Rigid Body Simulation

Rahil Baber

August 2006

Abstract

Rigid Body Simulation deals with trying to model real world physical situations computationally by applying Newton's laws of motion to perfectly rigid bodies. Work done in this area is used in the animation industry, haptic displays in robotics, and 3D games, so there is a real need for algorithms that work in realtime. The two main areas of research in rigid body simulation are collision detection and collision response. Collision detection involves using computational geometry to come up with algorithms with low complexities, as intersection detection can become computationally expensive when simulating a large number of bodies. In this thesis we will largely be looking at collision response. In the real world when objects collide they deform and it is this deformation that causes the forces between the objects. However simulating how objects deform takes a prohibitively long amount of time so instead the area of collision response looks at other empirical laws to calculate the forces, while keeping the objects rigid. A lot of the physics involved is no harder than what is taught in a high school mechanics course, but there are notable exceptions, and we find that when we deal with objects moving in 3D we have to include terms which allow for more interesting and un-intuitive movement such as gyroscopic stability. In this thesis we will look at the various schemes proposed to deal with collision response and the problems that arise from trying to implement such algorithms robustly.

Contents

1 Introduction

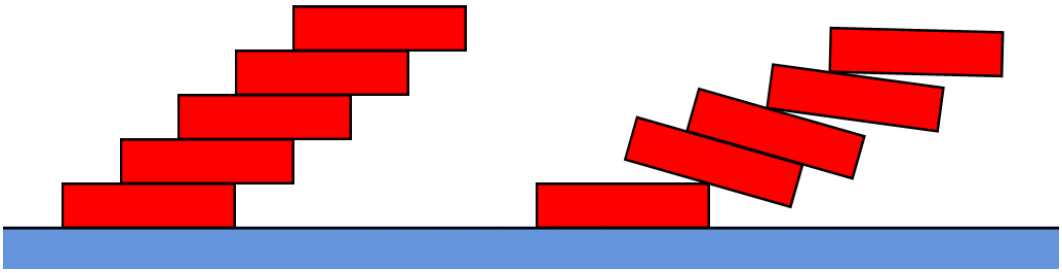


Figure 1: On the left we have an unstable stack of five identical bricks. On the right we see how the stack falls down under the influence of gravity

The area of classical dynamics largely stems from Newton's three laws of motion.

1. An object will remain at rest or in uniform motion unless compelled to change by the action of an external force
2. The rate of change of momentum of a body is equal to the resultant force acting on the body and is in the same direction.
3. For every action there is an equal and opposite reaction.

These are not very difficult concepts to understand, however to put them into practice, and create useful models and simulations is another matter. It turns out to be surprisingly difficult to do, and there are many programmers who have underestimated the challenges it poses. For example in Figure 1 we have a clearly unstable arrangement of five identical bricks, however it is not immediately apparent how this stack will fall down. As difficult as it is, it has many useful applications. Some of the more notable examples are those in 3D animations, engineering, haptic displays in robotics, and computer games. The last two of which often require the simulation to occur in real time. This thesis is largely concerned with applying classical dynamics to rigid bodies. In reality a body is never truly rigid but instead will deform on impact. It is this deformation that causes the forces. Such deformations can be accurately simulated using finite element methods, however this can take prohibitively long to simulate and so is not appropriate for a very complex simulation involving multiple bodies colliding. To avoid such problems we will be assuming that all objects are rigid, so objects that exhibit large scale deformations such as those made from materials like rubber and cloth are not suitable for this type of simulation. Also the period of impact between objects will be instantaneous even though in reality the period in which two objects collide and deform can be relatively long. With these and other approximations we can provide a reasonably accurate and fast simulation of very complex scenarios, not unlike that in Figure 1.

The area of rigid body simulation has two major areas, namely collision detection and collision response. The area of collision detection is largely based in computational geometry. Its purpose is to check whether any of the rigid bodies are in contact, and if so it is often useful to know things like the depth of the penetration, points of intersection, direction a force would act. Once we know two bodies are in contact or in other words colliding we need to consider collision response algorithms. In collision response we have to work out appropriate impulses and forces between the objects in order to

produce a realistic progression of the scenario according to the laws of physics, and the assumptions we are trying to enforce. After we have identified all the bodies in contact we apply impulses that keep the objects from intersecting and cause them to bounce away from each other. Then we calculate the bodies' motions due to external non-contact forces such as gravity. After evolving the bodies we repeat the procedure by testing for new object intersections. The last step of calculating the objects' motions due to non-contact forces is relatively quick and straightforward; the vast majority of the computation takes place (in roughly equal amounts) during the stages of identifying collisions and calculating an appropriate response.

This thesis is largely concerned with problems of robustness in creating an algorithm that accurately simulates rigid body motion. We will begin with a brief overview of collision detection algorithms, but the main body of work will be in collision response.

2 Collision Detection

Before we can resolve any collisions we must first detect them. In this section we will look very briefly at some of the methods employed in detecting collisions between the objects we are simulating. This is a very important area of rigid body simulation as the amount of time spent detecting collisions is roughly equal to the time taken to resolve them, making the need for efficient detection algorithms essential.

Firstly it should be noted that most of the algorithms in this area are for objects which are polyhedrons or polygons. All objects can be represented roughly by polyhedra, and most 3D graphics software (such as DirectX and OpenGL) in essence can only render triangles. This has meant that the boundary of an object is often represented by a series of triangles and this method of modeling complex shapes has become a widely accepted standard. In fact a lot of the algorithms in collision detection also require the polyhedra to be convex as well. Fortunately we can always split a non-convex polyhedron into a group of convex polyhedra. One way to do this is by using binary space partitioning, where we essentially recursively subdivide the polyhedron into convex smaller polyhedrons by using partitioning planes [16]. Convex decomposition is still an active area of research. This is because most partitioning methods tend to produce a large number of convex pieces, which is inefficient, and finding a minimal partitioning is known to be NP-hard. Research is also being done on developing detection algorithms that do not impose the condition of convexity on the objects, as this is advantageous when simulating objects which have a large minimal partitioning, such as torus-like objects.

Rather than test every pair of objects to see whether they intersect or not, we can save some time by using a space partition. We can partition the space that the objects will be moving in into a series of cells. For each cell we can maintain a list of which objects reside currently in it (wholly or partially). It is relatively quick to update these lists as the simulation progresses. Using this information we can skip testing whether two objects intersect if they do not occupy a single cell in common. This can greatly speed up collision detection when we are simulating a large number of objects. There are various different ways we can partition the space: some of the more notable ways are by Voxel Grids, Octrees, k-d Trees, and BSP Trees.

Another way of quickly testing object intersections is by using bounding volumes. A bounding volume is a primitive shape, often a sphere or a cuboid, which totally encloses the object. This means that if two objects' boundary volumes don't intersect then the objects themselves can't intersect. For bounding volumes to be most effective they should fit the object they encompass as tightly as possible, and testing whether two boundary objects intersect should be very quick to determine. Testing whether two spheres intersect is certainly a simple operation, however in general they don't fit most objects very tightly. Boxes provide a much tighter fit, but testing whether they intersect is more complicated. A relatively quick way of testing when two boxes intersect is provided by using the separating axis theorem [6]. It states that if two convex objects do not intersect then there exists a separating axis, i.e. an axis on which the projections of the two objects do not overlap.

We can speed up collision detection by storing and maintaining the shortest distance between the boundary of every pair of objects. The Lin-Canny [13] and Gilbert-Johnson-Keerthi [7] algorithms both do this. We can maintain for each pair of objects which parts of their boundaries are the closest together. When the objects move we can assume the closest pair of points lie close to the old pair of closest points, making it quick to find an update.

There are a variety of other methods and algorithms, each with their own advantages and disadvan-

tages, but hopefully this has been a brief overview of some of the main ideas and topics used in this area. More detailed information about collision detection can be found in [4, 5, 8, 21]. We will now proceed to the main focus of this work, collision response.

3 Collision Response

3.1 Introduction

We will begin this section with a brief look at the various different methods of collision response, and their various advantages and disadvantages. Then we will go through some preliminary definitions and assumptions that we will be making. In that section we will also discuss the way we will store the objects and how to calculate their mass distribution properties. Following this we will look at how these properties are used in calculating an object's motion under no external forces as well as how to deal with a simple collision. Finally we will end by formulating a robust algorithm that deals with complex simulations when friction is not present.

Finite Element Methods

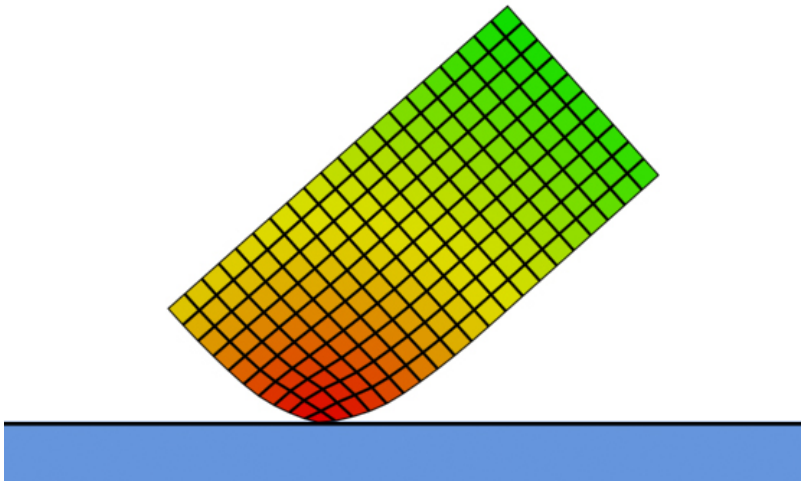


Figure 2: A rectangle colliding with a rigid floor. The rectangle is modeled by splitting it into 200 smaller quadrilateral elements.

This is the most accurate of all the collision response methods. It decomposes the body into a large but finite number of smaller elements and then proceeds to work out the stress and strain on each of the elements. In this way it can accurately work out the forces that occur during impact when the bodies deform. However this method is computationally expensive and simulations can not be done in real-time. This is fine for testing engineering structures but clearly is of little use when the simulation is meant to be interactive. For this reason we trade accuracy for speed and assert the condition that bodies should be modeled as rigid. The other alternatives that we will discuss are all specific to rigid body simulation.

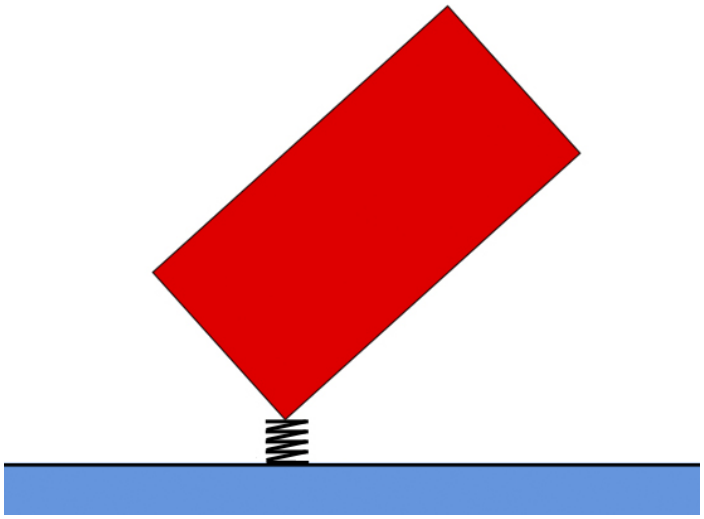


Figure 3: A rectangle colliding with the floor. In the spring model we imagine invisible springs pushing the objects apart. (For clarity the rectangle has been drawn not intersecting the floor.)

Penalty Methods

These methods are the most popular of the approaches largely due to the relative ease in which they can be implemented. When two objects are found to be colliding or intersecting, these methods query the resulting geometry for information such as penetration depth or the volume of intersection. They then use this to calculate an appropriate force between the objects. A common example would be the spring model, where we imagine there is an invisible spring between all contacts forcing them apart. So the greater the depth of penetration the larger the force should be that tries to push the objects apart.

The main problem with these sorts of methods are that they often require setting and tweaking a lot of variables manually to get realistic looking simulations. For example in the spring model all the properties of all the springs have to be set. This has to be done in an ad hoc manner, as the properties of the springs do not correspond to any familiar physical properties of the bodies being simulated. Also values that work well in one situation may produce unrealistic results when the initial configuration is slightly modified. Ideally we would prefer a situation where this does not occur and the same set of values can be used in all simulations. Although penalty methods seem reasonable there is little evidence that they accurately simulate what occurs in real life. Nevertheless they can be used effectively where accuracy is not an issue, where merely looking convincing is sufficient.

In the case of the spring model, as the stiffness of the spring is increased the collision time should decrease and we would hope that the limiting case is equivalent to the impulse methods that we will discuss next. However as the stiffness of the springs is increased the differential equations we need to solve become “stiff” in themselves making them increasingly more difficult to solve. So accuracy can be hard to achieve in real time.

Springs can also be used to force constraints, like a roller-coaster forced to follow the path laid out by track. In such an example springs would exist not just when there is penetration but when any object is close to the track. The springs push the roller-coaster away from the track if it penetrates it but they can also be used to pull it towards the track if it starts to deviate away due to numerical errors.

If the speed of the roller-coaster is sufficiently large we would see that in real life the roller-coaster would break away from the track and follow a ballistic trajectory. In this case the spring between the roller-coaster and the track would have to be destroyed. Choosing when a spring should be destroyed in such simulations is another problem this model has. It is not at all clear when it is a small deviation and the spring should pull, or it wants to break free and the spring should be destroyed.

Despite these problems it is relatively straightforward to incorporate frictional forces into a model using springs. We merely add a spring that lies in the direction that opposes motion and is perpendicular to the spring causing objects to move away from each other. To incorporate a more advanced model of friction that accounts for static friction requires a bit more work and involves a slightly more complicated setup of springs but the principles are the same [9]. Unfortunately the springs that simulate friction suffer from the same problems that the springs prohibiting penetration exhibit. They both require constants that need to be set heuristically, and often require manual adjustments on a per simulation basis. However, as we will see, incorporating friction accurately is a very hard problem.

Impulse-Based Methods

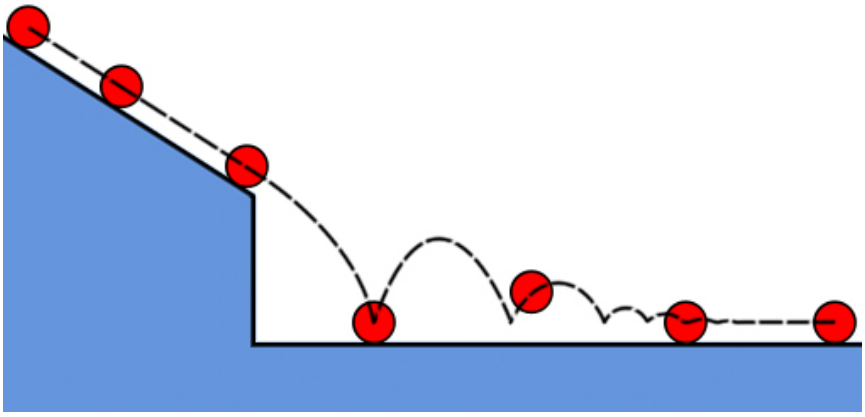


Figure 4: A ball rolling off a slope: its trajectory is given by the dashed line. The various different types of contact between the ball and the terrain are all modeled through impulses only.

Impulse-based methods model all the forces between bodies through a series of impulses. For example in Figure 4 the ball receives a series of impulses as it bounces and collides with the floor after falling off the slope. Even when the ball has stopped bouncing and is rolling we still model the forces between the ball and floor as impulses. Imagine a block resting on the ground. Instead of there being a constant force being applied upwards counteracting gravity and keeping it at rest, in the impulse model the block receives a rapid series of impulses being applied at each of its corners. It is these impulses that keep it from accelerating downwards. However they will also cause the block to vibrate rapidly but the values can be chosen such that the amplitude of the vibrations is less than a pixel, hence it appears stationary to the user. Although this seems like a departure from what happens in reality, it is in most cases an acceptable trade for speed. In fact if we take an average of the impulses applied over a time period we should get the same value as the force applied by the constraint-based methods described below. So in some sense the impulse method preserves the overall important physical quantities, such as the average velocity of the object and the average forces on the object.

Impulse-based methods have a clear advantage over the penalty methods as they do not require the manual tweaking of a large number of variables. However they can not completely handle a prolonged static contact, such as a brick resting on the floor. To remedy this a variable does need to be set, which artificially changes the properties of the object (specifically the coefficient of restitution) to avoid the object sinking and penetrating into the floor. Despite this it is still quite straightforward to implement. For simulations with long periods of static contact, constrained-based methods are better and we will discuss them in the next section. However where we have objects moving continually and never remaining static for too long, the impulse method does best. For further information on this approach see [15].

Constraint-Based Methods

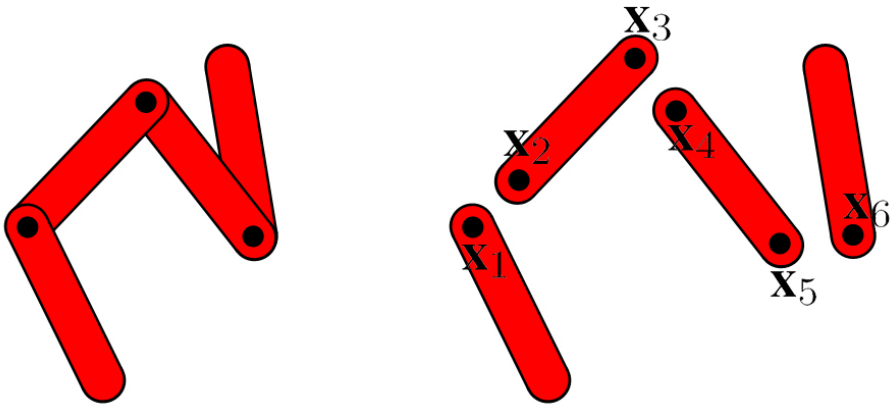


Figure 5: On the left we have four rigid bodies linked together by three hinges. The positions of the hinges on each body are given by the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_6$, for clarity the linked bodies are drawn on the right not intersecting.

Constraint-based methods work by applying a series of constraints on the rigid bodies [20]. For example consider the rigid bodies in Figure 5 which are hinged together. We calculate the forces on each of the four bodies and integrate them over time allowing each part to move as if it was not linked. The bodies appear to be linked when a simulation is run because, when we calculate the forces, we do so such that the constraints $\mathbf{x}_1 = \mathbf{x}_2,\ \mathbf{x}_3 = \mathbf{x}_4,\ \mathbf{x}_5 = \mathbf{x}_6$ will remain satisfied. Similarly we can enforce other constraints, such as bodies being forbidden to intersect. In this case we don't have a permanent constraint, like in the hinged example, but we get temporary constraints that appear when bodies come into contact. Also the constraints often take the form of inequalities such as *velocity of separation* ≥ 0 rather than strict equality constraints. The constraints, as we will see, are usually linear with respect to the forces, so we can use linear programming methods to calculate the forces.

Constraint-based methods have the disadvantage that we have to calculate the forces a large number of times to reduce the numerical error caused by integration. Since we may be trying to solve a large linear program each time we calculate the forces, this can cause the whole process to become very slow. However in modeling scenarios such as articulated and hinged objects it easily outperforms the impulse-based method which has to cope with a very large number of collisions between objects, as they are in continual contact with each other. The major advantage of this method is the lack of simulation-dependent variables that we need to set, however this unfortunately makes it the hardest of

all the methods to implement. Like the impulse method it makes use of the coefficient of restitution and other physical attributes of the bodies being simulated. This makes the simulation's approximation of reality more plausible.

The impulse and constrained-based methods complement each other well, and ideally it would be preferable to implement both, as in the scenarios where one performs badly the other may perform well. Although the impulse-based methods have notable strengths I will be adopting the constrained-based methods to handle collision response, as it seems to handle objects at rest more robustly and requires the least number of simulation-specific constants. We will look in more depth at the details of this approach in the rest of this thesis.

3.2 Preliminary Definitions and Assumptions

Before we delve too deeply into creating and implementing a robust collision response algorithm we should be clear what assumptions we are working under and what definitions and conventions we will be using.

