2D Simulation of Rigid Bodies

Alan Hazelden

0523756

Supervisor: Mike Paterson

2007-2008

Abstract

In the real world, rigid bodies may exert many forces upon one another. These forces may have several effects on the body, for example altering its velocity and rotation. Modelling these processes accurately within a computer is important within many different contexts.

This project consists of the design and implementation of a 2D rigid body physics engine in C + +. The engine supports circles and arbitrary convex polygons. Collisions are handled sequentially and resolved using physical laws of motion, including friction.

Keywords: physics, rigid bodies, 2D collision detection

Acknowledgements

I would like to thank Mike Paterson for agreeing to supervise this project, and for our discussions on how my ideas and simplified models would behave.

Many thanks to Adam Chester for helping me to organise the structure and scheduling of the project, and for his assistance while writing the report.

I am grateful to Nick Pope and Sam Gynn for writing the Warwick Game Design library and supporting me while I used it.

Finally, I would like to thank the entire Warwick Game Design society, without which I probably would not have chosen nearly so interesting a project.

Table of Contents

Introduction

A rigid body is some non-deformable shape with mass, position, orientation, velocity and angular velocity. The majority of objects found in everyday life can be considered rigid bodies – for example a teapot, a shelf or a wrecking ball could all be reasonably modelled in this manner.

When a teapot is placed on a shelf, neither object will move or change shape, but there are forces acting between them that prevent one falling through the other. The teapot will exert a force on the shelf and the shelf applies an equal force on the teapot. These contact forces are balanced, so neither will move relative to the other.

Forces can also act on a non-static body. When a ball is dropped from a height, gravity is constantly accelerating it towards the ground. The ball will continue to accelerate until air resistance becomes exactly opposite and the forces balance. Similarly, a ball rolling down a hill will accelerate due to gravity, but when it reaches the bottom, friction will decelerate it.

Momentum is defined as the product of a body's mass and its velocity. When two moving bodies collide, their combined momentum is unchanged, although it will have been partially transferred from one to the other. Their velocities before and after the collision are related - an instantaneous force is considered to have been applied relative to the mass and velocity of the colliding objects. This force satisfies the conservation of momentum. In reality, collisions are not perfectly elastic – energy will be lost, so the relative velocities after the collision will be less than the relative velocities before. The ratio between the separating velocity and the approach velocity is known as the coefficient of restitution.

There are many applications in which these forces are relevant. For example, structural engineers will have to consider in great detail the forces that will act on the structures they create. When something is overlooked or miscalculated, it can be expensive, or even catastrophic. The Millennium Bridge in London is a good example of this: unexpected resonance forces from pedestrians caused the bridge to be closed just three days after opening. It remained closed for over one and a half years while the problem was fixed, at an extra cost of £5 million [1].

Project Focus

I wanted to focus on the applications of rigid body dynamics in games. The requirements in this case are quite different to engineering simulations – the system must run in real time, but it is not constrained to absolute realism.

Modern games are becoming more and more aesthetically appealing, requiring ever-increasing levels of graphical detail to be stored and processed. However, there is no need for the physical simulation of a world to be as detailed as the visual appearance. A simplified model can be faster to process while not appearing obviously unrealistic.

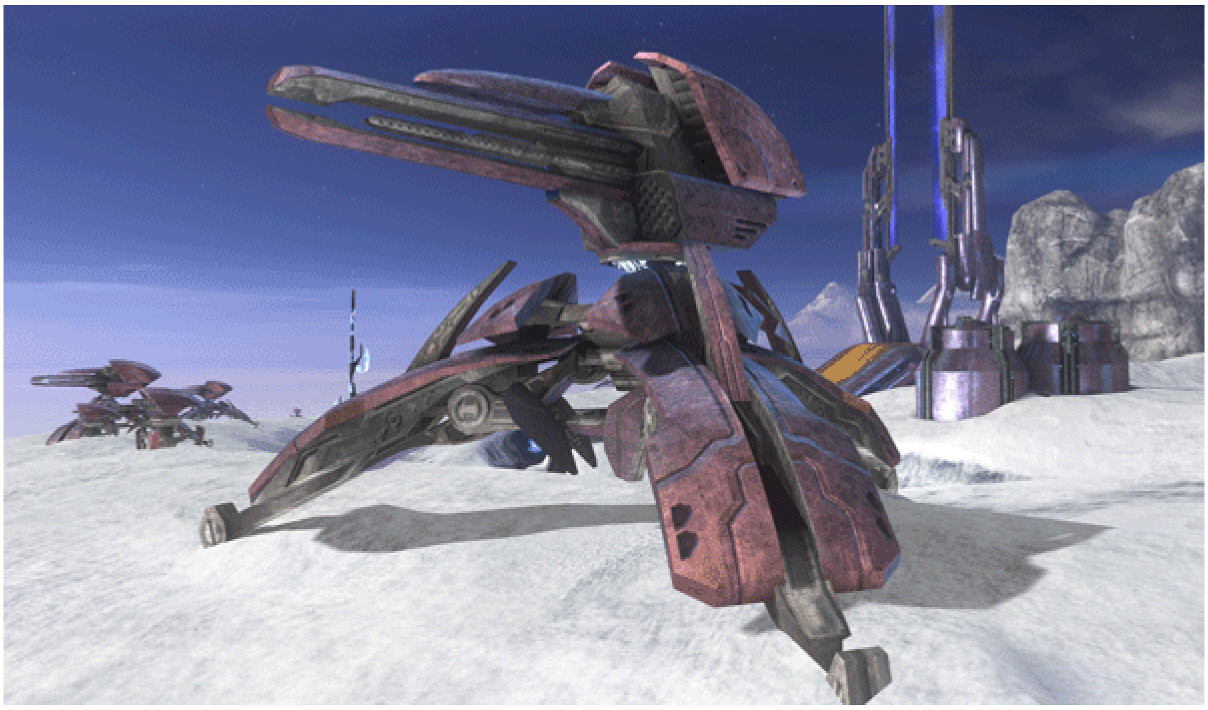


Image 1: screenshot from the game Halo 3

Take the gun placement shown in *Image 1*, for example. To prevent players from walking through it, the collision detection system could access the high-resolution geometry used to render it, and determine if any other objects (including players) are also inhabiting that space. However, as has already been mentioned, this rendering geometry is incredibly detailed and these collision queries would take a long time to compute. If all that is required is that the player not be able to walk through the object, it would be sufficient to simply place an invisible box or cylinder around the gun placement that the player cannot enter.

Some such simplifications must be made by the game designer, but some can be made in the physics engine. These simplifications are important because a game must run in real time. If a one second segment of gameplay takes two seconds to process and run, there are obvious intractable problems – the game will slow down and become unplayable.

The physics in a game is not required to be realistic – for example, it may be that allowing players to jump several times their height makes for a better gameplay experience, despite the obvious unrealism. But however realistic the physics, a key property is that two objects in the scene cannot occupy the same space.

