. Chapter 5 will detail the implementation of the Jenga™ simulation following the design as laid out in chapter 4 and will discuss any novel aspects encountered. Chapter 6 will evaluate the success of the program developed against the project requirements and will suggest any areas of improvement for the future. The simulation will be tested against a real Jenga™ tower to see how well it compares. Finally, chapter 7 will give a summary of the main findings of the project, what was done well, and what could have been done better.

Chapter 2: Physics-based Simulation

This section will look at the techniques currently in use in creating a general physics simulator. Initially, the methods used to model the motion of a particle will be discussed, as this is the basis of any physics simulator. The ideas will then be extended to that of modelling the motion of a body. After establishing how to simulate a free moving object, this chapter will then go on to discuss how to deal with the situation of colliding objects. This is divided into the two sections; collision detection and collision response. Collision detection is generally regarded as the most crucial part of any physics simulator and so particular importance will be paid to that section. Collision response will look at how collisions can be dealt with, as well as looking at a model for friction.

2.1 Modelling the Motion of a Particle

A particle, in physics simulation terms, can be thought of as a point mass. The important fact to note is that a particle has mass, but neglects the properties of volume and shape: as a consequence they are much simpler to simulate than a full body.

2.1.1 Newtonian Equations

Let's say that at a particular time a particle's position and velocity are both known. Sir Isaac Newton's First Law of Motion states that from these properties, assuming the particle is affected by no outside forces, the motion of the particle can be modelled forward in time indefinitely. If the velocity is zero the particle will simply remain still; if it is non-zero the particle will continue on at that velocity forever. This is described formally by Newton's First Law of Motion.

> *"Every object in a state of uniform motion will remain in that state of motion unless an external force is applied to it."*
>
> *Newton's First Law of Motion*

For a particle to change velocity therefore, a force needs to be exerted upon it. By definition, this will cause the particle to accelerate (acceleration is defined as change in velocity over time). This is described more formally in Newton's Second Law of motion.

$$\mathbf{F} = m\mathbf{a} \qquad \text{(Eq. 2.1)}$$

Equation 2.1 relates the acceleration of an object to its net force by its mass. This is why it is important for a particle to have mass even though conceptually it is infinitely small: a zero mass would produce a zero acceleration regardless of external force.

It has now been established that at any given time a particle has a mass, position, velocity and acceleration. From these four properties alone it is possible to simulate a particle forward in time indefinitely, so long as no external forces interfere with it. If a force does exert on the particle, a new acceleration can be calculated from Newton's Second Law of Motion and the particle can continue to be simulated forward in time. To do this we first of all need to understand the exact relationship between position, velocity and acceleration.

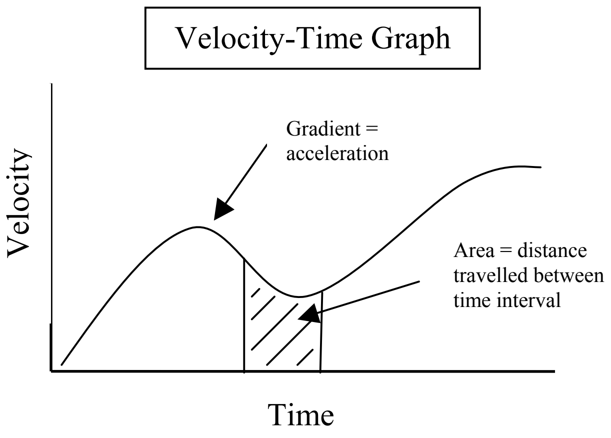


Figure 2.1: Velocity-Time Graph

By definition, velocity is the rate of change of position and similarly acceleration is the rate of change of velocity. The latter can be seen on the velocity-time graph above as the gradient. In general we cannot assume that these rates of change are linear and so the graphs they produce will, in general, form a curve. To calculate gradients of a curved graph, it is necessary to turn to calculus. The two relationships described above can therefore be expressed as:

$$\mathbf{v} = \frac{d\mathbf{r}}{dt} \qquad \mathbf{a} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{r}}{dt^2} \tag{Eq. 2.2}$$

where t is time, v is the velocity vector, r is the position vector and a is the acceleration vector. These equations allow us to go from position to acceleration by differentiating twice. This is not particularly useful however, as we wish to go from acceleration to position. Integration allows us to reverse the process. Velocity is obtained by integrating the acceleration; position can be obtained by integrating the velocity (the area under the velocity-time graph).

For integration to be possible, the function of the curve to be evaluated needs to be known. However, we do not have this in an explicit form: we can calculate acceleration at any point in time but do not know it as a function. We can use the fact that we can calculate the acceleration at any point, and the fact that acceleration is the differential of velocity, to integrate the acceleration to obtain velocity and then again to obtain position: this can be done using a numerical method called ordinary differential equations.

2.1.2 Ordinary Differential Equations

Ordinary differential equations can be used to integrate numerically any function, regardless of how complex, given that at any point along the curve of the function it is possible to calculate its derivatives. Baraff [1] states that an initial value Ordinary Differential Equation is of the form;

$$\dot{x} = f(x, t) \qquad \text{(Eq. 2.3)}$$

where f is a known function that can be used, given x and t, to give the differential of x with respect to t. This is known as a differential equation because it contains both x and the derivative of x in the same equation: the ordinary bit refers to the type of the derivative; in this case it contains only ordinary derivatives of the dependant variable (as apposed to partial derivatives).

The way in which ordinary differential equations work can be seen in Figure 2.2. The ODE describes what is known as a vector field, the vectors being the derivatives of the unknown function at a point in time along the x-axis. Depending on the initial value, it means that given an initial value $x(t_0)$, the curve can be traced out from then on using the vector field. A different curve will be swept out for different initial values: consequently this is called an initial value problem.

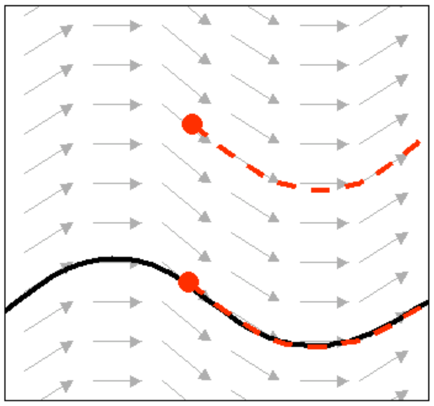


Figure 2.2: Vector Field. The arrows denote the vector field, the solid line represents the unknown curve. The dashed lines show how the curve is traced out given an initial value.

To implement this problem practically, numerical methods have to be used: it is neither practical nor even possible to sample the gradient infinitely along the curve. Whatever the numerical method used, it will always be an approximation of the actual curve and consequently can lead to inaccuracies. There are numerous numerical methods of solving ordinary differential equations: Euler's method is described below.

Euler's Method is the simplest method to implement and describe. Starting with an initial value, $x(t_0)$, Euler's method explains a way to obtain an approximation to the curve by using discrete time steps. From the initial time it takes the initial gradient (derivative) and steps forward using that gradient and step size, s (Eq. 2.4).

$$x(t_0 + s) = x_0 + s\,\dot{x}(t_0) \qquad \text{(Eq. 2.4)}$$

What this does in effect is to trace out a polygon path that approximates the curve. Intuitively Euler's method requires small step sizes to be accurate. It is simple to understand but is neither the most accurate nor the quickest method available.

Baraff [1] details exactly why Euler's method can fail and how to reduce the error by including more derivatives from the Taylor series. He also includes a more accurate ODE called the Midpoint method.

Returning to the particle example, at any point in time the acceleration can be obtained by summing the forces to produce a net force acting on the particle at any one time, then dividing by the mass of the particle (Eq. 2.1). Equation 2.2 shows acceleration to be the derivative of velocity with respect to time. It is therefore possible to use the ODE solution to numerically integrate acceleration to obtain the velocity, as at any point in time the derivative of velocity (acceleration) is known. That just leaves the initial value of the velocity; this is simply the initial velocity i.e. the velocity at time t_0. The same method can be used to go from velocity to position. Therefore, given a particle with mass, position, velocity and acceleration, it is possible by numerical integration using an ODE to obtain new values for velocity and position at future times.

2.2 Modelling the Motion of a Body

A body differs from a particle in the fact that it has volume and shape. Whereas a particle has only three degrees of freedom, a body generally has six. This section will outline the classic rigid-body method of modeling a body.

2.2.1 Classic Rigid-Body Method

By far the most popular way of representing a body is by separating its motion out into linear and angular components. The principle is that a body can not only move through space linearly but can also rotate around a given point. Hecker [2] shows that if we assume a body is made up of a set of particles, then these particles can be summed up to leave an "average" particle at the centre of mass of the body. This "average" particle can represent the linear component of motion and also represent the point around which the angular effects are performed.

For every linear property i.e. position, velocity, acceleration and force, there is an equivalent angular one. Orientation is the angular equivalent of position and is commonly represented as a 3x3 matrix. Given that an orientation in three dimensions has three degrees of freedom (three axes to rotate around), using nine scalars is quite a redundant representation. To represent three degrees of freedom, a 3x3 matrix has six constraints placed upon it to reduce its nine degrees of freedom down to just three. The columns in the matrix are all orthogonal to each other, and the rows are all of unit length. The problem with over specifying the problem is that error is more likely to build up in the coefficients of the matrix; the matrix will drift away from the constraints of a proper rotation matrix and consequently cease being a proper rotation matrix itself.

A better representation of an orientation is that of quaternions [3]. Quarternions use an axis vector and a scalar representing the rotation around that axis to describe an orientation. This is only four numbers representing the same orientation data so error build up will not be as bad.

Angular velocity, usually denoted by ω, is the angular equivalent of velocity. Like orientation it can be denoted as a magnitude around an axis. One important point to note is that each point on an object with angular velocity will be moving at a different velocity. The velocity of any point on a body can be calculated using Equation 2.5.

$$\mathbf{v}_{tot} = \mathbf{v}_{lin} + \omega \times \mathbf{r} \qquad \text{(Eq. 2.5)}$$

Equation 2.5 states that the total velocity of a point on a body is the sum of the linear velocity ($\mathbf{v}_{lin}$) and the angular velocity ($\omega \times \mathbf{r}$) around the body's centre of mass. ω is the angular velocity represented as a quaternion such that it is the vector of the rotation axis with magnitude equal to the angle around that axis. $\mathbf{r}$ is the radius vector, the vector from the centre of mass of the object to the point itself. The $\times$ operator is the cross product, which creates a vector perpendicular to both the radius and axis vectors.

$$\dot{L} = \tau \qquad \text{(Eq. 2.6)}$$

Force in angular terms is called torque. In linear motion, Equation 2.1 shows that force is linked to acceleration by mass. Both Baraff [4] and Hecker [2] detail a solution to link torque and angular velocity through linear momentum. First of all, the link between torque (τ) and angular momentum (L) can be seen in equation 6. It shows that the derivative of angular momentum is torque. It is possible then to show that angular momentum and angular velocity are linked by a 3x3 matrix known as the inertia tensor ($\mathbf{I}$). Baraff describes the inertia tensor as "the scaling factor between angular momentum and angular velocity". What it represents is the equivalent of the mass for linear motion, but more how the mass is distributed so that it affects the moment of the body.

$$\boldsymbol{\omega} = \mathbf{IL} \qquad \text{(Eq. 2.7)}$$

Finally, to get from torque to angular velocity it is necessary to multiply the integral of torque by the inertia tensor (Equation 2.7).

2.3 Collision Detection

Collision detection is generally understood to be the most crucial part of any physics simulator. The reason is simply that the vast majority of processing time is spent in the collision detection phase. The Havok documentation states that "Collision detection typically accounts for over 90% of the CPU time". For real-time physics simulators then, it is vital that a good collision detection strategy is used. This section will outline various collision detection techniques with particular importance being paid to optimisation strategies. To begin with though, the role of the collision detection routine will be defined precisely.

2.3.1 Overview of Role

Two objects at time t = 0 are assumed to be in a valid state; that is, the objects are initially separate. The simulator is to be updated to time t = 1, moving the objects in some way to a new state. The role of the collision detection routine therefore is simply this: given the situation just described, do the two objects collide, and if so, when and where.

2.3.2 Discrete/Continuous

Discrete methods test a collision by first assuming an object is not embedded within the other, basically a static time intersect test. Secondly, the objects are tested to see whether they are embedded at their updated positions. If they are not, it is concluded that there was no collision. The problem with this method is that no account of the motion of the two objects is taken into account, for example an object with high velocity may pass straight through a thin object. Indeed, this actually does happen in the Havok engine, a very popular solution for game physics currently.

Continuous methods take into account the motion of the objects and use the equations of motion to determine a time and potentially a contact set for a collision (if any). The downside to this method is obviously speed, as it is dependant upon the type of motion. Trying to solve motion equations with non-constant angular velocities can be extremely difficult and time consuming. However, the method has the advantage of not missing any collisions and having accurate collision information.

2.3.3 Methods

Continuous methods would always be used if speed wasn't an issue but in collision detection, speed is paramount and so a number of different methods have been created that balance speed and accuracy in different ways. Below are detailed a few popular methods in dealing with the task.

Separating Axis Method

Eberly [5] is one of the main proponents of the separating axis method, both in two and three dimensions, and has a website with exhaustive source code. Baraff [6] has a brief discussion on the method, but uses a separating plane instead of a separating line. The results are the same, but the separating line method is more scalable between 2D and 3D.

The separating axis method is used to determine whether or not two convex objects intersect one another at some instance in time. The method relies on the fact that if two convex objects are separate, then there exists a line such that when the two objects are projected onto it, their intervals do not overlap (Figure 2.3). The lines eligible for being a separating axis are the edge normals of each object and the normal to each edge pair (an edge from one object and an edge from the other object). Eberly [5] discusses the intuition for this method by considering the situation when two objects are just touching: there can either be a vertex-edge contact, a vertex-vertex contact, an edge-edge contact or a face-face contact. Either way, there will exist at least one line that will have the two intervals just touching each other.

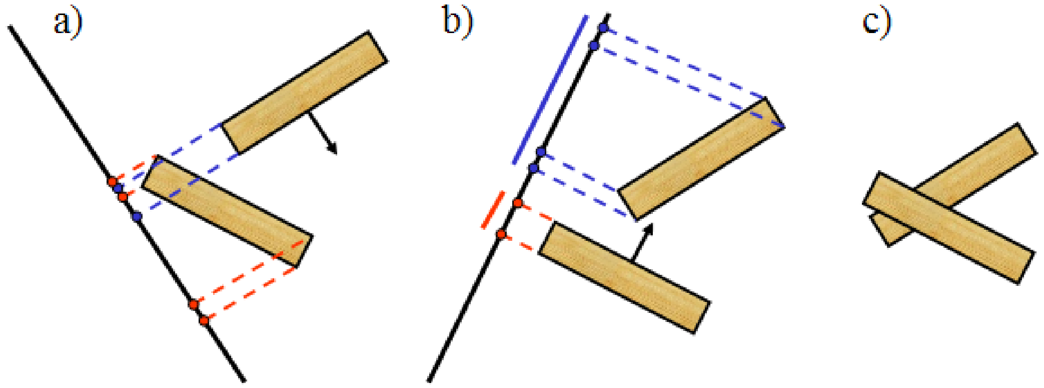


Figure 2.3: Separating Axes. The thick black lines represent a potential separating axis generated from the edge normal represented by an arrow. The dashed lines show the blocks being projected onto the separating axis.

Figure 2.3a shows a potential separating axis for two blocks using the normal represented by an arrow. However, by looking at the intervals projected onto the axis, it is clear that there is no separation. Figure 2.3b shows the same two blocks but with a different separating axis generated from a different edge normal, shown by the arrow. The projected points on the axis are separate demonstrating that the blocks are separate. In general, a line that has separate projected intervals proves for definite that the bodies are separate, but a line that doesn't have separate intervals does not prove the blocks are embedded. Figure 2.3b shows two embedded blocks: there does not exist a line that will have separate projected intervals.

Eberly [5] extends the idea of separating axes to moving objects by describing a method that will return the time of collision, if any. The method assumes that one object is stationary and the other is in motion. This can be done with no loss of generality by combining the velocities of both moving objects, moving one with that velocity and keeping the other one static. In effect, the collision is then viewed in the frame of reference of the static object, but the time until the collision remains the same.

At the time immediately before a collision, there will exist at least one separating axis that will separate the two objects: immediately afterwards, there will be no separating axes. This can be inferred quite simply from the separating axis method. Eberly [5] presents an algorithm that determines the period of time in which there exists no separating axis that can separate the pair of objects, if any, by projecting the intervals along each axis in time using a projected velocity vector: the start of the inseparable interval is the first collision time.

One downside to the method as described in [5] is that it assumes constant velocity and no angular velocity. If velocity is not constant then this could cause problems: more importantly is the effect of a quickly spinning object, which could cause the method to fail altogether.

Bisection

Bisection, as described by Baraff [6], is a collision detection method used to detect the time of a collision (if any) between two moving objects. It is a popular discrete method because it is easy to implement and robust, but relies on small step sizes to work properly.

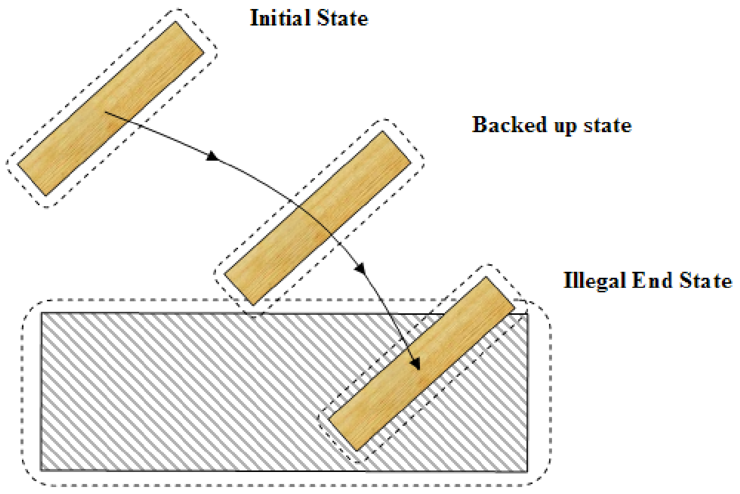


Figure 2.4: Bisection.

The method starts off by assuming that the two objects are initially separate. The new object states are first calculated as if there was no collision. If the objects are embedded at the new state then the simulation is backed up to time t/2. If there is a collision here too then the simulation is backed up again, this time to time t/4; if there wasn't a collision at time t/2 the simulation is moved forward to time t*3/4. The idea is to keep on bisecting time between an unembedded state and an embedded state until the two objects are within some threshold range such that the actual collision occurs somewhere between a time t and t+Δt, such that Δt is very small. The dashed lines in Figure 2.4 represent the threshold range around the objects. When a state is found that has the two blocks initially within the threshold range of each other, and a small time later the objects are embedded, the time can be returned.

Bisection assumes nothing about the motion of an object, and can take into account properties such as angular velocity and variable accelerations and velocities. Its approach is similar to the ordinary differential equations discussed in section 2.3.3: if you can calculate it at any time, then it will work. However, it does lead us to an approximate solution, which can be made more accurate by more iterations of the algorithm.

One point to note about bisection is that it does require quite small time steps to work properly. If a large enough time step is taken, the block in Figure 2.4 may pass straight through the static block without a collision being detected. This is because the method is a discrete one, as discussed in section 2.3.2.

2.3.4 Optimisation

Collision detection becomes very time-consuming when there are many objects in a scene. Collision detection routines are written to test one object against another one, so to test collisions between multiple objects it is necessary to deal with them in pairs. For a naive solution, an algorithm of order $O(n^2)$ is required (based on the combination formula when pairs are picked): for a real-time application it is not really possible, nor sensible, to perform this amount of detailed tests per frame. A few optimisation techniques are described below.