Firstly we are going to assume that all bodies are perfectly rigid, as this will allow the simulation to be run at speeds which allow real time interaction with the user. A consequence of this is that when two bodies collide the collision will happen instantaneously. This means all forces that are applied to stop the bodies from intersecting will occur over an infinitesimal time period and consequently can be thought of as impulses. The infinitesimal time period also means non-impulsive forces such as gravity will have no effect during the collision and that the positions of the bodies will remain constant throughout the collision. The rigid body assumption also allows us to fix some of the physical attributes of the body.

We will not be considering impacts with friction. Although friction is very important for a realistic looking simulation, its inclusion leads to a number of issues. These problems will be discussed in more detail in a later section. In this section we will be assuming all collisions are frictionless, which in any case is a good starting point in any collision response algorithm. We will also be assuming that bodies are not joined or hinged in any way. It is not difficult to incorporate hinged objects (especially in the constraint-based methods we will be using), however this complicates the equations and so, for clarity, we will be assuming all bodies are not intrinsically linked.

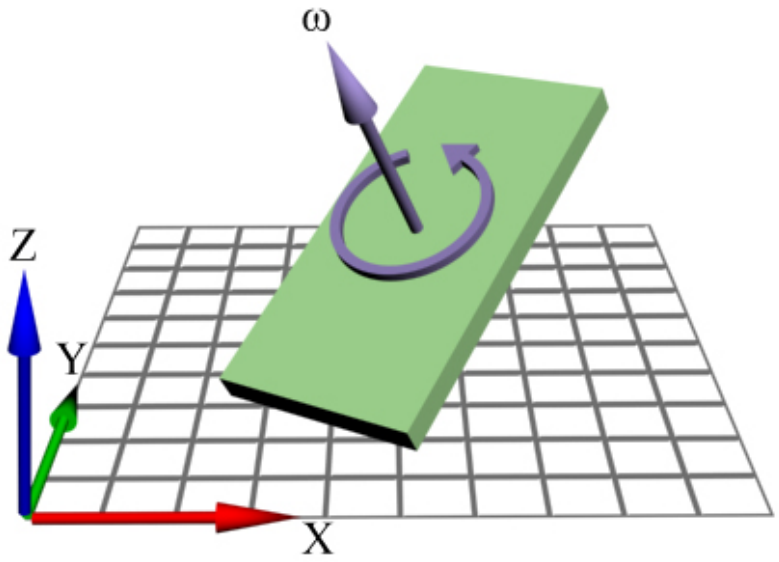


Figure 6: The arrows point in the direction of positive values.

Throughout this thesis we will be using the conventions illustrated in Figure 6. It is important to define a set of conventions and to stick to it, as it will greatly reduce confusion later and minimize sign errors when it comes to implementation. As the diagram shows we will be using the right-handed coordinate system of axis. We will assume that the Z direction will represent height so when we talk of gravity we will assume it is acting in the direction given by the vector $\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$. We may also discuss objects colliding or coming to rest on the floor or ground. For our purposes we will assume the floor

is immovable, stationary, and it lies in the XY plane with Z=0. The floor is illustrated in Figure 6 by a grid.

When working out the motion of an object we usually decompose it into a translational component and a rotational component. Since we are assuming the bodies are rigid we can assume the centre of mass is fixed relative to the object. Throughout the rest of this section we will be assuming that all rotations will be taken around the centre of mass. Although the equations will still give the same result if we calculate the rotational component about another point, choosing the centre of mass has the advantage of making the equations appear simpler as we usually have fewer terms to deal with. In two dimensions in order to rotate an object all we require is a pivot point (the centre of mass) and an angle. In three dimensions however we rotate about an axis. As a result angular velocity, which is usually denoted by the symbol ω, is a scalar in 2D, but in 3D where things are more complicated we represent it as a vector. The direction of the vector gives the axis of the rotation (assuming it passes through the centre of mass) whereas the magnitude gives us the number of radians per second. The convention that we will use is that positive values represent anticlockwise rotations. In the 3D case we rotate the object anticlockwise in the plane perpendicular to the vector as if we were looking at the object in the opposite direction to that which is represented by the vector. Figure 6 shows a cuboid with an angular velocity vector ω and a circular arrow showing the direction of rotation that this represents.

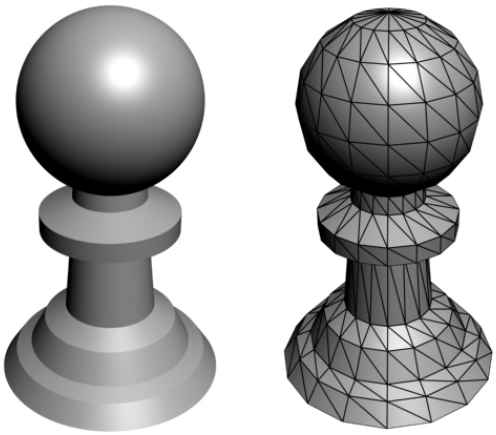
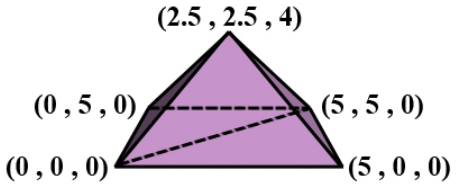


Figure 7: On the left we have a chess pawn. On the right we see that we can approximate its surface by a series of triangles.

For the purposes of collision detection and response we need to know each object's boundary. For a sphere its boundary can be calculated from the radius. Similarly for other primitive objects (such as cuboids, cones, cylinders) we can store the information about their boundaries by a few scalar values. However for each type of primitive object we need to write code which deals with it separately. This is not a major problem and this approach has the advantage that boundaries are exactly represented. However we can only do this for a few simple objects: there are many other more complex shapes that we may want to simulate. One of the most common ways to represent such complex shapes is to use a triangle mesh, as illustrated in Figure 7. The triangles that make up the mesh can be stored by recording the three position vectors of their vertices relative to some origin (that is common to the entire mesh). From Figure 7 it is clear that the triangles share their vertices with other triangles. Hence it is more efficient to store all the vertices' position vectors in an array separately and then each triangle can be defined by three integers which are index references to the array of vertices. This is illustrated for a square-based pyramid in Figure 8. For example Triangle[4] represents a triangle whose vertices' positions are stored in Vertex[1], Vertex[5], and Vertex[3]. Note that the elements of the array which

defines the triangles are ordered sets. Assuming the three vertices of a triangle are given by vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ (in that order) which are not collinear, we can work out a normal to the plane that the triangle lies in by the equation $(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$. If we interchanged $\mathbf{a}$ and $\mathbf{b}$ in the equation then the normal would point in the opposite direction, hence the order in which we take the vertices matters. The triangles' vertices in Figure 8 have all been ordered so that the normals point outwards away from the object. For example Triangle[5] gives us $\mathbf{a} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 5 \\ 5 \\ 0 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 5 \\ 0 \\ 0 \end{pmatrix}$ which results in the normal vector $\begin{pmatrix} 0 \\ 0 \\ -25 \end{pmatrix}$ which points away from the pyramid. Before, the three vertices just defined a plane, but with the additional information the ordering gives us it can be used to define a half-space. This is useful, as for convex objects if a point lies within each of the half-spaces given by each of the triangles it must lie in the object itself. Also for rendering purposes if a triangle's vertices would appear on the screen in a clockwise order then we can ignore it as it would be overdrawn by triangles with an anticlockwise ordering (this assumes the boundary of the object completely splits the space in two). We will be assuming the convention, for ordering the vertices in triangle meshes, given in Figure 8, throughout the thesis.



Vertex Array

Vertex[1] = (0 , 0 , 0)
Vertex[2] = (5 , 0 , 0)
Vertex[3] = (0 , 5 , 0)
Vertex[4] = (5 , 5 , 0)
Vertex[5] = (2.5 , 2.5 , 4)

Triangle Array

Triangle[1] = {1 , 2 , 5}
Triangle[2] = {2 , 4 , 5}
Triangle[3] = {3 , 5 , 4}
Triangle[4] = {1 , 5 , 3}
Triangle[5] = {1 , 4 , 2}
Triangle[6] = {1 , 3 , 4}

Figure 8: A square-based pyramid composed of 5 vertices and 6 triangles. The coordinates of the vertices are written next to them. The pyramid is stored as two arrays. One array holds the coordinates of all the vertices and the other holds ordered sets of three integers. These integers are index references to the vertex array.

When dealing with 3D objects we often refer to two spaces, the model space and the world space. The model space refers to the space the 3D object is defined in and stored in. So for example in Figure 8 the position vectors are vectors in the model space. The world space on the other hand is the space that all the objects will eventually be transformed into. It can be thought of as the space our simulation will occur in. Each object has its own model space but there is only one world space. To transform an object into the world space we need to know what orientation the object has in the world space and its position. The orientation can be characterized by a 3×3 matrix. Since it is a rotation we can further restrict it to being orthogonal with determinant equal to one (we will be assuming that the object will not need to be scaled or flipped). The position is given by a vector. So a vertex in model space given by a vector $\mathbf{x}$ has coordinates $\mathsf{R}\mathbf{x} + \mathbf{r}$ in the world space, where R is the rotation matrix and $\mathbf{r}$ is its new position. As a simulation progresses the objects will obviously rotate and move through the world space. We keep track of exactly where they are by storing, for each object, their position vectors $\mathbf{r}$ and their orientation matrices R. After each discrete time step we calculate new values for each R and $\mathbf{r}$ and update the values stored accordingly. Doing this calculation robustly is precisely what this thesis is about. Although we we will be storing objects position and orientation using a vector and a 3×3 matrix there are other ways of keeping track of this information. Some of the more notable ways are to use quaternions or the three Euler angles to represent the orientation, or to use a 4×4 matrix to keep track of the position and orientation simultaneously. We will discuss their relative advantages and disadvantages in a later section, but for clarity, at least for now, we will continue with our 3×3 matrix and vector pair. Also we will be assuming that the origin of each object's model space coincides with

its centre of mass, as this will simplify the equations we will be using. If this is not the case we can calculate where the centre of mass is in model space, and offset all the position vectors accordingly. Forcing the origin and centre of mass to coincide will have the added effect of speeding up the collision response algorithm as it will have fewer calculations to do.

As we have discussed, for collision detection all we need is the boundary of the object and its position in the world space. For collision response we also need to know how much mass each object has and how it is distributed over the object. Fortunately this information on mass distribution can be reduced to two quantities, the centre of mass, and the inertia tensor. The inertia tensor is given by a 3×3 matrix. This matrix is calculated by choosing a point and three orthogonal axis (that go through the point). The inertia tensor is real and symmetric. This means there is a choice of axes which will diagonalise the matrix. Such axes are known as the principal axes and each has a respective diagonal term known as the principal moment of inertia. As we are taking all rotations to be about an axis going through the centre of mass, this will be our choice of point to calculate the inertia tensor from. As well as the centre of mass coinciding with the origin in model space, we can also force the condition that the X, Y, and Z axes correspond to the principal axes. We will not however be assuming that this is enforced. Though it would speed up calculations slightly it makes setting up a simulation more complicated, and will not make the equations we will be using any clearer. In 2D, where things are simpler, we do not require the entirety of the inertia tensor. In 2D motion the object rotates about an axis perpendicular to the plane its motion is constrained to. This is always assumed to coincide with a principal axis (though this is often not explicitly stated). Because of this we only need to know one principal moment of inertia for 2D motion rather than the whole tensor. This is often referred to as the moment of inertia. We will show later that the more familiar 2D equations of motion arise as a special case of the 3D versions. Please note that by 2D motion we do not mean the object is necessarily a lamina. For example, a 3D sphere thrown through the air will follow a ballistic trajectory. Its centre of mass will trace out a parabola, this will lie completely in some 2D vertical plane, and hence the motion of the ball is 2D in nature even though the object itself is 3D.

We can of course manually set the values relating to mass distribution but this can lead to unrealistic simulations. See Video 01 and 02 on the accompanying cd-rom (note that the simulations are done assuming no friction, hence there are no horizontal forces and the centre of mass will have fixed x and y coordinates, which may look slightly odd). For example we could set the centre of mass to lie outside the object which is an impossible situation if the object is convex. Setting the values for the inertia tensor is even more problematic, as our intuition is poor, making it hard to guess reasonable values. To avoid such problems it is useful to have an algorithm to calculate the centre of mass and inertia tensor automatically assuming a uniform distribution of mass. We will go over the necessary calculations below.

Suppose the object is given by a triangle mesh (which exhibits our vertex ordering convention), it is a convex solid, and uniformly distributed. In our later calculations we will be assuming the model space's origin and centre of mass coincide. If this is not the case we will need to calculate the position of the centre of mass in model space. We do this by choosing a point inside the object, and using it, we decompose the object into a series of tetrahedrons. The tetrahedrons are defined by a triangle from the triangle mesh and the point we chose inside the object. This will give us the same number of tetrahedrons as there are triangles making up the mesh. Figure 9 shows such a tetrahedron, its vertices have positions in model space given by the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and $\mathbf{d}$. Note that all the tetrahedrons will have one vertex at $\mathbf{a}$, and the order of $\mathbf{b}, \mathbf{c}, \mathbf{d}$ is chosen according to our convention. The centre of mass of this tetrahedron is $\frac{1}{4}(\mathbf{a}+\mathbf{b}+\mathbf{c}+\mathbf{d})$ and its volume is $\frac{1}{6}(\mathbf{b}-\mathbf{a}) \cdot (\mathbf{c}-\mathbf{a}) \times (\mathbf{d}-\mathbf{a})$. Let N be the total number of tetrahedrons, and let $V_i, \mathbf{c}_i$ represent the volume and centre of mass of the ith tetrahedron

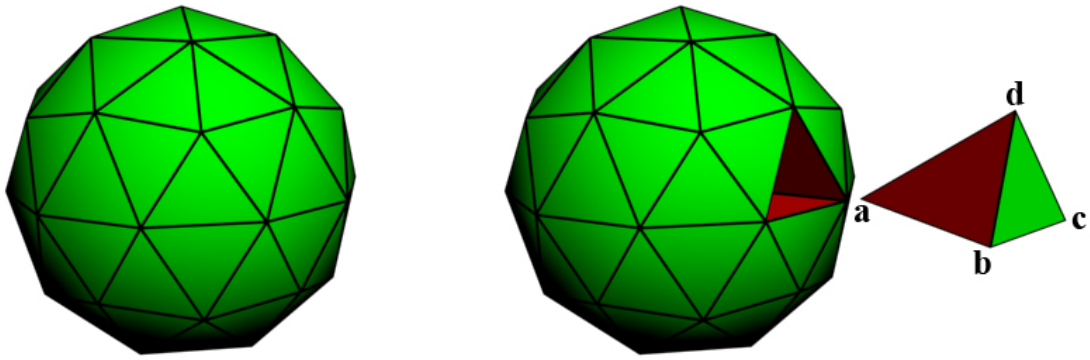


Figure 9: On the left we have a solid convex object. We can decompose it into a series of tetrahedrons. On the right we show such a tetrahedron.

respectively, then

$$\text{Centre of Mass of the Entire Uniform Solid Object} = \frac{\sum_{i=1}^{N} V_i \mathbf{c}_i}{\sum_{i=1}^{N} V_i}$$

If the object is in fact hollow instead of solid we can similarly decompose it into triangles. Each triangle is simply a member of the triangle mesh. Let us take the triangle given by $\mathbf{b}, \mathbf{c}, \mathbf{d}$ in Figure 9 as an example. Note that the point at $\mathbf{a}$ is no longer relevant. The triangle's centre of mass is $\frac{1}{3}(\mathbf{b}+\mathbf{c}+\mathbf{d})$, and its area is given by $\frac{1}{2}|(\mathbf{c}-\mathbf{b}) \times (\mathbf{d}-\mathbf{b})|$. To work out the centre of mass of the whole object we again take a weighted average. A_i represents the area and $\mathbf{c}_i$ represents the centre of mass of the ith triangle.

$$\text{Centre of Mass of the Entire Uniform Hollow Object} = \frac{\sum_{i=1}^{N} A_i \mathbf{c}_i}{\sum_{i=1}^{N} A_i}$$

Once we know where the centre of mass is in model space we can offset all the values so that the origin is forced to coincide with the centre of mass.

Working out the inertia tensor is a bit more complicated. Firstly we will be explicitly assuming that the centre of mass is at the origin. The inertia tensor, like the centre of mass, can be calculated from simpler constituent components like tetrahedrons. If the object is convex then the origin will lie inside the object, making it a convenient choice of point to use when splitting into tetrahedrons. Again we will be assuming the object is uniformly distributed, solid, has total mass M, and is convex. The inertia tensor for a solid object with a continuous mass distribution is given by

$$\int_V \rho(x,y,z) \begin{pmatrix} y^2+z^2 & -xy & -xz \\ -xy & x^2+z^2 & -yz \\ -xz & -yz & x^2+y^2 \end{pmatrix} dx\, dy\, dz$$

where V is the volume of the object, and $\rho(x,y,z)$ is the density of the object at the point (x,y,z). Since the object has uniform density $\rho(x,y,z)$ is a constant and can be taken out of the integral. The

inertia tensor is therefore

$$\frac{M}{\sum_{i=1}^{N} V_i} \times \sum_{i=1}^{N} \int_{V_i} \begin{pmatrix} y^2+z^2 & -xy & -xz \\ -xy & x^2+z^2 & -yz \\ -xz & -yz & x^2+y^2 \end{pmatrix} \, dx \, dy \, dz \tag{1}$$

where V_i is the volume of the ith tetrahedron. Let us assume that the tetrahedron is given by $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ as in Figure 9 (note that $\mathbf{a} = \mathbf{0}$ now). The volume is still given by $\frac{1}{6}(\mathbf{b}-\mathbf{a}) \cdot (\mathbf{c}-\mathbf{a}) \times (\mathbf{d}-\mathbf{a})$. To work out the integral part of the calculation it is sufficient to work out

$$\int_{V_i} \begin{pmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{pmatrix} \, dx \, dy \, dz$$

Each of the elements forming the matrix in our original integral are some linear combination of the elements in the above integral. We calculate the integral by, instead of integrating along the axes given by the vectors $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, changing to those given by $(\mathbf{b}-\mathbf{a}), (\mathbf{c}-\mathbf{a}), (\mathbf{d}-\mathbf{a})$. Now consider the point $\mathbf{a}+p(\mathbf{b}-\mathbf{a})+q(\mathbf{c}-\mathbf{a})+r(\mathbf{d}-\mathbf{a})$ where $p, q, r \in \mathbb{R}$. When $0 \leq p, q, r$ and $p+q+r \leq 1$ this point lies in the tetrahedron. These limits are much more convenient to work with. Changing variables gives

$$(\mathbf{b}-\mathbf{a}) \cdot (\mathbf{c}-\mathbf{a}) \times (\mathbf{d}-\mathbf{a}) \int_0^{1-p-q} \int_0^{1-p} \int_0^{1} (\mathbf{a}+p(\mathbf{b}-\mathbf{a})+q(\mathbf{c}-\mathbf{a})+r(\mathbf{d}-\mathbf{a})) \\ (\mathbf{a}+p(\mathbf{b}-\mathbf{a})+q(\mathbf{c}-\mathbf{a})+r(\mathbf{d}-\mathbf{a}))^T \, dp \, dq \, dr \tag{2}$$

To evaluate this new integral we need to calculate integrals that look like $\int_0^{1-p-q} \int_0^{1-p} \int_0^1 pq \, dp \, dq \, dr$, they are given below.

$$\begin{aligned}
\int_0^{1-p-q} \int_0^{1-p} \int_0^1 dp \, dq \, dr &= \frac{1}{6} \\
\int_0^{1-p-q} \int_0^{1-p} \int_0^1 p \, dp \, dq \, dr &= \int\int\int q \, dp \, dq \, dr &&= \int\int\int r \, dp \, dq \, dr &&= \frac{1}{24} \\
\int_0^{1-p-q} \int_0^{1-p} \int_0^1 pq \, dp \, dq \, dr &= \int\int\int pr \, dp \, dq \, dr &&= \int\int\int qr \, dp \, dq \, dr &&= \frac{1}{120} \\
\int_0^{1-p-q} \int_0^{1-p} \int_0^1 p^2 \, dp \, dq \, dr &= \int\int\int q^2 \, dp \, dq \, dr &&= \int\int\int r^2 \, dp \, dq \, dr &&= \frac{1}{24}
\end{aligned}$$

Substituting the above into (2), expanding and simplifying gives

$$\int_V \begin{pmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{pmatrix} \, dx \, dy \, dz$$
$$= \frac{1}{120}[(\mathbf{b}-\mathbf{a}) \cdot (\mathbf{c}-\mathbf{a}) \times (\mathbf{d}-\mathbf{a})][(\mathbf{a}+\mathbf{b}+\mathbf{c}+\mathbf{d})(\mathbf{a}+\mathbf{b}+\mathbf{c}+\mathbf{d})^T + \mathbf{a}\mathbf{a}^T + \mathbf{b}\mathbf{b}^T + \mathbf{c}\mathbf{c}^T + \mathbf{d}\mathbf{d}^T]$$

Using this and (1) we can easily calculate the moment of inertia for a tetrahedron and the entire object.