I chose to limit the project's scope to only deal with two dimensions. In the time frame available for the project, a three dimensional system was anticipated to be too complex to deal with. In two dimensions, the project would both require a more feasible amount of development time, and run faster when completed. In addition, keeping the project as a 2D system makes it easier to visualise and use for developers with no previous experience of physics simulation.

Current 2D Rigid Body Physics Engines

In order to define a useful feature set for the physics engine I decided to review a number of existing 2D rigid body physics engines. The physics engines that I have reviewed have each been in development for a number of years and could be utilised within many different contexts.

Farseer Physics Engine

Farseer Physics Engine is a 2D physics engine written by Jeff Weber in C# for Microsoft's XNA platform [19]. It supports both convex and concave polygons, and circles are approximated using regular polygons. It includes four different types of joints, and can use force generators to provide springs, air resistance or fluid drag. Any body can have multiple shapes attached to perform collision detection with, and it is possible to set up methods to be called when bodies collide.

Chipmunk

Chipmunk is a very fast 2D physics engine written in C by Scott Lembcke [14]. Its primitive shapes are circles, convex polygons and line segments. Basic springs and a variety of joints can be used. Any body can have multiple shapes attached to perform collision detection with, and it is possible to set up methods to be called when bodies collide.

Box2D

Box2D is a 2D physics engine written in C++ by Erin Catto [3]. It was originally written as a tutorial for creating the basics of a physics engine. It supports convex polygons and circles, and a body can have more than one of these attached at a time. There are many supported types of joint.

Common Features

I developed a feature matrix to enable me to easily compare the features of the different physics engines:

	Farseer	**Chipmunk**	**Box2D**
Convex polygons	Y	Y	Y
Concave polygons	Y	N	N
Circles	Approximated by regular polygons	Y	Y
Springs	Y	Y	N
Types of joint	4	4	6
Compound bodies	Y	Y	N
Friction	Y	Y	Y
Stacking / resting contacts	Y	Y	Y
Collision callbacks (code executed when collisions detected)	Y	Y	N

Technical Aims

From the table above it is obvious that the different physics engines have several common features. As they are all used within a gaming context, these common features, or a subset of these, should be a suitable basis for my physics engine.

Polygons are the most flexible shape available, so I think it is important that my project supports them. Restricting them to be convex polygons means that many simplifications can be made. Circles are a sufficiently simple shape that I feel it makes sense to support them directly. The primitive shapes I decided to use, then, were circles and convex polygons.

Some bodies might be more complicated than a convex polygon, so I decided that the ability to have compound bodies consisting of multiple primitive shapes would be useful. A concave polygon, for example, could be decomposed into multiple convex polygons. If these were attached to the same body, the result would be the same.

Since I do not plan to include non-convex polygons as a primitive shape, whenever I say "polygon" with no qualification in this report, it can be presumed that I mean "convex polygon". If I intend to refer to a concave polygon or a general polygon, I will make this clear.

When objects collide, they should behave as you would expect them to. My project should therefore use physically accurate models like momentum and restitution to determine the outcome of a collision.

One important feature of a physics engine is that objects should be able to stack and support each other. I determined that this would be required for realistic looking behaviour.

Friction was listed as a nice feature to have, but I anticipated that it would require a lot of time to get working. As such, it was determined to be a secondary goal, if time allowed. I came to realise during development, however, just how important friction is in creating a believable looking simulation.

It is vitally important that during development I am able to test the engine easily. I decided that creating an interactive simulation viewer would be essential – both for testing and demonstration purposes. It would need to support easily creating and manipulating objects.

Although all three engines I looked at included many different types of joint, I decided not to include this as an aim of my project. I did not see joints as being something that without which my project would be unusable, and with so many different variations, I saw that they might require more development time than it would be worth.

Project Methodology

Within a physics simulation, there are a huge number of interacting parts. Should one of these parts not work as anticipated, or need to be redesigned, it would significantly affect the others. For this reason, I chose to use an iterative development model – once one section worked, I could start designing the next.

I created a set of test cases – an initial state for the simulation in which I knew roughly the behaviour I expected. As I extended the engine to include more functionality, that expected behaviour changed to take the new features into account. This was used to test all changes and so avoid regressions.

I used Subversion as a revision control system, keeping all my code and writing stored on an online repository. This acted as a backup system in case of hardware failure, as well as allowing me to make spanning changes with the knowledge that if I broke something, I would be able to revert to a known working version.

Development

As already mentioned, I chose to use an iterative development model, although after iteration 5 was completed, the iterations were self-contained and to some extent developed concurrently.

Each stage of my work is detailed here.

Choice of Programming Language

There were two obvious choices of language to develop the project in: Java and C++. To choose which should be used, I completed an analysis of their strengths and weaknesses.

Since it is the dominant language taught on our course, I have significant experience with Java. Almost all projects I have been involved in during my time at university have been written in Java, including a simple game framework which would be suitable for the needs of this project. It is, however, considered to be a fairly slow language, and computation speed is likely to become a bottleneck in a potentially complex physics simulation.

I had only a passing knowledge of C++ as it is not covered on our course, although I have attended student-run lectures on the subject in my own time. It is much faster than Java, and has more language features too, some of which could be convenient. Additionally, a friend has created a C++ game library for Warwick Game Design (a student society we both belong to), which I could use. This would handle much of the graphical display for me, and has an internal database that I was interested in experimenting with.

I created a feature matrix comparing Java and C++ to help me make the decision:

	Java	**C++**
Cross-platform	Y	Somewhat
Personal experience	2 years	Vague familiarity
Speed	Medium	Fast
Operator overloading	N	Y
Game library	Yes (mine)	Yes (Warwick Game Design)
Used in games industry	N	Y

Both languages were viable candidates. Wanting to gain experience with C++ and expecting computation speed to be a relevant factor, I chose to make my first prototype using the Warwick Game Design (WGD) C++ game library, to see if it was a viable choice. If no major problems were encountered, I would use it for the whole project.

Iteration 1 – Creating and Drawing Bodies

The aim of this stage was to get myself acquainted with the Warwick Game Design library, while creating a test system that could create and display shapes on the screen.

Variable or Fixed Frame Rate

Any game or game-like system needs to continually update the state and position of the game objects and draw them to the screen. To achieve the illusion that these objects are in fact moving, this has to happen very quickly – a common target is 60 times per second, as the human eye is not capable of noticing much improvement past this point.

There are two simple ways of achieving this. The first is to have a constant frame rate – so that the simulation is updated in 1/60 s time intervals. Once the bodies have been updated, the simulation waits for the remainder of the 1/60 s to draw the results.

The alternative is to have the simulation run as fast as it can. When it finishes updating the bodies, it will immediately draw them and then immediately start the next update. It can calculate how long the frame lasts (and so how far to move the bodies) from how long the previous frame took to run. The result is that the display lags behind the update by some varying amount of time, but since it is only by a fraction of a second, it is not noticeable.