The bounding box method described by Lander [7] introduces a new test that can quickly eliminate pairs of objects that are obviously not colliding. A box is generated around each object such that each object is totally inside the box. A simpler test is then performed between the bounding boxes which is much quicker than an object–object test but, in general, less accurate: like with the separating axis method, if a test returns false (no collision) then the objects are definitely not colliding, but if a test returns true, you cannot infer anything about it. This is because there is some redundancy in the box representation, which represents space that may not have part of the object in it. There are two types of bounding box methods used. Axis aligned bounding boxes are boxes aligned to the axis and can be represented by just a minimum and maximum point; an object oriented bounding box is one aligned with the object it bounds. The axis aligned method is quicker but tends not to fit the objects as tightly under rotation which means more pairs are left in the collision routine when it comes to the slower but more accurate tests. Another popular method is that of bounding spheres, which uses the same principles.

Baraff [6] describes another method to eliminate collision pairs quickly by use of witnesses. A witness is any piece of information that can be utilised to quickly decide whether or not two objects are separate. The separating axis method described in section 2.3.3 uses witnesses in the form of separating axes. On the assumption that from one frame to the next the objects do not move that much, we can assume that in most cases a witness in one frame will also be a witness in the next frame. Therefore, we can cache separating axes over many frames providing us with a quick test for separation.

2.4 Collision Response

Collision response operates on the collision pairs returned by the collision detection routine. It happens at an instance in time (not over some small period of time) when there is at least one contact between two objects. It can broadly be split into two categories:

colliding contact, which occurs when two blocks hit each other, and resting contact which is when objects are in equilibrium with each other.

Let's say that at some exact point in time two blocks are in contact with each other. That they are in contact is not particularly of interest; what is more important is what is going to happen next. If left to continue as they are the two blocks could inter-penetrate, a case that is not allowed by the rigid body constraint. Therefore it would be necessary to intervene in some way to prevent inter-penetration. It may be the case that the blocks are moving away from one another: in this case it is not necessary to stop the simulator. It is also possible that the blocks are in equilibrium and as such are not going to interpenetrate at a future time, but stay in contact. This is known as resting contact. It may seem that nothing should be done in this case too but in fact a great deal has to be done. This is dealt with in section 2.4.2. So, what is required is some piece of information that will tell us, given that we have two objects with a contact point, whether they are moving toward each other, away from each other, or not moving either way. The quantity that tells us this information is the relative velocity.

$$v_{rel}(t) = \mathbf{n} \bullet (\mathbf{v}_b(t) - \mathbf{v}_a(t)) \qquad \text{(Eq. 2.8)}$$

where n is the collision normal, $\mathbf{v}_a$ is the velocity of the rigid body A at the contact point and likewise for B. Equation 2.5 states how to determine the velocity of a point on a rigid body. The relative velocity is the relative velocity of the two objects in the direction of the collision normal. The collision normal is the surface normal in a point-surface collision.

2.4.1 Colliding Contact

A contact is deemed to be a colliding contact if, were the simulator allowed to continue the two objects would become embedded: that is, they are traveling towards each other and because they are in contact cannot travel any further. Immediately after the point in time that the contact takes place, the blocks will be embedded so it is not possible to update the force, as this only affects the velocity over some period of time, not instantaneously as we need. The main method for dealing with this introduces a new term: impulse.

Impulse as described by Hecker [2] and Baraff [6] is a force applied to colliding objects that directly affects the velocities of colliding bodies. This is an implication of the rigid body constraint: a rigid body cannot deform in any way, so any collisions have to happen at a single instance in time, not over some short period of time like a soft body. If only the forces are updated our simulator will integrate the velocity change over time, meaning that at some point the objects are embedded: this would break the rigid body constraint. Hecker [2] explains the idea of impulse as ”a really huge force integrated over a really short period of time”, but in actuality it is calculated at an instance in time.

Impulse is defined as a scalar, j, and represents the magnitude of the collision force. The force itself is applied to both objects in equal and opposite directions in the direction of the collision normal as shown in Equation 2.9.

$$v_a = v_a + \frac{j}{M_a} \qquad v_b = v_b - \frac{j}{M_b} \tag{Eq. 2.9}$$

Equation 2.10 shows how the value of j is calculated for objects with both linear and angular motion. This equation is derived by Hecker [2] using the Law of conservation of momentum. The symbols M and I represent the mass and inertia tensor matrix respectively. Baraff [1] and Hecker [2] both show how the inertia tensor can be derived. The denominator of equation j essentially represents the mass of the two objects acting in the direction of the collision normal.

$$j = \frac{(-1+e)\mathbf{v}_{rel}}{\frac{1}{M_A}+\frac{1}{M_B}+\mathbf{n}\cdot(\mathbf{I}_A^{-1}(\mathbf{r}_A\times\mathbf{n}))\times\mathbf{r}_A+\mathbf{n}\cdot(\mathbf{I}_B^{-1}(\mathbf{r}_B\times\mathbf{n}))\times\mathbf{r}_B} \tag{Eq. 2.10}$$

The symbol e relates the relative velocity before the collision to the relative velocity after it: this relationship can be seen in Equation 2.11. The purpose of e is to simulate the loss of energy that usually takes place on an atomic level in real life. If e is set to 0 the objects will simply stick to each other, indicating that all energy is lost in the direction of the collision normal: if e is set to 1, no energy will be lost and it will simulate a perfectly elastic collision. Values between 0 and 1 will represent various degrees of collision elasticity.

$$\mathbf{v}_{rel\_new} = -e\mathbf{v}_{rel\_old} \tag{Eq. 2.11}$$

2.4.2 Resting Contact

Resting Contact arises when the relative velocity between two objects is 0. In reality, we class resting contact to be within a threshold range of 0 to account for numerical inaccuracies. There are several methods for performing resting contact, each with their own strengths and weaknesses.

Penalty Method

Baraff states in his 1992 thesis:

> *"The penalty method converts a constrained problem to an unconstrained problem where deviation from the constraint is penalized; that is, in the new problem, satisfaction of the constraint is encouraged, but not strictly enforced."*

The penalty method, in reference to bodies in contact, penalises objects for interpenetration but does not strictly enforce the constraint. A force is generated to separate the bodies, the magnitude of which is determined by how deep the penetration is. A penalty force can be thought of as a spring between the two bodies at each contact point. Equation 2.12 is a simple example of a penalty force equation, where the constant k determines how stiff the imaginary spring is: a high value of k will result in a high stiffness of spring.

$$F_{pen} = -k \times penetrationDepth \qquad \text{(Eq. 2.12)}$$

The penalty method is the easiest method to implement and as a consequence is very popular. Its main advantage, as Baraff states, is that it turns a constrained problem into an unconstrained one; that is, one that does not require the integrator to stop as only forces are added to the system, which can be integrated in. However, it does suffer from problems with very stiff springs that require very small time steps to integrate properly. There is also doubt about when the penalty force should break contact.

Impulse-Based Method

Mirtich [8] tries to extend the idea of impulse from just colliding contact to rolling, sliding and resting contact. For resting contact, the concept of micro-collisions is introduced in an attempt to emulate the correct macroscopic behaviour. The method uses a threshold value to determine if the relative velocity at a contact point is deemed small enough to be a resting contact: if it is then an impulse may need to be calculated to reverse the initial relative velocity.

Mirtich [9] concedes that a hybrid method is probably the best solution for a physics simulator. Currently there is no one method for collision response that is the best representation for all types of simulation. For example, impulse is the quickest way of calculating colliding contact cases, but doesn't simulate a constraint such as a pivot joint very well. Conversely, constraint based systems don't model situations that change frequently very well.

Constraint-Based Method

A constraint is simply used to describe the enforcement of a condition. For example, a cog can be constrained to only be able to rotate around its centre. To know how to use a constraint method for contact forces, first we must know what constraint to use, then it is necessary to provide a way to enforce it.

Baraff [6] provides a constraint-based method for computing contact forces. It starts off by assuming that all contact pairs in the list are resting contacts, i.e. have a relative velocity of 0 (within a threshold value). Baraff then states the constraints for resting contact.

$$\ddot{d}_i(t_0) \geq 0 \qquad \text{(Eq. 2.13)}$$

Equation 2.13 shows the first constraint, where d is the separation distance at a contact point, and $\ddot{d}$ is the acceleration of the objects away from each other. The constraint therefore states that the acceleration between the objects at a contact point has to be 0 or positive: a negative acceleration represents an acceleration towards each other, which is not allowed.

$$f_i \geq 0 \qquad \text{(Eq. 2.14)}$$

f_i represents the force that will be generated and added to the objects at the ith contact point to maintain resting contact. The constraint in Equation 2.14 constrains the contact forces generated to be greater or equal to 0. This means that the generated force cannot be one that attracts the bodies together.

$$f_i \ddot{d}_i(t_0) = 0 \qquad \text{(Eq. 2.15)}$$

The third constraint shown in Equation 2.15 states that either the contact force calculated or the acceleration between objects must be 0. If the objects are separating, ($\ddot{d}_i(t_0)$>0), f_i must be 0. This makes sense because f cannot be an attractive force, so if the objects are already separating, f must be 0. Conversely, given the previous constraints, if the objects are not moving apart, $\ddot{d}_i(t_0) = 0$, satisfying Equation 2.15.

Given these constraints, an algorithm is needed that will find the contact forces for each contact point. The derivations of why the following is the case are lengthy and complex, but Baraff [6] goes through it in his appendices. It shows that the contact point forces to be generated are related to the contact point accelerations by Equation 2.16:

$$\mathbf{a} = \mathbf{A}\mathbf{f} + \mathbf{b} \qquad \text{(Eq. 2.16)}$$

where **a** is a column vector of the contact point acceleration magnitudes, **f** is a column vector of the contact point forces we wish to calculate, **A** is a square matrix representing the masses and contact geometries of the objects, and **b** is a column vector representing the forces in the system. Exactly how **A** and **b** are calculated is a lengthy process but is described in detail in Baraff [6]. **A** and **b** are constant and known. What is required is an algorithm that will find values for each element in **a** and **f** that will satisfy the constraints in Equations 2.14 to 2.16. One way is to express it as a Linear Complimentary Problem (LCP): Baraff however turns it into a Quadratic problem (QP). His 94 paper [10] details a complete algorithm to solve such a QP using Dantzig's algorithm. The forces in **f** are then applied to the respective contact points to maintain resting contact.

2.4.3 Friction

The most widely used model for friction is the Coulomb model, developed in the 18th century by Charles Coulomb (1736-1806). The Coulomb model is an approximation to real friction, but models it well for most common applications. The Coulomb model splits friction into two categories: static and dynamic. Coulomb observed that there is a certain amount of force that must be applied to an object before it will begin to move. Before this force is reached, friction will keep the object in equilibrium by equaling the force put on the object: this is known as static friction. Dynamic friction arises when the force has exceeded the maximum static friction force and the object begins to move.

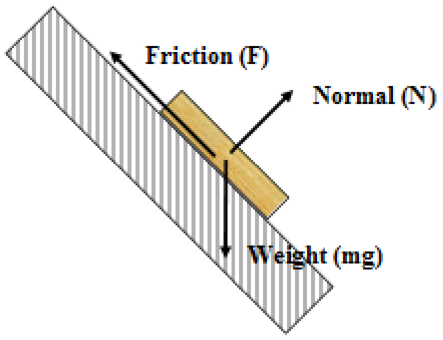


Figure 2.5: Friction on a slope. Friction is a force in the opposite direction to the force parallel to the surface.

$$|\mathbf{F}_{s\max}| = \mu_s |\mathbf{N}| \tag{Eq. 2.17}$$

Equation 2.17 shows that the maximum static friction force is related to the normal force on an object by the constant μ_s. μ_s is called the coefficient of static friction and represents the roughness of the contact between the two objects: the higher the number the rougher the contact surfaces are. When the force parallel to the surface is below or equal to $|\mathbf{F}_{s\max}|$, the friction force generated is equal and in the opposite direction causing the object to remain in equilibrium. Any force parallel to the contact surface greater than $|\mathbf{F}_{s\max}|$ will cause the object to slip: when this happens, dynamic friction takes over.

$$|\mathbf{F}| = \mu_k |\mathbf{N}| \tag{Eq. 2.18}$$

Equation 2.18 represents the equation for dynamic friction: the friction generated is related to the surface normal force by the constant μ_k, where μ_k is defined as the coefficient of kinetic friction. μ_k is generally less than μ_s, but not always. Whenever objects in contact are moving relative to one another, dynamic friction is involved.

2.5 Summary

The topic of physics simulation is a huge one; collision detection, for example, is a research area on its own. Currently there is no one standard way of modelling objects in a physically correct manner. The reason for this can be attributed largely to the trade off between speed and accuracy, with different methods yielding different balances between the two. Tailoring that balance to the specific needs of a particular physics simulator is key to its success.

Chapter 3: Analysis and Requirements

This chapter will analyse the game of Jenga™, specifically that which is needed of a physics simulation to simulate the game. In physics terms, Jenga™ is the problem of simulating a stable stack of blocks and this is the approach to which the physical analysis will be conducted. However, the game of Jenga™ also has specific needs in terms of gameplay, and it is this aspect that will be discussed first. The findings of the analysis will be summarised in the requirements section. The chapter will end with a discussion of an evaluation strategy for the project.

3.1 Project Outline

The aim of this project is to create a real-time physics simulator for the game of Jenga™. The project will focus mainly on simulating the actual tower in a physically believable manner. Being a real-time project, accuracy may not always be achievable, but believability has to be because the player must not feel cheated by the game at any point. Interaction with the tower is therefore also important. Jenga™ is a game of skill and as such the player should be able to play the game in such a way that the skill of judgement from the real game can be used in the simulated one.

3.2 Analysis of the game of Jenga™

This section will look at what the game of Jenga™ is, what you need to play it and how to play it.

3.2.1 Overview

Jenga™ is a stacking game consisting of 52 wooden blocks. The blocks are stacked up in a tower configuration in rows of 3, 18 rows in total. Players take it in turns to remove a block from the tower and place it back on top of the tower. The player to make the tower collapse is the looser.

3.2.2 Physical Analysis

As mentioned, the game of Jenga™ is made up solely of 52 wooden blocks. These wooden blocks therefore make up a large part of what it is that makes the game of Jenga™. The following table (Table 3.1) is based on an analysis of 10 blocks chosen at random from a Jenga™ tower.

Block	Mass (g)	Width (cm)	Depth (cm)	Height (cm)
1	17	8.1	2.7	1.8
2	23	8.1	2.7	1.9
3	19	8.1	2.6	1.8
4	19	8.1	2.7	1.8
5	20	8.1	2.7	1.8
6	21	8.1	2.6	1.8
7	21	8.1	2.6	1.8
8	21	8.1	2.7	1.8
9	18	8.1	2.7	1.8
10	17	8.1	2.7	1.9
Average	19.6	8.1	2.6	1.82
Standard Deviation	1.854723699	0	0.050452498	0.04

Table 3.1: Measurements taken from a random sample of 10 Jenga™ blocks

Table 3.1 shows that the blocks in a game of Jenga™ actually do vary slightly in size and weight, which can be attributed to inconsistencies in the manufacturing process. This fact should be taken into account as it may have an effect on the gameplay.

3.2.3 Gameplay

All aspects of gameplay involved in Jenga™ should be modelled. The real game requires skill based on the physical implications of not knocking the tower over. In the real game of Jenga™ there are two things a user can do to the tower: remove a block, and place a block back on top.

- **Removing a block:** When a player removes a block from the tower they try to disturb the tower as little as possible. One way of doing this is to choose a block that is loose in the first place such that pulling it out will have little effect on the rest of the blocks. If the chosen block is not loose, then skill is required to not pull other blocks out with the selected one. The player has control over the speed and direction the block is pulled.

- **Placing a block on top of the tower:** The player will be careful to place the block so that the tower is disturbed as little as possible. It is also necessary to place the block such that the tower is still stable when the block is released.

3.2.4 Analysis of Skill

From a player's point of view, the element of skill lies in the choice of block to remove, the way in which it is removed and subsequently placed back on top of the tower. When removing a block from the tower the aim is to minimise the disturbance to the other blocks in the tower as much as possible. The decision as to which block is to be removed is based on how much it is perceived to affect the tower when removed.

The reason the tower is disturbed when a block is removed from it is due primarily to the friction forces generated by the block being removed. The greater the friction force generated, the greater the effect on the tower. It was discussed in section 2.4.3 that the magnitude of the friction force generated by an object is directly proportional to the normal force exerted upon it. In Jenga™, all normal force is attributed to the weight of the blocks under gravity. The distribution of weight throughout the tower has a direct affect on how it responds when a block is removed from it.

The explanation of weight distribution explains why some blocks are loose, not affecting the tower much when removed. Section 3.2.2 showed that the blocks in a Jenga™ tower are actually slightly different sizes due to imperfections in the manufacturing process. This means that in a row of 3 blocks in the tower, a slightly smaller block will be loose because the upper rows are supported on the slightly larger blocks (Figure 3.1). The smaller block will not have the weight of all the rows above on it and so will create less friction force when removed. As a consequence it can be pulled out relatively easily, minimising disturbance to the tower (Fig 3.1). This is what the player expects to happen in the real game and so should be a part of the simulation.

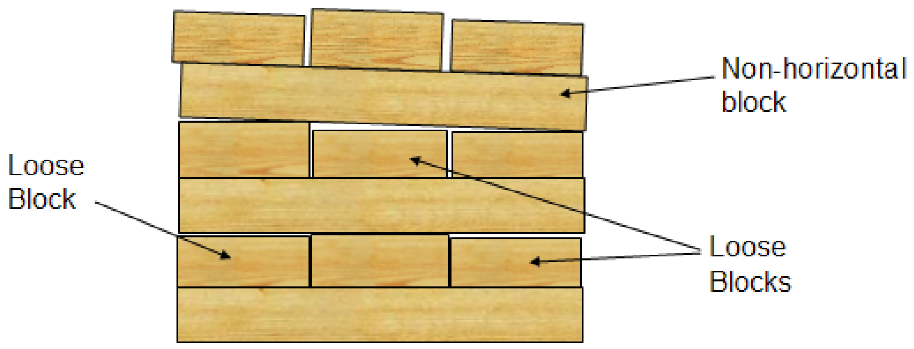


Figure 3.1: A side view of a 3D Jenga™ tower

Another factor that complicates the issue of weight distribution throughout the tower is taking into consideration that the blocks are not all horizontal. Figure 3.1 shows that if a taller block is on the edge of a row, the subsequent rows above will be angled slightly. If many rows have this situation, the weight distribution throughout the tower will be very uneven.

It is crucial, not just desirable, that block size vary slightly, and randomly, in the simulation of Jenga™. Friction must be implemented to model the disturbance to the tower when a block is removed and the simulator must be able to cope with non-horizontal blocks. Also, it has to be taken into account that when a block is removed from the tower the weight distribution is changed, potentially to a state in which the tower is not stable, and thus will collapse.

3.2.5 Player Interaction

The player needs to be able to pull or push a block out of the tower, move it around, and drop it back onto the top of the tower in any orientation they like. In the real game of Jenga™, the speed with which a block is pulled out of the tower is important to whether or not the tower will collapse or not, likewise for putting a block back on top of the tower. Analog motion is preferable as this better represents hand motion than a digital system. It also means that there is more of an element of skill. A mouse would be a good device to represent the movement of the hand in the real game. A mouse can only move in two dimensions so perhaps keyboard buttons could be used as control modifiers. The magnitude of the force can be specified, perhaps, by the speed of the mouse movement.

Moving a block in the simulator can simply be encoded as an external force that just affects the selected block.