By a similar process we can also calculate the moment of inertia for a uniformly distributed hollow object. Again we will be assuming the centre of mass and the origin of the model space coincide, and

the total mass is M. Instead of (1) we now use

$$\frac{M}{\sum_{i=1}^{N} A_i} \times \sum_{i=1}^{N} \int_{A_i} \begin{pmatrix} y^2+z^2 & -xy & -xz \\ -xy & x^2+z^2 & -yz \\ -xz & -yz & x^2+y^2 \end{pmatrix} \, dx \, dy \, dz$$

where A_i is the area of the ith triangle. If a triangle is given by $\mathbf{b}, \mathbf{c}, \mathbf{d}$ then its area is $\frac{1}{2}|(\mathbf{c}-\mathbf{b})\times(\mathbf{d}-\mathbf{b})|$. Similarly we have

$$\int_A \begin{pmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{pmatrix} \, dA$$
$$= \frac{1}{24}|(\mathbf{c}-\mathbf{b}) \times (\mathbf{d}-\mathbf{b})| \times [(\mathbf{b}+\mathbf{c}+\mathbf{d})(\mathbf{b}+\mathbf{c}+\mathbf{d})^T + \mathbf{b}\mathbf{b}^T + \mathbf{c}\mathbf{c}^T + \mathbf{d}\mathbf{d}^T]$$

Our choice of point used to decompose an object into tetrahedrons does not appear in any of the equations relating to a uniformly distributed hollow object, and so is completely irrelevant. Also it is not too hard to see that the condition that the object be convex can also be dropped for hollow objects. When the object is solid, the convexity condition and the choice of a point that lies inside the object, are there to give an obvious way to decompose the object into tetrahedrons. Surprisingly these conditions can both be dropped even when the object is solid and not hollow. We still need to choose a point but it need not lie inside the object. The object does not have to be convex and may even have holes like a torus. The only condition required, for the equations to hold, is that the boundary splits the space into two regions, i.e., a region inside the object and a region outside. For a hollow object this may not hold true, but the boundary of a solid object should always satisfy this condition. The reason the equations still work is due to the property that we can work out the centre of mass and inertia tensor of the overall object by summing its decomposition of tetrahedrons. As well as summing we can also subtract tetrahedrons. So we are allowed to decompose an object into a series of tetrahedrons with "positive" and "negative" volumes. This is illustrated for a pyramid in Figure 10, which is split into two tetrahedrons with positive volumes and two with negative volumes. Since we are working with a triangle mesh that has its vertices ordered in a special way (see Figure 8) the sign of the volume is automatically taken care of and the equations will still hold.

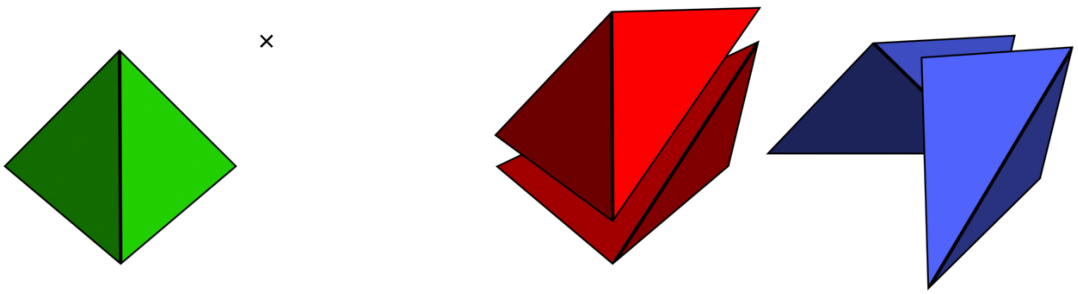


Figure 10: On the left we have a solid triangle-based pyramid and a point outside the pyramid whose position is indicated by a cross. We can as usual decompose the pyramid into 4 tetrahedrons that share a common vertex at the cross (shown on the right). Each tetrahedron corresponds to a triangular face of the original pyramid on the left. However the 2 tetrahedrons on the far right have "negative" volume.

In order to reduce numerical inaccuracies it is preferable to choose a point that avoids long thin tetrahedrons. So although we are free to choose a point outside the object often a point inside the object

will yield more accurate values. For speed and convenience the origin of the model space is often a good choice for decomposition purposes. However since the inertia tensor and centre of mass only needs to be computed once speed is usually not a major concern. A better choice is the average of the vertices' positions that make up the object. When working out the inertia tensor the origin should be at the centre of mass, which is also a good choice of point that will reduce long thin tetrahedrons. This has the added advantage of making the equations simpler and so quicker to evaluate. Another numerical accuracy issue is that of calculating the inertia tensor from a triangle mesh approximation of the boundary. Unfortunately the inertia tensor is significantly affected by mass at a large distance away from the centre of mass, even if the mass in question is small. For this reason the mass near the boundary of the object is quite important. For objects that are angular and have flat faces, like boxes, this is not a problem. For objects with curved surfaces, like spheres, the approximation to a triangle mesh can have a noticeable impact on the inertia tensor. Obviously the more triangles that are used to approximate the surface the smaller the error will be, but the more slowly the simulation will run. Of course we do not have to force ourselves to use this method to calculate the inertia tensor and centre of mass. We can set it to a more accurate value manually and use a less refined mesh for the purposes of the rest of the simulation. Another common way to calculate the inertia tensor is to use the Monte Carlo method to randomly sample points that lie inside the object then approximate the object as a series of particles. Although this method is simple (hence its popularity), it doesn't give such accurate results, and calculating exactly the centre of mass and inertia, as we have shown, isn't that complicated. See [11, 12, 15] for other ways of computing the mass properties of an object.

In the next section we will look at how we use the inertia tensor to calculate a body's motion. We will also describe a way to handle simple collisions via impulses.

3.3 Motion Under No External Forces and Impulse-Based Collisions

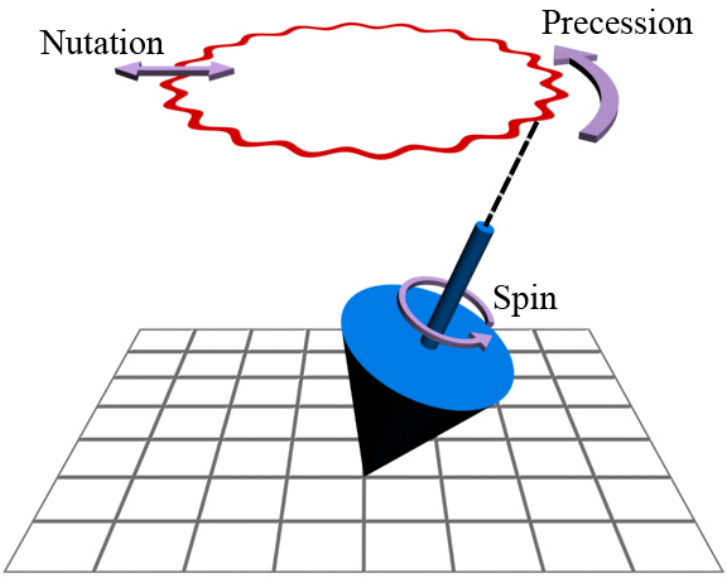


Figure 11: The bottom of the spinning top remains fixed if there is sufficient static friction. The axis of rotation however moves through the curve displayed above the top.

In 2D motion the rotation is wholly restricted about one axis which has its direction fixed and is perpendicular to the plane in which the object is free to move in. This axis usually corresponds to a principal axis. In 3D however we have more freedom and the axis of rotation often moves rather than staying fixed. This can be witnessed by looking at a spinning top, see Figure 11 and Video 03 on the accompanying cd-rom. The axis of rotation goes through motion known as precession and nutation. The stability caused by such gyroscopic motion can only be explained when looking at the equations of motion in 3D, consequently the equations are more complex than their 2D counterparts. To begin with, let us look at the equations of how an object moves under no external forces in 3D. The equations we will need for our simulation do not tend to appear in the standard text books, for maths or computer science, as they are often deemed to be too complicated or too generalized. For this reason we will be deriving them ourselves.

Equations of Rotational Motion

Suppose we convert a vector $\mathbf{r}$ in model space to the world space, as described in Section 3.2, via the matrix R (which represents the new orientation) and vector $\mathbf{p}$ (representing the position of the object's centre of mass). This gives us the point $\mathsf{R}\mathbf{r} + \mathbf{p}$ in the world space. If the object is rotating with an angular velocity given by the vector $\boldsymbol{\omega}$, and its centre of mass is moving with velocity $\mathbf{v}$ then the point at $\mathsf{R}\mathbf{r}+\mathbf{p}$ is moving with velocity $\boldsymbol{\omega} \times (\mathsf{R}\mathbf{r}) + \mathbf{v}$. This result is obtained by considering the contribution to the point's velocity from rotational motion and linear motion respectively. By the definition of how we represent angular velocity (as discussed in Section 3.2) we get the rotational contribution to be $\boldsymbol{\omega} \times$ *relative position vector from the centre of mass*. If we differentiate the position with respect to time we get $\dot{\mathsf{R}}\mathbf{r} + \dot{\mathbf{p}}$. Note that position differentiated with respect to time is velocity, so $\dot{\mathbf{p}} = \mathbf{v}$ and

$\dot{\mathsf{R}}\mathbf{r} + \dot{\mathbf{p}} = \boldsymbol{\omega} \times (\mathsf{R}\mathbf{r}) + \mathbf{v}$. This gives us $\dot{\mathsf{R}}\mathbf{r} = \boldsymbol{\omega} \times \mathsf{R}\mathbf{r}$. Vector cross multiplication is linear, so given

$$\boldsymbol{\omega} = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} \quad \text{and} \quad \mathbf{r} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix}$$

we have

$$\boldsymbol{\omega} \times \mathbf{r} = \begin{pmatrix} \omega_2 r_3 - \omega_3 r_2 \\ \omega_3 r_1 - \omega_1 r_3 \\ \omega_1 r_2 - \omega_2 r_1 \end{pmatrix} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} = \widetilde{\boldsymbol{\omega}}\mathbf{r}$$

where

$$\widetilde{\boldsymbol{\omega}} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix}$$

For convenience, from now on a tilde above a vector will represent the matrix associated with cross multiplication by that vector. So under this notation we can write

$$\dot{\mathsf{R}} = \widetilde{\boldsymbol{\omega}}\mathsf{R} \tag{3}$$

which will be useful later on.

When there are no external forces, an object conserves momentum. Since we are decomposing the motion into a linear component and rotational component about the centre of mass, we can consider the respective momenta independently of each other. Linear momentum is defined as mass times velocity (here we take the linear velocity of the centre of mass). Since the mass will not be changing this means the velocity is constant. The angular momentum is given by the inertia tensor times the angular velocity. In 2D motion we can simplify this to the moment of inertia times the angular velocity. Everything is a scalar in 2D, and the moment of inertia remains fixed, which means that the angular velocity will also stay constant just like the case for linear velocity. The fact that both the linear and angular velocity remain constant makes the 2D motion of an object easy to calculate. In 3D motion however the inertia tensor does not remain fixed but instead changes as the orientation of the object changes. Let I_0 be the inertia tensor of the object in model space. Then $\mathsf{I} = \mathsf{R}\mathsf{I}_0\mathsf{R}^T$ where I is the inertia tensor in world space coordinates. (Since R is orthogonal $\mathsf{R}^T = \mathsf{R}^{-1}$.) Hence this gives

$$\mathbf{L} = \mathsf{I}\boldsymbol{\omega} = \mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} \tag{4}$$

where $\mathbf{L}$ is the angular momentum and is constant, when there are no external forces. Rearranging (4) gives

$$\begin{aligned} \boldsymbol{\omega} &= (\mathsf{R}\mathsf{I}_0\mathsf{R}^T)^{-1}\mathbf{L} \\ &= (\mathsf{R}^T)^{-1}\mathsf{I}_0^{-1}\mathsf{R}^{-1}\mathbf{L} \\ &= \mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\mathbf{L} \end{aligned}$$

We can take the inverse of the inertia tensor, as under an appropriate orthonormal basis it is a diagonal matrix with non-zero diagonal terms.

Force is defined as the rate of change of momentum with respect to time. Since the mass is a constant, this is often written in the more convenient form

$$\mathbf{F} = m\mathbf{a} \tag{5}$$

where $\mathbf{F}$ is the force, m the mass, and $\mathbf{a}$ the acceleration. We can define torque $\boldsymbol{\tau}$ in relation to the forces being applied to an object via

$$\boldsymbol{\tau} = \mathbf{r} \times \mathbf{F} \tag{6}$$

where $\mathbf{r}$ is the vector from the centre of mass to the point which the force $\mathbf{F}$ acts through. For 2D we just take the modulus, $\tau = |\mathbf{r} \times \mathbf{F}|$. We also have a similar result to (5) for rotations in 2D

$$\tau = I\dot{\omega} \tag{7}$$

where I is the moment of inertia, and $\dot{\omega}$ the angular acceleration. Using (6) and (7) gives us a way to calculate the angular acceleration from the forces acting on the object. Torque is the rate of change of angular momentum with respect to time. This is consistent with (7) as the moment of inertia is a constant in 2D, but when we move into the third dimension, (7) no longer holds and things are more complex. Using (3) and (4) we have

$$\begin{aligned}
\boldsymbol{\tau} &= \dot{\mathbf{L}} \\
&= \dot{\mathsf{I}}\boldsymbol{\omega} + \mathsf{I}\dot{\boldsymbol{\omega}} \\
&= \dot{\mathsf{R}}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0\dot{\mathsf{R}}^T\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0\mathsf{R}^T\dot{\boldsymbol{\omega}} \\
&= \widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0(\widetilde{\boldsymbol{\omega}}\mathsf{R})^T\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0\mathsf{R}^T\dot{\boldsymbol{\omega}} \\
&= \widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} - \mathsf{R}\mathsf{I}_0\mathsf{R}^T\widetilde{\boldsymbol{\omega}}\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0\mathsf{R}^T\dot{\boldsymbol{\omega}} \\
&= \widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} + \mathsf{R}\mathsf{I}_0\mathsf{R}^T\dot{\boldsymbol{\omega}}
\end{aligned} \tag{8}$$

When there are no external forces $\boldsymbol{\tau} = 0$, substituting this into (8) and rearranging gives

$$\dot{\boldsymbol{\omega}} = -\mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} \tag{9}$$

From this formula we can show that the axis of rotation remains fixed if and only if the axis of rotation corresponds to a principal axis. It's not too hard to see that rotating about a principal axis will result in no angular acceleration, but proving that all the fixed rotations are principal axes is more complicated. The proof of this is given in Appendix A. Rotating about a principal axis also fixes the angular velocity as $\dot{\boldsymbol{\omega}} = 0$. This simplification is exactly what is used when initially studying rotational motion in 2D; it should be clear that it is in fact a very special case of the 3D motion. It is important to know under what conditions we can model a simulation using the 2D equations of motions, as doing so will significantly increase the speed of the simulation. Unfortunately in general we can not solve for $\boldsymbol{\omega}$ in a closed form and are instead forced to use numerical methods.

As well as momentum another important quantity that remains conserved under no external forces is energy. In 2D, rotational energy is given by $\frac{1}{2}I\omega^2$. Since I and ω are constant it is easy to see that the energy remains constant. However due to the complexity of (9) this is no longer obvious. In 3D, rotational energy is given by $E = \frac{1}{2}\boldsymbol{\omega}^T\mathsf{I}\boldsymbol{\omega} = \frac{1}{2}\boldsymbol{\omega}^T\mathbf{L}$. Differentiating with respect to time gives

$$\begin{aligned}
\dot{E} &= \frac{1}{2}\dot{\boldsymbol{\omega}}^T\mathbf{L} \\
&= \frac{1}{2}(-\mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega})^T(\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega}) \\
&= -\frac{1}{2}\boldsymbol{\omega}^T\mathsf{R}\mathsf{I}_0\mathsf{R}^T\widetilde{\boldsymbol{\omega}}^T\mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega} \\
&= -\frac{1}{2}\boldsymbol{\omega}^T\mathsf{R}\mathsf{I}_0\mathsf{R}^T\widetilde{\boldsymbol{\omega}}^T\boldsymbol{\omega} \\
&= \frac{1}{2}\boldsymbol{\omega}^T\mathsf{R}\mathsf{I}_0\mathsf{R}^T(\widetilde{\boldsymbol{\omega}}\boldsymbol{\omega}) \\
&= 0
\end{aligned}$$

hence under the motion given by (9) energy remains conserved, which is what we expect. Although we will not explicitly be using the fact that energy is conserved, it is still useful to know how to calculate it, and how it fits in with the general picture. If nothing else, calculating an object's energy as it moves is a useful way to catch programming errors and helps us check on the accumulation of numerical errors.

We now have the necessary formulae to work out how an object moves when there are no external forces.

Simulating Motion when in Free Fall

In this section we want to look at how we calculate the motion of a solitary object in free fall. Since we are not considering any other objects we don't need to consider collision detection and we do not need to deal with the object's boundary. We do however need to know information about its mass distribution (but not its total mass). We will use the notation we introduced in the previous section.

$$
\begin{aligned}
t &= \text{Time} \\
\mathsf{I}_0 &= \text{Inertia tensor of the object in model space} \\
\mathbf{p}_0 &= \text{Initial position of the centre of mass in the world space} \\
\mathsf{R}_0 &= \text{The matrix initially representing orientation of the object in the world space} \\
\mathbf{v}_0 &= \text{Initial velocity of the centre of mass} \\
\boldsymbol{\omega}_0 &= \text{Initial angular velocity} \\
\mathbf{p} &= \text{Current position of the centre of mass in the world space} \\
\mathsf{R} &= \text{The matrix currently representing orientation of the object in the world space} \\
\mathbf{v} &= \text{Current velocity of the centre of mass} \\
\boldsymbol{\omega} &= \text{Current angular velocity}
\end{aligned}
$$

The values $\mathbf{p}, \mathsf{R}, \mathbf{v}$, and $\boldsymbol{\omega}$ all vary with time t. Our aim is to calculate how $\mathbf{p}$ and R evolve with time. To do this we need to know the inertia tensor I_0 (which is constant) and the initial state of the object at $t = 0$, given by the fixed values $\mathbf{p}_0, \mathsf{R}_0, \mathbf{v}_0$, and $\boldsymbol{\omega}_0$. These fixed values need to be set by the user before the simulation begins. From the previous section we know that, if there are no external forces, the following hold

$$
\begin{aligned}
\dot{\mathbf{p}} &= \mathbf{v} \\
\dot{\mathsf{R}} &= \widetilde{\boldsymbol{\omega}}\mathsf{R} \\
\dot{\mathbf{v}} &= \mathbf{0} \\
\dot{\boldsymbol{\omega}} &= -\mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\widetilde{\boldsymbol{\omega}}\mathsf{R}\mathsf{I}_0\mathsf{R}^T\boldsymbol{\omega}
\end{aligned}
$$

So to calculate $\mathbf{p}$ and R all we need to do is numerically integrate the above equations. This can be easily done using Euler's method or using the Runge-Kutta method. In my implementation I used the fourth order Runge-Kutta method, as it gives better results than Euler's method and is straightforward to implement. Without much difficulty we can generalize to simulate the motion of an object when it is under a constant uniform force, like that due to gravity. Since it is uniform, the force contributes no torque, so the rotational aspect of the motion is completely unaffected. If the acceleration caused by the force is $\mathbf{g}$ then all we do is replace $\dot{\mathbf{v}} = \mathbf{0}$ with $\dot{\mathbf{v}} = \mathbf{g}$ in the equations above, and integrate as before.