This variable frame time could cause changes in the simulation – two 1ms updates will not perform the same operations as one 2ms update. However, neither should look more correct than the other, and since we don't require a totally deterministic simulation, this should be acceptable.

When running at a constant frame rate, obvious issues arise if the computer is too slow to update and draw the simulation within the allotted time. If the drawing takes longer than the updating, this can be somewhat resolved by performing multiple updates between every frame drawn, but this is unlikely to be the case in a physics simulation.

Other solutions exist – one recommended by Fiedler [12] is to have a constant time between updates but to draw independently, using interpolation between the current and previous states to prevent problems when they go out of sync. However, this still has the potential for trouble when running on a slow computer and would be significantly more work to implement.

I decided that a variable frame rate was the simplest and best choice, as its disadvantages would hopefully not be relevant within the scope of this project.

Design

Every body needs to store its current position and orientation, so that it can be drawn to the screen and so that other bodies know where it is. Each body also requires a velocity and a rotational velocity (the speed and direction it rotates), so that its movement can be calculated. Every frame these have to be updated based on the time elapsed.

Also required for each body is a mass, although this will not change with time. Certain bodies should be static and fixed to a position – in a game context this would usually include the terrain that you stand and walk about on. It should not be possible to move these, and forces such as gravity should not affect them. If a body's mass is set to infinity, the body should be considered static.

In my design I listed two basic shapes that I needed to support – circles and (convex) polygons. These would be two different types of body, each requiring some additional attributes – circles have a radius, and polygons have a list of vertices. These vertices would be stored as an offset from the polygon's position, so that they would not have to be updated every frame when the polygon moved.

Since I am creating a rigid body simulator, technically the bodies shouldn't be capable of changing shape, but I realised it would be convenient to be able to construct bodies dynamically. It should be possible to construct a polygon vertex-by-vertex or to change the size of a circle, and these actions obviously involve changing their shape. However, so long as the bodies remain a fixed shape for the duration of a frame (even if they can change shape between frames), it should be possible to treat them as rigid bodies.

Warwick Game Design Library – Database and Other Features

The internal database of the library is called DOSTE. DOSTE is based on empirical modelling concepts [18], so many of its features go way over my head, however at its most basic level (as I understand it), it consists of key-value pairs. Each of these can contain more key-value pairs, creating a hierarchical structure of data.

The Instance class included with the library is the base class that every game object is expected to subclass. Every instance has a database object associated with it, where its properties are stored. This information includes a position and an orientation, among others that will not be required for this project. Because this information is stored in the database, it can be easily accessed from the in-game console for debugging and testing, which I love.

You can very easily define accessor methods in C++ to get data from the database, so C++ code does not have to know anything about the database – this nicely fits the principle of encapsulation. If it is required, however, it is possible to access any section of the database directly, from C++ or from the in-game console.

Since this means that is possible to change a property of an instance without calling any methods of that instance, it is important that there is some way of being notified when a value is changed. It is possible to attach an event handler to a property in the database. When that value changes, the event handler will be called and a user-defined function can be run.

It is also possible to set up dependencies between values in the database, so that one will dynamically update if another changes. This uses concepts from empirical modelling, and since I did not understand those and also saw no need for this feature in my project, I never used it.

In addition to the database, the WGD library provides useful classes such as Vector2D (an implementation of the mathematical concept of a vector in two dimensions, with x and y coordinates) and will set up an OpenGL drawing context. This means that I don't have to learn how to create and initialise a window myself, and can concentrate on the physics aspect of the project.

Implementation

A Body class was created, with Circle and Polygon subclasses. Body defines `update` and `draw` methods which can be overridden; the default implementation of `update` simply moves and rotates the body according to the elapsed time, the velocity and the rotational velocity of the body.

The bodies are drawn using the primitive drawing operations in OpenGL. They are block filled with a colour (initially green so that they could be turned red while colliding) and a black border. OpenGL provides methods to draw convex polygons, but not to draw circles. As a result, circles must be drawn as a 100-sided polygon, with lines to show rotation. Anti-aliasing is enabled to make the border lines appear less pixelated. *Image 2* shows the result.

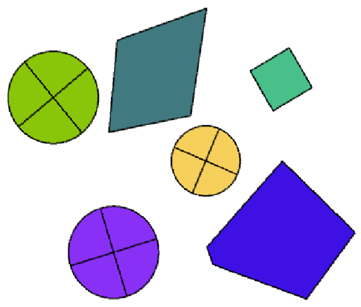


Image 2: screenshot of shapes drawn in OpenGL

Bodies have a linear velocity and a rotational velocity, which are stored in the database and accessor methods are created for them. Circles have a radius which is similarly stored in the database, and polygons store the relative positions of their vertices.

If a body rotates, then it will rotate around its centre of mass. For a circle that's easy enough, but if a polygon's vertices can be at any arbitrary offset from the position, then so can the centre of mass. It would be useful if the vertices were offset from the centre of mass so they could be more easily rotated. We can accomplish this by saying that the position of a polygon is always at its centre of mass. If the centre of mass moves (perhaps because a vertex is added), the new centre of mass can be calculated, relative to the polygon's current position. Then the current position can be moved to compensate and all the (relative) vertex positions moved in the opposite direction so their absolute position does not change.

I defined a `remakeCOM` (remake centre of mass) method, which performs the calculations described in the previous paragraph. It is called at the end of `addVertex` (the only method at this stage that changes the centre of mass). Really, this should be accomplished with an event handler, but at this stage I was still learning how to use the WGD library and wanted to keep things simple. Having laid down these preparations for rotation, I did not actually implement rotation until a later step. Non-rotating bodies were more convenient to work with at this stage.

Initially, all bodies had a default mass of one. I later added a method to get the area of a body, and stored the density of a body rather than its mass. This meant that the mass could be calculated by multiplying the density by the area, so that large bodies would by default have greater mass than small bodies.

The class structure I created is shown in the UML diagram in *Figure 1*, however the data members listed are actually stored in the database rather than directly in the class. The Instance class is included with the WGD library.