3.3 Physics-based Analysis

In terms of physics simulation, Jenga™ is the problem of creating a stable stack of cuboids. This section will discuss current solutions to modelling stable stacks, and then proceed onto discussing the technical aspects of the specific problem of stable stacks.

3.3.1 Existing Solutions

There is already software readily available, either commercially or via the Internet for free, that will simulate physics with varying degrees of accuracy. A stack is a specialised problem of a physics simulator and so any physics simulator should be able to model stacks, but the level to which they do so will vary greatly.

An article published in Game Developer Magazine in September 2000, tested three commercial physics simulators with various types of tests. One test was to see how well they could each model stacks. The magazine had this to say:

> **MathEngine:** When more then 10 to 12 boxes were dropped, the system crashed. Otherwise the boxes bounced badly, even adding energy. The system was very unstable, though no boxes appeared to interpenetrate.
>
> **Ipion:** Stable, but really started slowing as boxes were added to the system. Some boxes eventually fell through each other and the boxes at the bottom jiggled a bit. Parts of the stack slept while others jittered and interpenetrated.
>
> **Havok:** Perfectly stable for a while, but with the maximum stack, the boxes started falling through each other and the floor.

Jeff Lander and Chris Hecker - Product Review of Physics Engines, Part One: The Stress Tests (Sept 2000)

It is important to note that Ipion was bought by Havok, and that developments since then will inevitably have improved the simulations. However, what the tests do demonstrate is that modeling stacks is not a trivial problem.

One game that tried to implement stacks of blocks was a game called Trespasser: the game was not a success. In an article also published in Game Developer Magazine in 1999, a designer on the game wrote about why it went wrong. In some of his comments he described how they had attempted to have stacking puzzles in the game but had had to give up on them.

> *"The friction problem was the main reason we gave up on stacking puzzles - a box on top of another box would more often than not start to vibrate until it had slid off one side or another, and multiple-box stacks were nearly impossible to keep together. It is this very friction problem which has steered many other physics coders away from the penalty force method. The effects of static and dynamic friction are so important to realistic behavior that their absence will compromise the simulation."*
>
> *Richard Wyckoff - Postmortem: DreamWorks Interactive's Trespasser*

The game used the penalty method, which requires friction to be added in a very ad hoc way. With numerical inaccuracies, any flaw in the method is revealed and will cause the stack to collapse, or the program to crash. Friction then is crucial to the stacking problem.

3.3.2 Collision Detection and Response

The collision phase of the simulator will be the crucial one in terms of speed. The following is a list of properties that are specific to the problem of Jenga™.

- Low velocity collisions.
- High number of contacts indirectly connected to each other.
- High percentage of resting contacts.

The high number of resting contacts and the fact that they are all interconnecting makes the problem of a stack a difficult one in terms of speed to resolve.

A cuboid is a convex object; therefore all objects in the Jenga™ simulation will be convex. This fact makes it possible to use the separating axis method for collision detection. This can be extended to the idea of witnesses by caching axes over frames. Eberly [5] also detailed a method to extend the idea of separating axes to moving objects. However, the method uses constant velocity to predict the time of a collision. The Jenga™ simulator cannot be constrained to constant velocities, as whenever a force is applied the velocity will be, by definition, variable.

Object orientated bounding box collision detection is a perfect solution to the problem and so does not provide a quicker test that the full one. Axis aligned collision detection is quicker but will remove less collision pairs.

3.3.3 Speed/Stability trade-off

In general, speed can be traded for accuracy in a simulation and visa versa. In the case of a tower, reduced accuracy results in jitter and instability. This can be attributed to the large number of contacts specific to a tower configuration. Small errors calculated in blocks in contact with one another are propagated throughout the tower: if all the contacts are calculated just slightly wrong, the tower will become very unstable and begin to jitter, possibly even to collapse. This is unacceptable behavior for the simulation and so there is a limit on how much accuracy can be traded for speed. This means that optimisations will be a crucial part of the project, because they not only give more speed, but more potential accuracy, and therefore stability.

3.3.4 Optimisation

Optimisation is an important factor in the project due to its real-time nature. A well-optimised program will be able to handle more blocks but also difficult situations that may arise much better than an unoptimised one. By examining the specific problem of Jenga™, it may be possible to optimise it to quite a high degree: inevitably though this will lead to a loss of generality.

By far the slowest part of the simulator will be coping with large numbers of collision pairs for collision detection and collision response. For example, a full Jenga™ tower with 52 blocks will require:

$$\frac{n!}{r(n-r)!} = \frac{52!}{2(52-2)!} = \frac{52\times 51}{2} = 1326 \text{ collision pairs.}$$

Any method that can reduce the number of collision pairs will significantly increase the speed of the game.

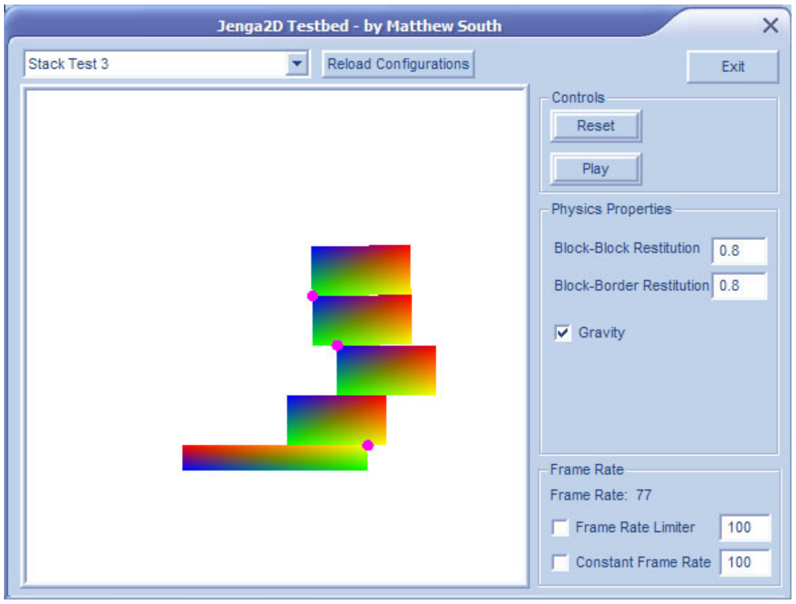
Dealing with collisions in pairs means that the simulator is ignoring all other objects at that time. Resolving a collision pair invariably results in having to resolve another collision pair that has been developed due to the first pair being resolved. The amount of iterations the blocks go through before all are resolved should be looked at as an important part of the optimisation process. It would be useful to measure the amount of iterations processed till resolution of a standard tower and try to reduce the number in various ways. One possible method would be to order the collision pairs in some way that would reduce the amount of new collision pairs generated. Possible ways would be to order collision pairs to be dealt with by order they appear in the tower, either bottom to top, or top to bottom. These should be compared to the naïve algorithm for speed.

A method used by the Havok engine to reduce the number of collision pairs is to simply "turn off" objects that have been stationary for a long period of time. Turning off, or deactivating an object, basically removes the object from the collision routine and effectively makes it into a static object. The object is "turned on" again if something interacts with it. One beneficial side effect of this is that it reduces jitter in the tower: when blocks are turned off they are stationary and therefore do not jitter at all.

3.4 Prototypes

Two prototypes were created to see how the main methods would handle the situation of a stack of blocks. The prototypes were implemented in two-dimensions, but the principles used apply to three-dimensions. The programs were written in C++ for speed and OpenGL was used for the graphics.

3.4.1 Rigid Body



Block-Block Restitution 0.8
Block-Border Restitution 0.8
☑ Gravity
Frame Rate: 77
☐ Frame Rate Limiter 100
☐ Constant Frame Rate 100

Figure 3.2: 2D rigid-body prototype based on Baraff's constraint based method

The rigid body prototype was implemented using Baraff's SIGGRAPH 1997 notes [6] with the linear complimentary algorithm for the resting contact problem based on Baraff's 1994 paper [10].

The simulation was very fast for situations involving colliding blocks. However, it invariably could not deal adequately with the situation of blocks in resting contact. Part of the problem was that the resting contact coefficient matrix could not be guaranteed to be solvable and would occasionally crash the program. The major problem though was that the collision routine could not be time limited and in complicated situations would take a long time to complete. Invariably the collision routine would cause a stack overflow or just not terminate at all. The problem is demonstrated in Figure 3.3.

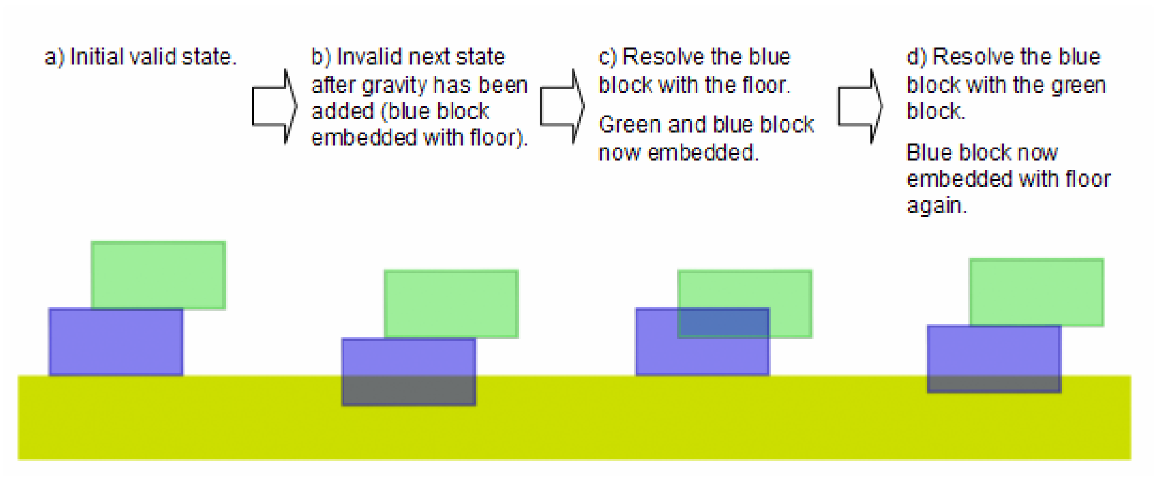

a) Initial valid state.
b) Invalid next state after gravity has been added (blue block embedded with floor).
c) Resolve the blue block with the floor.
Green and blue block now embedded.
d) Resolve the blue block with the green block.
Blue block now embedded with floor again.

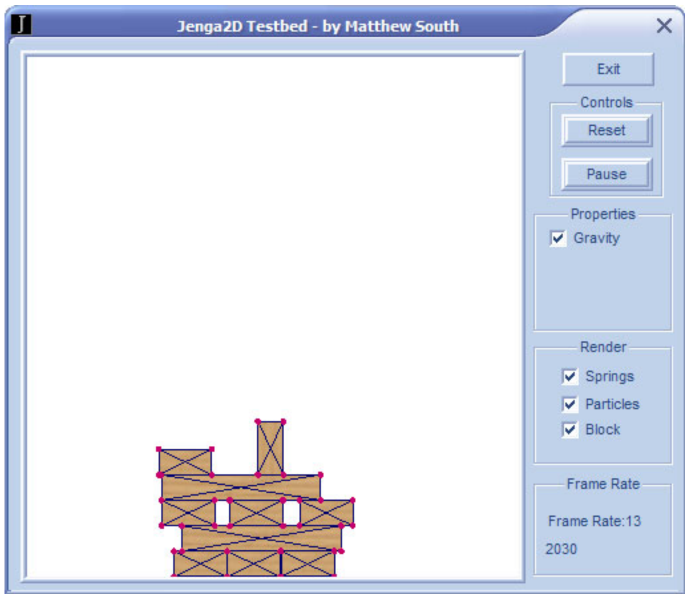
Figure 3.3: Problem with resolving rigid-body collisions

Figure 3.3 demonstrates a problem that occurs when resolving rigid-body collisions in pairs. For the time that two objects are being considered by the collision routine, all other objects are ignored. By resolving one pair, another pair of embedded objects can occur. By resolving the new pair of objects, the previous pair may become embedded again, and so on. As the simulation has to be left in a stable state, that is, one with no embedded objects, the collision routine either has to run its natural course or be prematurely stopped with previous states being restored. The latter was implemented but dropped because it resulted in jerky movement.

The configuration of a stack is the epitome of a configuration that requires a large amount of iterations for completion, and so where the rigid-body method may provide a good solution to a wide range of situations, it does not suit the case of a stack all that well.

3.4.2 Spring Based

When the problem described in Figure 3.3 was discovered it was decided to look at an alternative method to physics simulation. A spring based approach was looked at whereby particles representing the corners of an object are constrained by springs to form an object. The program was written from scratch using basic principles, with Jeff Landers articles [11] on the subject used as a reference. The only articles that could be found on spring based modelling of objects only modelled one object and neglected the interaction between two or more. However, adding collision detection and response to the objects proved to be relatively simple with encouraging results.



[✓] Gravity
[✓] Springs
[✓] Particles
[✓] Block
Frame Rate:13
2030

Figure 3.4: 2D prototype using a spring based method

The spring based method produced instantly improved results over the constraint based method. The first noticeable advantage was the removal of resting contact: this was replaced by very small, high frequency collisions between blocks. The second advantage was the vastly reduced number of iterations needed to resolve all collisions. Objects are still resolved in pairs as in the constraint based approach but because of the spring like nature of the blocks, resolving one collision generally does not result in more collisions. This can be seen in Figure 3.5. Resolving a collision on one side of the block does not knock the other side of the block into another block. In general, it means that there is always a solution to resolving collisions and, crucially, the collisions are always resolved within a couple of iterations of each contact point.

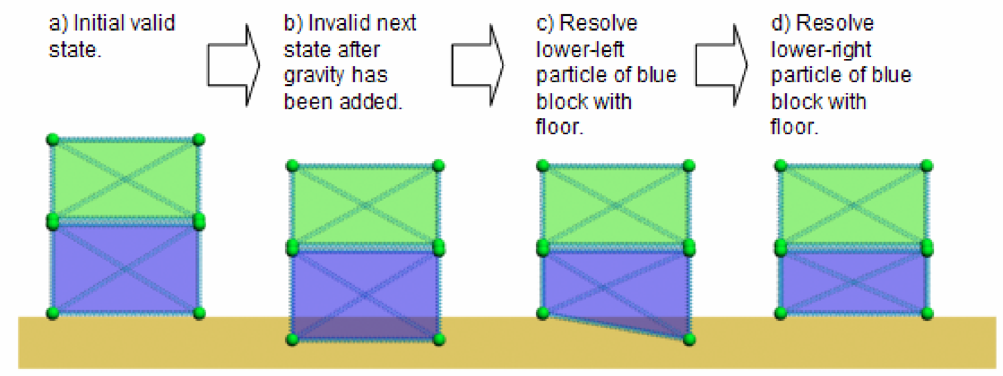

a) Initial valid state.
b) Invalid next state after gravity has been added.
c) Resolve lower-left particle of blue block with floor.
d) Resolve lower-right particle of blue block with floor.

Figure 3.5: Resolving contacts with a spring-based method

Although collisions are resolved much quicker the side effect of deformation of blocks is clearly undesirable. In the prototype pictured in Figure 3.4 however, deformation was barely noticeable for a number of reasons. The springs were made to be quite stiff and 5 iterations were performed each frame to resolve the blocks to normal size. A squashed block has higher tension in the springs and so will force itself outwards more in the next iteration. The main factor as to how much a block will deform is the velocity of the impact: fortunately, the situation of a stack doesn't have any high velocity impacts and so the spring based method worked very well.

The spring based method proved to be far simpler to implement and far more stable than the constraint based method for stacks, largely because it took advantage of the fact that a stack involves no high velocity collisions.

3.5 Project Constraints

A fully functional physics simulator will not be used for the project; instead one will be created to the specific needs of the game.

What forces should be simulated?

Only forces that are essential to the believability of the simulation should be modelled. Gravity is an important part in the game play adding weight to all the blocks, and being the reason the stack can be unstable and collapse. It has been seen that friction is also important in the simulation of Jenga™, both in gameplay aspects and believability. Forces such as air resistance offer negligible effect to the game of Jenga™ and so can be ignored.

How many blocks should the simulator have to simulate?

In the standard game of Jenga™ there are 52 blocks, 18 rows of 3. This may be too many to actually simulate in real time, so the amount of blocks that the simulator will simulate may have to be reduced. 3 rows of 3 blocks is an absolute minimum as a lot of interesting behavior will be lost with such a small amount of blocks. The project should aim at simulating a minimum of 6 rows (one third of the full tower) in real time, but aim at around at least 9 rows (half of the full tower) as being satisfactory. Obviously the aim will be to simulate all 18 rows.

Is it acceptable to model the blocks as perfect cuboids?

In the real game of Jenga™ the blocks are almost perfect cuboids except for their slightly chamfered edges. The difference will be negligible on gameplay, and as the render model is independent of the physical model, this should not matter.

3.5.1 Rendering

Rendering is a trivial issue for the game of Jenga™. Assuming each block consists of 12 triangular polygons, the maximum tower in Jenga™ of 52 blocks will result in a total polygon count of 624 polygons. This is a low amount of polygons to render and so the rendering process should not factor much at all into the speed issue.

3.6 Requirements

So far this chapter has served to analyse the game of Jenga™. The findings from the analysis will now be summarised in the following list of requirements.

1. Gameplay

1.1 A player should be able to remove a block from the tower.
1.2 A player should be able to place a block back on top of the tower.
1.3 Input should be analogue to simulate a player's movement in the real game.
1.4 The camera should allow for full view of all sides of the tower.

2. Physic's based

2.1 Gravity should be modelled.
2.2 Friction should be modelled.
2.3 The tower should stay up when in a stable state.
2.4 The tower should collapse when in an unstable state.
2.5 The tower should behave as 'expected' when a block is removed from it.
2.6 The tower should behave as 'expected' when a block is placed on top of it.
2.7 Blocks should not be able to pass through other blocks.
2.8 Blocks should not be able to become embedded with other blocks.
2.9 The blocks should not jitter.

3. Program

3.1 The program should not crash or freeze.
3.2 The game should run in real-time, no less than 12 frames per second.
3.3 The program should be simple and intuitive to use.

3.7 Evaluation Strategy

The program will be evaluated against how well it meets the list of requirements in the previous section. Some of these requirements are a case of just yes or no, some are more subjective and will need testing.

Requirements 2.3 to 2.6 will be evaluated on how well they behave in comparison to a real Jenga™ tower. A representative sample of configurations will be set up on both the real and simulated Jenga™ towers and the results compared. To test for a wide range of typical configurations experienced in a Jenga™ game, a sequence of moves will be played out on both towers to see how similarly they perform.

Chapter 4: Spring-based Physics Simulation

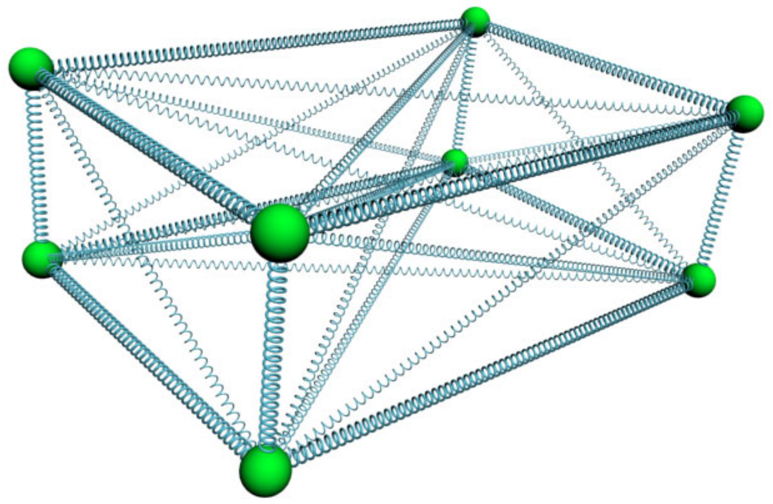


Figure 4.1: A cuboid consisting of particles constrained by springs

The chosen implementation for this project is a spring-based method. The primary reason for this choice is speed and stability when dealing with the high number of resting contacts associated with the game of Jenga™. The analysis section showed the rigid-body methods not to be suitable for the high number of resting contacts that the game of Jenga™ requires. The spring-based method on the other hand proved to work well in the situation of a stack. This chapter will detail the design of a spring-based system for the game of Jenga™. Though methods will broadly be borrowed from existing solutions, the design presented in this chapter will be one tailored specifically for this project.