Although it's fairly simple to achieve our goal of calculating an object's motion under free fall, there are a number of improvements that can be made. Firstly, although the rotational aspect of the motion in general can not be written down in a closed form, the translational motion can. We get $\mathbf{v} = \mathbf{g}t + \mathbf{v}_0$ and $\mathbf{p} = \frac{1}{2}\mathbf{g}t^2 + \mathbf{v}_0 t + \mathbf{p}_0$. Unsurprisingly these are the equations of motion for a ballistic trajectory. So although we can not get rid of the numerical integration step in our simulation we can make it more efficient by computing the translational component separately (and more accurately). We can, in fact, improve the efficiency when calculating the rotational motion as well. It is natural to calculate R by looking at $\boldsymbol{\omega}$ but a better choice is to use the angular momentum $\mathbf{L}$. Initially $\mathbf{L} = \mathsf{R}_0\mathsf{I}_0\mathsf{R}_0^T\boldsymbol{\omega}_0$ and remains constant throughout the simulation. This means we have one less equation to integrate, in fact all we are integrating now is $\dot{\mathsf{R}} = \widetilde{\boldsymbol{\omega}}\mathsf{R}$. To get the value of $\boldsymbol{\omega}$ needed for this integration we merely apply (4), so $\boldsymbol{\omega} = \mathsf{R}\mathsf{I}_0^{-1}\mathsf{R}^T\mathbf{L}$.

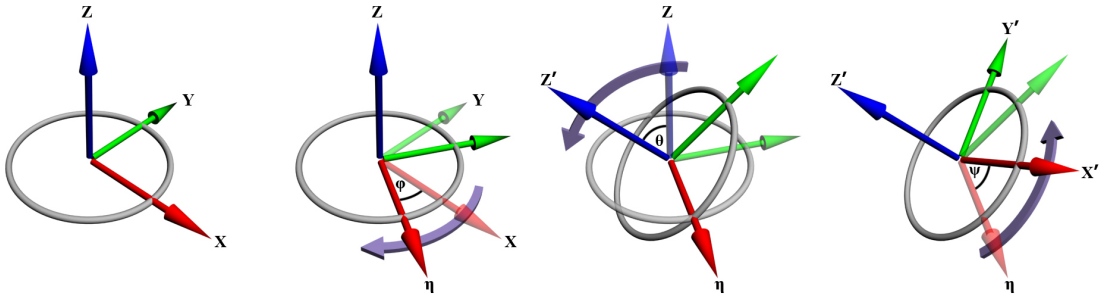


Figure 12: On the left we have the original orientation represented by the axes X, Y, Z (a circle lying in the XY plane is also shown for clarification). In order to transform it to the axes X′, Y′, Z′ we apply three rotations. First we rotate about Z by an angle ϕ, this maps the X axis to the η axis. Next we rotate about η by an angle of θ, this maps Z to Z′. Finally we rotate about Z′ by an angle ψ. The angles (ϕ, θ, ψ) are known as the Euler angles.

Representing orientations by a 3 by 3 matrix is inefficient. We can in fact represent it by just 3 scalars instead of 9. These 3 scalars represent angles of rotation about 3 specially chosen axes and are usually called the Euler angles. There is no standard definition of Euler angles but Figure (12) shows a popular version. This more economical representation means we would be doing less integration, which would speed up our algorithm. Given an orientation represented by the triple (ϕ, θ, ψ), see Figure (12), we have

$$\mathsf{R} = \begin{pmatrix} \cos\psi\cos\phi - \cos\theta\sin\phi\sin\psi & \cos\psi\sin\phi + \cos\theta\cos\phi\sin\psi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi - \cos\theta\sin\phi\cos\psi & -\sin\psi\sin\phi + \cos\theta\cos\phi\cos\psi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{pmatrix}$$

So although we can get back to R from the Euler angles we have to calculate sines and cosines of three angles which is computationally expensive. Due to the minimalist way the Euler angles represent orientations they appear quite clumsy in nature and produce rather complicated formulae. There is a way however to represent orientations that regains much of the simplicity that our matrix representation gave us, whilst still giving us the economy that Euler angles exhibited. We do this by using quaternions.

Quaternions are a generalization of complex numbers. They are represented by 4 scalars and take the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k$$

where i, j, k satisfy

$$\begin{aligned} i^2 = j^2 = k^2 &= -1 \\ ij &= -ji = k \\ jk &= -kj = i \\ ki &= -ik = j \end{aligned}$$

The conjugate of $\mathbf{q}$ is a quaternion, denoted by $\mathbf{q}^*$ which equals $q_0 - q_1 i - q_2 j - q_3 k$. Just like complex numbers, quaternions act like a vector space under addition and scalar multiplication. We define the norm of a quaternion as the square of the length of the four dimensional vector it represents

$$N(\mathbf{q}) = q_0^2 + q_1^2 + q_2^2 + q_3^2 = \mathbf{q}\mathbf{q}^* = \mathbf{q}^*\mathbf{q}$$

Under these definitions, given two quaternions $\mathbf{p}, \mathbf{q}$ we have $N(\mathbf{pq}) = N(\mathbf{p})N(\mathbf{q})$. So this means the set of quaternions satisfying $N(\mathbf{q}) = 1$ forms a subgroup with unit 1 $(= 1 + 0i + 0j + 0k)$ and $\mathbf{q} = \mathbf{q}^*$. We can map rotations to unit quaternions using

$$\mathbf{q} = \cos\frac{\theta}{2} + \hat{\mathbf{n}}\sin\frac{\theta}{2}$$

where $\mathbf{n}$ is a unit vector representing the axis of rotation, and θ is the angle we rotate about that axis. $\hat{\mathbf{n}}$ represents the vector $\mathbf{n}$ as a quaternion. Given vector $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ we define $\hat{\mathbf{x}}$ as $x_1 i + x_2 j + x_3 k$. Using this notation we can write a simple formula: $\hat{\mathbf{x}}' = \mathbf{q}\hat{\mathbf{x}}\mathbf{q}^*$ rotates a point $\mathbf{x}$ about the origin, where the rotation is given by the quaternion $\mathbf{q}$, and $\mathbf{x}'$ is the position of the point after being rotated. Using this formula also allows us to calculate the matrix R for the associated quaternion.

$$\mathsf{R} = \begin{pmatrix} 1 - 2(q_2^2 + q_3^2) & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & 1 - 2(q_1^2 + q_3^2) & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & 1 - 2(q_1^2 + q_2^2) \end{pmatrix}$$

Since we don't have to calculate sines and cosines, converting a quaternion into a matrix is much quicker than doing it from Euler angles. Also we have

$$\dot{\mathbf{q}} = \frac{1}{2}\hat{\boldsymbol{\omega}}\mathbf{q}$$

which is more succinct and so easier to work with. We can also store quaternions as three scalars just like Euler angles. Each orientation corresponds to exactly 2 unit quaternions, $\mathbf{q}$ and $-\mathbf{q}$. So we can enforce the rule that we will only represent orientations with quaternions which have a positive q_0 value. So now if we store only q_1, q_2, q_3 we can still retrieve q_0 as it is $\sqrt{1 - q_1^2 - q_2^2 - q_3^2}$. Note that although this reduces the amount of memory we use this may not justify the added time needed to calculate q_0 at each step. Another advantage quaternions hold over Euler angles is in key frame animation. Interpolating quaternions give much smoother animations than trying to interpolate the Euler angles, see [18]. Since we are numerically integrating the equations of motion, numerical error will be introduced and the values we get will drift away from the actual solution with time. This means that eventually there will be noticeable errors. When solving for R the matrix will stop being orthogonal, and similarly when solving for $\mathbf{q}$ the quaternion will cease to have norm 1. After each integration step we can of course correct for this drift by forcing the matrix to be orthogonal and re-normalizing the quaternion respectively. However re-normalizing a matrix takes a lot more time to do than a four dimensional vector does. For these reasons when implementing rigid body simulation it is usually best to represent orientations as quaternions. Throughout the rest of the report I will still be using matrices to represent orientations, this is just for clarity though, and all the equations can of course be rewritten to use quaternions instead.

Now that we know how an object moves when there are no other objects about, the next step is to look at a simple treatment of collisions.

Simple Impulse-Based Collisions

So far our algorithm consists of: progressing time by a small time step, updating the position and orientation by numerical integration accordingly, displaying the object, and repeating. If we have more than one object and they don't collide we can adapt this algorithm easily by updating all the positions and orientations for each object separately. However as we will see later it is often more convenient to integrate all the properties of all the objects simultaneously. If the objects collide then we need to detect this at some point; this is handled by collision response algorithms. In this section we will be assuming that when the objects collide they are heading towards each other at some nonzero velocity, that there is no friction, the objects are only in contact at a single point, and that each object can only be in contact with at most one other object.

Since the bodies are rigid this means that when the objects collide we model the deformation caused by collision as occurring over an infinitesimal time period. This means that the forces between the objects will also occur over an infinitesimal time period and so can be replaced by impulses. Also an infinitesimal time period means that we can ignore uniform external forces like gravity in our collision calculations as they will make no contributions. In order for our collision response algorithm to be successful we will need to know the point of contact as this is where the impulse will occur. We will also need to know the direction that the impulse will act. (Note that the impulse that acts on one object will be equal in magnitude and opposite in direction to the impulse on the other object due to Newton's third law of motion.) This can be a problem as there are some degenerate cases where such information is not available. In 2D if the object is a polygon then there are only two ways we can get a single point collision, they are when a vertex from one object collides with an edge of the other, or when it collides with another vertex. These two types of collision are indicated in Figure 13. It is clear the direction of the impulses will be perpendicular to the edge in a vertex-edge collision, but it is unclear in which direction it will act in vertex-vertex collision. Such a collision is degenerate and we can easily choose an arbitrary but reasonable direction for the impulse to act. In 3D we have a similar situation, two boundary mesh triangles can collide at a single point in one of 4 ways, vertex-vertex, vertex-edge, vertex-face, and edge-edge collisions. The first two are degenerate cases and the impulse direction will have to be arbitrarily chosen. For a vertex-face collision the direction will be normal to the plane that the face lies in. For an edge-edge collision the direction impulses will act is normal to the plane both edges lie in; we can easily calculate it by taking the cross products of the vectors representing the edges. We will assume that the collision detection side of the algorithm will take care of finding the point of intersection and supplying an appropriate direction to apply impulses.

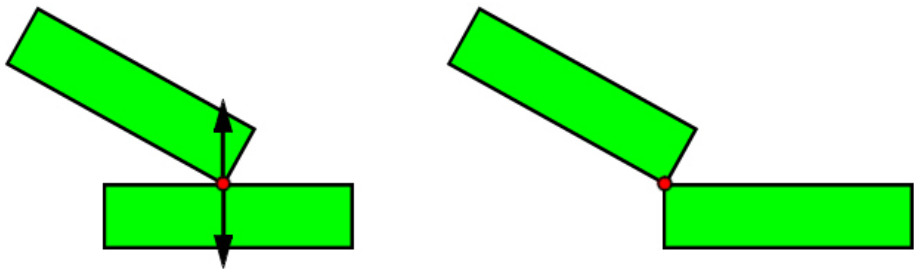


Figure 13: 2D rectangles colliding. The point of collision is indicated by the dot. On the left we have a vertex-edge collision, the impulses applied on the objects act in a direction perpendicular to the edge it collides with. On the right we have a vertex-vertex collision. It is unclear in exactly which direction the impulse will be applied. This case is degenerate.

So now our algorithm has the following form:

1. Check for collisions using a collision detection algorithm. For each pair of objects found to be intersecting provide the point at which they intersect and the direction any impulses will act along.
2. If there are any objects that are colliding, work out the appropriate impulses between the objects and apply them so the objects will move away from each other.
3. Integrate the positions and orientations of all the objects forward in time by a small finite time step Δt (as if they were in free fall).
4. Display the new state of the simulation.
5. Repeat from step 1.

In order to calculate the magnitude of the impulses required in step 2, we require an empirical law. What we use is Newton's law of restitution. This says that if two particles collide we have

$$\textit{Speed of Seperation} = e \times \textit{Speed of Approach}$$

where e is a constant known as the coefficient of restitution, which is dependent only on the material of the two particles. e takes values in the region of 0 to 1. $e = 0$ corresponds to an inelastic collision where kinetic energy is lost. $e = 1$ corresponds to a perfectly elastic collision where all the kinetic energy is conserved. This law assumes the collision is frictionless and is essentially only for objects which are spheres and can be modeled as particles. However it is often also used to model collisions between a ball and the ground or a wall. Evidence for the correctness of this formula can be seen when dropping a ball onto the ground. The ball will make several bounces before coming to rest, and the maximum height it reaches on each bounce is proportional to the square of the speed it leaves the ground at. So according to Newton's law of restitution the max height of each of the bounces should form a decreasing geometric series, which is what is observed. We will use this law to calculate impulses for general objects not necessarily spherical. Although there is not much evidence why this should hold, it appears to give reasonable looking simulations.

Let us assume that object A collides with object B at a point $\mathbf{r}$ in world space and impulses will act in the direction given by the unit vector $\mathbf{n}$. The direction of $\mathbf{n}$ is chosen so that it points away from object B. For example in Figure 13 if the top rectangle is object A and the bottom rectangle is object B then $\mathbf{n}$ would point vertically upwards. Let J be the magnitude of the impulses that will be applied during the collision. Note that object A will receive an impulse of $\mathbf{J} = J\mathbf{n}$ and object B will receive an impulse of $-\mathbf{J}$. Let us define the other variables we will be using

$$\begin{aligned}
m_A &= \text{Total mass of object A} \\
\mathsf{I}_A &= \text{Inertia tensor of object A in world space} \\
\mathbf{p}_A &= \text{Position of the centre of mass of object A in the world space} \\
\mathbf{v}_A &= \text{Velocity of object A's centre of mass before the collision} \\
\boldsymbol{\omega}_A &= \text{Object A's angular velocity before the collision} \\
\mathbf{v}'_A &= \text{Velocity of object A's centre of mass after the collision} \\
\boldsymbol{\omega}'_A &= \text{Object A's angular velocity after the collision}
\end{aligned}$$

$$
\begin{aligned}
m_B &= \text{Total mass of object B} \\
\mathsf{I}_B &= \text{Inertia tensor of object B in world space} \\
\mathbf{p}_B &= \text{Position of the centre of mass of object B in the world space} \\
\mathbf{v}_B &= \text{Velocity of object B's centre of mass before the collision} \\
\boldsymbol{\omega}_B &= \text{Object B's angular velocity before the collision} \\
\mathbf{v}'_B &= \text{Velocity of object B's centre of mass after the collision} \\
\boldsymbol{\omega}'_B &= \text{Object B's angular velocity after the collision}
\end{aligned}
$$

Our aim is to calculate $\mathbf{v}'_A, \boldsymbol{\omega}'_A, \mathbf{v}'_B, \boldsymbol{\omega}'_B$ from the other variables, we do this by calculating the magnitude of the impulse required to make Newton's law of restitution hold. Note that for the duration of the collision the orientations and positions of the objects don't change due to the fact that it takes an infinitesimal time. Also for the same reason constant uniform forces like gravity can be ignored. Force and torque are defined as the rate of change of linear momentum and angular momentum with respect to time. Impulses are similar to forces but, due to their discontinuous nature, instead of looking at the rate of change of momenta we just use the difference in momenta before and after the application of an impulse. So we have

$$
\begin{aligned}
\mathbf{J} &= m_A\mathbf{v}'_A - m_A\mathbf{v}_A && (10)\\
(\mathbf{r}-\mathbf{p}_A)\times\mathbf{J} &= \mathsf{I}_A\boldsymbol{\omega}'_A - \mathsf{I}_A\boldsymbol{\omega}_A && (11)\\
-\mathbf{J} &= m_B\mathbf{v}'_B - m_B\mathbf{v}_B && (12)\\
-(\mathbf{r}-\mathbf{p}_B)\times\mathbf{J} &= \mathsf{I}_B\boldsymbol{\omega}'_B - \mathsf{I}_B\boldsymbol{\omega}_B && (13)
\end{aligned}
$$

The velocity at the point $\mathbf{r}$ for object A is $\mathbf{v}_A+\boldsymbol{\omega}_A\times(\mathbf{r}-\mathbf{p}_A)$ before the collision and $\mathbf{v}'_A+\boldsymbol{\omega}'_A\times(\mathbf{r}-\mathbf{p}_A)$ after the collision. Similarly for object B the point at $\mathbf{r}$ has velocity before and after the collision of $\mathbf{v}_B + \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B)$ and $\mathbf{v}'_B + \boldsymbol{\omega}'_B \times (\mathbf{r} - \mathbf{p}_B)$ respectively. Note that the speed of approach and separation is that of the point of contact and not the centre of mass. Substituting this into Newton's law of restitution gives us

$$
[(\mathbf{v}'_A+\boldsymbol{\omega}'_A\times(\mathbf{r}-\mathbf{p}_A))-(\mathbf{v}'_B+\boldsymbol{\omega}'_B\times(\mathbf{r}-\mathbf{p}_B))]\cdot\mathbf{n} = -e[(\mathbf{v}_A+\boldsymbol{\omega}_A\times(\mathbf{r}-\mathbf{p}_A))-(\mathbf{v}_B+\boldsymbol{\omega}_B\times(\mathbf{r}-\mathbf{p}_B))]\cdot\mathbf{n}
$$

We can use (10), (11), (12), (13) to substitute for $\mathbf{v}'_A, \boldsymbol{\omega}'_A, \mathbf{v}'_B, \boldsymbol{\omega}'_B$ respectively in the above equation.

$$
\begin{aligned}
[\mathbf{v}_A + \frac{1}{m_A}\mathbf{J} + \boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A) + (\mathsf{I}_A^{-1}[(\mathbf{r} - \mathbf{p}_A) \times \mathbf{J}]) \times (\mathbf{r} - \mathbf{p}_A) \\
- \mathbf{v}_B + \frac{1}{m_B}\mathbf{J} - \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B) + (\mathsf{I}_B^{-1}[(\mathbf{r} - \mathbf{p}_B) \times \mathbf{J}]) \times (\mathbf{r} - \mathbf{p}_B)] \cdot \mathbf{n} \\
= -e[(\mathbf{v}_A + \boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A)) - (\mathbf{v}_B + \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B))] \cdot \mathbf{n}
\end{aligned}
$$

To help simplify the above, let $\mathbf{r}_A = \mathbf{r} - \mathbf{p}_A$, and $\mathbf{r}_B = \mathbf{r} - \mathbf{p}_B$. Now we have

$$
\begin{aligned}
[J(\frac{1}{m_A}\mathbf{n} + \frac{1}{m_B}\mathbf{n} - \widetilde{\mathbf{r}}_A\mathsf{I}_A^{-1}\widetilde{\mathbf{r}}_A\mathbf{n} - \widetilde{\mathbf{r}}_B\mathsf{I}_B^{-1}\widetilde{\mathbf{r}}_B\mathbf{n}) + \mathbf{v}_A - \widetilde{\mathbf{r}}_A\boldsymbol{\omega}_A - \mathbf{v}_B + \widetilde{\mathbf{r}}_B\boldsymbol{\omega}_B] \cdot \mathbf{n} \\
= -e[\mathbf{v}_A - \widetilde{\mathbf{r}}_A\boldsymbol{\omega}_A - \mathbf{v}_B + \widetilde{\mathbf{r}}_B\boldsymbol{\omega}_B] \cdot \mathbf{n}
\end{aligned}
$$

Rearranging to solve for J gives

$$
\begin{aligned}
J &= \frac{-(1+e)[\mathbf{v}_A - \widetilde{\mathbf{r}}_A\boldsymbol{\omega}_A - \mathbf{v}_B + \widetilde{\mathbf{r}}_B\boldsymbol{\omega}_B] \cdot \mathbf{n}}{\frac{1}{m_A} + \frac{1}{m_B} - \mathbf{n} \cdot (\widetilde{\mathbf{r}}_A\mathsf{I}_A^{-1}\widetilde{\mathbf{r}}_A\mathbf{n}) - \mathbf{n} \cdot (\widetilde{\mathbf{r}}_B\mathsf{I}_B^{-1}\widetilde{\mathbf{r}}_B\mathbf{n})} \\
&= -(1+e)\frac{(\mathbf{v}_A - \mathbf{v}_B) \cdot \mathbf{n} + (\mathbf{r}_A \times \mathbf{n}) \cdot \boldsymbol{\omega}_A - (\mathbf{r}_B \times \mathbf{n}) \cdot \boldsymbol{\omega}_B}{\frac{1}{m_A} + \frac{1}{m_B} + (\mathbf{r}_A \times \mathbf{n}) \cdot [\mathsf{I}_A^{-1}(\mathbf{r}_A \times \mathbf{n})] + (\mathbf{r}_B \times \mathbf{n}) \cdot [\mathsf{I}_B^{-1}(\mathbf{r}_B \times \mathbf{n})]} \qquad (14)
\end{aligned}
$$

This equation is consistent with those derived in [15] and [20]. So if a collision is detected we use (14) to calculate the magnitude of the impulse. Then we substitute it into (10), (11), (12), (13) to get $\mathbf{v}'_A, \boldsymbol{\omega}'_A, \mathbf{v}'_B, \boldsymbol{\omega}'_B$. Once we update the linear and angular velocities (remembering to change the angular momentum as well) we can continue with the simulation as normal.