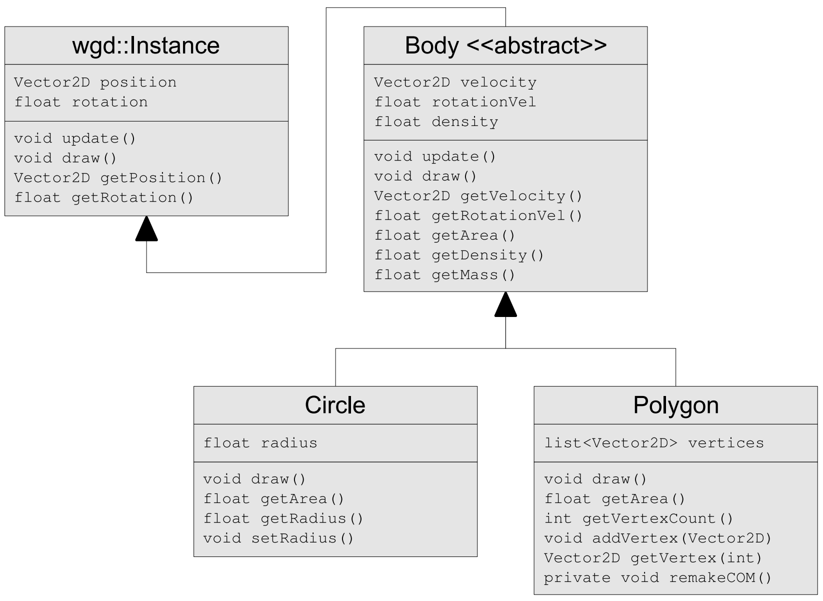


Figure 1: UML class diagram

Testing

In the course of my testing I encountered a couple of problems with the WGD library – then in its early stages. Thankfully, however, they were small issues that could either be worked around or were fixed soon after I reported them.

The first such problem involved negative integers stored in the database. If asked to retrieve such a value as an integer, it would give the correct result; however if asked to retrieve the value as a floating point number, it would instead return a number on the order of 10^9 – probably the result of treating it as an unsigned value. A workaround was to define it as a floating point number (e.g. -2.0 instead of -2).

A second problem was similar – it occurred when storing the value infinity in the database. When converted to a floating point number, the database would return the value zero instead. The reverse problem also existed, in that setting a value in the database from the floating point value infinity would also give erroneous results. When I posted about the issue on the WGD forums, it was quickly fixed.

Another problem came out of unexpected behaviour in OpenGL, which by default will not draw polygons which are facing away from the camera. This improves performance in 3D applications, where the back face of a polygon will usually be hidden from view, but in 2D it is less helpful. Whether a polygon is facing towards the camera or not is determined by whether the polygon's vertices are ordered clockwise or anticlockwise – the result of this is that some polygons were not drawn. This feature was disabled in OpenGL so that both clockwise and anticlockwise polygons could be created.

Evaluation

Other than a few minor problems, the WGD library worked well. It took longer than I expected to set it up and begin work, but has some interesting features that I had not yet begun to experiment with. I was happy with my work so far and wanted to continue using the library, so I saw no reason to stop working in C++.

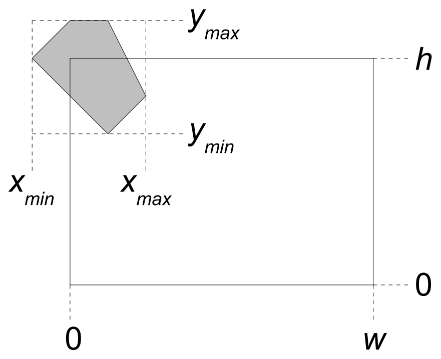
Iteration 2 – Collision Detection with Screen Edges

After iteration 1, bodies would travel in a straight line, leaving the screen after a few seconds. It is necessary to be able to see them to test anything, so for the next iteration I decided to make the bodies bounce off the edges of the screen.

Design

Considering only collisions with the edge of the screen allows some simplifications to be made initially. The edges of the screen are aligned with the world axes – this means that a collision with the left or right edges will only affect the x component of velocity and position.

It's also trivial to detect a collision – the minimum and maximum x and y coordinates of the body are compared to the screen's width and height, as shown in *Figure 2*:



$x_{min} < 0$, so colliding with left screen edge
$x_{max} < w$, so not colliding with right screen edge
$y_{min} > 0$, so not colliding with bottom screen edge
$y_{max} > h$, so colliding with top screen edge

Figure 2: colliding with the top and left screen edges

A body can potentially collide with a vertical screen edge and a horizontal screen edge in the same frame. It does not matter what order we process these collisions in, however, since the x and y components are independent.

When a collision is found, we want to reverse the direction of movement of the body and change its position so it is fully onscreen.

Implementation

To do this, we need to be able to find the minimum and maximum x/y coordinates of the bodies. With non-rotating polygons, these can be easily precomputed relative to the body's centre, since they will not change unless the vertices also change – this can be done in `remakeCOM` because this will always be called when the vertices are changed. For circles, it is a simple matter of adding or subtracting the radius from the x/y coordinate of the circle's position.

Four extra methods were defined to access the data we have calculated: `getMinX`, `getMinY`, `getMaxX` and `getMaxY`. These are defined in the Body class and implemented in the Circle and Polygon classes.

Testing

While testing, I found that the centre of mass for polygons was not where I was expecting it to be. The polygon constructor was calling `remakeCOM` (the method responsible for positioning the polygon at the centre of mass), but this did not seem to be doing anything. On further inspection, it emerged that the database values being accessed did not even exist.

Eventually I discovered that the data associated with the object (including vertex positions) was only being initialised after the object's constructor had been called. As a result, calling `remakeCOM` manually, after creating the object, would give the correct behaviour. This was not ideal, but I decided to go with it as a temporary solution. Later, the use of event handlers would solve the issue properly.

Evaluation

This was not a difficult section to complete, but it needed to happen before implementing anything else. Now I would be able to continue development without the bodies going offscreen where I wouldn't be able to observe them.

Iteration 3 – Collision Detection Between Two Bodies

The collision detection is possibly the most important stage of a physics engine. Before any collision response can be programmed, the collisions must first be found and any relevant information about the collision recorded.

Research

I initially planned to implement collision detection as a test that would return the time of a collision. Once a collision had been detected, those bodies would be reverted to their position at the time of collision. Then the correct collision response would be calculated, and the simulation would continue from that point in time. Having researched the problem, this seems to be a far less sophisticated version of the Timewarp system [16].

For bodies moving with constant linear velocities, finding the time of collision is feasible. However, when the bodies can also rotate, the problem is complicated – I failed in my research to find an algorithm capable of calculating the time of collision between moving bodies which have both linear and rotational movement.

Ericson [8] briefly mentions a possible solution: to move bodies only linearly, with instantaneous rotational changes at the start or end of movement. This is not strictly accurate, but may give results close enough to reality that it will not be significant.

An even simpler approach suggested by Millington [15] is to move bodies simultaneously, and only check for collisions once they have all moved. These would then be processed in order of severity, so that bodies interpenetrating the most are handled first. Again, this is not strictly accurate, but would be considerably simpler to implement.

As it was the easiest to implement, I decided to use this approach unless I discovered that it produced noticeably incorrect behaviour. Since the simulation will be running very fast – on the order of fifty to a hundred updates a second, any inaccuracies in the model should hopefully not be noticeable to a user.

Design

There are only two types of body (circles and polygons), so there are only three distinct collision tests at this second stage – circle against circle, polygon against polygon, and circle against polygon.