4.1 Method Overview

The spring-based method has no notion of an object such as the cuboid in Figure 4.1. An implementation has a list of particles and a list of springs, each spring consisting of a reference to two of the particles in the particle list. Springs can just as easily attach particles in a nonsensical way as they can to form objects. Figure 4.1 would be created with eight particles connected by 28 springs of the correct length to be at rest in the shape of a cuboid.

As mentioned earlier, a spring-based method has many desirable features. The following is a short list of its main advantages and disadvantages.

Advantages

- No Angular Motion
- No Resting Contact Required

- Simple to implement

Disadvantages

- Modelling rigid-bodies results in stiff spring equations

The disadvantage listed is quite a major one. It won't be possible to make the springs completely rigid, but as was found with the prototype in section 3.4.2, because there are no high velocity collisions in Jenga™, the deformation of blocks will not be a major factor.

At the heart of the method is the spring. Section 4.1.1 will now describe exactly what is meant by a spring in mathematical terms.

4.1.1 Springs

A spring is a constraint between two particles. The constraint requires that two particles maintain a specified distance apart. The constraint is not strictly enforced, merely penalised if broken.

A spring has a number of properties. The crucial one is its rest length, the distance between its two particles that it is trying to maintain. Another property of a spring is loosely described as its stiffness. The more a spring is penalised for breaking a constraint, the stiffer it is. Equation 4.1 shows how the magnitude of the penalty force can be calculated.

$$F_{mag} = stiffness \times (springLength - restLength) \qquad \text{(Eq. 4.1)}$$

F_{mag} is positive for stretched springs and negative for compressed ones. The sign indicates whether the applied force will pull the particles together or push them apart.

The penalty force is added to each particle equally and in opposite directions along the length of the spring. The equation for the penalty force below (Equation 4.2a) calculates the force exerted on particle $\mathbf{p}_2$ by particle $\mathbf{p}_1$, but the force on $\mathbf{p}_1$ due to $\mathbf{p}_2$ is just the negative of it. The forces are added to the spring's particles as shown in Equation 4.2b.

$$\mathbf{F}_{pen} = F_{mag} \times \frac{(\mathbf{p}_{2\,pos} - \mathbf{p}_{1\,pos})}{springLength} \qquad \text{(Eq 4.2a)}$$

$$\begin{aligned} \mathbf{p}_{2\,force} &= \mathbf{p}_{2\,force} + \mathbf{F}_{pen} \\ \mathbf{p}_{1\,force} &= \mathbf{p}_{1\,force} - \mathbf{F}_{pen} \end{aligned} \qquad \text{(Eq 4.2b)}$$

The final property of a spring that will be implemented is that of damping. Damping apposes the direction of motion and can be thought of as a kind of friction or drag, as if the spring were in oil.

$$\mathbf{F}_{damp} = \frac{-damping \times \left(velDiff \bullet posDiff \right)}{springLength} \qquad \text{(Eq. 4.3a)}$$

where:

$$\begin{aligned} posDiff &= \mathbf{p}_{2\,pos} - \mathbf{p}_{1\,pos} \\ velDiff &= \mathbf{p}_{2\,vel} - \mathbf{p}_{1\,vel} \end{aligned} \qquad \text{(Eq. 4.3b)}$$

The damping force (Equation 4.3a) is added to the particles in exactly the same way as the spring force is.

Using just particles and springs it is possible to model the motion of an object such as the cuboid in Figure 4.1. However, more data structures need to be introduced to consider collisions between two or more such objects.

4.2 Data Structures

There are five different geometric data structures that will be implemented:

- Particle
- Spring
- Edge
- Face
- Block

These will now be described in detail.

4.2.1 Particle

At the heart of the simulator is a list of all the particles being simulated. All other geometric data structures are derived in someway from the particles in the particle list as it is the particle that stores the mass, position, velocity and acceleration data. When the simulator is updated by the integrator, the list of particles is processed as if no other geometric data structures existed and the particles are free moving. For this processing to take place in practice, each particle needs to contain a duplicate set of states so that a previous state can be restored if the new one is invalid. To do this, the motion data will be partitioned into a separate state structure and a particle will contain both a reference to a current state structure and a next one.

Particle	
Type	**Name**
float	mass
State	curr
State	next

Particle::State	
Type	**Name**
vec3D	position
vec3D	velocity
vec3D	force

Figure 4.2: Particle and particle state structure

Figure 4.2 shows the basic data structures necessary for the particle. Interestingly, it is force that is represented in the particle state instead of acceleration. The reason for this is in the way the simulator calculates the forces. The forces acting upon on a particle are built up from a number of sources, such as gravity, spring forces, user forces etc, before the particle ever reaches the integrator. This is necessary because acceleration is related to the net force of an object. It is necessary therefore to have storage capacity for force in a particle for the force to be built up. When integration is performed, acceleration is simply the net force over the particle's mass and so is used purely as a temporary value in calculating the particles velocity.

The separation of the state from the particle comes in useful when more complicated integrators are used such as the midpoint method where numerous temporary states are required.

4.2.2 Spring

A spring is a constraint between two of the particles on the particle list and so the spring structure must reference two of the particles from the particle list in some way. In a typical C style language, pointers can be used. The spring structure must also store the spring properties discussed in section 4.1.1.

Spring	
Type	**Name**
Particle*	p1
Particle*	p2
float	restLength
float	stiffness
float	damping

Figure 4.3: Spring structure

The '*' symbol after the word 'Particle' in Figure 4.3 is the syntax used in the C programming language to indicate a pointer and will be used in the same manner in this report.

Similarly to the particles, the springs will be stored in a global list of all springs in the current scene. When the spring forces are calculated, the entire list of springs will be processed regardless of geometric context such as which block it belongs to. The forces are calculated using the equations in section 4.1.1. The stiffness and damping constants are decided based on the specific needs of the simulator and tailored accordingly.

4.2.3 Edge

Edges are important in the collision detection phase of a simulator. The edges of an object, such as the cuboid in Figure 4.1, are not implicitly defined as they would be in a rigid-body simulator. From Figure 4.4 though, it can be seen that the list of edges is a subset of the list of springs: an edge is a spring but a spring is not necessarily an edge.

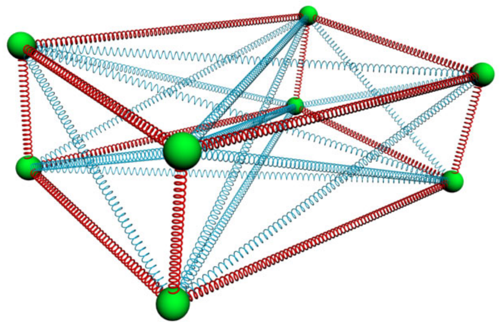


Figure 4.4: Cuboid modelled by springs, red indicating edge springs

As an edge does not have any properties of its own, and as they are simply a subset of the springs, it would be simpler to add an edge flag to the spring structure.

Spring	
Type	**Name**
…	…
bool	edgeFlag

Figure 4.5: Modified spring structure to include edges

The red springs shown in Figure 4.4 are edge springs and their edge flag value will be set to true: all other springs will have an edge value of false.

4.2.4 Face

The notion of a face, essential for collision detection, is not an obvious addition to a spring based simulator. When trying to model the faces of the cuboid depicted in Figure 4.1, a first attempt might be to model them as six rectangular polygons, one for each side. However, unlike a rigid-body, the 4 particles making up a side do not necessarily lie in a plane, indeed are not likely to. This means that triangular faces have to be used, as three points always share a common plane. To do this then, a standard way of deriving how a cuboid should be divided up into faces is needed.

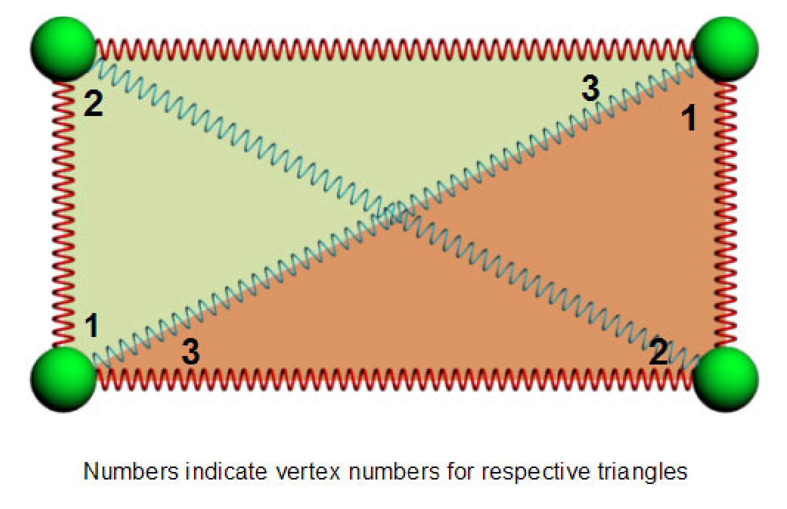
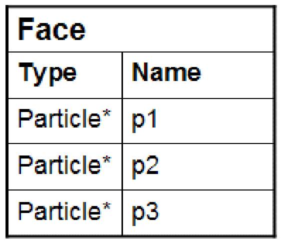


Figure 4.6: Side on view of cuboid containing 2 3-sided polygons

Figure 4.6 shows how the two triangular polygons for a side are to be assigned. Each polygon consists of two edges and one internal spring. The order of the vertices is assigned such that the first two sides of the triangle are edges. The vertices are also ordered in a clockwise manner, important in determining which way an edge facing.

Defining a structure for a face can conceivably be done in 2 different ways: a face can either contain pointers to its 3 particles (corners) or pointers to its 3 springs (sides). It is the former representation that will be used because when they come to be used in the collision detection routine, it is the particles that are used to calculate the face's motion.

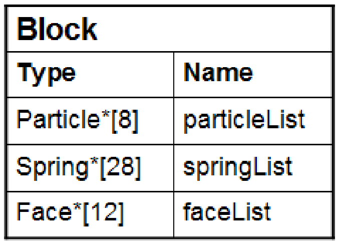


Face	
Type	**Name**
Particle*	p1
Particle*	p2
Particle*	p3

Figure 4.7: Face data structure

4.2.5 Block

The block data structure contains lists of pointers to all the geometric structures that it is built up from (Figure 4.8).



Block	
Type	Name
Particle*[8]	particleList
Spring*[28]	springList
Face*[12]	faceList

Figure 4.8: Block data structure

Figure 4.8 shows that a block consists of a list of pointers to its 8 particles, a list of pointers to its 28 springs and a list of pointers to its 12 faces. The block structures are then stored in a list of blocks, completing the geometric data structures needed for the simulator.

The geometric data will be generated at the block level and will be encapsulated in a "CreateBlock" function. The "CreateBlock" function will take a block's position and dimensions as parameters and will generate the relevant data structures adding them to their respective lists.

4.3 Main Loop Algorithm

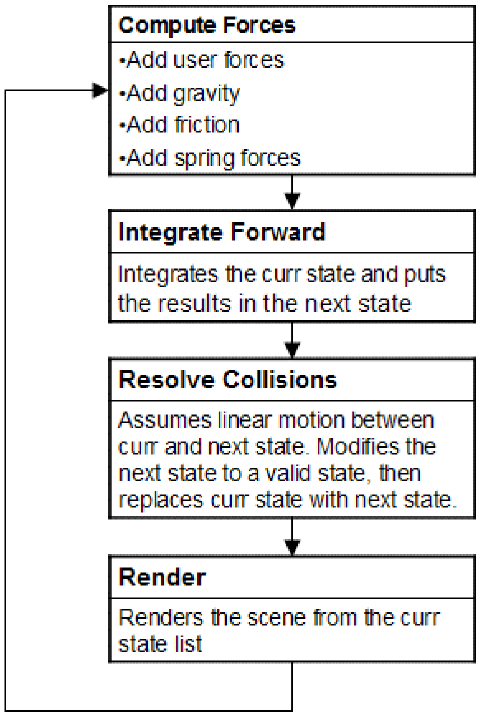

Compute Forces
•Add user forces
•Add gravity
•Add friction
•Add spring forces
Integrate Forward
Integrates the curr state and puts the results in the next state
Resolve Collisions
Assumes linear motion between curr and next state. Modifies the next state to a valid state, then replaces curr state with next state.
Render
Renders the scene from the curr state list

Figure 4.9: Overview of the main loop algorithm

The term "main loop", sometimes referred to as the heartbeat function, is the routine called each frame to update the simulator. Figure 4.9 outlines the general structure of the main loop algorithm. The 4 main processes will now be discussed.

4.3.1 Computing Forces

The first task the simulator performs each frame is to calculate the net force on each particle: this value is then placed in the current state's force vector. Previous forces are zeroed before this is done. Forces are obtained from a number of sources. User forces are external forces placed on the particles by the user: these will be discussed in section 4.8. Gravity simply adds a constant downwards force to each particle. The spring forces are calculated as described in section 4.1.1. Finally, friction is added. Friction relies upon the type of contact it is in with another block to determine its direction and magnitude: it is only practical to use the last frame's contact information to do this. The process will be described in section 4.6.

4.3.2 Integrating Forward

The integrators role is to move the simulation forward in time. The current state is used as the initial state, while the results are put in the next state. Simple one-step Euler integration can be performed as in equation 4.4.

$$\begin{aligned}\text{next.velocity} &= \text{curr.velocity} + \left(\frac{\text{curr.force}}{\text{particle.mass}}\right) \times \text{time} \\ \text{next.position} &= \text{curr.position} + \text{next.velocity} \times \text{time}\end{aligned} \qquad \text{(Eq. 4.4)}$$

This, however, will not result in a stable simulator unless extremely small time steps are taken. More accurate methods cannot be performed in one step and require the use of temporary lists of states. For a temporary state list to be used, the integration function and the "ComputeForces" function have to be parameterised on state lists as shown in Figure 4.10 below.

```
ComputeForces(StateList source, StateList dest)
EulerIntegrate(StateList initial, StateList source, StateList dest, float time)
```

Figure 4.10: Function declarations parameterised on state lists

The one step Euler method would then be written as follows:

$$\begin{aligned}\text{dest.velocity} &= \text{initial.velocity} + \left(\frac{\text{source.force}}{\text{particle.mass}}\right) \times \text{time} \\ \text{dest.position} &= \text{initial.position} + \text{dest.velocity} \times \text{time}\end{aligned} \qquad \text{(Eq. 4.5)}$$

Previously it was assumed that the initial and source states would be the current state and the destination state would be the next state. With more complicated integration methods temporary states have to be used and so the former is not always true. A one-step Euler integration can still be performed as follows.

```
ComputeForces(curr,next)
EulerIntegrate(curr,curr,next,time)
```

Figure 4.11: One step Euler Integration using state lists

However, more accurate methods such as the midpoint method can now be implemented too as shown in Figure 4.12: the midpoint method updates the current state with the forces generated at half the time step to be taken as apposed to the forces generated at the start of the step. The initial state is still the current state and the destination state is still the next state, but a temporary state is used in the process.

```
ComputeForces(curr,temp)
EulerIntegrate(temp,curr,temp,time/2)
ComputeForces(temp,curr)
EulerIntegrate(temp,curr,next,time)
```

Figure 4.12: Midpoint integration method implemented using state lists

4.3.3 Resolving Collisions

By the time the simulator reaches the collision routine, each particle has 2 different states: the current state, which is the last valid state (i.e. no embedded blocks) and the next state which is the new unchecked state. The task of the collision routine is to check the next state for embedded blocks and if found resolve them in a physically believable manner. Once the next state is a valid one the collision routine can exit, the current state is overwritten with the next state and the simulator continues.

Collision detection and response will be discussed in detail in sections 4.4 and 4.5 respectively.

4.3.4 Rendering

The rendering phase is responsible for rendering the blocks in their current positions. Although the rendering model can be different to the physical model, this project will use the faces of the blocks to render them.

4.4 Collision Detection

As mentioned in section 4.3.3, by the time the collision detection routine is reached each particle has two different positions: an initial valid one and an unchecked destination one. The role of the collision detection routine is to detect when and where collisions take place.

The chosen method for collision detection is a continuous method: section 2.3.2 explained that a continuous method is one where the equations of motion are solved directly to determine exact times, and therefore locations, of collisions. Between states it will be assumed that the particles travel in a straight line with constant velocity. In actuality this is generally not the case but because the simulator will be updated at least 12 times a second, the motion each frame will be so close to that of a straight line that the difference noticed by the user will be negligible. Optimisations like this can be performed because the simulator has to be believable, not necessarily accurate.

For three-dimensional collision detection two types of collision need to be considered: point-face and edge-edge. Collisions such as point-point and point-edge are degenerate situations and as such will not be considered separately.

4.4.1 Point-Face Collision

The point refers to a single particle that travels from one position to another in a straight line. The face refers to a face structure as defined in section 4.2.4 but the motion is somewhat more complex. A face consists of 3 particles, each moving in a straight line from one position to another, but in potential different directions. This has the effect that the face can actually rotate and skew; however, nothing can be assumed about its motion.

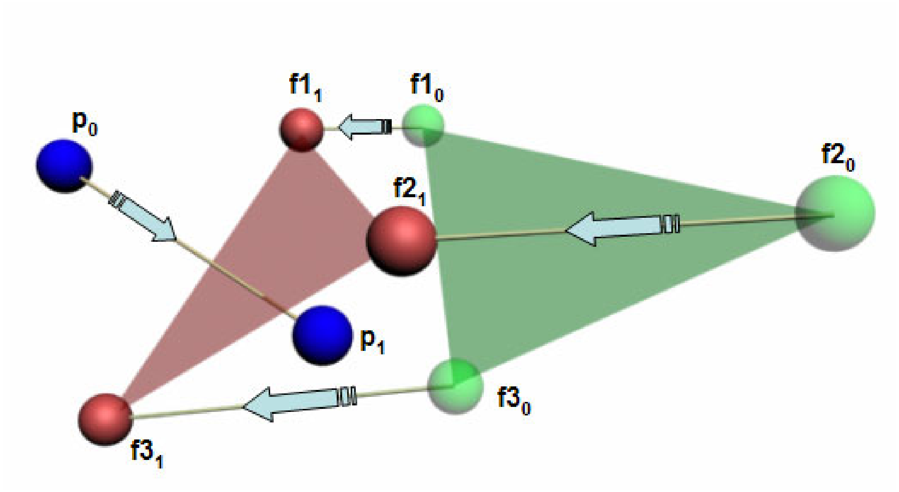


Figure 4.13: Point-Face collision.

At time t=0, each particle (either the single point or the three in the face) is at its start position: at t=1 each particle is at its end state. Between positions each particle moves at a constant velocity so that by t=1 it has reached its end state (effectively velocity = end position – start position). We wish to know if at some time between t=0 and t=1 the particle passes through the face, and if so at what time.

The problem can be divided into two parts: does the particle pass through the plane defined by the three face particles and if so does it pass through the triangle defined by the three face particles.