Often in a simulation we wish to have objects which are modeled as being immovable or having infinite mass, such as the ground. The impulse from a collision with an object of finite mass will not affect such an object's linear or angular velocity. We need to change (14) to take this into account. Without loss of generality let us assume object B is of infinite mass, then

$$J = -(1+e)\frac{(\mathbf{v}_A - \mathbf{v}_B)\cdot\mathbf{n} + (\mathbf{r}_A \times \mathbf{n})\cdot\boldsymbol{\omega}_A - (\mathbf{r}_B \times \mathbf{n})\cdot\boldsymbol{\omega}_B}{\frac{1}{m_A} + (\mathbf{r}_A \times \mathbf{n})\cdot[\mathsf{I}_A^{-1}(\mathbf{r}_A \times \mathbf{n})]}$$

This is equivalent to setting $\frac{1}{m_B} = 0$ and $\mathsf{I}_B^{-1} = \mathsf{0}$. Interestingly the total mass does not matter when colliding with an infinitely heavy object, an object with unit mass and identical mass distribution will behave in exactly the same way. Note that both object A and B cannot be of infinite mass as this leads to a division by 0 and is undefined. When $e = 1$ we had conservation of kinetic energy before and after the collision with particles. This property reassuringly still holds when setting $e = 1$ in (14), see Appendix B. Setting all the coefficients of restitutions to 1 in a simulation and checking no energy is lost in collisions, can be a useful way of catching programming errors.

Problems with the Simple Impulse-Based Model

Although this model is quite useful it has its drawbacks and problems, especially when we relax some of the conditions that we imposed. A lot of the problems in rigid body simulation are related to robustness of algorithms. It is important not to treat such things hypothetically and just deal with the equations. Often problems in the algorithm come from important problems with the model we are trying to implement.

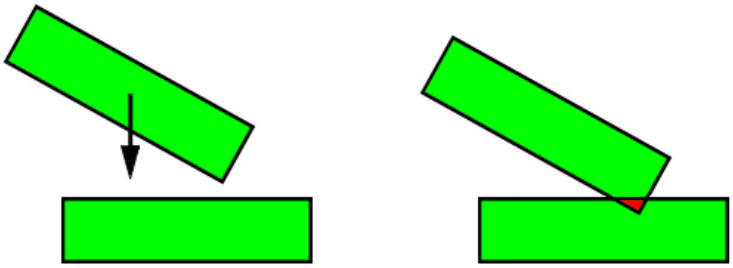


Figure 14: On the left we have 2 rectangles that are not intersecting. The top rectangle is moving down towards the bottom stationary rectangle. On the right we see the simulation one time step later. The rectangles have collided and their intersection is a small triangle as opposed to a single point.

In our algorithm we have to be very careful how we define collisions. Since we are evolving the simulation in discrete finite time steps, when two objects are found to be in contact it is unlikely that their intersection is a point; it is more likely to be a small area or volume, see Figure 14. We can of course, having found a collision, move the simulation backwards applying essentially a root finding algorithm to find the exact time the collision occurred (usually this is done to find the first collision between all pairs of objects). This adaptive time stepping solution would ensure we get point

intersection. Note that when we go back and find the exact time the collision occurred the other objects will have moved and there may be intersections that were previously missed. Unfortunately we can not always do this as we will see later (objects that have come to rest on the ground for example will cause problems). In any case there is a limit to how small we can make the time step before numerical error overwhelms the results we get. This is not as big a problem as it would first appear, as if the time step is small enough then the area of intersection will also be relatively small. Although we can not provide the exact point at which the impulses will be applied, any point inside the intersection will be a good enough approximation. Fixing the time step and allowing a small amount of penetration upon collision was the method I initially used.

A result of this was that the objects upon collision could have difficulty separating. This can be seen in Video 04 on the accompanying cd-rom. The problem was that objects that intersect each other are classed as colliding and so an impulse is applied to separate them. This works fine in theory but, after the impulse is applied, the object's separation velocity will in general be smaller than their approach velocity. This means that after another time step although the objects are moving away from each other they may not have sufficient velocity to have moved far enough to separate. Since they are still intersecting they will be deemed to be colliding and so an impulse will be applied. The formulae we defined in Section 3.3, assumed the approach velocity was positive, it is however now negative, so the impulse will act to push the bodies together not apart. Hence the simulation proceeds to apply an unwanted impulse in the wrong direction. This causes the objects to move towards each other again and keep them intersecting. To avoid such problems we have to redefine when we should apply an impulse: the objects should be intersecting *and* their approach velocity should be positive. This is a typical example of the gap between a hypothetical algorithm and an actual robust implementation.

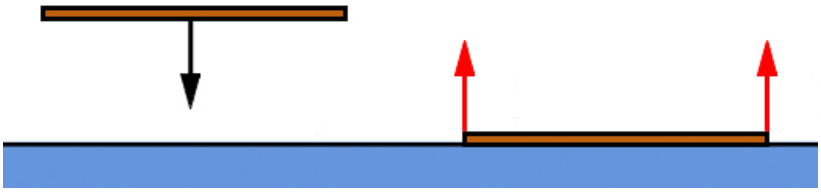


Figure 15: On the left we have a horizontal stick with no angular velocity. When it hits the ground both its ends will be in contact with the ground simultaneously

The impulse-based model is a pretty good model but there are a number of restrictions that we applied in Section 3.3 that severely restrict its use. Let us consider the restriction that the objects only intersect at one point. As we have already seen, when implementing an algorithm, the point is often really a small region, but let us concentrate on the aspect that there is only *one* point of intersection. Consider the degenerate case of a stick colliding with the ground such that both ends hit simultaneously see Figure 15. By symmetry we would expect that equal impulses would be applied at both ends simultaneously. However the magnitude of the impulse is unclear. Due to the lack of rotation caused by the symmetry, it is reasonable to expect the stick to behave as a particle, and we could calculate the impulse to be half that of a particle colliding at the same speed. However it is not hard to imagine other cases where we can not make such assumptions. In general our formulae will no longer hold as there is no longer a unique speed of approach. Suppose the stick made an infinitesimal angle with the horizontal ground, then we have one end colliding before the other. We can then apply our formulae as before. In such a case we would expect that after the first end hits the ground the other will follow almost instantaneously. The ends could (alternately) hit the ground a large number of times before the stick eventually bounces away from the floor. Unfortunately, though both ends may receive a series of

impulses it will in most cases not leave with zero angular velocity. In general if we have simultaneous contact points what we do is:

1. Choose an arbitrary ordering of the contacts.
2. Go through the ordered list of contacts till we find a contact point with a positive speed of approach.
3. If such a contact point exists apply the appropriate impulse and go to step 2 starting from the beginning of the list.
4. If no contact point exists with a positive speed of approach then the objects are moving away from each other, and so we need not apply any more impulses.

Note that after every application of an impulse we have to start all over again as points that were previously separating may now be approaching each other. This method is known as the propagation model and is what I have chosen in my implementation. There is a way to calculate all the impulses simultaneously (known as the simultaneous model) which we will discuss briefly in a later section but it is more complicated and uses equations which are less justified. See [1] for a comparison of these two models.

The algorithm we have described for the propagation model, although fairly simple, may not terminate. We can show with a bit of work that if we have a stick with its centre of mass midway along its length, and the coefficient of restitution is 1, then the algorithm will eventually terminate (see Appendix C). This makes it a good object to test our programs on. The proof in Appendix C also tells us the theoretical values of the impulses applied, and the velocities of both ends of the stick after each collision, which can be used to check if the simulation is progressing as intended. Unfortunately a simple simulation will show that our propagation model algorithm will fail to terminate if e is sufficiently small. This is because after each collision kinetic energy is lost for $e < 0$. So if sufficient collisions happen then the object will lose sufficient energy that it will not be able to move away from the floor. The end point's velocities will decrease to zero approximating a geometric series. For other more complicated simulations we can not guarantee that the algorithm will terminate even for $e = 1$. The best we can do to sort out this problem is to force the algorithm to stop when the velocities are sufficiently small (by setting an artificial threshold for the approach velocity). It should be noted that although there are situations where the algorithm may not terminate, in general this rarely occurs. This problem is related to the problem of resting contacts which we will discuss later in this section, which is a much bigger problem.

The algorithm we described can also be adapted to handle the case when more than one object is in contact with another. We simply look at all the contact points between all objects and apply the algorithm as described. Unfortunately as soon as we apply one impulse we have to start again. This rapidly slows down the simulation when there are a large number of contact points. In such situations, if we require real time results, we have to use the simultaneous model. From Figure 16 we see that both models satisfy Newton's law of restitution and conserve energy, but produce totally different results. Such ambiguity is a direct consequence of forcing the bodies to be rigid, and choosing the correct model for a simulation is impossible.

Another important problem with our current algorithm is that of sinking. See Videos 05 and 06 on the cd-rom. Due to lack of friction the objects shown there can never move horizontally, which may appear slightly odd. The objects when left to come to rest on the ground will actually slowly sink into the ground, ie. they have a small downward velocity instead of the expected zero velocity. We

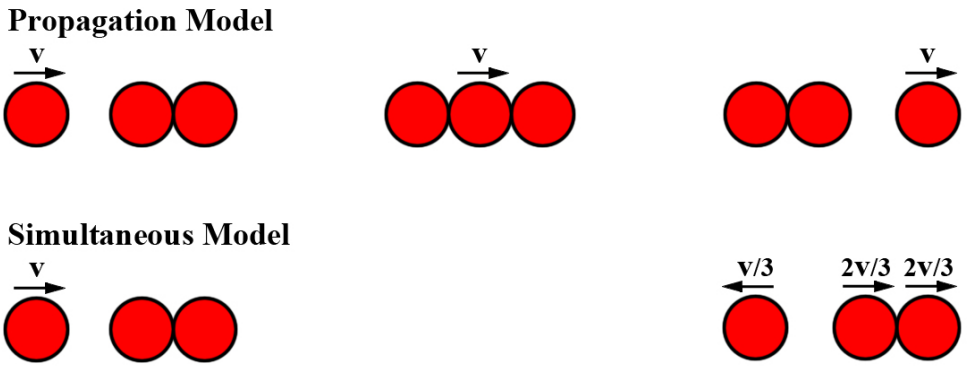


Figure 16: On the left we have one ball moving with velocity v towards two identical stationary balls. On the right we see the final result according to the propagation and simultaneous models.

can try to fix this by artificially moving the object to lie above the ground. This has the unfortunate side effect of adding energy to the object (we have added potential energy). This results in the objects never actually coming to rest, as the energy they lose coming to rest is offset by the energy they gain by forcing them to lie above the ground. We now see objects continually vibrating just above the floor. See Videos 07 and 08 on the cd-rom. This vibration is still preferable to the sinking effect, and as a result can be seen in many implementations of rigid body simulation algorithms. It is in fact quite a deep problem which cannot be completely removed from any algorithm. It arises from the simple fact that if we allow a ball to bounce on the ground (with $e < 1$) and come to rest, then the time taken for each bounce forms a geometric decreasing sequence. So the ball will make an infinite number of bounces in a finite time. Any algorithm obviously can not deal with each collision and terminate in a finite time. In such a simple case we can use our knowledge of the motion to allow the computer to deal with multiple bounces in one go, but in general the motion is sufficiently complicated that we need to deal with it on a per collision basis.

To understand why the object sinks let us look at a ball coming to rest on the ground with $e = 0$. As soon as it hits the ground the ball receives an impulse that sets the velocity to zero. Since all the collisions have been resolved, the algorithm moves all the objects as if they were in free fall. This results in the ball moving slightly downward due to gravity, and slightly penetrating the ground. Our collision detection algorithm detects this as a collision and our algorithm resets the balls velocity to zero again. Then the ball is moved as if under free fall again, causing it to penetrate the ground even further, and so on. A similar situation, resulting in this unwanted sinking effect, occurs in more complicated simulations. Also this example provides problems with adaptive time stepping algorithms as we discussed earlier. As the object is constantly in contact with the ground the collisions occur continuously causing such algorithms to fail.

This problem by its very nature can not be got rid of, but we can minimize the effects it causes, and we do this by analytical methods (also known as constraint-based methods) [1], which we will review in detail in the next section.

The last remaining restriction we placed on the application of the formulae in Section 3.3 was that there was no friction. Without friction we can use Newton's law of restitution. It gives fairly realistic and believable results. Without it we have no way of calculating the magnitude of the impulses during collisions. In order to have physically accurate looking simulations we need to include friction at some point, but that is easier said than done. We will briefly discuss friction in a later section and look at the

challenges and problems it poses. For now we will simply continue assuming that there is no friction and try to develop our algorithm to reduce the sinking effect.

3.4 Analytical Force Determination

Introduction

To get rid of the sinking effect we need to model a resting contact point (ie. a contact point formed when an object comes to rest) and we must do so with some care. In the impulse-based method all contacts are modeled using impulses [15]. Often what is used is a rapid succession of small magnitude impulses, known as microcollisions. We will however opt to model resting contacts using forces instead of impulses, which is more natural [1]. As we will show in the rest of this section, we can make some reasonable constraints on the system of forces and reduce this to a linear programming problem.

Firstly since objects are coming to rest there are certain types of contacts that were considered degenerate using the simple impulse-based method of the previous section, but which are not under a resting contact model. Edge-face and face-face resting contacts are now likely to occur when objects are coming to rest. The force acting through such contacts can be decomposed into a series of forces acting in the same direction and through the vertices making up the intersection between the two objects. This decomposition can be done in such a way that the overall linear force and the torque produced is the same, hence for our purposes the two are equivalent (note however that the decomposition may not be unique). So we need only work out forces at the vertices making up the intersection. This means we can treat an edge-face resting contact as two vertex-face resting contacts (the two vertices taken from the end points of the colliding edge). Similarly a face-face resting contact can be dealt with as a collection of edge-edge and vertex-face collisions.

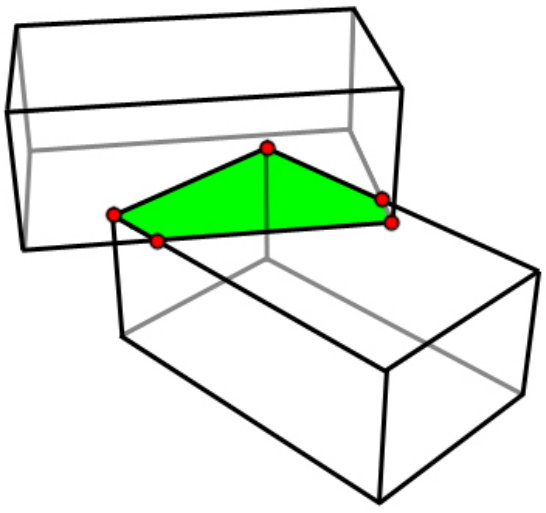


Figure 17: Outline of two boxes on top of each other, forming a face-face resting contact. The contact region has 5 vertices, 2 made from edge-edge intersections and 3 from vertex-face intersections. All forces acting between the objects through the contact region, can be reduced to 5 forces acting through the 5 vertices on the region's boundary.

Since we are dealing with forces instead of impulses we need to integrate them over time. Our algorithm for rigid body simulation now changes to:

1. Check for collisions using a collision detection algorithm. For each pair of objects found to be intersecting provide a list of vertex-face and edge-edge contact information. This information should include the point and direction any impulses or forces will act.
2. Search for a contact point that has a positive approach velocity. If such a contact exists apply an appropriate impulse to keep objects moving away from each other. The magnitude of the impulse is chosen so that Newton's law of restitution holds.
3. Check that all contacts have a negative approach velocity (ie. all objects are moving away from each other). If there exists a contact with a positive approach velocity go to step 2.
4. Identify all the contact points that have a small separation velocity. Such contact points we will artificially force to be resting contact points.
5. Work out the magnitudes of the forces at each of the resting contact points (simultaneously). Integrate forward the positions and orientations of all the objects under these forces. During the integration the magnitudes of the forces need to be continually recalculated and updated.
6. Display the new state of the simulation.
7. Repeat from step 1.

So given a list of resting contact points and the current state of the objects we need to somehow work out what system of forces is acting between the objects. We will do this by forming a list of constraints that we believe the system of forces should adhere to.

Non-penetration Constraints

To avoid the sinking effect an important constraint we want to impose is that objects are not allowed to penetrate. In this section we will formulate this constraint by defining a formula that represents a pair of object's relative acceleration towards each other. First we need to go through some definitions and notation.

Let us consider a single resting contact point between object A and object B. From our collision detection algorithm we are given that the force will act through point **r** in the world space and in the direction **n**, where **n** points away from object B and is a unit vector. Note that we will be using a lot of the same notation that we set up for our treatment of simple impulse-based collisions.

Let $\mathsf{R}_A, \mathsf{R}_B$ be matrices representing the orientations of objects A and B respectively. Now let us consider the point $\mathsf{R}_A^{-1}(\mathbf{r} - \mathbf{p}_A)$ in object A's model space and the point $\mathsf{R}_B^{-1}(\mathbf{r} - \mathbf{p}_B)$ in object B's model space. These points lie on their respective object's boundaries and furthermore when we transform them into the world space they both map to the position **r**. We will keep these points fixed in their model space with respect to time, and look at how their relative positions in world space change. Define $\boldsymbol{\alpha}(t), \boldsymbol{\beta}(t)$ to be object A's and object B's (respectively) point in world space at time t after the initial contact. At $t = 0$ we have $\boldsymbol{\alpha}(t) = \boldsymbol{\beta}(t) = \mathbf{r}$. Now we define a measure of their relative positions using

$$\chi(t) = [\boldsymbol{\alpha}(t) - \boldsymbol{\beta}(t)] \cdot \mathbf{n}(t)$$

Note that **n**, the direction the force will act, will also change with time as the objects move. If the objects are no longer in contact, $\mathbf{n}(t)$ is a bit ambiguous. To remedy this let us make some assumptions.

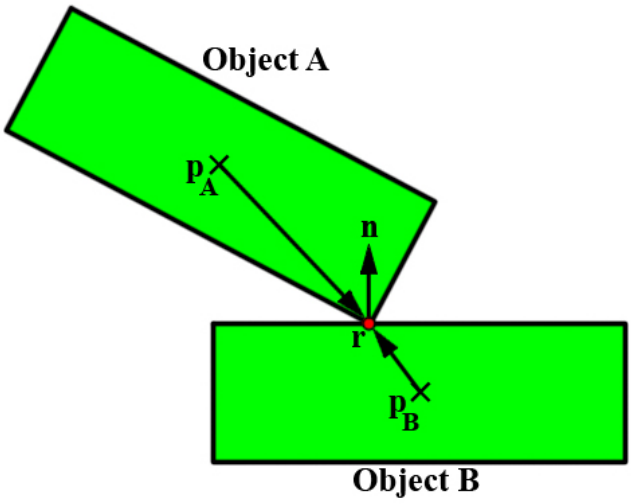


Figure 18: Objects A and B collide at **r** in world space. The normal of the contact is given by vector **n** which points away from object B. Also shown are the positions of the centre of masses, $\mathbf{p}_A$, $\mathbf{p}_B$, and the vectors representing the relative position of the contact from the centre of masses, $\mathbf{r} - \mathbf{p}_A$, $\mathbf{r} - \mathbf{p}_B$.