Testing whether a circle is colliding with another circle is trivial – if the distance between the centres of the circles is less than the sum of the radii, they are colliding; otherwise they are not (see *Figure 3.a* and *Figure 3.b*). In fact, this is just a special case of the separating axis test.

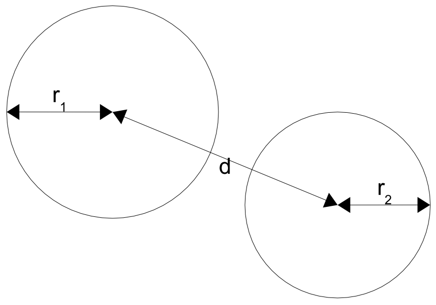


Figure 3.a: $r_1 + r_2 < d$

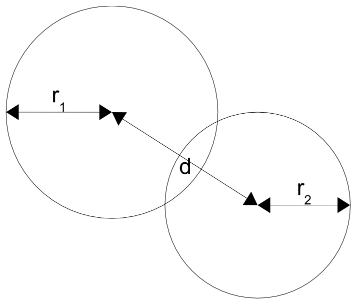


Figure 3.b: $r_1 + r_2 > d$

Testing whether two arbitrary polygons are colliding is complicated, but when dealing with convex polygons, the problem is simpler – the separating axis test can be used. This is the reason that I initially chose not to support concave polygons as a primitive shape. A very similar approach can be used when checking circle-polygon collisions.

The extreme x and y coordinates calculated in iteration 2 form a rectangle known as a bounding box. It is extremely simple to test whether two bounding boxes overlap, so it can be used as the first step of collision detection between two bodies. If two bodies are nowhere near one another, this check will inform us that they cannot be intersecting, preventing us from having to perform the separating axis test as a second (more computationally expensive) test.

Separating Axis Test

The separating axis test is described well in Millington [15] and Eberly [5]; I shall only attempt to give a brief overview here. It uses the fact that if two shapes are not intersecting, then there is some line separating them (in 2D; in 3D they are separated by a plane). Perpendicular to this line, there is the axis along which the shapes are separated. By projecting the shapes onto an axis, it can be easily determined if they are separated in that direction or not. The two bodies in *Figure 4* are separated, and a separating axis is shown.

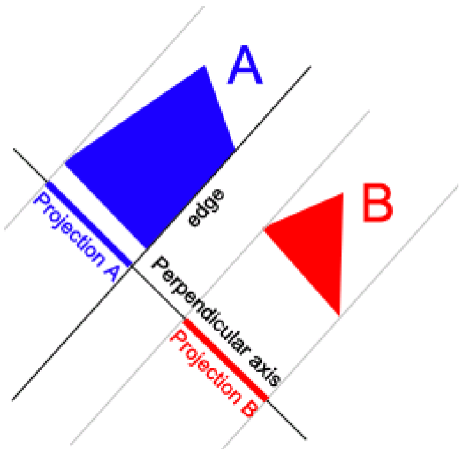


Figure 4: a separating axis for two bodies

For two polygons, there are only a limited number of potential separating axes, and it is possible to enumerate through all of them. If you find any axis along which they are separated, then you know that they cannot be colliding. If they overlap along all axes, then they must be colliding.

With two polygons, the axes to check are perpendicular to the edges of the polygons (as shown in *Figure 4*). With a polygon and a circle, the axes are perpendicular to the edges of the polygon, and along the lines between each vertex of the polygon and the circle's centre.

To use the separating axis test, I must assume at this stage that all polygons are convex. However, this was not a problem – I hoped to add compound bodies later in the project, and any non-convex polygon is representable as multiple convex polygons. Some pre-processing would be required to detect and perform the conversion. If compound bodies were not implemented, then non-convex polygons would have to be omitted from the simulator.

Collision Detection with Ellipses

After Mike Paterson expressed interest in whether ellipses could be supported by the simulator, I briefly looked into the problem. A simple technique of colliding ellipses against lines or polygons is to perform a scaling transformation on both bodies so that the ellipse becomes circular. This transformation will not affect the results of the test, and can be inverted afterwards to get the correct normal and penetration information.

Unfortunately, colliding two ellipses is less feasible. The separating axis test cannot be applied, because surprisingly there is no simple way of finding a separating axis. An alternate method involves specifying both ellipses as geometrical equations and obtaining a fourth order polynomial equation. It is possible (but not trivial) to find the roots of this equation, which provide the intersection points [6]. This will not, however, detect one ellipse containing the other, and it also does not provide the required information for collision resolution.

Implementation

The separating axis test is reasonably simple to implement. There are a finite number of possible axes to test, and for each one you perform exactly the same test – do the bodies overlap in this direction? For convenience, I define the axis to always point outwards from the first body towards the second.

I define some way of getting the extreme point of a body in any given direction. For a polygon, this involves iterating through the vertices, performing the dot product with the direction vector, and returning the vertex with the largest result. For a circle, it is easier: the extreme point is simply the centre of the circle translated along the direction vector by the radius.

These extreme points can be used to simplify the test for whether two bodies overlap in a given direction.

Testing

I decided the easiest way to test the collision detection was to be able to click and drag the bodies around the screen. Doing this required a test for whether a given point (the mouse pointer) is inside a body.

For circles, this is trivial: you can just check the distance to the centre, and compare to the radius. For polygons, you must iterate through the edges and for each one check if the point is inside or outside that edge. If the point is outside any of the edges, it is outside the polygon. Otherwise the point is contained by all edges and is inside the polygon also.

One notable bug was discovered at this point in testing. Polygons were colliding slightly before they should have, as if the edges were oriented slightly differently to how they were displayed. After much investigation, I discovered that the bodies were being tested for separation along the wrong axes. The offending code follows:

```
Vector2D side = v1[i+1] - v1[i];
Vector2D axis = Vector2D(axis.y, -axis.x);
```

The second line incorrectly used `axis.y` and `axis.x` when it should have been `side.y` and `side.x`. Note that `axis` was being initialised and referenced on the same line – this error was not caught by the compiler, but meant that the x and y values would contain whatever data was in the memory location previously.

When the second line was corrected, the collision detection code worked correctly.

Iteration 4 – Collision Response (without Rotation)

With a working collision detection system, I now needed to use this information to make the bodies bounce away from one another when a collision is found. I chose to initially implement a linear collision response, which meant that the bodies would not rotate at all. This is because I already knew the laws of motion that are involved and so would easily be able to implement these and confirm the results.

Design

When two bodies are in contact there are two active forces: a contact force keeping them apart and a frictional force opposing motion. *Figure 5* demonstrates this. These forces act at right angles to one another, with the contact force acting along the normal to the collision plane and the frictional force acting along that plane.