Determining whether or not the particle passes through the plane defined by the face's particles is relatively simple. All that needs to be done is to check the side the particle is on before and the side it is on afterwards: if these are different then it must have passed through the plane at some time between t=0 and t=1. In actuality, it's even simpler than that because only particles that pass through a face from front to back need to be considered. Defining our faces in a clockwise manner now allows us to determine whether a particle is on the front side or back side of a face. The face's normal will always point outwards in the direction of the front of the face if defined as follows:

$$faceNorm = \left(p2 - p1\right) \times \left(p3 - p1\right) \qquad \text{(Eq. 4.6)}$$

where p1, p2 and p3 are its respective points ($\times$ denotes the cross product).

The following equation will determine the shortest distance to a plane from a point if given a unit normal (if not, the result will be scaled by the length of the normal).

$$\text{distToPlane} = \left(\text{point - pointOnFace}\right) \bullet \text{faceNormal} \qquad \text{(Eq. 4.7)}$$

"distToPlane" will be a positive number if the point is on the front side of the face, negative if behind it. To test if a point crosses a plane we need only to test if initially, at t=0, the point is in front of the face, i.e. "distToPlane" is a positive number, and at time t=1 "distToPlane" is negative.

Once it is known that a particle has passed through the plane defined by the face's 3 particles a more involved process has to be performed to determine whether or not it passes through the triangle defined by the 3 points. Firstly, an exact time of collision has to be calculated so that a static time test can then be performed to determine whether or not the point is within the triangle at the exact moment it crosses the plane.

When a point is on a plane its distance to it is intuitively 0, hence the previously defined "distToPlane" will be 0. This fact can be taken advantage of to calculate an exact time of collision. The only other case that "distToPlane" can be 0 is if the plane normal is of zero length: it will be assumed that this event cannot happen.

Using Figure 4.13 as a reference, if we let p stand for the particle's position, f1, f2, and f3 stand for the respective face particles positions with subscripts indicating time (0 = current state, 1 = next state), then each particles position can be defined with respect to t as follows:

$$p_t = p_0 + (t \times pVel) \qquad \text{(Eq. 4.8a)}$$

where:

$$pVel = p_1 - p_0 \qquad \text{(Eq. 4.8b)}$$

and similarly for f1,f2,and f3.

Plugging in the equations for each particle's position with respect to time into equation 4.7 for the distance of a point to a plane and setting the distance equal to 0 will give us an equation with time (t) as the only unknown.

$$\left((p_0 - f1_0) + t(pVel - f1Vel)\right) \bullet faceNorm = 0 \qquad \text{(Eq. 4.9a)}$$

where:

$$faceNorm = \left(f2_0 - f1_0 + t(f2Vel - f1Vel)\right) \times \left(f3_0 - f1_0 + t(f3Vel - f1Vel)\right) \qquad \text{(Eq. 4.9b)}$$

Taking into account that every variable is a constant except for t, and by using the distributive properties of the dot product and cross product, it is possible to write an equation in terms of t.

$$at^3 + bt^2 + ct + d = 0 \qquad \text{(Eq. 4.10)}$$

The exact values of the constants a, b, c and d are given in appendix 1.1. It can be seen that equation 4.10 is a cubic equation, which means that there is a possibility of up to 3 solutions. The solution that will be chosen will be the smallest solution between 0 and 1 inclusively.

Plugging t into equation 4.8 for each particle will give the positions of each particle at the time of collision, the point's position being the exact intersect point. The final step in determining a point-face collision is now to check whether or not the point is within the triangle defined by the face's 3 particles. To do this we can take advantage of equation 4.7 again for the distance from a point to a plane. The cross product of an edge vector with the face normal will produce an outwards facing normal. If the distance to this plane is positive, the point must be outside the bounds of the triangle. Therefore, if the distance to each plane, created by each edge of the face, from the intersect point is negative, the particle did indeed pass through the triangle and a collision is flagged.

4.4.2 Edge-Edge Collisions

Edge-edge collisions can be handled in much the same way as point-face collisions. An edge is a line segment joining two particles that can be travelling in different directions at different speeds. Unlike the point-face test, 2 lines do not have a front and back and so the initial test used in the point-face test cannot be used. However, the distance from a point to a plane equation can be used yet again.

When two lines cross each other in three-dimensions they both lie in a common plane with normal perpendicular to both lines. Another way this can be expressed is by saying that the distance between the two lines along the direction of their common normal is zero. The only time two lines lie in the same plane but do not cross is when they are parallel: the parallel case should be dismissed by checking whether the cross product of the two lines is zero (or suitably close to it).

Let p1 and p2 be points on one line and p3 and p4 be points on the other line. Then, when two lines share the same plane:

$$\left((p1_0 - p3_0) + t(p1Vel - p3Vel)\right) \bullet norm = 0 \qquad \text{(Eq. 4.11a)}$$

where:

$$norm = \left(p2_0 - p1_0 + t(p2Vel - p1Vel)\right) \times \left(p4_0 - p3_0 + t(p4Vel - p3Vel)\right) \qquad \text{(Eq. 4.11b)}$$

This is similar in structure to Equation 4.9a for the point-face collision and does in fact produce a cubic equation in t, albeit with different constants (see appendix 1.2). The value of t picked will again be the lowest value between 0 and 1 inclusively.

Now the two lines are considered at the point in time that they intersect. The lines intersect but the line segments, that is, the part on each line between the two points, do not necessarily cross.

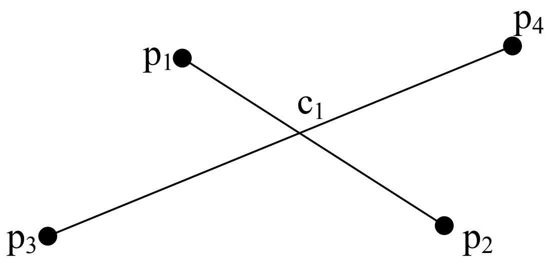


Figure 4.14: Two lines intersecting at c_1

$$\begin{aligned} c_1 &= p_1 + \mu_1 (p_2 - p_1) \\ c_1 &= p_3 + \mu_2 (p_4 - p_3) \end{aligned} \qquad \text{(Eq. 4.12)}$$

The line segments intersect if, and only if, μ_1 and μ_2 are both between 0 and 1. A standard line intersecting test can be used to determine these constants.

4.4.3 Optimisation Techniques

The bottleneck of the simulator will be solely due to the large amount of potential collisions that have to be tested for each frame. With no optimisation performed at all the total amount of collisions to perform each frame is as follows:

$$\text{number of point-face collisions:} \quad (8\times12)\times\left(\frac{n!}{(n-2)!}\right)$$

$$\text{number of edge-edge collisions:} \quad (12\times12)\times\left(\frac{n!}{2(n-2)!}\right)$$

$$\therefore \text{total number of collisions} = 168n(n-1)$$

where n is the number of blocks to be simulated. This means that a full Jenga™ tower of 52 blocks will require 445536 collision tests per frame (possibly even more if a collision resolves into another one) if the collision routine is not optimised in some way. Considering each collision requires at least a cubic equation to be solved, the result of performing this number of tests each frame would clearly not be a real-time solution.

The optimisation technique chosen is that of axis aligned bounding boxes. The axis aligned bounding box test will be performed before the more accurate tests detailed in sections 4.4.1 and 4.4.2. The extra test will be performed not only between blocks but between each particle-face and edge-edge test. The bounding boxes will bound both the current and next states in one, thus covering all motion in-between states.

4.5 Collision Response

It has already been discussed that the spring-based method does not require resting contact to be performed; only colliding contact. The colliding contact that will be performed will be the impulse method as described in detail in section 2.4.1. The equation for impulse with only linear motion is defined by Hecker [2] and is shown in Equation 4.13. This is then applied to the contact point of each object in equal and opposite directions along the collision normal in the same way that section 2.4.1 describes.

$$j = \frac{-(1-e)\mathbf{v}_{rel}}{\mathbf{n}\bullet\mathbf{n}\left(\frac{1}{M_a}+\frac{1}{M_b}\right)} \qquad \text{(Eq 4.13)}$$

The impulse method requires the relative velocity of the two colliding objects to be known at the point of collision. With a point-face collision, the exact velocity of the point of contact on the face has to be derived from its 3 particles, as these will generally be traveling at different velocities. This can be achieved by using a linear weighting of the three particles as follows:

$$faceContactVel = \left(w_1 \times p1Vel\right)+\left(w_2 \times p2Vel\right)+\left(w_3 \times p3Vel\right) \qquad \text{(Eq 4.14a)}$$

where:

$$w_1 + w_2 + w_3 = 1 \qquad \text{(Eq 4.14b)}$$

The values of w_1, w_2 and w_3 can be worked out from the contact point and face data as shown in Figure 4.15 below.

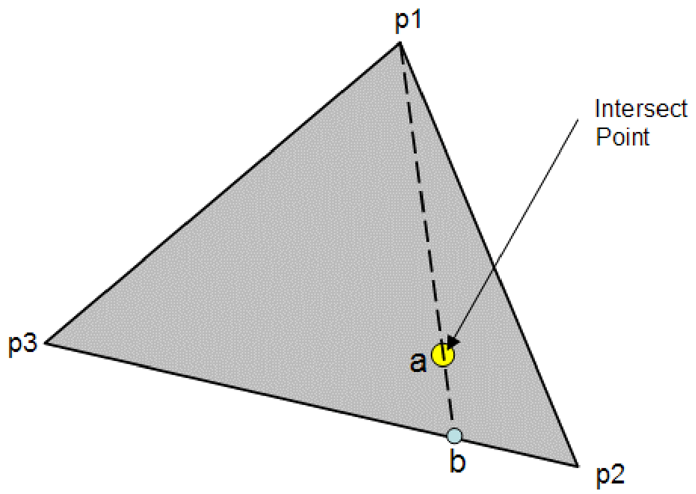


Figure 4.15: Calculating the vertex weights from a point in a triangle

The dashed line in Figure 4.15 is constructed from p1 to the contact point. The point b is produced from the projection of this line onto the opposite side of the triangle, the line segment from p2 to p3. Two values are then calculated, μ_a and μ_b, which represent the percentages the points **a** and **b** are along their respective lines. They are defined as follows:

$$\mu_a = \frac{|a - p1|}{|b - p1|} \qquad \mu_b = \frac{|b - p3|}{|p2 - p3|} \qquad \text{(Eq. 4.15)}$$

w_1, w_2 and w_3 can now be calculated accordingly using μ_a and μ_b by linearly interpolating down the p2/p3 edge, then again down the line segment from p1 to b.

$$\begin{aligned} w_1 &= 1 - \mu_a \\ w_2 &= (1 - \mu_b) \times \mu_a \\ w_3 &= \mu_a \times \mu_b \end{aligned} \qquad \text{(Eq. 4.16)}$$

The three weighting values are used again when the change in velocity is applied at the point on the face and distributed accordingly around the three particles. The equations are listed in section 2.4.1.

When creating the prototypes described in section 3.4 it was discovered that modifying the velocities alone during a collision is not sufficient. The positions have to be modified in some way because just restoring their old states results in jerky motion. The position, therefore, will be updated immediately by the newly calculated velocities from the impulse calculations.

Edge-edge collisions are to be dealt with in exactly the same manner as point-face collisions with weighting values uses to distribute the collision response appropriately amongst the particles involved.

4.6 Friction

Friction forces are added to the simulator at the start of each frame along with all other forces acting on a particle. For this to happen, a list of contacts from the previous frame will be used. The contact information needs to contain the two objects that collided (point-face, edge-edge, point-floor), the force at which they collided and the direction of the collision normal.

Only dynamic friction will be implemented in this project. The friction force applied to each contact should be applied equally and in opposite directions for each contact point. The direction of the force to be applied is in the relative velocity direction in the plane perpendicular to the collision normal. The magnitude of the force is dependant on the normal force as shown in Equation 2.18 in section 2.4.3.

4.7 Disabling Blocks

When a block has not moved much for a length of time it should be removed from the simulation, effectively disabling it. This will have a two-fold benefit: disabled blocks can be removed from the collision routine thus reducing the amount of potential collisions, and jitter will be reduced because a disabled block is a still one. A block should be reactivated when it is involved in a big enough collision, that is, an impulse force generated from a collision involving the block is above a certain threshold value.

4.8 User Manipulation

User manipulation of blocks will be performed via the mouse. The left mouse button will drag and drop blocks around much like an icon can be in a WIMP GUI. The motion of the block will take place in a plane coplanar to the screen formed by the x and y axes of the camera. The right mouse button will be used for pulling blocks out of the tower: motion will take place in the camera's z-axis direction, directly into, or out of, the screen.

4.8.1 Selection

Selection of a block will be achieved by projecting a ray from two-dimensional screen space into three-dimensional world space. A ray-face intersect test will be performed on each face in the simulator: the block that the first face intersected belongs to will be the one selected.

Chapter 5: Implementation

This chapter will discuss the implementation of a simulator for Jenga™ based on the design from the previous chapter. Figure 5.1 shows a screenshot from the final program.

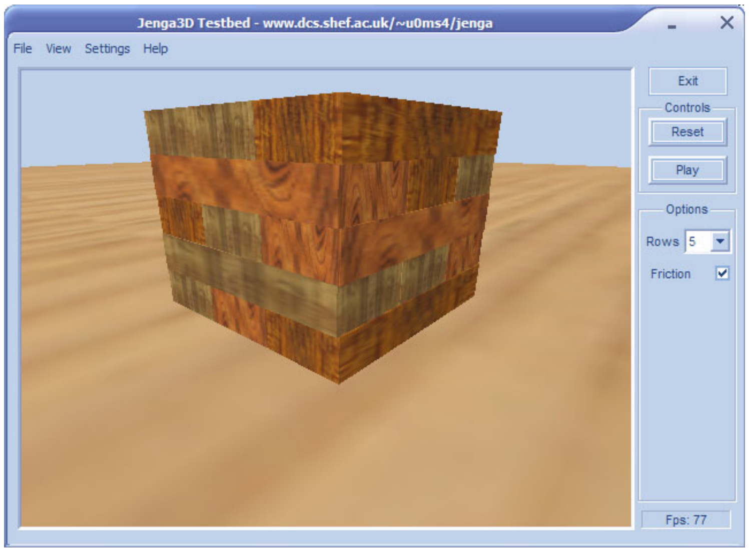


Figure 5.1: Screenshot of Jenga3D

5.1 Chosen Languages

The language chosen for this project is C++. The C programming language compiles faster code compared to languages such as Java and speed is critical to this project. C++ is an object oriented language that lends itself well to the type of structures required for this project. It also has a major benefit in its ability to overload operators, an extremely useful tool when working with vector math.

Microsoft Visual C++ 6 will be the software used to write the code because of the author's familiarity with it and its ability to create Windows based programs featuring GUIs relatively simply.

The OpenGl API was chosen for the 3D graphics rendering, largely because of its simplicity of setup compared with DirectX. The project is not graphics intensive and so the only factor taken into account is simplicity of code.

5.1.1 Language Dependant Features

Pointers are used heavily in the simulation for geometric structures such as the particle, spring and block. Comparing whether or not two objects, such as two particles, actually refer to the same instance was thought to be unsafe with pointers and so the notion of IDs is used. Each particle, for example, contains a unique integer "particleID", each block has a unique integer "blockID", and so on.

The use of IDs proved to be a useful one for identifying relationships between geometry without using pointers all the time. For example, each particle is also given a “blockID”, which is the ID of the block to which it belongs. These IDs are frequently used in the program as a simple check when the actual object is not needed, just the relationship.

The project relies heavily on lists of different types of structures which may themselves contain lists; for example, a list of blocks where each block contains a list of particles. It was decided early on that the project should use the vector structure from C++’s standard library. The vector encapsulates all the requirements of a list needed for the types of lists in the project such as array access and variable size. The following code fragment shows how a vector of pointers to particles can be defined:

```
std::vector<Particle*> particleList;
```

This is effectively how the global list of all particles is defined in the JengaMain class.

5.2 JengaMain Class

The JengaMain class is the main simulation class of the program. The class is independent of the platform it’s on, the interface of the program that runs it and the API used for rendering: its sole purpose is to simulate the game of Jenga™. Other classes are used to implement platform dependant code and then use this class to get the relevant data out of it.

class JengaMain		
Public Functions		**Description**
int	Init()	Initialises all the simulation data
int	Loop()	Called every frame to update the simulator
int	Reset()	Resets the simulator to the default stack of blocks. Essential destroys every object and calls Init()
int	JengaMain()	Creates the class, initialises default values
int	JengaMain~()	Destroys the class and all of its objects
Data		**Description**
bool	friction	Boolean value to set whether friction is on or off
bool	disableBlockMode	Boolean value to set whether the disable block mode is enabled or not
bool	paused	Boolean value to pause the simulation. No physics is performed but the tower is still rendered each frame
int	rowCount	Number of rows for the tower to simulate, effective upon a reset
vector<Particle*>	particleList	List of all particles in the simulation
vector<Block* >	blockList	List of all blocks in the simulation
vector<Spring*>	springList	List of all springs in the simulation
vector<Face* >	faceList	List of all faces in the simulation

Table 5.1: Public member functions and data for the JengaMain class

The class contains the main methods and data for the simulation. The class also contains numerous private functions for collision detection and integration but these are encapsulated away from its external use. In terms of use, the class should be initialised using Init() and then Loop() should be called every frame.

5.2.1 Initialisation

All initialisation is performed in the Init function. The Init function creates each block by calling the following function:

```
CreateBlock(vec3D pos, vec3D dims)
```

The create block function creates the relevant structures placing them on their respective lists. For example, the rest length of each spring is calculated here and all relationships between geometry are determined by the pointers and IDs this function creates.

Each block is created with a random variation in its height. It was considered varying width and depth to be irrelevant to the game as Jenga™ as it is only the variation in height that produces the loose block situation as described in section 3.2.3.

The rows in the tower are created to be initially spaced apart in the height direction. When the simulation is started, the blocks fall on top of each other and settle into the tower configuration.

5.2.2 Main Loop

The main loop function is as follows:

```
JengaMain::Loop()
{
     if(!paused)
     {
          for(int loop=0;loop<midLoopCount;loop++)
          {
               Integrate();
               CollisionPhase();
               AcceptNextState();
          }
     }

     renderer->Render();
}
```

Figure 5.2: JengaMain::Loop() function

The Integrate function computes the forces as well as integrating the simulation forward. Typically the forces will be calculated more than once per iteration, depending on the type of integration used, and so decoupling these two tasks is not possible. Code snippets from the functions involved in calculating forces can be found in appendix 2.

The collision phase resolves all collisions between blocks, making the next state a valid one. This is then copied to the current state by the AcceptNextState function and then rendered by calling the Render function of the Renderer class.

The for loop is there for stability. It is not possible to take large step sizes because the spring equations will cause the springs to explode. As a result, smaller step sizes are taken but performed more than once per frame to make up for it. The more times the for loop iterates, the further the blocks will move in a given frame, but the slower the frame-rate. The frame rate can be improved by reducing this value but the simulation will appear to move in slow motion. An adaptable frame rate, whereby the simulation step size is proportional to the length of time the last frame took to process, was not implemented because of the risk of stiff equations in some circumstances. This is an area that can benefit from future improvement.

5.3 Data Structures

All geometric data structures (particle, spring, block etc…) are implemented as structures in C++ rather than classes. This is mainly due to their simplicity and orientation around data as apposed to functions.