Let us assume that the contact is of the type vertex-face (we will deal with edge-edge contacts later). Also without loss of generality let us assume the vertex and face, involved in the contact, belong to A and B respectively. Hence we more formally define $\mathbf{n}(t)$ as the normal to object B's face. When $t = 0$ this is our **n** as defined by the direction the force acts but even if the objects are no longer touching, $\mathbf{n}(t)$ still makes sense.

Let us now look at $\chi(t)$ in more detail. When it is positive it implies the A's vertex lies outside B, and when it is negative it lies inside B and there is penetration. $\chi(t) = 0$ occurs when A's vertex lies exactly on B's face. This is the initial scenario $\chi(0) = 0$. Differentiating gives us

$$\dot{\chi}(t) = [\dot{\boldsymbol{\alpha}}(t) - \dot{\boldsymbol{\beta}}(t)] \cdot \mathbf{n}(t) + [\boldsymbol{\alpha}(t) - \boldsymbol{\beta}(t)] \cdot \dot{\mathbf{n}}(t)$$

at $t = 0$. This can be written as

$$\begin{aligned} \dot{\chi}(0) &= [\dot{\boldsymbol{\alpha}}(0) - \dot{\boldsymbol{\beta}}(0)] \cdot \mathbf{n} \\ &= [\boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A) + \mathbf{v}_A - \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B) - \mathbf{v}_B] \cdot \mathbf{n} \end{aligned}$$

which is the relative speed of separation. If $\dot{\chi}(0)$ is positive then the objects are moving away from each other, in which case we have no collision to deal with. If $\dot{\chi}(0)$ is negative then we can apply impulses as we did in our simple impulse-based model. When $\dot{\chi}(0) = 0$ then we have a resting contact, and so we can assume that this is true for all the resting contacts we are dealing with. Since $\chi(t)$ represents relative position, and $\dot{\chi}(t)$ represents relative velocity, then unsurprisingly $\ddot{\chi}(t)$ will represent relative acceleration.

$$\ddot{\chi}(t) = [\ddot{\boldsymbol{\alpha}}(t) - \ddot{\boldsymbol{\beta}}(t)] \cdot \mathbf{n}(t) + 2[\dot{\boldsymbol{\alpha}}(t) - \dot{\boldsymbol{\beta}}(t)] \cdot \dot{\mathbf{n}}(t) + [\boldsymbol{\alpha}(t) - \boldsymbol{\beta}(t)] \cdot \ddot{\mathbf{n}}(t)$$

At $t = 0$ this is

$$\begin{aligned} \ddot{\chi}(0) = \quad & [\ddot{\boldsymbol{\alpha}}(0) - \ddot{\boldsymbol{\beta}}(0)] \cdot \mathbf{n} + 2[\dot{\boldsymbol{\alpha}}(0) - \dot{\boldsymbol{\beta}}(0)] \cdot \dot{\mathbf{n}}(0) \\ = \quad & [\dot{\boldsymbol{\omega}}_A \times (\mathbf{r} - \mathbf{p}_A) + \boldsymbol{\omega}_A \times (\boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A)) + \dot{\mathbf{v}}_A \\ & -\dot{\boldsymbol{\omega}}_B \times (\mathbf{r} - \mathbf{p}_B) - \boldsymbol{\omega}_B \times (\boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B)) - \dot{\mathbf{v}}_B] \cdot \mathbf{n} \\ & +2[\boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A) + \mathbf{v}_A - \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B) - \mathbf{v}_B] \cdot (\boldsymbol{\omega}_B \times \mathbf{n}) \end{aligned}$$

For edge-edge contacts this formula is slightly different. Let us define $\mathbf{e}_A$ and $\mathbf{e}_B$ as vectors that lie in the same direction as the edges of A and B, respectively. Then the force will act in the direction $\mathbf{e}_A \times \mathbf{e}_B$. We can choose the directions of $\mathbf{e}_A$ and $\mathbf{e}_B$ such that $\mathbf{e}_A \times \mathbf{e}_B$ points away from B. Let us redefine $\mathbf{n}(t)$ as $\mathbf{e}_A \times \mathbf{e}_B$ (the fact that it is not a unit vector is inconsequential). $\ddot{\chi}(0)$ now becomes

$$\begin{aligned}\ddot{\chi}(0) = \quad & [\dot{\boldsymbol{\omega}}_A \times (\mathbf{r} - \mathbf{p}_A) + \boldsymbol{\omega}_A \times (\boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A)) + \dot{\mathbf{v}}_A \\ & -\dot{\boldsymbol{\omega}}_B \times (\mathbf{r} - \mathbf{p}_B) - \boldsymbol{\omega}_B \times (\boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B)) - \dot{\mathbf{v}}_B] \cdot (\mathbf{e}_A \times \mathbf{e}_B) \\ & +2[\boldsymbol{\omega}_A \times (\mathbf{r} - \mathbf{p}_A) + \mathbf{v}_A - \boldsymbol{\omega}_B \times (\mathbf{r} - \mathbf{p}_B) - \mathbf{v}_B] \cdot [(\boldsymbol{\omega}_A \times \mathbf{e}_A) \times \mathbf{e}_B + \mathbf{e}_A \times (\boldsymbol{\omega}_B \times \mathbf{e}_B)]\end{aligned}$$

The following variables $\dot{\mathbf{v}}_A, \dot{\boldsymbol{\omega}}_A, \dot{\mathbf{v}}_B, \dot{\boldsymbol{\omega}}_B$ all depend on the forces that we are trying to work out. Since $\chi(0) = 0$ and $\dot{\chi}(0) = 0$, to avoid object A and B penetrating each other we must have $\ddot{\chi}(0) \geq 0$. This gives us our first set of constraints on the system. For each resting contact we can define a function like $\chi(t)$. Let us denote it by $\chi_i(t)$ for the ith resting contact. So we have $\ddot{\chi}_i(0) \geq 0$ for all i. Next let us explicitly calculate how $\ddot{\chi}_i(0)$ depends on the forces.

Suppose a force $\mathbf{F}$ acts on object A in our resting contact. Then we know that the force on object B must be $-\mathbf{F}$, by Newton's third law of motion. Furthermore we know that the direction the force acts in is $\mathbf{n}$ (at $t = 0$). So we can write $\mathbf{F} = f\mathbf{n}$, where f is the magnitude of the force on object A. Hence our problem reduces to finding f for each contact. Let us define f_i as the magnitude of the force on A in contact i. Since forces at contacts can only push objects apart (not pull them together) this gives us another set of constraints on our system of forces, for all i, $f_i \geq 0$. Before we can continue, we need to make some more definitions and go over some notation that should hopefully make the equations clearer.

Let the number of contacts in our simulation be given by N_c. For contact i let us define the following (at $t = 0$)

$$\begin{aligned}A_i &= \text{Object A} \\ B_i &= \text{Object B} \\ \mathbf{n}_i &= \text{Direction force acts (chosen to point away from } B_i\text{).} \\ \mathbf{r}_i &= \text{The contact point in world space} \\ f_i &= \text{A scalar representing force } (f_i\mathbf{n}_i \text{ is the force that acts upon } A_i\text{).}\end{aligned}$$

Note that when the contact is of type vertex-face, $\mathbf{n}_i$ is just a normal to the plane containing object B_i's face. When the contact is of type edge-edge then $\mathbf{n}_i$ is the cross product of the two edge vectors. Our formulae will not need this distinction and so we have represented it by the one symbol. Now let us define the number of objects in our simulation by N_{obj}. For object i let us define (at $t = 0$) the following

$$\begin{aligned}m_i &= \text{Total mass of object } i \\ \mathsf{I}_i &= \text{Inertia tensor of object } i \text{ in world space} \\ \mathbf{p}_i &= \text{Position of the centre of mass of object } i \text{ in the world space} \\ \mathsf{R}_i &= \text{The matrix representing the orientation of object } i \\ \mathbf{v}_i &= \text{Velocity of object } i \\ \boldsymbol{\omega}_i &= \text{Object } i\text{'s angular velocity} \\ \boldsymbol{\tau}_i &= \text{The overall torque on object } i \\ \mathbf{L}_i &= \text{The angular momentum of object } i\end{aligned}$$

We will also need to make use of the following function

$$\phi(i,j) = \begin{cases} 1 & \text{if } j = A_i \\ 0 & \text{if } j \neq A_i \text{ and } j \neq B_i \\ -1 & \text{if } j = B_i \end{cases}$$

So $\phi(i,j)$ tells us whether object j is involved in contact i or not, and if it is involved whether it acts as object A or B. Such a function can easily be created from the contact information and can be stored as a simple 2-dimensional array. We could consider it as a matrix but there is little computational advantage to doing so. Since most rigid body simulations involve gravity, we need to give the user some additional control over the simulation, to allow other forces to act other than those created due to resting collisions. Let us assume that object i feels a user-defined acceleration of $\mathbf{g}_i$ and an angular acceleration of $\mathbf{h}_i$. So to simulate motion under no external forces we set all $\mathbf{g}_i$ and $\mathbf{h}_i$ to $\mathbf{0}$. To simulate motion under a gravitational acceleration g, we set $\mathbf{h}_i = \mathbf{0}$ and $\mathbf{g}_i = \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix}$ for all i.

Given this notation we can write equations for each object's acceleration. Object i has the following linear acceleration and torque

$$\dot{\mathbf{v}}_i = \mathbf{g}_i + \frac{1}{m_i} \sum_{j=1}^{N_c} \phi(j,i) f_j \mathbf{n}_j \tag{15}$$

$$\boldsymbol{\tau}_i = \mathsf{I}_i \mathbf{h}_i + \sum_{j=1}^{N_c} (\mathbf{r}_j - \mathbf{p}_i) \times [\phi(j,i) f_j \mathbf{n}_j] \tag{16}$$

From (8) we have

$$\boldsymbol{\tau}_i = \dot{\mathbf{L}}_i = \mathsf{I}_i \dot{\boldsymbol{\omega}}_i + \widetilde{\boldsymbol{\omega}}_i \mathsf{I}_i \boldsymbol{\omega}_i$$

rearranging and substituting in (16) gives

$$\begin{aligned} \dot{\boldsymbol{\omega}}_i &= \mathsf{I}_i^{-1}(\boldsymbol{\tau}_i - \widetilde{\boldsymbol{\omega}}_i \mathsf{I}_i \boldsymbol{\omega}_i) \\ &= \mathbf{h}_i - \mathsf{I}_i^{-1} \widetilde{\boldsymbol{\omega}}_i \mathsf{I}_i \boldsymbol{\omega}_i + \sum_{j=1}^{N_c} \phi(j,i) f_j \mathsf{I}_i^{-1} [(\mathbf{r}_j - \mathbf{p}_i) \times \mathbf{n}_j] \end{aligned} \tag{17}$$

Note that both $\dot{\mathbf{v}}_i$ and $\dot{\boldsymbol{\omega}}_i$ are linear with respect to the force magnitudes f_j. This property translates across when we substitute into $\ddot{\chi}(0)$. If contact i is a vertex-face contact we have

$$\begin{aligned} \ddot{\chi}_i(0) =& [\dot{\mathbf{v}}_{A_i} - \dot{\mathbf{v}}_{B_i} + \dot{\boldsymbol{\omega}}_{A_i} \times (\mathbf{r}_i - \mathbf{p}_{A_i}) - \dot{\boldsymbol{\omega}}_{B_i} \times (\mathbf{r}_i - \mathbf{p}_{B_i})] \cdot \mathbf{n}_i \\ &+ [\boldsymbol{\omega}_{A_i} \times (\boldsymbol{\omega}_{A_i} \times (\mathbf{r}_i - \mathbf{p}_{A_i})) - \boldsymbol{\omega}_{B_i} \times (\boldsymbol{\omega}_{B_i} \times (\mathbf{r}_i - \mathbf{p}_{B_i}))] \cdot \mathbf{n}_i \\ &+ 2[\mathbf{v}_{A_i} - \mathbf{v}_{B_i} + \boldsymbol{\omega}_{A_i} \times (\mathbf{r}_i - \mathbf{p}_{A_i}) - \boldsymbol{\omega}_{B_i} \times (\mathbf{r}_i - \mathbf{p}_{B_i})] \cdot (\boldsymbol{\omega}_{B_i} \times \mathbf{n}_i) \end{aligned}$$

Substituting in for $\dot{\mathbf{v}}_{A_i}, \dot{\mathbf{v}}_{B_i}, \dot{\boldsymbol{\omega}}_{A_i}$ and $\dot{\boldsymbol{\omega}}_{B_i}$ gives

$$\begin{aligned} \ddot{\chi}_i(0) =& \sum_{j=1}^{N_c} f_j \left(\phi(j, A_i) \left[\frac{\mathbf{n}_j}{m_{A_i}} - (\mathbf{r}_i - \mathbf{p}_{A_i}) \times \mathsf{I}_{A_i}^{-1} [(\mathbf{r}_j - \mathbf{p}_{A_i}) \times \mathbf{n}_j] \right] \right. \\ &\left. - \phi(j, B_i) \left[\frac{\mathbf{n}_j}{m_{B_i}} - (\mathbf{r}_i - \mathbf{p}_{B_i}) \times \mathsf{I}_{B_i}^{-1} [(\mathbf{r}_j - \mathbf{p}_{B_i}) \times \mathbf{n}_j] \right] \right) \cdot \mathbf{n}_i \\ &+ [\mathbf{g}_{A_i} - \mathbf{g}_{B_i} + \boldsymbol{\omega}_{A_i} \times (\boldsymbol{\omega}_{A_i} \times (\mathbf{r}_i - \mathbf{p}_{A_i})) - \boldsymbol{\omega}_{B_i} \times (\boldsymbol{\omega}_{B_i} \times (\mathbf{r}_i - \mathbf{p}_{B_i})) \\ &- (\mathbf{r}_i - \mathbf{p}_{A_i}) \times (\mathbf{h}_{A_i} - \mathsf{I}_{A_i}^{-1} \widetilde{\boldsymbol{\omega}}_{A_i} \mathsf{I}_{A_i} \boldsymbol{\omega}_{A_i}) + (\mathbf{r}_i - \mathbf{p}_{B_i}) \times (\mathbf{h}_{B_i} - \mathsf{I}_{B_i}^{-1} \widetilde{\boldsymbol{\omega}}_{B_i} \mathsf{I}_{B_i} \boldsymbol{\omega}_{B_i})] \cdot \mathbf{n}_i \\ &+ 2[\mathbf{v}_{A_i} - \mathbf{v}_{B_i} + \boldsymbol{\omega}_{A_i} \times (\mathbf{r}_i - \mathbf{p}_{A_i}) - \boldsymbol{\omega}_{B_i} \times (\mathbf{r}_i - \mathbf{p}_{B_i})] \cdot (\boldsymbol{\omega}_{B_i} \times \mathbf{n}_i) \end{aligned} \tag{18}$$

For edge-edge contacts we simply replace the very last $\cdot(\boldsymbol{\omega}_{B_i} \times \mathbf{n}_i)$ term by $\cdot[(\boldsymbol{\omega}_{A_i} \times \mathbf{e}_{A_i}) \times \mathbf{e}_{B_i} + \mathbf{e}_{A_i} \times (\boldsymbol{\omega}_{B_i} \times \mathbf{e}_{B_i})]$. Also this formula must be modified when dealing with immovable objects or equivalently objects with infinite mass. In order for the equations to work, it is best to let the object have some finite mass (and finite inertia tensor) and then merely set $\phi(i, j) = 0$, when j is an immovable object. This just ensures that the object never feels the effects of any forces that are applied to it. When two objects of infinite mass collide, we obviously cannot resolve the collision, so care must be taken to identify and then ignore such collisions.

Although (18) is quite a large and somewhat unwieldy formula, the only terms we can not obtain from the state of the simulation are the force magnitudes f_j. This shows that $\ddot{\chi}_i(0)$ is in fact just a linear combination of the force magnitudes (plus the constant term that spans the last three lines of the equation). Hence for simplicity let us now rewrite the formula as

$$\ddot{\chi}_i(0) = b_i + \sum_{j=1}^{N_c} M_{ij} f_j$$

Where b_i and M_{ij} are scalars that are terms in (18). Let $\mathbf{b}$ be an N_c-dimensional vector whose elements are b_i. Similarly let $\ddot{\boldsymbol{\chi}}$ and $\mathbf{f}$ be N_c-dimensional vectors formed from $\ddot{\chi}_i(0)$ and f_i respectively. Finally let M be an $N_c \times N_c$ matrix with elements M_{ij}. This gives us

$$\ddot{\boldsymbol{\chi}} = \mathsf{M}\mathbf{f} + \mathbf{b}$$

Since we know $\ddot{\boldsymbol{\chi}} \geq \mathbf{0}$ we can write our constraints as

$$\mathsf{M}\mathbf{f} \geq -\mathbf{b}$$
$$\mathbf{f} \geq \mathbf{0}$$

Linear constraints in this form can be solved by linear programming. Unfortunately, as we will see in the next section, we need to add an additional constraint which will make solving the forces using only linear programming untenable.

Vanishing Contact Points

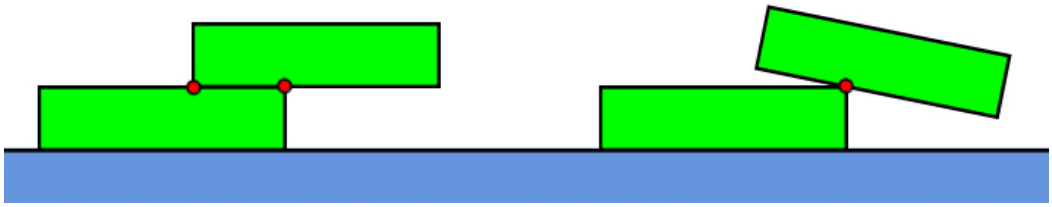


Figure 19: On the left we have 2 rectangles resting on top of each other. On the right we see them after one time step. The contact points between the rectangles are indicated by the small dots.

Consider a box resting on the ground (under gravity). Currently the constraints we have formulated merely tell us that when we have two objects that are in resting contact we should apply forces to ensure that they do not accelerate towards each other. This means it is perfectly acceptable to make the forces between the box and the floor so large that not only does it not accelerate toward the floor but it in fact accelerates away from it. This is clearly a problem as in reality the box would continue to

remain at rest neither accelerating away nor towards the floor. We resolve this issue by introducing the concept of a vanishing contact point, see Figure 19. The two rectangles are initially at rest and have two resting contact points, however after a small time step gravity causes the situation to change to that on the right of Figure 19. We now have one less contact point, the one on the left has vanished. Even though the two objects were initially at rest, they were accelerating away from each other at the contact point on the left, as a result the contact point vanished immediately. From now on let us call a resting contact point at which the bodies are accelerating away from each other a vanishing contact point.

If two bodies are not in contact the contact force between them is zero. So in the case of a vanishing contact point the contact force at that point will be zero after an infinitesimal time step as the contact point will vanish. Since forces are continuous, we can conclude that the contact force at the vanishing contact point must be zero. By definition two bodies must be accelerating towards each other at all their non-vanishing contact points. We can choose positive contact forces at these points to ensure penetration does not occur. In fact we choose the forces such that the acceleration between the bodies at these points are zero, i.e. the smallest force necessary to stop penetration. This gets rid of our problem of allowing bodies to accelerate away from each other when they are in a state of stable rest. Also this is consistent with Newton's law of restitution. Since the approach velocity is zero at a resting contact point, the separation velocity after a collision should be zero as well. In other words we should apply forces that keep the relative velocities at resting contact points zero, unless the force required to do this would pull instead of push the bodies apart (which is what would occur at a vanishing contact point).