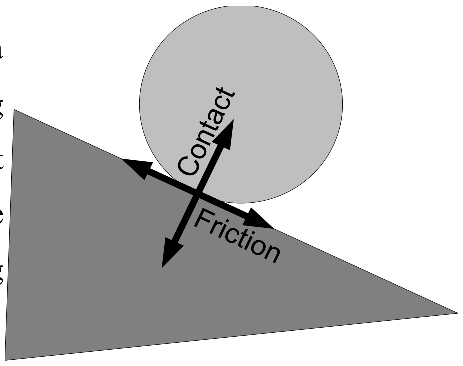


Figure 5: contact and friction forces

We can model a collision by considering the two bodies involved to be in contact at exactly one instant. Before the collision they are moving together with constant velocities, and afterwards they are moving apart with new velocities. At the time of collision, an impulse (a force acting only at this one instant) is applied which provides this change of velocity.

For the time being I am ignoring friction, so this impulse will be entirely in the direction of contact. We can split the velocity of a body into its components in the direction of contact and in the direction of friction. The component of the velocity in the direction of friction will remain unchanged.

Note, however, that when a collision is not instantaneous, and the bodies remain in contact over extended periods of time, this analysis does not hold. At this stage, I planned only to simulate instantaneous collisions, with resting contact to be added in a later iteration when I could see the issues with using this approach.

The conservation of momentum and the coefficient of restitution are the two properties needed to model collisions without rotation. In the following analysis I use u to denote a velocity before collision and v to denote a velocity afterwards.

Suppose we have two bodies b_1 and b_2 with masses m_1 and m_2, initially travelling at velocities u_1 and u_2. The conservation of momentum states that if they collide, the total momentum of the system remains constant, so if v_1 and v_2 are the velocities after the collision, the following equality applies:

$m_1 u_1 + m_2 u_2 = m_1 v_1 + m_2 v_2$ (Conservation of momentum)

The coefficient of restitution (μ) is a ratio between the separation speed and the approaching speed. Equivalently, where u and v are now scalar vectors in the direction of collision:

$-v_1 + v_2 = \mu(u_1 - u_2)$ (Coefficient of restitution)

Together, these give us a set of simultaneous equations which, given the coefficient of restitution and the initial velocities of the bodies, will give us the final velocities of the bodies after impact.

Implementation

The collision detection subsystem had to be modified to identify the contact normal and the penetration distance – fortunately enough this was simple to do, as this information is required for the separating axis test. This information was stored in a container type named, at various points, Collision, CollisionData and finally Contact.

To get the scalar velocity along the direction of contact, the dot product of the collision normal with the relevant velocity can be used. The collision normal is always taken to be from the first body towards the second, so if they are moving towards one another (which is expected since they have collided), it should always be the case that $u_1 > u_2$.

A small number of additional checks are required to see if one or both of the bodies are static objects fixed to the scene, in which case they should not move at all.

The consequences of the earlier decision to move bodies simultaneously and only then check for collisions begin to reveal themselves. The bodies will have travelled since the time of collision, so will be overlapping at the end of the frame. This means that a final addition is required to slightly move the bodies so that they are no longer interpenetrating.

Evaluation

The system at this stage worked significantly better than I expected it to. Bodies could remain in resting contact with one another without strange behaviour, and everything seemed to work as it should. It was good to see that the inaccurate model did not cause any visible problems.

Iteration 5 – Collision Response (with Rotation)

At this point, the simulation worked well, but the lack of rotation meant that any non-circular body would behave in a noticeably unrealistic manner.

Design

The changes of velocity calculated in the previous section are the result of an impulse applied at the moment of collision. We were able to simplify this to only consider velocities. The velocity of the bodies was used before; now the important thing is the velocity of the collision points on the bodies. Rotational movement can affect this as well as linear movement. The collision detection would have to be modified to find the collision points.

In the previous section, I used the conservation of momentum and the coefficient of restitution to directly solve equations to find the new velocities of the bodies. Now the bodies can rotate as well, this is not a viable solution. Instead, we need to consider impulses.

An impulse is applied at the moment of collision, which alters the velocity of both bodies. We know that the same impulse must be applied to both bodies, but in opposite directions (as with forces, they must be equal and opposite). Given the magnitude of the impulse, we can calculate the relative change of velocity. But it is also possible to calculate the magnitude of the impulse given the relative change of velocity, and this can be determined from the speed at which they are approaching and the coefficient of restitution. This approach was suggested by Millington [15].

For a full description of the dynamics involved, see Eberly [5]. This includes discussion on how to calculate the moment of inertia of a body and how to use that to determine the ratio of linear movement to angular movement per unit impulse.

Implementation

The bodies now need to be able to rotate. While I had laid the groundwork for this in the first iteration, I never implemented it, so this had to be the first thing I did. With a now greater understanding of the how the database works and how best to use it, I decided to cache the vertices and invalidate the cache whenever the body moves, rotates, or has its vertices altered.

With rotating bodies, you can no longer precompute the min/max x/y coordinates – these have to be generated at the same time as the cache. In theory, you could use a hill-climbing algorithm to quickly find the new values from the old, but I decided that it would be more complicated to code than it would be worth for the insignificant speed improvement.

Testing

When adding a new vertex to a polygon that had been rotated, the new vertex appeared at a totally different place to where it should have. I quickly realised that the `addVertex` method assumed that the body was not rotated, and fixed the oversight.

After implementing caching, the program began to crash when creating a new polygon with no vertices. Running a debugger, I found that it was crashing after trying to assign a value to the first value of a vector, which I thought dynamically resized themselves on assignment. Eventually I realised that only applies if you use accessor methods rather than directly accessing elements. After manually allocating enough space, it worked fine.

Sometimes the bodies would interpenetrate and then chase each other around the screen. This was the weirdest bug I had ever seen, and I couldn't work out how it could possibly be happening. After days of enduring this, I finally chanced upon the answer – I was calling the WGD library method `translate`, when what I wanted was in fact called `move`. During the penetration resolution, instead of being moved along the collision normal, the bodies were moved relative to their orientation, which could easily push them further into contact.

Evaluation

Despite having fixed the crazy bug that made bodies dance about the screen, the system was not as stable as it had been at the end of the previous iteration. When in resting contact, bodies would vibrate and jitter for no readily apparent reason. Instantaneous collisions, on the other hand, were working perfectly.

Iteration 6 – Resting Contacts

I tried a large variety of approaches to avoid jitter. None were completely successful, but some were more successful than others.

This work was carried out in parallel with later sections.

Don't remove 100% of penetration

I wondered if the bodies might be colliding in one frame, then not colliding in the next. If this was the case, I thought, it could help to leave some small amount of overlap (1 pixel or so) at the end of every frame. Then they would definitely still be colliding at the start of the next frame.

This did not have any noticeable effect.

Calculate new velocities for all bodies before removing any penetration

Rather than modifying the velocity and position for each pair of bodies in turn, I thought it might give better results to perform all the velocity calculations first, only moving the bodies when that was complete.

This did not have any noticeable effect, but was probably a good thing to do anyway.