During implementation it was decided that the geometric data structures would store extra data and functions to that mentioned in the design. The Block structure, shown in Figure 5.2, is a prime example of this. It was decided during the implementation that the bounding boxes would be cached over frames and only updated when the particles moved. Instead of employing a separate list of bounding box structures, the bounding box's min and max vectors were placed directly in the actual structure they bounded. This was also done when it was decided that the blocks could have different textures: a textureNum variable was added to the block structure. Finally, the function getCentre was added instead of having a global function such as "vec3D getCentre(Block* block)", which is passed a block. All of these examples could have been implemented without adding to the Block data structure, but were done so for ease of implementation.

```
struct Block
{
	std::vector<Particle*>	particleList;
	std::vector<Spring*>	edgeList;
	std::vector<Face*>	faceList;

	bool active;
	int blockID;
	int textureNum;
	vec3D min,max;

	Block()
	{
		blockID = -1;
		active = true;
		textureNum = 0;
	}

	// returns the centre of a block
	vec3D getCentre()
	{
		vec3D ret;
		for(int i = 0;i<8;i++)
		{
			ret+= particleList[i]->curr.position;
```

```
            }

            ret /=8;

            return ret;
        }
    };
```

Figure 5.3: Code listing of Block structure

The particle structure, arguably the most important structure in the simulation, has a few additions to it also which were added for simplicity rather than necessity.

```
    struct Particle
    {
        double mass;

        int blockID;
        int particleID;

        int faceNum;
        Face* faceList[6];
        Spring* springList[3];
        vec3D min,max;

        struct State
        {
            vec3D position;
            vec3D velocity;
            vec3D force;
        }curr,next,*state[2];

        Particle()
        {
            mass = 0;
            blockID = -1;
            particleID = -1;
            faceNum = -1;
        }
    };
```

Figure 5.4: Code listing of Particle structure

Figure 5.4 shows the code listing for the particle structure. There has been the addition of the faceList and springList. The face list is a list of all faces that the particle is a part of; similarly for the spring list. These lists were used to make it quicker to determine which bounding boxes need to be updated when a particle has been updated; they are not essential to the working of the simulator.

The State structure is defined inside the declaration for a particle, explicitly linking the two together. A state structure is declared as follows:

```
Particle::State myState;
```

The current and next states are defined along with the definition for a particle state. “state[2]” is an alternative way of accessing the current and next states.

5.4 Collision Routine

The structure of the main collision routine, implemented in the JengaMain::CollideBlocks() function, can be seen in Figure 5.5. The two outer loops iterate through each block pair whilst the inner loops iterate through each of the block pair's point-face and edge-edge tests. If an AABB test, seen in blue, returns' false (indicating no collision), the current loop's iteration is skipped. The initial block-block AABB test will skip the most amount of tests, but is less accurate than the more tightly fitting bounding boxes of the other AABB tests.

```
for(Block b1 = blockList[0 to blockListSize-1])
{
      for(Block b2 = blockList[1 to blockListSize])
      {
          if(!AABBTest(b1,b2))  continue;

              for(Face f2 = b2.faceList[0 to 12])
              {
                  if(!AABBTest(f2,b1))  continue;

               for(Particle p1 = b1.particleList[0 to 12])
                  {
                      if(!AABBTest(p2,f1))  continue;
                   //perform detailed collision test between p1 and f2
                  }
              }

              for(Face f1 = b1.faceList[0 to 12])
              {
                  if(!AABBTest(f1,b2))  continue;

                  for(Particle p2 = b2.particleList[0 to 12])
                  {
                      if(!AABBTest(p2,f1))  continue;
                   //perform detailed collision test between p2 and f1
                  }
              }

              for(Spring s1 = b1.edgeList[0 to 12])
              {
             if(!AABBTest(s1,b2))  continue;

                  for(Spring s2 = b2.edgeList[0 to 12])
                  {
                      if(!AABBTest(s1,s2))  continue;
                   //perform detailed collision test between s1 and s2
                  }
          }
      }
}
```

Figure 5.5: Pseudo-code outline of main collision algorithm

5.4.1 Optimisation

This section will present information obtained from the simulation of Jenga™ to show how the various optimisation techniques employed reduced the number of detailed collision tests to be performed. Table 5.1 shows the number of collisions left after various levels of optimisations, from no optimisation all the way down to the actual amount of collisions encountered in a given frame.

Test performed	6 Blocks	9 Blocks	18 Blocks
No optimisation tests performed	102816	235872	961632
Block-block AABB tests	30240	48384	102816
All AABB tests (tests for every individual test pair)	8274	13480	27584
Full optimisation, All AABB and plane intersection tests	174	213	767
Full accurate tests (Actual amount of collisions left)	40	67	158

Table 5.2: Number of potential collisions left each frame after various tests

The plane intersection tests mentioned in table 5.2 refer to the tests that determine whether a particle has passed through the plane defined by a face (not necessarily the face itself), or that two lines (not line segments) formed by two edges have crossed. These are more costly tests than the AABB tests, but less costly than the full test. As can be seen from Table 5.2, with all optimisations on (2nd to last row), the amount of potential collisions left is about 4-5 times more than the actual amount of collisions.

The tests in table 5.2 are listed in descending order of how time consuming they are to perform, the bottom test being the most time consuming. Each successive test reduces the amount of collisions to be tested for but in turn is more complex to perform. This reduction of potential collisions via increasingly complex tests is known as multi-phase collision testing: the tests get more accurate but slower to perform.

The difference between no optimisation and full optimisation for this one function is an overall speed increase for the simulation of at lease 10 fold. However, the function is nowhere near fully optimised. No Jenga™ specific optimisations were performed; in fact, nothing was assumed about the type of objects being simulated. One potentially huge optimisation that was considered, but never implemented because of time, involves the use of deactivated blocks. When the tower is at rest, all blocks are deactivated and there are no collision tests to be performed. When a block is pulled out of the tower, only a few blocks around it will be activated and will need collision tests to be performed. This method would suit the game of Jenga™ perfectly as for most of the time the vast majority of blocks are motionless.

5.5 Stability

The actual design of the simulator does not incorporate any instability; it is the specific implementation of it that introduces the notion of numerical accuracy and computational efficiency. One thing that was noticed during the implementation was the amount of threshold values that were required. The values were often arbitrarily set to what worked well and had no explicit or intuitive meaning. An example of this would be the threshold used to determine whether a block should be disabled or not. Another example is the use of thresholds when testing for 0: a double precision value is not likely to be exactly 0, but within a threshold around 0. Deciding how generous this threshold is is purely down to the specific needs of the simulator.

5.5.1 Springs

The springs are at the heart of the stability issue. The stiffer the spring, the more likely it is to explode when integrated. The trade-off then became between rigidity of blocks and speed. Obviously, the stiffer the spring the better but deciding when to stop making them stiffer to maintain the speed of the simulator is a difficult thing to judge. When the simulator is slow it is easy to get more speed out of it by reducing the stiffness of the springs and then increasing the step size but going too far makes the blocks “spongy”, a very subjective thing to judge exactly when it is detrimental to the simulation. Another issue is the amount of over compensation given to the integrator to make sure it can integrate the more extreme situations it may be faced.

Issues like this are very subjective. All relevant constants such as stiffness were tailored over time and adjusted as necessary. These values are all placed in an ini file for ease of editing, which loads up when the JengaMain class initialises.

5.5.2 Integration

The Euler, midpoint and Runge-Kutta 4 methods were all implemented for integration. The one that actually ended up being used was a multi-step Euler method. This gave the best speed-accuracy trade off because the amount of iterations of the Euler method could be tailored to the exact needs of the simulator, specifically that as described in section 5.5.1.

5.6 User Input

The idea, as set out in the design chapter, is to have users drag and drop blocks around in the simulator much like an icon in a Windows based environment. Effectively, the blocks are to be moved in a two-dimensional plane coplanar with the screen. Implementation wise, the plane was simple to define using the cameras z-axis as plane normal and the selected block’s centre as a point on the plane. However, manipulation of the blocks did not work out as expected.

The original idea for moving blocks was to add a force to the selected block (equally to each of the block’s particles) in the direction of the user’s movement and of magnitude proportional to the distance between the mouse point and the current position of the block. This method works well for removing a block from the tower but cannot drag a block through the air: as soon as the block is in free space it will bounce around the mouse cursor, effectively as if it were on a spring. Modifying the forces alone therefore is not sufficient to drag a block effectively through free space.

A method that does drag a block through free space as expected was achieved by overwriting the velocity directly with the vector from block to mouse point. This resulted in a smooth drag through free space. However, overwriting the velocities directly meant that the user manipulation was directly violating the simulation properties. Effectively the user had "super strength" and could drag a block straight through the tower regardless of blocks being in the way. This clearly was not acceptable.

The solution implemented is a hybrid of the two methods described above. When a block is in contact with any other block, or the floor, the first method of modifying the forces is used. When the block is in free space, the second method of directly overwriting the velocities is used. This has the result that the block moves correctly in free space but modifies the force correctly when in contact with the tower or floor.

5.7 Problems

5.7.1 Disabling Blocks

Disabling blocks that are not very active has a two fold benefit: fewer collisions to test for and reduced jitter in the tower. The difficulty arises in deciding a calculation that will determine whether or not a block is moving much. Unfortunately, whether a block is perceived to be moving or not its particles will still be moving in some way to correct the spring forces when forced down under gravity. The test used, the full code listings for which are shown in appendix 2.6, simply uses the total distance each particle in the block has moved through. If this value is below a certain threshold, the block is disabled. What this test does not take into account is the fact that a block can compress resulting in motion in each particle, but no overall motion through space. Finding out how much the centre of the block has moved is possible, but would not take into account angular motion. Also not taken into account is the length of time a block has been still, the implemented method works on a frame by frame basis.

5.7.2 Friction

Friction remains the greatest problem to overcome in the Jenga™ simulator. It was decided that only dynamic friction should be implemented because static friction was too complex to implement in the time frame. Even so, dynamic friction proved to be extremely difficult to implement and in the end was not done so adequately. As a result blocks can occasionally be forced out of the tower when they should be still, and do not slide when they should do so.

Friction is added to the particles according to the contact information from the last frame. Each collision's contact information is stored in a friction contact class (see appendix 2.6 for code listings). Each contact contains the standard properties of a collision such as the collision normal from the FrictionContact class. The three individual types of contact, particle-floor, particle-face and edge-edge then inherit from this class adding their own relevant information.

The problem with friction is adding the correct amount of force, considering that each block may have a number of edge-edge and particle-face contacts with other blocks. The force applied has to be equal and opposite for each point in contact. It is in trying to balance these forces to make a block come to rest that problems were encountered and never fully fixed. Occasionally the friction forces would cause a block to rotate rather than come to a rest. To reduce the problem, but not remove it altogether, a form of drag was added to each contact point in the direction perpendicular to the collision normal. This is not an ideal solution.

Chapter 6: Results and Evaluation

This chapter will evaluate the program developed in chapter 5 against the requirements as set out in section 3.7. The first part of this chapter will display the results of the simulation with direct comparison to a real Jenga™ tower. The second part will evaluate the success of the project based on how well the requirements were met and how believable the simulation of Jenga™ is.

6.1 Results

There are many aspects to measuring the success, or failure, of the program developed but ultimately the test should be how well it manages to model the game of Jenga™. The first part of this section will model various configurations in the real Jenga™ tower and try to emulate them in the simulated one. The second part of the results will play out a sequence of moves typical in a normal Jenga™ game and see how well the simulator compares to it. This will demonstrate every aspect of the simulation, from user manipulation of blocks to physical believability.

6.1.1 Static Configuration Tests

A suitable set of configurations to test has been chosen. First of all, a code will be devised to accurately describe a given configuration. These will be referred to during each test.

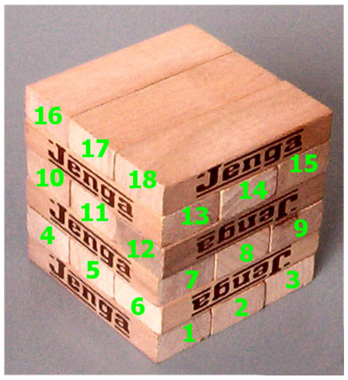


Figure 6.1: Jenga tower depicting block order

Figure 6.2 shows the order in which the blocks will be referred to. Each row will be considered ordered from left to right using the sides facing the camera. A block can then be referred to by its row (from the bottom) and number: for example, block 7 can be referred to as the 1st block in 3rd row, or 3-1 for short.

A specific configuration will be referred to by a bit sequence in the order of the blocks in the tower. A 1 will signify the block is present, a 0 that it is not. Figure 6.2 for example has a bit sequence of (010)-(010)-(011)-(010)-(101).

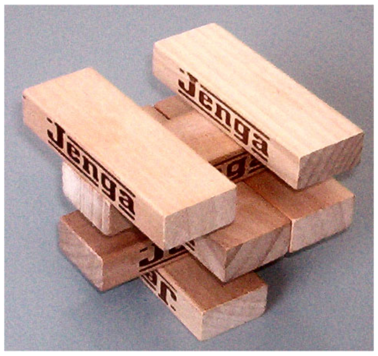


Figure 6.2: Jenga tower configuration

In each test the real tower will be shown on the left with the simulated one on the right. Small towers of just a few blocks are tested because it is the point of collapse of the tower that is of interest, not the whole tower. Tests will be referred to by their bit sequence.

Test 1: (010)-(010)-(010)-(010)-(010)-(010) – Stable

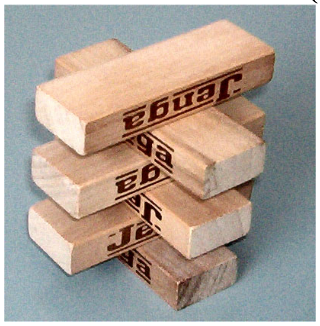


Figure 6.3a

This 6 row high tower contains only its middle blocks. The simulation performed as expected. In general, all symmetrical towers can be simulated because forces throughout the tower are balanced. No more symmetrical towers will be considered.

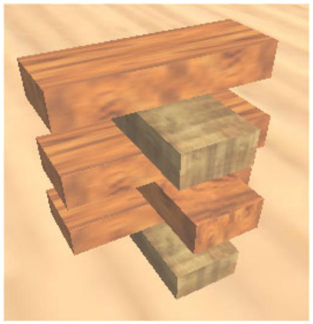


Figure 6.3b

Test 2: (001)-(010)-(001) – Stable

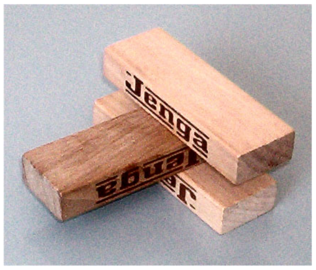


Figure 6.4a

This test is stable until the top block is removed. The top block is holding the middle one in place. The simulator simulates it perfectly.

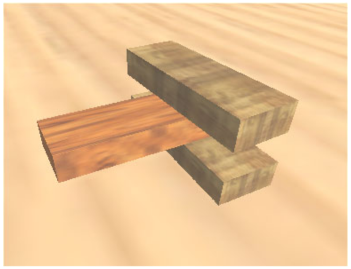


Figure 6.4b

Test 3: (001)-(101)-(001) – Unstable

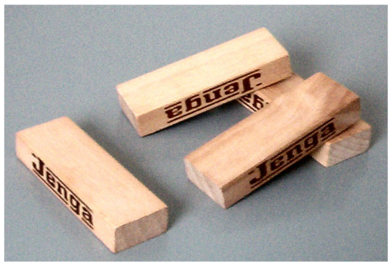


Figure 6.5a

Similar to the last test but with two blocks in row two instead of one, this tower collapses making the top block slide down the middle two. The simulator performed perfectly.

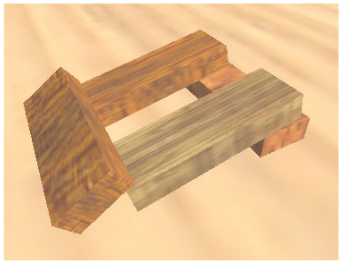


Figure 6.5b

Test 4: (100)-(010)-(001) – Unstable

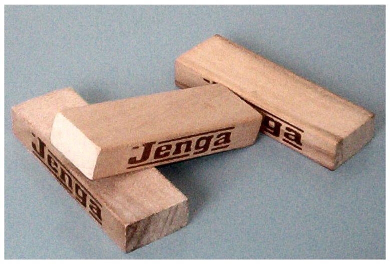


Figure 6.6a

Obviously unstable in the real tower, the simulator performed in a physically correct manner.

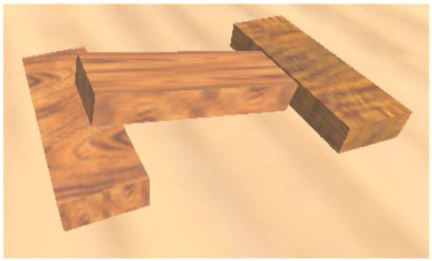
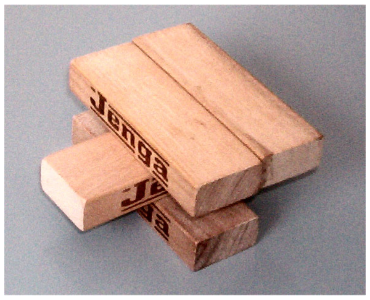


Figure 6.6b

Test 5: (010)-(010)-(011) – Stable



Both towers are stable in this configuration.

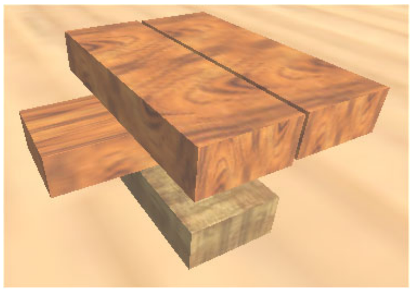
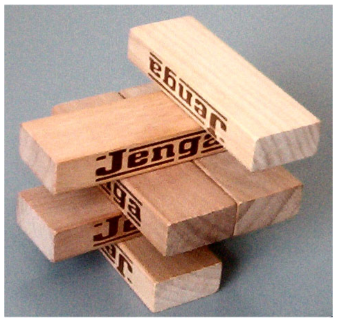


Figure 6.7a **Figure 6.7b**

Test 6: (010)-(010)-(011)-(010)-(001) – Stable



This tower can cause the simulator some problems. The tower puts more force down on its right side and this can occasionally lead to the block in row 2 position 2 being forced out of the tower incorrectly.

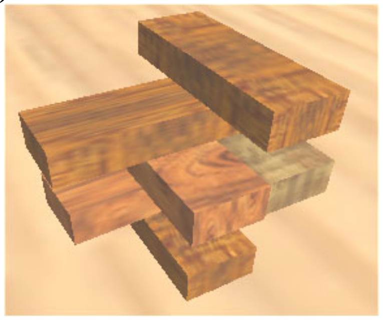


Figure 6.8a **Figure 6.8b**

The static tests performed very well compared to the real tower. Configurations that caused a problem for the simulator are ones with a large force placed on one side of the tower, and a lone middle block somewhere lower down in the tower. Friction, which is not implemented well, does not always stop the lone middle block being forced out slowly leading to the collapse of the tower.

6.1.2 Testing a Sequence of Moves

Now a sequence of moves will be played out on the real tower and emulated in the simulated one. The initial tower will consist of six full rows of blocks, 18 blocks in total, because this is what can currently be simulated in real-time. Using the bit notation as in the previous section, the following is the configuration of loose blocks.

(L00)-(00L)-(00L)-(0L0)-(L0L)-(LLL)

The simulator will model the locations of the loose blocks by making those blocks slightly smaller in height than the rest.

Step 1: Remove loose block 3-3 slowly

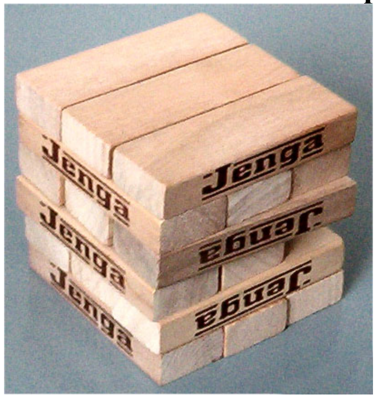


Figure 6.9a

Both towers simulate the block being removed with no movement in the either tower.