In our previous notation, for the resting contact point i, we write these extra conditions as

$$\begin{array}{llll} \ddot{\chi}_i(0) \geq 0 & \textit{and} & f_i = 0 & \textit{if } i \textit{ is a vanishing contact point.} \\ \ddot{\chi}_i(0) = 0 & \textit{and} & f_i \geq 0 & \textit{if } i \textit{ is a non-vanishing contact point.} \end{array}$$

We can write the above into one expression $f_i\ddot{\chi}_i(0) = 0$. Note that regardless of this new rule $f_i\ddot{\chi}_i(0) \geq 0$, since we know that $\ddot{\chi}_i(0) \geq 0$ and $f_i \geq 0$ (from our previous constraints). This means we can rewrite the statement $f_i\ddot{\chi}_i(0) = 0$ for all i as

$$\sum_{i=1}^{N_c} f_i\ddot{\chi}_i(0) = 0$$

which is equivalent to

$$\mathbf{f}^T\ddot{\boldsymbol{\chi}} = \mathbf{0}$$

and

$$\mathbf{f}^T\mathsf{M}\mathbf{f} = -\mathbf{f}^T\mathbf{b}$$

This is in fact our last constraint on the system of forces. The last version of our vanishing contact constraint is clearly quadratic with respect to $\mathbf{f}$, which changes the problem of finding $\mathbf{f}$ from a linear to a quadratic programming problem. Unfortunately quadratic programming problems are in general NP-hard, so in the next section we will look at a heuristic method for getting around this last constraint, and returning it to a linear programming problem.

Finding the Vanishing Contact Points

Currently we have to find a system of forces **f** that satisfies the following three constraints

$$\begin{array}{lrcl} \text{(a)} & \mathsf{M}\mathbf{f} & \geq & -\mathbf{b} \\ \text{(b)} & \mathbf{f} & \geq & \mathbf{0} \\ \text{(c)} & \mathbf{f}^T\mathsf{M}\mathbf{f} & = & -\mathbf{f}^T\mathbf{b} \end{array}$$

If we were given a list of which contact points would be vanishing and which would not be, we could incorporate (c) into (a) and (b) leaving the problem solvable by linear programming. For contact i, (a) tells us that $\sum_{j=1}^{N_c} M_{ij}f_j \geq b_i$. But if we are given that i is not a vanishing contact point we can incorporate this information into (a) by changing the inequality to equality so it now reads $\sum_{j=1}^{N_c} M_{ij}f_j = b_i$. If however i is a vanishing contact point we leave (a) as an inequality and merely add the additional constraint that $f_i = 0$. If $f_i = 0$ we can save some time finding a feasible solution by setting $M_{ki} = 0$ for all k (we are essentially just removing all terms that contain f_i). Just to be clear let us illustrate this with an example. Suppose we are trying to find three forces subject to

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \geq \begin{pmatrix} -b_1 \\ -b_2 \\ -b_3 \end{pmatrix} \qquad \begin{array}{l} f_1 \geq 0 \\ f_2 \geq 0 \\ f_3 \geq 0 \end{array}$$

$$\mathbf{f}^T\mathsf{M}\mathbf{f} = -\mathbf{f}^T\mathbf{b}$$

Now if we are given that contact 1 and 3 are non-vanishing and contact 2 is a vanishing contact point we can rewrite our constraints as

$$\begin{pmatrix} M_{11} & 0 & M_{13} \\ M_{21} & 0 & M_{23} \\ M_{31} & 0 & M_{33} \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} -b_1 \\ -b_2 \\ -b_3 \end{pmatrix} \qquad \begin{array}{l} f_1 \geq 0 \\ f_2 = 0 \\ f_3 \geq 0 \end{array}$$

which can be solved using linear programming. In general making such changes will force (c) to hold and keep the problem linear. Unfortunately it is not always clear which contact points will be vanishing and which ones won't, and for this method to work we need to know this information before we solve for **f**.

We can always guess which contacts will be vanishing. If we try to solve for **f** using linear programming and it fails to find a feasible solution we will know that our guess was incorrect and can try another guess. Unfortunately for n contacts there are 2^n configurations of vanishing contact points, and searching through them all becomes quickly computationally expensive as the number of contacts increases. Nevertheless this works well for simple simulations with relatively few objects (which have few vertices). It should be noted that there is no evidence to suggest that there is only one unique configuration of vanishing contact points that will lead to a feasible solution. There may be several configurations that lead to a solution but this was never observed during any of the simulations that were run. Also even with a correct configuration of vanishing contact points there may be several values of **f** that satisfy the constraints. For example Figure 20 shows a beam resting on three pillars. There are three points of contact one at the centre of the beam and two more equidistance from the centre. Clearly these three contact points are non-vanishing, however the forces required to keep the beam at rest are not unique. One solution is that the entire weight of the beam is supported by the central pillar and no force is applied by the pillars at either end. Another valid solution is that no force

is applied by the central pillar but instead the weight is equally supported by the two outer pillars. This lack of uniqueness is a direct result of bodies being simulated as rigid. Even though the solution to the linear program may not be unique we hope that the overall motion is.

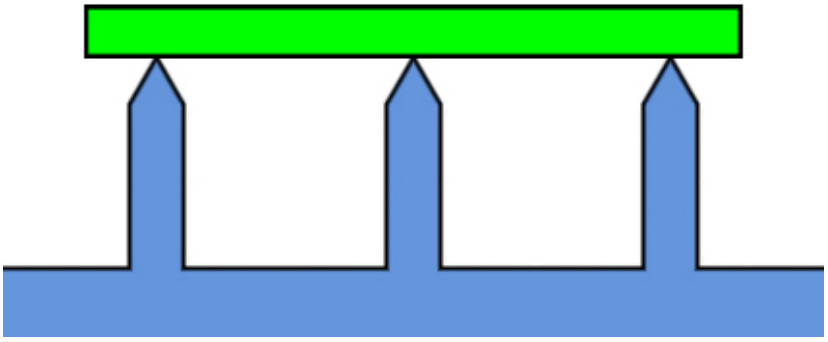


Figure 20: A beam resting on three pointed pillars.

A good initial guess is that there are no vanishing contact points. This is reasonable because vanishing contact points do not last very long, if one exists then by the next time step the bodies will have accelerated away causing the contact to vanish. Vanishing contact points tend to occur very rarely. However if there is no feasible solution for $\mathbf{f}$, when we assumed all points were non-vanishing, then we need to employ a heuristic method to guess which points are vanishing.

Suppose we are given an approximate solution to $\mathbf{f}$ that satisfies constraints (a) and (b); let us denote it by $\mathbf{f}_a$. Now using $\mathbf{f}_a$ let us calculate $\ddot{\chi}_i(0)$ for all i. If $\mathbf{f}_a$ was an exact solution then $\ddot{\chi}_i(0) = 0$ if contact i is non-vanishing, and $\ddot{\chi}_i(0) \geq 0$ if i is a vanishing contact point. Our hope is that, even if $\mathbf{f}_a$ isn't an exact solution but only an approximate one, then looking at whether $\ddot{\chi}_i(0)$ is zero or not will indicate which points are non-vanishing or vanishing. We can calculate an $\mathbf{f}_a$ by minimizing the objective function $\sum_{i=1}^{N_c} f_i$ subject to the constraints given by (a) and (b) (which is a linear programming problem). The idea is that by minimizing $\sum f_i$ we are also minimizing $\sum f_i \ddot{\chi}_i(0)$, and if we manage to minimize it to zero then that will mean we have an exact solution as opposed to an approximate one. If $\mathbf{f}_a$ is an exact solution then the forces will do no net work, so by choosing it such that it minimizes the total force, we hope that it will do little net work.

In general this method is quite good at predicting vanishing points, which allows us to calculate an exact value for $\mathbf{f}$, but there are of course situations where it fails. In such situations we just use the approximate force solution $\mathbf{f}_a$. This will violate constraint (c) and add energy to the system, however the way we calculated $\mathbf{f}_a$ means the energy we add will be small. This seems to be a necessary tradeoff we need to make for speed. Because vanishing contact points seldom exist, and if they do we have a good way of predicting them, we rarely have to resort to violating (c) and so the overall motion looks reasonable.

Using this method eliminates the sinking effect we experienced with the simple impulse-based model (see Video 09 on the cd-rom). There is still some vibration of the object, but its amplitude has been greatly reduced and with a small time step it can be made smaller than a pixel, so it's imperceptible. This effect stems from the fact that before we do any analytical force determination we have to apply impulses to ensure any non-resting contacts are resolved. Our constraint-based method assumed that the contact points we were working with have zero velocity, however as an object comes to rest it is impossible to simulate the infinite collisions needed for it to reach zero velocity. As a result we allowed

contacts to be classified as resting as long as their separation velocity was small, but not necessarily zero. Calculating and applying the forces as we have done in this section will merely ensure that the objects in contact will have zero relative acceleration towards each other. Now if the objects' centres of mass have a positive approach velocity (no matter how small) then applying our forces will not stop the sinking but merely slow it down. This would not occur if the resting contacts were of zero velocity, but they are very unlikely to be. So to get around this problem we ensure that every contact point has a positive separation velocity before we choose resting contact points. This forces the objects' centres of mass to have a positive separation velocity which is usually very small. This stops sinking but it means eventually the objects will move away from each other, and when they are no longer in contact gravity will bring them back together, causing the cycle of events to repeat and resulting in a small vibration between the objects.

The method we have used to calculate the forces between objects can also be modified to calculate impulses for the simultaneous model (that deals with multiple impacts). Currently we are using the propagation model for handling what happens when an object has more than one non-resting contact. However as the number of contacts increases the propagation method becomes computationally expensive. For example it would struggle simulating several boxes stacked on top of each other. In such a situation it is useful to be able to apply all the impulses simultaneously in one go. Unfortunately how Newton's law of restitution extends to simultaneous contacts is somewhat ambiguous. We will briefly go over how to calculate the impulses for the simultaneous model.

First let us identify all the contact points. Let $\mathbf{n}_i$ be the direction the impulse will act in for contact i. Let u_i be the speed of approach before the collision, and v_i be the speed of separation after the collision (in the direction $\mathbf{n}_i$). Let j_i be the magnitude of the impulse and e_i the coefficient of restitution. Now if the contact point has $u_i < 0$ then the bodies are moving away from each other at that point. We can ignore such contact points for now, but after we apply the simultaneous impulses to all the other points we will need to recheck that these points are still separating, and if they are not we will need to repeat the entire procedure. The number of times we have to repeat the entire procedure will still be much smaller than the number that is carried out by the propagation method. Newton's law of restitution tells us that for a single collision $v_i = e_i u_i$, for multiple collisions we will extend it to

$$v_i \geq e_i u_i \quad \text{for all } i$$

We require it to be an inequality as another collision may inadvertently cause the separation speed to be larger than what we expected. Just like when we were calculating forces we have the constraint that impulses push not pull, which we write as $j_i \geq 0$. These two constraints by themselves allow disproportionately large values of j_i, which we rectify with the constraint that if $v_i > 0$ then $j_i = 0$. So now we have for all i

$$\begin{array}{lrcl} \text{(a)} & v_i & \geq & e_i u_i \\ \text{(b)} & j_i & \geq & 0 \\ \text{(c)} & j_i(v_i - e_i u_i) & = & 0 \end{array}$$

Note that v_i is dependent on the impulses and is in fact linear with respect to them. This means that (a) and (b) could be solved by linear programming if we can get around (c) which is quadratic. We use the same heuristic as we outlined in this section to solve for j_i. It can be shown, when $e_i = 1$ for all i, that the impulses calculated will conserve the kinetic energy, and that when there is only one collision it reduces to Newton's law of restitution, which are useful properties of this formulation.

3.5 Summary

For clarity let us give a brief summary of the entire algorithm in this section. This should help us see how all the components we have discussed fit together and give a general overview of the algorithm.

Before we begin we collate information about the objects we will be simulating. We need to know an object's boundary, its centre of mass, its total mass, and its inertia tensor. We also need to know the initial state of the object which means we need to know where in world space the object resides, at what orientation, and what its angular and linear velocity are. We can also allow the user to specify any forces or torques that will act on the object, for example the force due to gravity.

Once we know the initial state of the simulation, we move on to evolving the simulation using the information given. First we check for penetrations between objects. This is done by some collision detection scheme. For each penetration we store the normal of the impact, and the point at which the contact occurred.

If any contact points exist we check whether the objects involved have a positive speed of approach. If they are separating we can ignore the penetration, otherwise we apply impulses to force them to have a positive speed of separation. We continually check the contact points and apply impulses until we have ensured that at all contact points the speed of separation is positive (this is necessary to stop the unwanted sinking effect).

Next we identify from the list of contact points which ones are good candidates to be treated as resting contact points. We do this by setting an upper bound on the separation speed. If a contact point has a speed of separation that is sufficiently small (i.e. lower than that of our upper bound) we deem it to be a resting contact point.

We now need to find the magnitude of the forces that we should apply at the resting contact points to produce a realistic looking and physically accurate motion. We do this by reformulating the problem into a linear program. The values we need for the linear program can be obtained from the coefficients in equation (18). Also we need to determine which of the resting contact points are vanishing contact points. Given the set of vanishing contact points we can modify our constraints and keep them linear (as opposed to quadratic). We initially guess there to be no vanishing contact points (as they occur rarely). We check whether this yields any feasible solutions. If not we use a heuristic method: specifically we solve the linear program with an objective function that minimizes the sum of the forces. This approximate solution allows us to make a better informed guess as to which points are vanishing and which are not. If this new guess still yields no solution to our problem, then we resort to just using the approximate solution (a tradeoff of accuracy for speed).

Given that we now know what forces are acting between the objects, we need to integrate the simulation forward in time. Substituting the magnitude of the forces into equations (15) and (16) tells us each object's linear acceleration and torque (optionally (17) can be used if angular velocity is being used instead of angular momentum). We now integrate over a small time step, for each object independently its linear acceleration, torque, linear velocity, and rate of change of orientation, which gives us new values for the linear velocity, angular momentum, position, and orientation of the object.

The simulation has now been evolved forward in time by a small time step, and we repeat our procedure by again checking for penetrations.

This algorithm was implemented in C++. Videos 10 to 15 on the accompanying cd-rom show some simple test simulations using this algorithm. Also on the cd-rom I have included the source code used to make the videos. This thesis is largely concerned with collision response as opposed to collision detection. Finding the correct normal to a collision is not a simple task and falls in the domain of collision detection; as such there are some restrictions on what simulations my code will model. The *Floor Collision* code will model a single object bouncing off the ground, whereas the *2D Boxes* code allows multiple objects to interact and collide but restricts the objects' motions to be 2D in nature and the objects themselves must be cuboids.

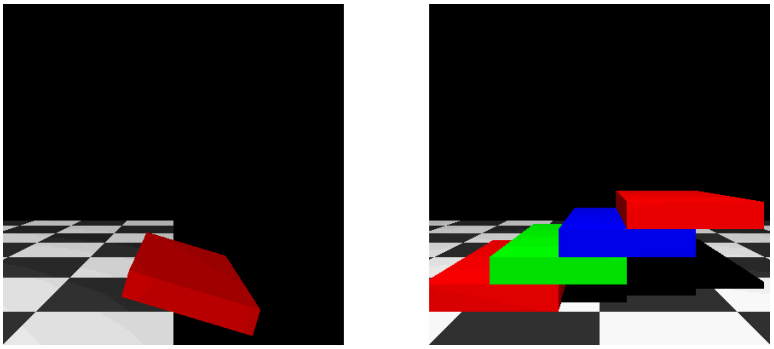


Figure 21: On the left we have a still from Video 10 and on the right a still from Video 11

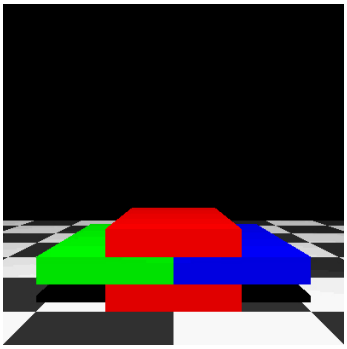


Figure 22: A still from Video 12

Video 10 shows a box sliding smoothly off the edge of a table and Video 11 shows a simulation of an unstable stack of four overhanging boxes, see Figure 21. Videos 12 to 15 show simulations done on boxes stacked in diamond shapes. Video 12 is of the 2-diamond (i.e. it has "side length" 2), see Figure 22. Videos 13, 14, and 15 are of a 3-diamond, 4-diamond, and 5-diamond respectively. The 2-diamond, 3-diamond, and 4-diamond should be stable and the 5-diamond unstable. This can be shown from the equations derived in [17]. The 2-diamond after a long period of time stays completely at rest, but the boxes in the 3-diamond slowly develops some horizontal velocity causing it eventually to collapse, see Figure 23. This is because the boxes are never truly at rest but are vibrating (though with a small amplitude). Since we are not simulating friction any horizontal motion is free to accumulate until it becomes significant. When we simulate the 4-diamond the errors accumulate much faster due to the increase in the number of contacts we are trying to resolve, see Figure 24. Both the 3-diamond and 4-diamond collapse despite initially being in a stable equilibrium. We expect the 5-diamond to collapse because it is initially unstable but it collapses in an unsymmetrical way, see Figure 25. This is due to the fact that multiple collisions are resolved using the propagation method, which breaks symmetry as it only deals with one collision at a time.

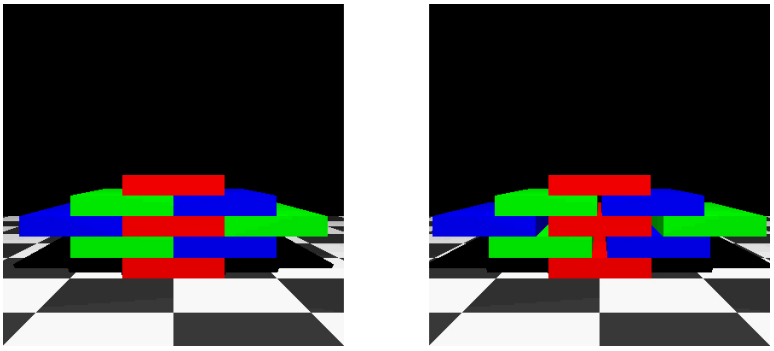


Figure 23: Stills from Video 13. On the left we have the initial arrangement, and on the right we see that the boxes have drifted slowly apart.

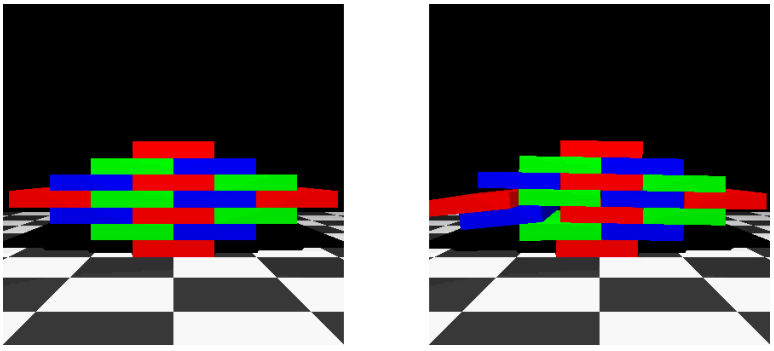


Figure 24: Stills from Video 14. On the left we have the initial arrangement, and on the right we see that it quickly becomes unstable.

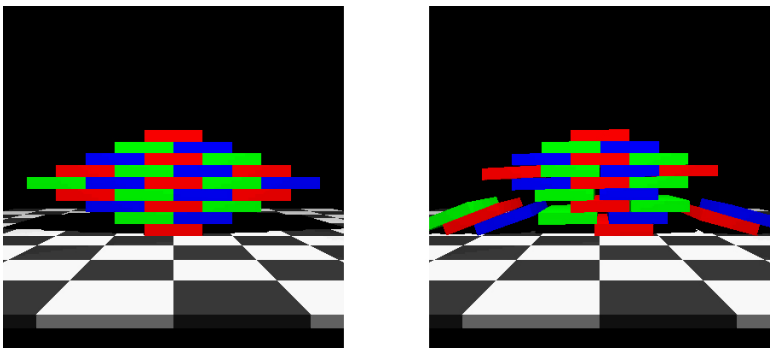


Figure 25: Stills from Video 15. On the left we have the initial arrangement, and on the right we see it collapse unsymmetrically.

4 Friction

Our current rigid body simulation algorithm appears to give realistic motion when all the surfaces are frictionless. To get truly realistic looking motions we need to incorporate friction into our collision response scheme. Unfortunately this is easier said then done.