For low-speed collisions, make coefficient of restitution = 0

A technique I read in Eberly [5] was to remove any restitution when the bodies are colliding very slowly. The idea, I believe, is that a bouncing ball may in theory bounce forever, simply bouncing to a smaller and smaller height each time. At some point, it will be bouncing so little that it is not noticeable, but the simulation would continue to calculate increasingly smaller and smaller bounces. This change would mean that after some point, the ball would stop bouncing.

This did not have any noticeable effect on the stability of resting contacts.

Resolve half the penetration each frame

If the problem manifests itself by bodies changing position quickly between frames, then one obvious solution is to limit or reduce the movement per frame. If, instead of resolving all the penetration, we resolve just half of it, then in the next frame we can resolve half of the remainder – it should smooth the jitter out over several frames.

This seemed to help very slightly, but it did not resolve the central issue, however.

Don't resolve any penetration

Mike Paterson wondered if the penetration resolution step was necessary at all: after the collision the bodies should be moving away from one another, so in theory they wouldn't be able to move into one another too much.

I tried disabling this step, but this clearly did not work: the bodies would slip through one another and fall out the bottom. I think that due to the complex interactions between bodies when you have resting contact, you cannot guarantee that any given pair of bodies will be moving apart at the end of the frame.

Non-linear penetration resolution

Millington [15] suggests that allowing the bodies to rotate when resolving penetration may give more realistic results. Trying this suggestion results in larger code rewrites than I was expecting.

This did not seem to have any noticeable effect, which was annoying, as it was one of the more complicated ideas to implement.

Debug information: draw contact points

To help me to debug the situation, I added a small piece of code which would read the contact points generated by the collision detection system, and draw them to the screen. This was an extremely useful and interesting source of information, which helped me to think of the next idea:

Create contacts between bodies which are nearby but not colliding

I noticed that sometimes two bodies would be interpenetrating quite severely, but no contact point would be displayed. I realised that while they were not initially colliding, one or both of them was being displaced to resolve other collisions, and this was moving them into contact.

Because no collision between them was initially detected, they had no way of knowing that they could not be moved in this direction until the next frame. The solution was to create a contact between two bodies if they are colliding or if they are very close to colliding (within some fixed distance).

This meant that if two bodies are moved into one another in the collision resolution stage, the problem can be fixed immediately, rather than having to wait for this to be reported next frame. This approach worked very well, and finally the jitter became manageable.

Simultaneous contacts

The collision detection only finds one contact point at a time. I noticed that when two polygons lie edge to edge, the contact points would. I believe that if my system was capable of detecting multiple contact points, this would solve the remaining issues.

There is an algorithm described in Eberly [5] that would seem to detect multiple contact points, but it appears to require that the bodies not be overlapping. Since my bodies will always be overlapping before a collision can even be detected, this was not suitable.

I discovered a persistent algorithm that claimed to be able to do what I wanted [17]. Each frame, only one contact point is found, but it is added to a list containing all previous contacts. Contacts are invalidated when they become out of date. I attempted to understand how the algorithm worked, but did not have much success implementing it.

After much wasted time, I realised that the time I was putting into the problem was not justified by the results I was getting. Regrettably, I called a halt to my development in this area.

Iteration 7 – Friction

Without friction, bodies would slide over one another in an unrealistic way, and circles would not rotate because the contact forces on a circle always act through the centre. This lack of realism was jarring, and I decided that the project should definitely support friction. Previously friction was considered an optional feature, so I decided to compensate for this priority change by making compound bodies optional instead.

Design

Friction depends on the magnitude of the contact force, the coefficient of friction between the two bodies, and the direction of motion of the bodies relative to one another. Friction is always perpendicular to the contact force, and it always acts in the opposite direction to motion. The magnitude of the force has a maximum value equal to the magnitude of the contact force multiplied by the coefficient of friction.

Of course, my system uses impulses rather than forces. I hoped that the principles that apply with frictional forces would also work with frictional impulses. This seemed a reasonable assumption, but I knew that I could not be sure until it was implemented.

In reality, the coefficient of friction depends on the materials touching – two blocks of ice have much less friction than two blocks of concrete. However, as with the coefficient of restitution, I initially chose to give all contacts the same coefficient, with the possibility of extending this later.

Implementation

Calculating the magnitude of the frictional impulse is done in a very similar way to calculating the contact force; you take the desired change of velocity and determine how large an impulse would be required to create such a change.

To determine the relative movement of the bodies and so the direction of friction, the velocities of the bodies' collision points need to be found. This is nothing new either; it is done in exactly the same way to find the speed of approach.

Testing

My first tests produced some very strange behaviour – bodies dropped to the floor would bounce back with far more rotation than would be expected. After a time they would settle, but then they would start to bounce again. I checked several times and the problem was not as simple as a force being applied in the wrong direction.

Eventually, the problem was solved by applying the contact force before calculating the relative motion of the bodies, rather than after. Even after I had seen this fix the problem, for a while I was almost convinced that this should not have changed anything. Finally I realised that the contact force was changing the rotation, which was included in the relative motion.

Evaluation

I was really pleased to see circles rolling down a slope without any issues, and it was good to see bodies come to a rest naturally. They were also now able to prop themselves up against the walls, and could rest on slightly sloped surfaces without slipping.

Overall, it added a lot of subtle touches to the simulation that weren't obviously missing, but would subconsciously bother someone. It was amazing just how effective this step was in increasing the perceived realism, and I am incredibly glad that it got implemented.

Iteration 8 – Ensuring Polygon Convexity

The collision detection algorithms assume that all polygons are convex. However, with users able to dynamically create polygons, there is no guarantee that this assumption will hold. If a concave polygon is created, the system would not warn the user; however the collision detection system would give strange results. The final addition I made to the project was to ensure that a polygon remains convex at all times, thus avoiding this problem.

Design

The convex hull of a set of points is the smallest convex polygon that encloses all of the points, akin to stretching an elastic band around a set of pins – see *Figure 6.a*. This is what how a polygon should behave when a vertex is added – the vertices of the new polygon should be the convex hull of all the old vertices plus the new vertex, as shown in *Figure 6.b*.