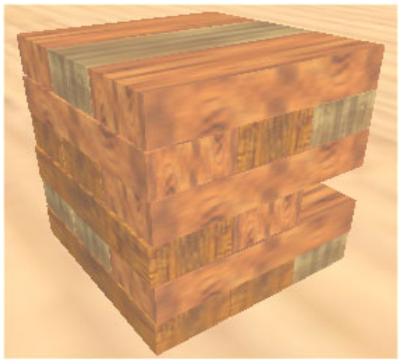


Figure 6.9b

Step 2: Place block slowly back on tower in position

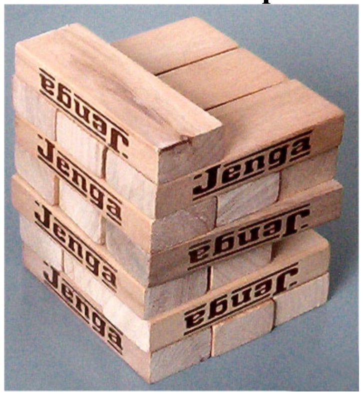


Figure 6.10a

The real tower does not move at all when the block is placed on top of it. The simulated tower moves a small amount and then stops.

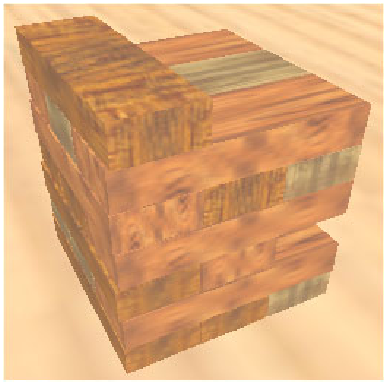


Figure 6.10b

Step 3: Remove block 3-1 slowly

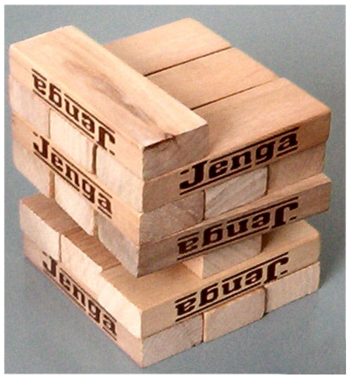


Figure 6.11a

The real tower doesn't move when the block is removed from it.

The row below the block being removed in the simulated tower does move slightly.

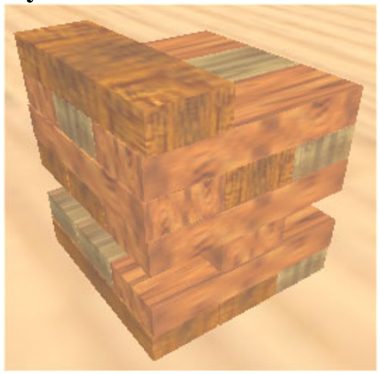


Figure 6.11b

Step 4: Place block back on tower in 7-2 position

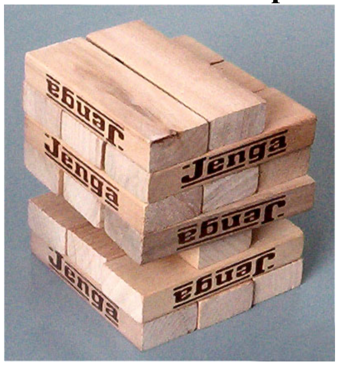


Figure 6.12a

Both towers don't move when the block is placed on them.

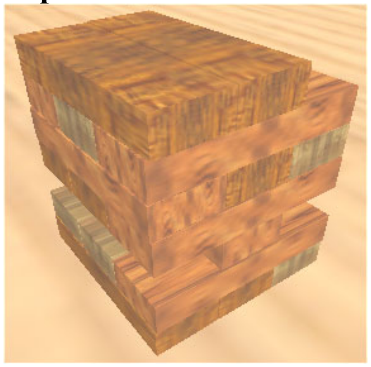


Figure 6.12b

Step 5: Remove block 4-3 quickly

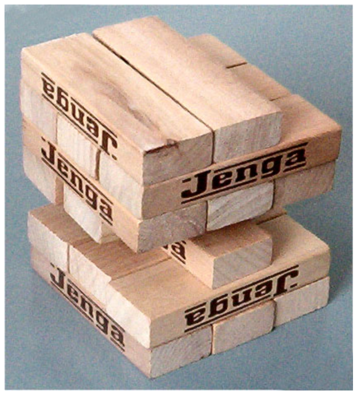


Figure 6.13a

Both towers move a small amount in the direction the block is removed from the tower.

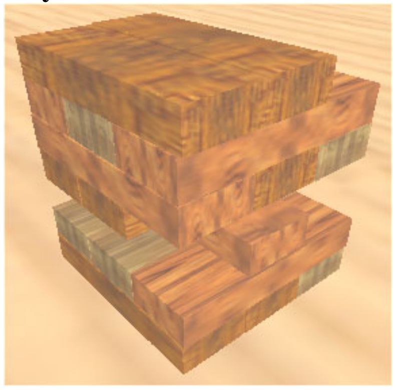


Figure 6.13b

Step 6: Drop block from 1 cm above top of tower in 8-1 position

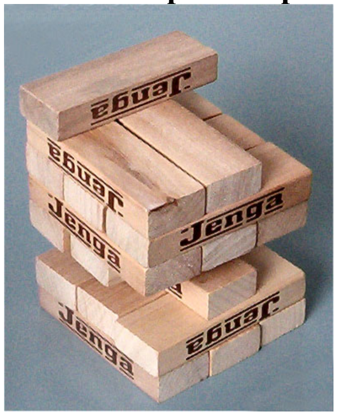


Figure 6.14a

The real-tower moves a slight amount on the side of the tower the block is dropped on.

The simulated tower also moves down on the side the block is dropped on, but takes longer to come to a rest.

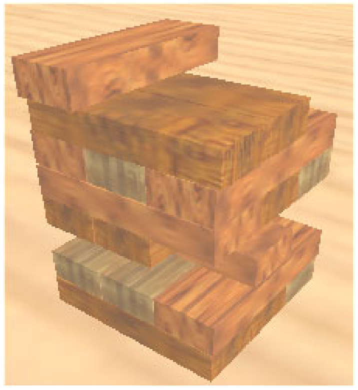


Figure 6.14b

Step 7: Remove loose block 2-3 quickly

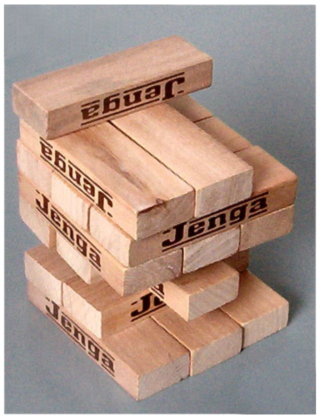


Figure 6.15a

The real-tower does not move.

The simulated tower sinks slightly in the direction of the removed block.

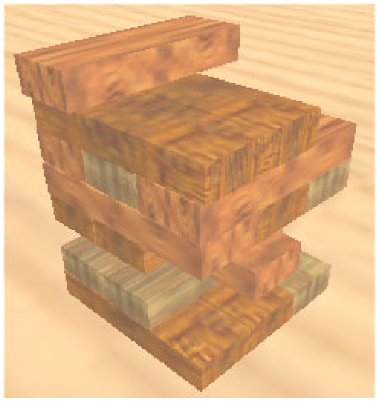


Figure 6.15b

Step 8: Drop block from 2cm from top of tower in 8-2 position

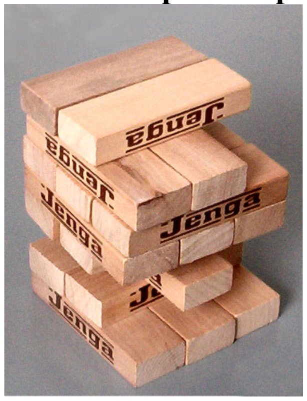


Figure 6.16a

Both towers shake slightly, especially block 8-1, when the block is dropped. All blocks down to row 3 compress slightly in the simulated tower before coming to a rest.

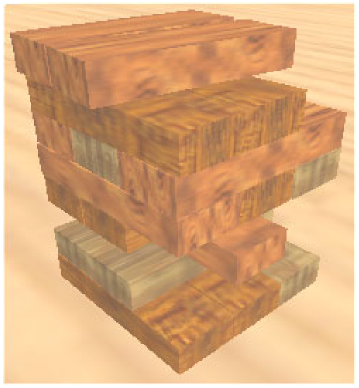


Figure 6.16b

Step 9: Remove loose block 2-1 slowly

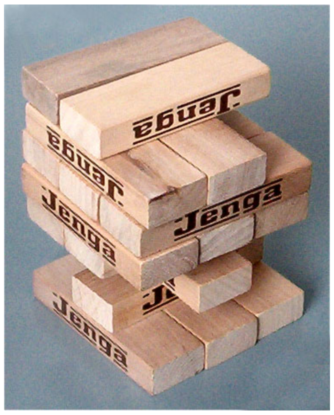


Figure 6.17a

In the real-tower the block is removed with no movement in the tower. The simulated tower though is weighed down on the loose block and so pulling it out does have a slight effect on the tower.

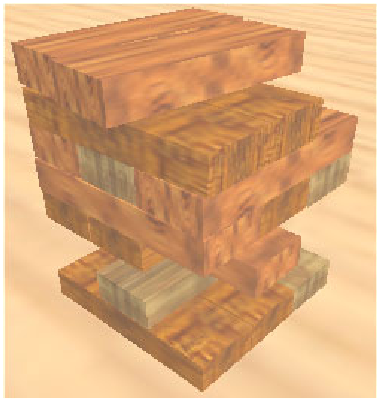


Figure 6.17b

Step 10: Place block on tower in 8-3 position

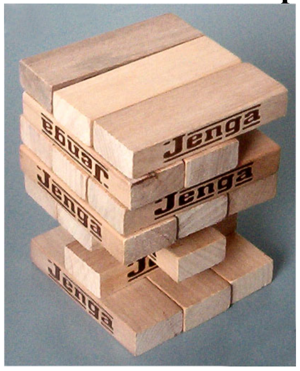


Figure 6.18a

The real tower doesn't move.

The simulated tower doesn't move, but the block slides slightly to a stop.

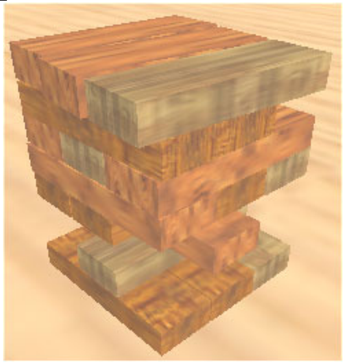


Figure 6.18b

Step 11: Remove loose block quickly from position 5-1

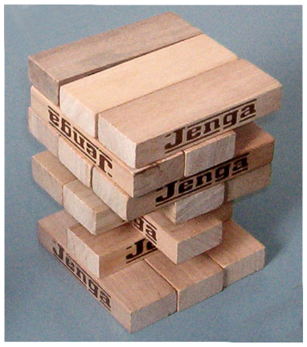


Figure 6.19a

The block in the real tower is removed easily.

The block in the simulated tower is also removed easily, but block 8-1 moves slightly, unrealistic.

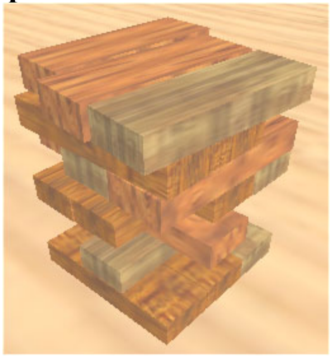


Figure 6.19b

Step 12: Drop block from 1cm from top of tower in 9-3 position

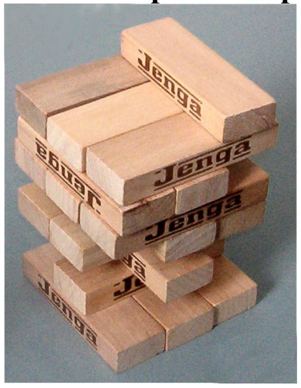


Figure 6.20a

The top row of the real tower moves a small amount.

The top row of the simulated tower slides a bit and then comes to a rest.

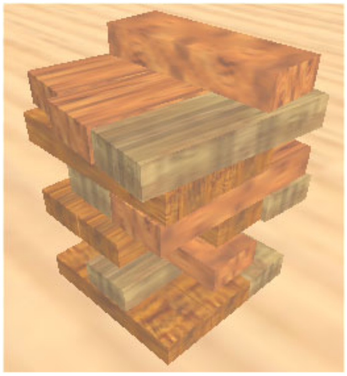


Figure 6.20b

The simulated tower performs well against the real one. Steps 1-10 are modelled very believably by the simulator, the only noticeable difference being slightly more movement in the tower when a block is dropped back on top of it. Generally the simulated tower took longer to damp out the impact of a block being dropped on top of itr. Steps 11-12 show signs of unrealistic behaviour when the top blocks start sliding for no apparent reason. They do come to a halt promptly but could just as easily have slid off the top of the tower.

6.2 Evaluation

The system developed will now be discussed in detail in the context of its requirements.

6.2.1 Gameplay

The gameplay requirements are summarised as follows.

- A player should be able to remove a block from the tower.
- A player should be able to place a block back on top of the tower.
- Input should be analogue to simulate a player's movement in the real game.
- The camera should allow for full view of all sides of the tower.

Obviously, removing a block and placing it back on top of the tower is crucial to the game of Jenga™. The way in which a user does so though is equally important. In the real game of Jenga™ the skill lies not only in the judgement of which block to remove but in the speed, angle and steadiness at which it is removed. Equally, when placing a block back on top of the tower, steadiness of placement is essential, not just an extra feature.

The way in which a player interacts with the blocks in the program developed in chapter 5 is a combination of the camera movement and the movement of the blocks. The camera is free to rotate round the tower around both the x and y axes, whilst always looking at it. When a block is moved, it is moved in the camera's current x/y plane or alternatively in the z-axis direction. This combination of movement allows for any combination of block movement whilst remaining an intuitive system to use. The camera represents the way in which a player moves around a real tower to survey their choice, and the movement emulates the way in which a player will remove a block relative to their eye position.

The mouse is the sole method of movement, either camera or block, satisfying the requirement that the player's movement should be analogue. Just one modifier key is used in conjunction with the mouse to perform all movement within the game, a very simple system to use. An extra key is used for temporarily pausing the game to allow the user to readjust the camera whilst still holding onto a block. Just like the design dictated, the blocks are dragged around like an icon in windows by holding the left mouse button down. As a result, the input to the game is extremely simple and intuitive to use.

It was a deliberate intention not to allow the user to lock the camera at 90° angles to the tower enabling the user to pull a block out at the perfect angle. The position of the camera and the subsequent removal of a block is solely the judgement of the player based on the way they interpret what they see. No unnatural guides such as numbers are used as a reference.

When a player moves a block they are actually applying a force to it proportional to the distance between the mouse position and the block. This generally has the effect that the faster the mouse is moved, the faster block moves, but not necessarily so. A block that has other large forces acting on it, such as being at the bottom of the tower, will take more force to move than a loose one and so the same amount of mouse movement will result in a smaller resultant velocity. This is crucial feedback to the user as to how "stuck" a block is and consequently how removing it from the tower will affect the other blocks.

In terms of gameplay, the program fulfilled all of its requirements, and successfully too. The only action a player cannot do in the simulation that is possible in the real game is to rotate a block around. This was deliberately left out as it added an unintuitive control to the input system and added little benefit to the game of Jenga™.

6.2.2 Physics

It was decided after the analysis section that a spring-based method was going to be used to implement the physics instead of the common rigid-body method. The spring-based method was developed from scratch making use of common techniques where applicable. It was this fundamental design choice that had the major impact on the results of the simulator. The results of the method will now be discussed, making references to the rigid-body method that was rejected in the analysis section.

Beneficial Features

The spring-based method's major advantage is that all collisions are guaranteed to be solved within just a couple of iterations. This means that the simulator does not slow down dramatically in certain frames due to a complicated situation, but stays simulating at a constant frame rate. The only factor that does affect the speed of resolving all collisions is the amount of contact points there are to resolve.

The situation of Jenga™ is not just one with a high number of contacts, but one where each contact is linked indirectly to every other one. The rigid-body method performed poorly in this situation because resolving collision pairs almost always produced new ones. Being able to solve all collisions, no matter what the situation, in just a couple of iterations was a major benefit in the solution developed.

Another beneficial feature to the spring-based method is its simplicity and intuitiveness. The spring-based method has no notion of resting-contact or angular motion and so forgoes any of the problems associated with them.

A continuous collision detection method was implemented which means that blocks cannot pass straight through each other, satisfying requirement 2.7.

Limitations

The spring-based method's main limitation lies, unsurprisingly, in its springs. Stiff springs have to be used to give the appearance of modelling rigid-bodies, but these result in stiff equations. As a consequence, the integrator has to take small time steps. To take larger time steps, the simulator has to perform numerous smaller steps, slowing the simulation down some what.

Another problem unique to the spring-based method is the sheer number of springs that have to be used to model an object. A cuboid, for example, took 28 springs, and this can be considered a simple object to model.

Problems

The simulator developed does not always produce physically believable behaviour. Occasionally a block can become embedded with another block causing the blocks to stretch apart: this problem is down to numerical accuracy. Other than that, there were 3 main problems the simulated faced that are attributed to creating unrealistic physical behaviour.

- **Springs**

The stiffness of the springs was always going to be a crucial factor to the success of the simulator, but it turned out that the springs could be made stiff enough to make the blocks seem rigid. However, tall towers pose a problem to the spring-based method because of the sheer amount of force being put upon the blocks lower down in the tower. Figure 6.21 demonstrates what happens to a full Jenga™ tower when the springs are stiff enough make the blocks appear solid, but not stiff enough for the huge compression force placed on the tower.

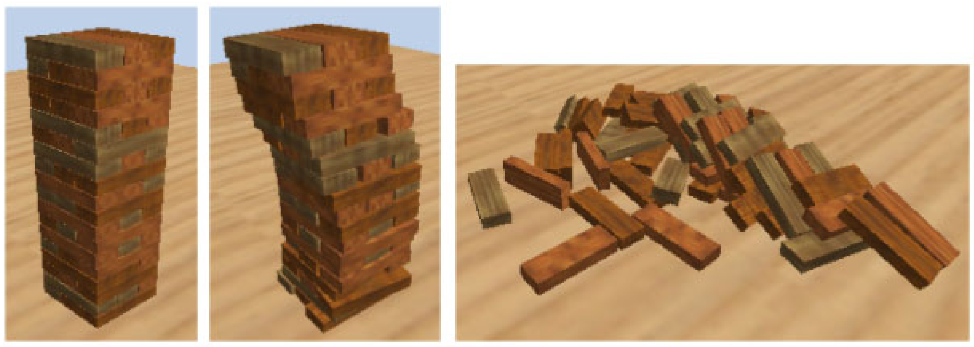


Figure 6.21: The results of simulating a full tower with soft springs

The situation is easily resolved with stiffer springs, but at the expense of frame-rate. The only true solution to the problem is to optimise the simulator, then trade-off some of the speed for spring stiffness.

Another problem with the springs was that of jitter. From a distance it is barely noticeable but zoomed in it can be seen that the edges of the block are actually fluctuating by a small amount. The overall position of the block appears to be motionless but it is the jitter than is effectively acting as the block's resting contact. A rigid-body simulator does not have this jitter problem.