The most standard and simplest model of friction is that given by Coulomb's law of friction which states that

$$F \leq \mu N$$

where F is the force due to friction, N is the normal force between two objects in contact, and μ is a constant known as the coefficient of friction (which is dependent only on the materials the two objects in contact are made from). When $F < \mu N$ then the friction is sufficient to stop the object from moving, and when $F = \mu N$ the object is either about to start moving or is already moving (more commonly it is said to be sliding). There are two main types of friction, static friction and kinetic friction. In general the coefficient of friction is slightly higher for static friction than it is for kinetic, but they are often set to the same value. For the purposes of our discussion we will treat them as the same thing.

By introducing friction we can no longer use Newton's law of restitution. This could add a significant amount of energy to the system, making the resulting motion look unrealistic, [19]. This is a major problem, as our entire impulse calculations were based on Newton's law of restitution. Impulses are merely the limit of the forces being applied between objects as we let the period the collision occurs in tend to zero. If instead we assumed the collision takes a small but finite time and try and look at what forces occur between objects when they collide under the presence of friction, we find even more problems. During the collision the objects may start off with a period where sliding is occurring but then the normal force may increase so that the objects stop sliding and enter a "sticking" phase. As the normal force decreases the objects could re-enter a sliding phase. The result of this is that during a collision we can not assume that $F < \mu N$ always holds or $F = \mu N$ always holds. In fact we can not even assume that the frictional force always acts in the same direction. This makes it very difficult to calculate impulses that incorporate friction.

Calculating the frictional force using the constraint-based method is also problematic. Firstly the algorithm we've outlined is only for resting contacts, the first step was to apply impulses to handle the non-resting contacts. As we've seen this is difficult, but let us suppose that we can do this. It turns out that there exist scenarios for which there is no system of forces that will satisfy the constraints of non-penetration and Coulomb's law. Worse still is that identifying such configurations is NP-complete [2].

Another problem which needs careful resolving is that of objects resting on slopes. Imagine a box on a ramp with a sufficiently low gradient that the box is prevented from slipping down the slope due to friction. We would expect the box to merely remain stationary resting on the ramp, however under our current algorithm we would actually observe the box slowly slipping down the ramp. The reason for this is that the box is not really resting on the ramp but is actually vibrating slightly. This means that occasionally it will break contact with the surface of the ramp. When this happens the frictional force will immediately drop to zero, and the box will free fall down the ramp. After a short period of time the box will eventually come into contact with the ramp again which will cause it to stop due to friction. However, this cycle repeats continuously which results in the box appearing to creep slowly down the ramp.

Simulating friction is still an active area of research. However, there are ways around the problems which incorporating friction creates, but they are outside the scope of this thesis. For further information see [2, 3, 10, 14, 15].

5 Concluding Remarks

This thesis has highlighted some of the numerous issues associated with rigid body simulation. Applying Newton's three laws of physics to rigid bodies is a much more formidable task than we would first expect, and one that requires a lot of thought. A lot of previous work done in this area has concentrated on the mathematical results and on improving the efficiency of algorithms. My aim throughout has been to show that there are also significant problems in producing a robust algorithm. Understanding these problems as well as solving them is just as important. Work in this area is by no means complete, and there are a variety of topics in need for further research. Some of the more notable ones are modeling friction, simulating articulated models, and handling closed-chain constraints. No doubt as computers increase in processing speed this area will attract increasing attention, and research such as this will increase in importance.

APPENDICES

A Fixed Axis of Rotation

Here we will prove that the axis of rotation remains fixed under free motion if and only if it corresponds to a principal axis. From equation (9) we have

$$\dot{\boldsymbol{\omega}} = -\mathsf{I}^{-1}(\boldsymbol{\omega} \times \mathsf{I}\boldsymbol{\omega}) \tag{19}$$

where $\boldsymbol{\omega}$ is the angular velocity and I is the inertia tensor.

If the rotation is in the direction of a principal axis then $\boldsymbol{\omega}$ must satisfy $\mathsf{I}\boldsymbol{\omega} = \lambda\boldsymbol{\omega}$ where λ is the respective eigenvalue. Substituting this into (19) gives us

$$\begin{aligned}\dot{\boldsymbol{\omega}} &= -\mathsf{I}^{-1}(\boldsymbol{\omega} \times (\lambda\boldsymbol{\omega})) \\ &= \mathbf{0}\end{aligned}$$

Hence the angular velocity and therefore the axis of rotation will remain fixed.

Let us now show that if the axis of rotation remains fixed then we must be rotating around a principal axis. When $\boldsymbol{\omega} = \mathbf{0}$ there is no rotation and from (19) we have $\dot{\boldsymbol{\omega}} = \mathbf{0}$ so the object will remain not rotating. The axis we are rotating about is therefore irrelevant, and we can assume for our purposes that the axis is a principal one. Hence for the rest of the proof we will assume that $|\boldsymbol{\omega}| > 0$.

For the axis of rotation to remain fixed we must have $\dot{\boldsymbol{\omega}} = \mu\boldsymbol{\omega}$ where μ is a scalar (which may vary with time). Substituting this into (19) and rearranging yields the following

$$\begin{aligned}\mu\boldsymbol{\omega} &= -\mathsf{I}^{-1}(\boldsymbol{\omega} \times \mathsf{I}\boldsymbol{\omega}) \\ -\mu\mathsf{I}\boldsymbol{\omega} &= \boldsymbol{\omega} \times \mathsf{I}\boldsymbol{\omega} \\ -\mu(\mathsf{I}\boldsymbol{\omega}) &= \widetilde{\boldsymbol{\omega}}(\mathsf{I}\boldsymbol{\omega})\end{aligned} \tag{20}$$

where $\widetilde{\boldsymbol{\omega}}$ represents the matrix

$$\begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix}$$

given that $\omega_1, \omega_2, \omega_3$ are the components of the vector $\boldsymbol{\omega}$. From (20) we can see that $\mathsf{I}\boldsymbol{\omega}$ is an eigenvector of $\widetilde{\boldsymbol{\omega}}$ with $-\mu$ as its corresponding eigenvalue. We can explicitly calculate the eigenvalues and eigenvectors of $\widetilde{\boldsymbol{\omega}}$. There are in fact three distinct eigenvalues (when $|\boldsymbol{\omega}| > 0$), and they are $0, i|\boldsymbol{\omega}|, -i|\boldsymbol{\omega}|$. Since μ and $\mathsf{I}\boldsymbol{\omega}$ are both real, and have no imaginary components, they must correspond to the eigenvalue 0 and its corresponding eigenvector $\boldsymbol{\omega}$. This means that $\mathsf{I}\boldsymbol{\omega}$ and $\boldsymbol{\omega}$ must be equivalent eigenvectors, and must only differ by some scalar factor. So

$$\mathsf{I}\boldsymbol{\omega} = \lambda\boldsymbol{\omega}$$

for some real scalar λ. This by definition means that the rotation is about a principal axis, and so we are done.

B Conservation of Energy After Applying an Impulse

We will be using the formulae and notation set out in Section 3.3 to show that equations (10), (11), (12), (13), and (14) conserve kinetic energy before and after applying the impulses (when $e = 1$). Kinetic energy is defined as

$$E = \frac{1}{2}m\mathbf{v}\cdot\mathbf{v} + \frac{1}{2}\boldsymbol{\omega}\cdot(\mathsf{I}\boldsymbol{\omega})$$

So the energy before the collision was

$$\frac{1}{2}m_A\mathbf{v}_A\cdot\mathbf{v}_A + \frac{1}{2}\boldsymbol{\omega}_A\cdot(\mathsf{I}_A\boldsymbol{\omega}_A) + \frac{1}{2}m_B\mathbf{v}_B\cdot\mathbf{v}_B + \frac{1}{2}\boldsymbol{\omega}_B\cdot(\mathsf{I}_B\boldsymbol{\omega}_B)$$

and the energy after was

$$\frac{1}{2}m_A\mathbf{v}'_A\cdot\mathbf{v}'_A + \frac{1}{2}\boldsymbol{\omega}'_A\cdot(\mathsf{I}_A\boldsymbol{\omega}'_A) + \frac{1}{2}m_B\mathbf{v}'_B\cdot\mathbf{v}'_B + \frac{1}{2}\boldsymbol{\omega}'_B\cdot(\mathsf{I}_B\boldsymbol{\omega}'_B)$$

Using (10), (11), (12), (13) we can substitute for $\mathbf{v}'_A, \boldsymbol{\omega}'_A, \mathbf{v}'_B, \boldsymbol{\omega}'_B$ in the above equation to get

$$\begin{aligned}&\frac{1}{2}m_A\left(\frac{J^2}{m_A^2} + \frac{2J}{m_A}\mathbf{v}_A\cdot\mathbf{n} + \mathbf{v}_A\cdot\mathbf{v}_A\right) + \frac{1}{2}(J\mathsf{I}_A^{-1}(\mathbf{r}_A\times\mathbf{n}) + \boldsymbol{\omega}_A)\cdot(J\mathbf{r}_A\times\mathbf{n} + \mathsf{I}_A\boldsymbol{\omega}_A)\\ +&\frac{1}{2}m_B\left(\frac{J^2}{m_B^2} - \frac{2J}{m_B}\mathbf{v}_B\cdot\mathbf{n} + \mathbf{v}_B\cdot\mathbf{v}_B\right) + \frac{1}{2}(-J\mathsf{I}_B^{-1}(\mathbf{r}_B\times\mathbf{n}) + \boldsymbol{\omega}_B)\cdot(-J\mathbf{r}_B\times\mathbf{n} + \mathsf{I}_B\boldsymbol{\omega}_B)\end{aligned}$$

There are clearly terms which look like those in the equation for kinetic energy before the collision. So to make things simpler we will look at twice the difference in energy. If we can show it equals zero then we are done. Twice the difference in energy is

$$\begin{aligned}&\frac{J^2}{m_A} + 2J\mathbf{v}_A\cdot\mathbf{n} + J^2(\mathbf{r}_A\times\mathbf{n})\cdot(\mathsf{I}_A^{-1}(\mathbf{r}_A\times\mathbf{n})) + J(\mathbf{r}_A\times\mathbf{n})\cdot\boldsymbol{\omega}_A + J(\mathsf{I}_A\boldsymbol{\omega}_A)\cdot(\mathsf{I}_A^{-1}(\mathbf{r}_A\times\mathbf{n}))\\ +&\frac{J^2}{m_B} - 2J\mathbf{v}_B\cdot\mathbf{n} + J^2(\mathbf{r}_B\times\mathbf{n})\cdot(\mathsf{I}_B^{-1}(\mathbf{r}_B\times\mathbf{n})) - J(\mathbf{r}_B\times\mathbf{n})\cdot\boldsymbol{\omega}_B - J(\mathsf{I}_B\boldsymbol{\omega}_B)\cdot(\mathsf{I}_B^{-1}(\mathbf{r}_B\times\mathbf{n}))\end{aligned}$$

This simplifies to

$$\begin{aligned}J^2\left[\frac{1}{m_A} + \frac{1}{m_B} + (\mathbf{r}_A\times\mathbf{n})\cdot(\mathsf{I}_A^{-1}(\mathbf{r}_A\times\mathbf{n})) + (\mathbf{r}_B\times\mathbf{n})\cdot(\mathsf{I}_B^{-1}(\mathbf{r}_B\times\mathbf{n}))\right]&\\ + 2J[(\mathbf{v}_A - \mathbf{v}_B)\cdot\mathbf{n} + (\mathbf{r}_A\times\mathbf{n})\cdot\boldsymbol{\omega}_A - (\mathbf{r}_B\times\mathbf{n})\cdot\boldsymbol{\omega}_B]&\end{aligned}$$

These terms look a lot like those that appear in (14) in Section 3.3. Substituting in the equation for J once into the left hand term gives

$$-J(1+e)[(\mathbf{v}_A-\mathbf{v}_B)\cdot\mathbf{n}+(\mathbf{r}_A\times\mathbf{n})\cdot\boldsymbol{\omega}_A-(\mathbf{r}_B\times\mathbf{n})\cdot\boldsymbol{\omega}_B]+2J[(\mathbf{v}_A-\mathbf{v}_B)\cdot\mathbf{n}+(\mathbf{r}_A\times\mathbf{n})\cdot\boldsymbol{\omega}_A-(\mathbf{r}_B\times\mathbf{n})\cdot\boldsymbol{\omega}_B]$$

When $e = 1$ this is obviously zero, so energy is conserved and we are done. (When one of the objects is immovable the equations are slightly different as discussed in Section 3.3. Showing that energy is conserved for these equations is done in exactly the same way, and so has been omitted).

C Proof that the Propagation Model Terminates for a Stick when there is No Energy Loss

In this section we will be showing that under the propagation model, a stick colliding horizontally with the ground will eventually move upwards away from the ground, when there is no energy loss. See Figure 15 in Section 3.3. Without loss of generality let us assume that the stick has $mass = 1$, and *total length* $= 2$, also let us set $e = 1$ and the centre of mass to lie midway along its length. Since the object is 1D in nature we only need consider the motion in 2D. Let the moment of Inertia be represented by I, the linear velocity and angular velocity before the collision by v, ω respectively, and after the collision we denote them by v', ω'. The velocity of the right end of the stick is therefore $v + \omega$, and the left end is $v - \omega$. Assuming $v + \omega < 0$ let us apply an impulse to the right end of the stick. The impulses acting on the stick will clearly be acting vertically upwards and the magnitude of the impulse is given by

$$J = \frac{-2(v+\omega)}{1+\frac{1}{I}}$$

This means the new velocity and angular velocities are

$$v' = v + J = v + \frac{-2I(v+\omega)}{1+I}$$
$$\omega' = \omega + \frac{J}{I} = \omega + \frac{-2(v+\omega)}{1+I}$$

and the new velocities of the end points are

$$v' - \omega' = (v-\omega) + \frac{2(1-I)}{1+I}(v+\omega)$$
$$v' + \omega' = -(v+\omega)$$

We can rewrite this as

$$\begin{pmatrix} v'-\omega' \\ v'+\omega' \end{pmatrix} = \begin{pmatrix} 1 & 2\frac{1-I}{1+I} \\ 0 & -1 \end{pmatrix} \begin{pmatrix} v-\omega \\ v+\omega \end{pmatrix}$$

Now if $v' - \omega' < 0$ we need to apply an impulse to the left hand side. We can save some time and effort by merely swapping the values of either end's velocities and re-applying the above formula. Using this idea we have

$$\begin{pmatrix} p_n \\ q_n \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 2\frac{1-I}{1+I} \end{pmatrix}^n \begin{pmatrix} p_0 \\ q_0 \end{pmatrix} \tag{21}$$

where p_n is the velocity, after the nth impulse has been applied, of the right end when n is odd and the left end when n is even. Similarly q_n is the velocity of the other end, after n impulses. So our problem now reduces to showing that when $p_0 = q_0 = -1$, there exists an n for which q_n is positive (note that for $n > 0$, p_n is automatically positive when q_{n-1} is negative).

By the definition of inertia, I lies between 0 and 1. Let us define $\alpha = \frac{1-I}{1+I}$. By the limits of I we have $\alpha \in (0, 1)$. The matrix in (21) becomes

$$\begin{pmatrix} 0 & -1 \\ 1 & 2\alpha \end{pmatrix}$$

and it has eigenvalues $\lambda_1 = \alpha + i\sqrt{1-\alpha^2}$ and $\lambda_2 = \alpha - i\sqrt{1-\alpha^2}$ with eigenvectors $\mathbf{v}_1 = \begin{pmatrix} 1 \\ -\alpha - i\sqrt{1-\alpha^2} \end{pmatrix}$ and $\mathbf{v}_2 = \begin{pmatrix} 1 \\ -\alpha + i\sqrt{1-\alpha^2} \end{pmatrix}$ respectively. Note that we can write

$$\begin{pmatrix} p_0 \\ q_0 \end{pmatrix} = (a+ib)\mathbf{v}_1 + (a-ib)\mathbf{v}_2$$

where $a, b \in \mathbb{R}$ and are

$$a = \frac{p_0}{2}$$
$$b = \frac{p_0\alpha + q_0}{2\sqrt{1-\alpha^2}}$$

This means (by the property of eigenvectors) that

$$\begin{pmatrix} p_n \\ q_n \end{pmatrix} = (a+ib)\lambda_1^n\mathbf{v}_1 + (a-ib)\lambda_2^n\mathbf{v}_2$$

Hence $p_n = (a+ib)(\alpha + i\sqrt{1-\alpha^2})^n + (a-ib)(\alpha - i\sqrt{1-\alpha^2})^n = 2Re[(a+ib)(\alpha + i\sqrt{1-\alpha^2})^n]$. Since α is between 0 and 1 we can rewrite it as $\cos\theta$ and $\sqrt{1-\alpha^2} = \sin\theta$ for some value of θ between 0 and $\frac{\pi}{2}$. So $p_n = -q_{n-1} = 2Re[(a+ib)e^{i\theta n}]$ and thus as n increases, $a+ib$ rotates by an angle θ in the argand plane. Eventually it will rotate into the lower half plane as theta is acute. Hence there exists an n which makes $q_n > 0$ and consequently the propagation method will terminate.

References

[1] D. Barraf, *Analytical methods for dynamic simulation of non-penetrating rigid bodies*, SIGGRAPH '89, 23(3):223-232, 1989.

[2] D. Baraff, *Coping with friction for non-penetrating rigid body simulation*, SIGGRAPH '91, 25(4):31-40, 1991.

[3] D. Baraff, *Fast contact force computation for non-penetrating rigid bodies*, SIGGRAPH 94, 1994.

[4] G. Barequet, B. Chazelle, L. J. Guibas, J. S. B. Mitchell, and A. Tal, *BOXTREE: A hierarchical representation for surfaces in 3D*, Eurographics'96, 1996.

[5] G. V. D. Bergen, *Collision detection in interactive 3D environments*, Morgan Kaufmann Publishers, San Francisco, 2004.

[6] D. Eberly, *Intersection of convex objects: the method of separating axes*, Geometric Tools Inc, 2003.

[7] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, *A fast procedure for computing the distance between complex objects in three-dimensional space*, IEEE Journal of Robotics and Automation, 4(2):193-203, 1988.

[8] S. Gottschalk, M. C. Lin, and D. Manocha, *OBBTree: A hierarchical structure for rapid interference detection*, SIGGRAPH '96, 171-180, 1996.

[9] A. Kecskemethy, C. Lange, and G. Grabner, *Analysis of impact responses using the regularized model approach*, Euromech Colloquim 397, 1999.

[10] P. R. Kraus, A. Fredriksson, and V. Kumar, *Modeling of frictional contacts for dynamic simulation*, Proceedings of IROS 97 Workshop on Dynamic Simulation: Methods and Applications, 1997.

[11] Y. T. Lee and A. A. G. Requicha, *Algorithms for computing the volume and other integral properties of solids*, Communications of the ACM, 25(9):635-41, 1982.

[12] S. Lein and J. T. Kajiya, *A symbolic method for calculating the integral properties of arbitrary nonconvex polyhedra*, IEEE Computer Graphics and Applications, 4(10):35-41, 1984.

[13] M. C. Lin and J. F. Canny, *A fast algorithm for incremental distance computation*, IEEE International Conference on Robotics and Automation, 1008-1014, 1991.

[14] M. T. Mason and Y. Wang, *Two dimensional rigid body collisions with friction*, Journal of Applied Mechanics, 59(3), 1992.

[15] B. V. Mirtich, *Impulse-based dynamic simulation of rigid body systems*, PhD Thesis, University of California, Berkeley, 1996.

[16] M. Paterson and F. F. Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete and Computational Geometry, 5:485-503, 1990.

[17] M. Paterson and U. Zwick, *Overhang*, Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA'06), 231-240, 2006.

[18] K. Shoemake, *Animating rotations with quaternion curves*, SIGGRAPH '85, 19:245-254, 1985.

[19] W. J. Stronge, *Rigid body collisions with friction*, Proceedings of the Royal Society of London, Series A vol 431:169-181, 1990.

[20] A. Witkin, D. Baraff, and M. Kass, *An introduction to physically based modeling*, SIGGRAPH Course Notes 32, ACM SIGGRAPH, 1994.

[21] G. Zachmann, *Exact and fast collision detection*, Diploma Thesis, Technical University Darmstadt, Department of Computer Science, 1994.