- **Friction**

Friction was without doubt the biggest problem the simulator faced. The problem lies in the implementation of friction as apposed to any specific problem applying it to the spring-based method. Friction is applied at contact points, as expected, but because two blocks can have multiple contact points between them, balancing out the friction becomes very difficult to achieve when only applied in a local context. The effect is that the friction forces can twist a block around as apposed to bringing it to a rest. This was remedied slightly by adding friction drag, essential a form of drag only applied to contact points that does not act in the direction of the collision normal. This solution though is not an ideal one.

- **Disabling Blocks**

Disabling blocks was largely a beneficial feature to the simulation: it reduced jitter and helped to speed it up slightly. However, occasionally blocks are disabled that should not be. This misclassification lies in the test used to determine when a block should be disabled.

A block is disabled if the distance each of its particles moves in a given frame is below a certain threshold. This test is done because even though a block may look still, its particles are actually moving slightly. However, the case where a block's centre of gravity is just overhanging an edge can sometimes be misclassified as a disabled block because the movement is so slight when it should be tipping over.

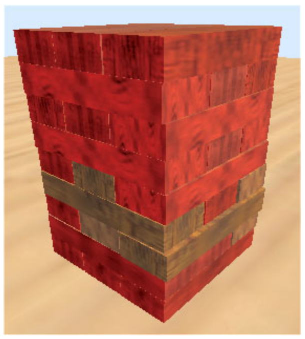


Figure 6.22: Red blocks indicate disabled blocks

Figure 6.22 shows that the method used for disabling blocks is not a perfect solution. The red blocks indicate blocks that are disabled. Baring in mind that the tower in figure 6.22 is at rest, there are 6 blocks that are not disabled. The reason for this is that even though the blocks have no overall net movement, the particles are moving enough to categorise them as moving blocks. The situation becomes worse with more blocks in the tower, with fewer and fewer blocks become disabled.

6.2.3 Program

The requirements for the program are summarised below.

- The program should not crash or freeze.
- The game should run in real-time, no less than 12 frames per second.
- The program should be simple and intuitive to use.

The first requirement, that the program should not crash or freeze was met. The only way that the simulator could possibly freeze is if the simulator got into an invalid state and could not resolve it. Though the simulator has got into an invalid situation occasionally, it has always been possible to reset it after a short period of time.

The second requirement states that the simulator should run in real-time, a frame rate of no less than 12 frames per second. As has already been discussed, there are many factors that can affect the overall speed of the simulator. The results displayed in Figure 6.23 are conducted at a high standard of physical simulation. The graph shows how the number of rows in the tower affects the overall frame-rate.

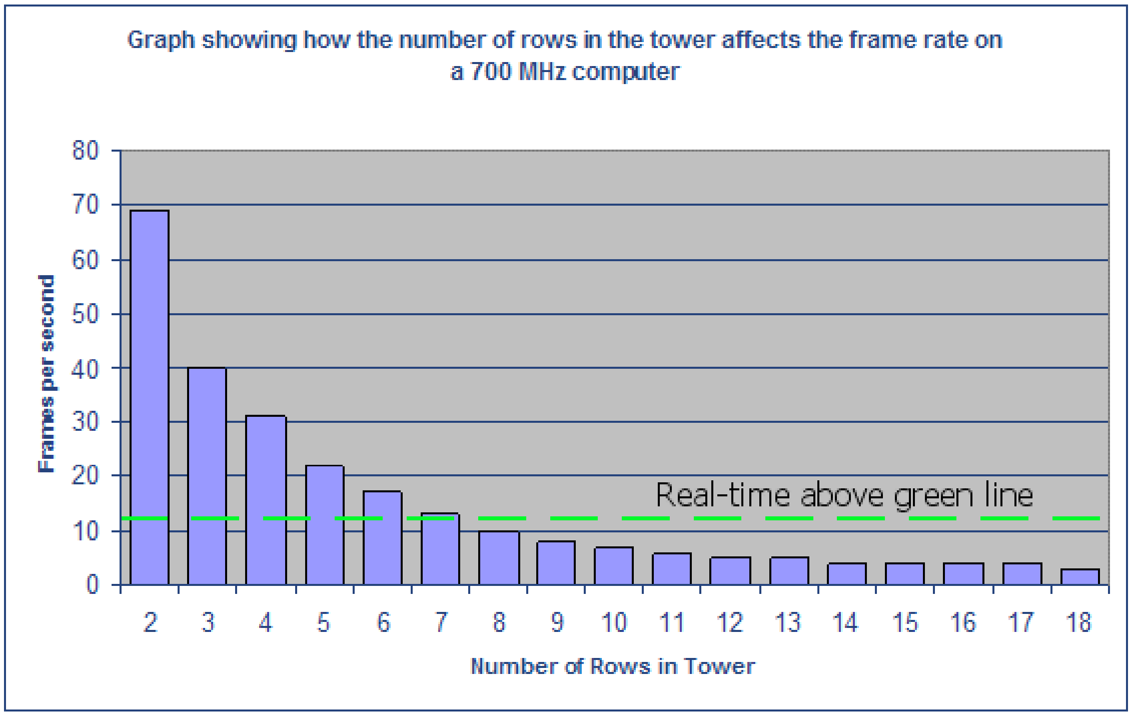


Figure 6.23: Graph of frame-rate against rows in tower

From Figure 6.23 it can be seen that a 7 row tower can just be simulated in real-time. In the analysis section, it was suggested that 6 rows would be an acceptable minimum to simulate, with 9 being a good target. 7 rows is a reasonable achievement considering the amount of blocks being simulated all in contact with each other. Higher rows could be simulated in real-time if the spring stiffness was reduced but that would compromise the integrity of the simulation.

Finally, the program is simple and easy. There is no clutter on the interface and as explained earlier, the way the player interacts with the game is extremely simple to understand.

6.3 Future Work

The spring-based method developed shows very promising signs of being a competitive physics simulator. Its major advantage is its quickness at resolving all contacts, regardless of how complex a situation. However, the method is limited by the stability of its springs, which require the simulation to be stepped forward slowly. It also requires a large amount of springs to be used for any one object. If these issues were addressed, the simulator could be made extremely competitive.

There are three main areas not associated directly with the spring-based method in which the simulator can be greatly improved.

Friction

Friction was not implemented adequately in the simulator, though this was not due to the spring-based method, rather a poor implementation. Implementing friction properly would immediately remove a lot of the inaccurate behaviour in the simulation. Adding static friction would also improve the quality of the simulation, though this would be difficult to implement.

Disabling Blocks

Disabling blocks is not only useful, but essential to the spring-based method in stopping jitter. The test used to determine when to disable a block does not take into account the fact that a block can compress or expand, and so these cases are categorised as moving blocks. If a suitable metric could be used to classify a block as at rest, the benefits to the game of Jenga™ would be huge. The whole tower would be disabled most of the time, speeding up the simulation vastly.

Optimisation

The simulation was by no means fully optimised. Only axis-aligned bounding box tests were used to remove collision pairs for the collision routine, a situation which could be vastly improved upon.

Conclusion

The project set out to create a real-time simulator for the game of Jenga™. Initially, a common rigid-body approach, closely following Baraff's method [6], was looked at but soon found to be inadequate for the specific problem of Jenga™. It was discovered that Jenga™ posses a very specific, problematic situation whereby at any one time a large number of objects are at rest and all in contact with one another. A different approach had to be taken.

In looking at different methods, and trying out new approaches to the problem, a spring-based method was discovered. A two-dimensional prototype using the approach was implemented and proved to be far more stable for the situation of Jenga™ than the rigid-body one. Fortunately, it also proved to be a much simpler method to implement. It was this method that was elaborated on to produce the final design.

The spring based method has the advantages that resting contact and angular motion do not have to be modelled. The major advantage though of the spring-based method is that all collisions can be guaranteed to be resolved within a couple of iterations. For Jenga™, a situation with a high number of contacts all connected to each other indirectly, this was essential. The method's prime weakness, one that the rigid-body method doesn't suffer from, is that objects modelled can deform and squash under pressure or high velocity collisions. This undesirable property can be reduced to the point that it is not noticeable, but only at the expense of speed of the simulation. This trade-off proved to be at the heart of making the simulator a successful one.

The spring-based method has two speed disadvantages compared with the rigid-body method. One is the fact that stiff spring equations require small time steps to be taken: the rigid-body method, broadly speaking, has no such problem. Two is the fact that an object requires a lot of springs to model it adequately. Optimisation became a key part of the project as more speed also meant more stability and realism. Though the optimisation performed sped the simulator up by at least 10 fold, there was still quite a bit of room for improvement.

The results from the implementation were encouraging. Chapter 6 shows that the simulated tower performs very favourably against the real one for towers of about 6-9 rows. The program developed was easy to use and the simulator produced physically believable behaviour. The simulator did have problems however with taller towers due to the compression of blocks on lower rows, but this could have been remedied by stiffer springs if the frame-rate had allowed. Generally, the simulator performed very well.

The implementation of friction let the simulation down somewhat. The full complexity of the problem of applying friction was not fully grasped until late on in the project. The spring-based method can support friction as well as any other method; it was just a bad implementation that was used. Friction would be top of the list of improvements to be made to the simulator. Another area that would have to be looked at in any future improvement of the simulator would be the test used to disable blocks. The test that was used neglects the fact that blocks can compress and expand and so these cases are always classified as moving blocks. Disabling blocks is crucial to the simulator to reduce jitter and increase overall speed, so improving this method would be greatly beneficial to the simulator.

In conclusion, it should be known that the spring-based method is not the perfect method of simulating objects in real-time, not even really a viable alternative to the rigid-body method for a lot of physic's-based applications. However, at the time of writing there still is no perfect solution to modelling physics in real-time. The spring-based method, though by no means perfect, shows promise and could potentially be adapted to make a competitive physics simulator. Maybe by exploring such novel techniques as the spring-based method, a better solution may be found.

References

[1] A. Witkin and D. Baraff, An Introduction to Physically Based Modelling: Differential Equation Basics, SIGGRAPH, 97.

[2] Chris Hecker, Physics Parts 1-4, http://www.d6.com/users/checker/dynamics.htm, first published in "Game Developer Magazine", October 96.

[3] Altmann, Simon L. Rotations, Quaternions, and Double Groups. Oxford, England: Clarendon Press, 1986.

[4] D. Baraff, An Introduction to Physically Based Modeling: Rigid Body Simulation I - Unconstrained Rigid Body Dynamics, SIGGRAPH, 97.

[5] D. Eberly, Testing for Intersection of Convex Objects: The Method of Separating Axes, Magic Software, http://www.magic-software.com

[6] D. Baraff, An Introduction to Physically Based Modeling: Rigid Body Simulation II - Nonpenetration Constraints, SIGGRAPH, 97.

[7] Jeff Lander, When two hearts collide: Axis-Aligned Bounding Boxes, article from February 1999 edition of Game Developer Magazine.

[8] B. Mirtich, J. Canny, Impulse-Based Simulation of Rigid Bodies, University of California Berkeley, 95.

[9] B. Mirtich, Hybrid Simulation: Combining Constraints and Impulses, University of California Berkeley, 96.

[10] D. Baraff, Fast Contact Force Computation for Nonpenetrating Rigid Bodies, SIGGRAPH, 94.

[11] Jeff Lander, Collision Response: Bouncy, Trouncy, Fun, article from March 1999 edition of Game Developer Magazine.

Appendix 1: Derivations

1.1 Constants for point-face collision equation

Section 4.4.1 produced the result that the time, t, of a collision can be found by the equation:

$$at^3 + bt^2 + ct + d = 0$$

The constants a, b, c and d are as follows:

$$\begin{aligned} a &= d3 \bullet c1 \\ b &= (a3 \bullet c1) + (d3 \bullet c3) \\ c &= (a3 \bullet c3) + (d3 \bullet c2) \\ d &= a3 \bullet c2 \end{aligned}$$

where:

$$\begin{aligned} a1 &= f2_0 - f1_0 \\ a2 &= f3_0 - f1_0 \\ a3 &= p_0 - f1_0 \\ d1 &= f2Vel - f1Vel \\ d2 &= f3Vel - f1Vel \\ d3 &= pVel - f1Vel \\ c1 &= d1 \times d2 \\ c2 &= a1 \times a2 \\ c3 &= (d1 \times a2) + (a1 \times d2) \end{aligned}$$

1.2 Constants for edge-edge collision equation

The constants a, b, c and d for the cubic equation are as follows:

$$\begin{aligned} a &= d3 \bullet c1 \\ b &= (a3 \bullet c1) + (d3 \bullet c3) \\ c &= (a3 \bullet c3) + (d3 \bullet c2) \\ d &= a3 \bullet c2 \end{aligned}$$

where:

$$a1 = p2_0 - p1_0$$
$$a2 = p4_0 - p3_0$$
$$a3 = p3_0 - p1_0$$
$$d1 = p2Vel - p1Vel$$
$$d2 = p4Vel - p3Vel$$
$$d3 = p3Vel - p1Vel$$
$$c1 = d1 \times d2$$
$$c2 = a1 \times a2$$
$$c3 = \left(d1 \times a2\right) + \left(a1 \times d2\right)$$

Appendix 2: Code Snippets

This section contains relevant code snippets from the implementation written in chapter. The code is written in C++.

2.1 JengaMain::Integrate()

```
JengaMain::Integrate()
{

	int numSteps = iniFile->getDouble("Integration","iterationsPerFrame");
	double timeStep = frameTime/numSteps;

	ComputeForces(&currStateList,&currStateList);
	IntegrateEuler(&currStateList,&currStateList,&nextStateList,timeStep);

	for(int i = 0;i<numSteps-1;i++)
	{
		ComputeForces(&nextStateList,&nextStateList);
		IntegrateEuler(&nextStateList,&nextStateList,&nextStateList,timeStep);
	}
}
```

2.2 JengaMain::ComputeForces(sourceStateList,destStateList)

```
JengaMain::ComputeForces(std::vector<Particle::State*>* sourceStateList,
               std::vector<Particle::State*>* destStateList)
{

	ZeroAllForces(destStateList);

	ComputeAndAddUserForces(sourceStateList,destStateList);
	AddGravity(sourceStateList,destStateList);
	ApplySpringForces(sourceStateList,destStateList);

	if(friction)
	{
		ComputeAndAddFrictionForces(sourceStateList,destStateList);
	}
}
```

2.3 JengaMain::IntegrateEuler(startStateList,sourceStateList,endStateList,time)

```
JengaMain::IntegrateEuler(std::vector<Particle::State*>* start,
                          std::vector<Particle::State*>* source,
                          std::vector<Particle::State*>* end,
                          double time)
{
	int particleListSize = particleList.size();

	for(int xi = 0;xi<particleListSize;xi++)
	{
		Particle* p = particleList[xi];
		Particle::State* startState = start->at(xi);
		Particle::State* sourceState = source->at(xi);
		Particle::State* endState = end->at(xi);

		endState->velocity = startState->velocity + (sourceState->force/p->mass)*time;
		endState->position = startState->position + endState->velocity*time;
	}
}
```

2.4 JengaMain::CollisionPhase()

```
JengaMain::CollisionPhase()
{
	ClearFrictionList();
	FloorCollision();
	ReactivateAllBlocks();
	CollideBlocks();
	DisableMotionlessBlocks();
	CollideBlocks();
}
```

2.5 JengaMain::DisableMotionlessBlocks()

```
JengaMain::DisableMotionlessBlocks()
{
	if(!disableBlockMode) return -1;

	double disableThreshold = 2e-4;

	for(int ib = 0;ib<blockList.size();ib++)
	{
		if(selectedBlock==ib) continue;

		Block* b = blockList[ib];

		double testVal = 0;

		for(int ip = 0;ip<8;ip++)
		{
			Particle* p = b->particleList[ip];
			double diffLength = GetLength(p->next.position - p->curr.position);
			testVal += diffLength;
		}

		if(testVal<disableThreshold)
		{
			b->active=false;
			for(int ip = 0;ip<8;ip++)
			{
				Particle* p = b->particleList[ip];
				p->next.position = p->curr.position;
			}
		}
	}

	return 0;
}
```

2.6 Friction contact class definitions

```
	class FrictionContact
	{
	public:
		int ID;
		int type;
		vec3D norm;
		double normForce;
	};

	class ParticleFaceContact : public FrictionContact
	{
	public:
		Particle* p;
```

```
          Face* f;
          double f1Perc,f2Perc,f3Perc;

          ParticleFaceContact()
          {
                type = 1;
          }
     };

     class SpringSpringContact : public FrictionContact
     {
     public:
          Spring* s1, *s2;
          double s1aPerc,s1bPerc,s2aPerc,s2bPerc;

          SpringSpringContact()
          {
                type = 2;
          }
     };

     class ParticleFloorContact : public FrictionContact
     {
     public:
          Particle* p;

          ParticleFloorContact()
          {
                type = 3;
          }
     };
```

2.7 IntersectMovingPointFace(point, face, time, &intersectPoint, &f1, &f2, &f3)

The accurate collision test for a point and face moving with constant velocity. Fills the intersectPoint, f1, f2 and f3 with the positions of the particles at the point of collision, if any.

```
bool IntersectMovingPointFace(Particle* p, Face* f,double* time,
                                          vec3D* intersectPoint, vec3D* f1, vec3D* f2,
vec3D* f3)
{

     vec3D v1 = f->p1->next.position - f->p1->curr.position;
     vec3D v2 = f->p2->next.position - f->p2->curr.position;
     vec3D v3 = f->p3->next.position - f->p3->curr.position;
     vec3D v  = p->next.position - p->curr.position;

     vec3D a1 = f->p2->curr.position - f->p1->curr.position;
     vec3D a2 = f->p3->curr.position - f->p1->curr.position;
     vec3D a3 = p->curr.position - f->p1->curr.position;
     vec3D d1 = v2 - v1;
     vec3D d2 = v3 - v1;
     vec3D d3 = v - v1;

     vec3D c1 = CrossProduct(d1,d2);
     vec3D c2 = CrossProduct(a1,a2);
     vec3D c3 = CrossProduct(d1,a2)+CrossProduct(a1,d2);

     double a = DotProduct(d3,c1);
     double b = DotProduct(a3,c1) + DotProduct(d3,c3);
     double c = DotProduct(a3,c3) + DotProduct(d3,c2);
     double d = DotProduct(a3,c2);

     const double coeff[]={d,c,b,a};
     double x[4];
```

```
	int ansTot = SolveCubicEquation(coeff,x,1e-13);

	double sol = 1000;
	double threshold = 1e-2;

	for(int i = 0;i<ansTot;i++)
	{
		if(x[i]<-0-threshold||x[i]>1+threshold)	continue;

		vec3D tf1 = f->p1->curr.position + (v1 * x[i]);
		vec3D tf2 = f->p2->curr.position + (v2 * x[i]);
		vec3D tf3 = f->p3->curr.position + (v3 * x[i]);

		vec3D tintersectPoint = p->curr.position + (v * x[i]);

		vec3D intersectNorm = CrossProduct(tf2-tf1,tf3-tf1);

		vec3D norm1 = CrossProduct(tf2 - tf1,intersectNorm);
		vec3D norm2 = CrossProduct(tf3 - tf2,intersectNorm);
		vec3D norm3 = CrossProduct(tf1 - tf3,intersectNorm);

		double dist1 = DotProduct(tintersectPoint - tf1,norm1);
		double dist2 = DotProduct(tintersectPoint - tf2,norm2);
		double dist3 = DotProduct(tintersectPoint - tf3,norm3);

		if((dist1<=0&&dist2<=0&&dist3<=0)&&(x[i]<sol||sol==1000))
		{
			sol = x[i];
			*intersectPoint = tintersectPoint;
			*f1 = tf1;
			*f2 = tf2;
			*f3 = tf3;
		}
	}

	if(sol==1000)	return false;

	*time = sol;

	return true;
}
```