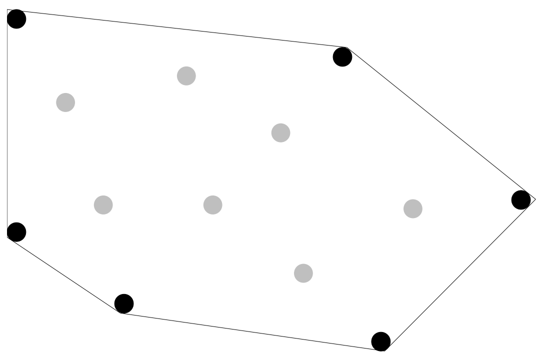


Figure 6.a: convex hull of a set of points

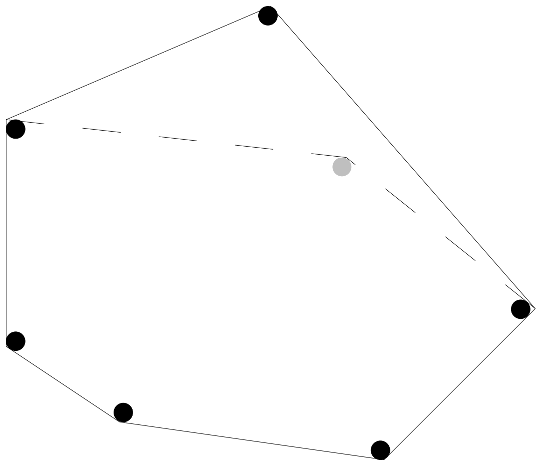


Figure 6.b: adding a vertex to a polygon

The current system allowed a polygon to have their vertices ordered either clockwise or anticlockwise. This necessitated a check every time an edge normal was generated, to ensure that the normal actually was pointing outwards. Running the vertices through a convex hull algorithm would mean that I could ensure that all polygons had their vertices listed in clockwise order. This means that the normals can be created without needing to check their direction. Similarly, calculating the area of a polygon depends on the ordering of the vertices (it needs to be multiplied by one if they are anticlockwise).

Implementation

There are many different convex hull algorithms. I looked at several, including Jarvis March, Graham Scan and QuickHull. In the end, I decided to use the simplest algorithm, Jarvis March, due to its ease of implementation. Since polygons in the simulation are unlikely to have very many vertices and the algorithm only needs to be called rarely, having a better algorithm would be highly unlikely to noticeably improve performance.

Jarvis March (also known as the gift wrapping algorithm) runs in O(nh) time, where n is the total number of points and h is the number of points on the convex hull. It begins by finding a point known to be on the convex hull. Given a point p_i on the convex hull, the next point p_j on the hull can be found because all other points will be on the same side of the line through p_i and p_j. The convex hull is generated point-by-point and proceeds until the first point is reached again.

My first implementation of finding the next point on the convex hull ran in O(n^2) time, resulting in O(n^2h) overall running time. While I knew this was not optimal, I wanted to get something that worked, and had not managed to find an example implementation in my brief search. Also, as noted previously, this algorithm was unlikely to ever become a bottleneck. My implementation looped through all points, for each one checking the line between that point and the previous point on the convex hull against all *other* points.

I later realised the simpler O(n) time implementation that I had not thought of: there is a candidate for the next point on the convex hull, initially the first point (or arbitrarily any other). Then you iterate through all the points. For each one, you ask: does this point lie inside or outside the halfspace formed from the last point on the convex hull and the candidate? If the point lies outside, then the candidate cannot be the next point on the convex hull, so we make the new point the candidate. After we have looped through all points, we know our candidate must be the next point.

Testing

The algorithm to find the next point on the hull uses the last point to be added to the hull and a candidate for the next point. These form a line which divides the plane in two, with one half being inside and the other outside. Sometimes, however, the algorithm will start with these points equal. In this case, it is very important that any other point is considered to be outside rather than inside. Otherwise, the algorithm will terminate early – this manifested itself in my testing by only allowing a polygon to have one vertex at a time. Another solution to this issue would simply be to ensure that the last point on the hull can never be the candidate.

Evaluation

There are numerous places in the code which assume that polygons are convex, and now this assumption is guaranteed to be true. As it is impossible to create concave shapes even if you try, no work has to be done to handle this case and the user does not have to be warned. Also, further simplifications can be made now that all polygons have their vertices defined clockwise.

The implementation described here should be reasonably efficient.

Iteration 9 – WGD Library Integration & Compound Bodies

As has already been discussed, there are some components of a physics engine that are found in almost every game. Collision detection and the enforcement of non-penetration are especially general, and I found myself wanting to use these sections of the code in another project.

The ideal solution would be to package the project into a library, which could be used without any knowledge of the algorithms used internally. It could even be directly integrated with the WGD library.

Design

The class structure I initially designed, with a body class which could either be a circle or a polygon, was not well designed for packaging the project as a library. It meant that to use the physics system, a body would have to directly extend either the polygon class or the circle class, which seemed like a messy solution. A more sensible class structure would have been to have a body class which had as a property the shape which the collision detection would use. An advantage of this is that compound bodies could be easily implemented by allowing a body to own more than one shape.

These shapes would ideally be definable both in the database and in C++ code, and I thought it would be a useful feature if they did not have to be attached to a body. Obviously, a shape that *was* attached to a body would have to move with it, so its position would have to be updated every frame. All methods should return positions in world coordinates, not coordinates relative to the body.

In the currently implemented system, all collisions have the same coefficients of restitution and friction. Some user-friendly system to allow these properties to be set on a per-collision basis would be important. The ability to inform bodies that they have just collided with something would also be incredibly useful. One possible solution would be to be able to set a function or functions which would be called for every collision. This function would be responsible for setting the collision properties and informing any relevant parties.

Implementation

Creating the distinction between a body and a collision shape was a simple enough matter of splitting the Body class into a Body class and a Shape class, and defining the right methods in each. Re-implementing these methods, however, was not so simple. The Body class now had to handle arbitrarily many shapes being attached to it, and the Shape class had to remain synchronized with the body it was attached to. This in particular proved to be frustrating, with increasingly large amounts of complex interaction with the database seeming to be necessary.

It became obvious that the rewrite required to implement this would be a much larger undertaking than I had expected or could be justified. With limited time remaining for development, I sadly decided that work put in here would simply not pay off.

Evaluation

Time constraints meant that completing this section during the development phase of the project would have been impractical. In many ways this is a shame, as this is potentially the most useful area – allowing others who have no knowledge of collision detection or physics to use my code.

When writing on this report has concluded, I intend to return to this section and complete the work.

Author's Assessment of Project

This project does not break any new ground – there are 2D physics engines far faster, more stable and with more features. However, this has been the work of months rather than years, and I am incredibly pleased with what I have created.

Writing a physics engine is a deceptively complex task, and in the course of this project, I have learnt much. It is my hope that others would be capable of using my code in their application, or to read this report and learn from my experiences.

Most of the features detailed by the original specification have been implemented, and I intend to tackle the remainder in my continuing work on the project – to enable it to be used as a general purpose physics library. It will be integrated with the WGD library and made available to all users of that library.

The project could also be used as an example of how to use the Warwick Game Design library to make a simple 2D game. While it only displays uncomplicated block-filled shapes, it does demonstrate how the internal database works and how it can be used.

When dealing with large numbers of interacting bodies, the nature of the simulator causes the simulation to slow down, which in turn creates visible stability issues. Because of this, the project is not suited to modelling highly complex physical situations, but it is adequate to enforce the simple non-penetration constraints for most games.

Project Management

The following is a summary of the work done on a week-by-week basis. A recurring theme is that when drawing up the initial timetable, I did not have a good enough understanding of what I would be implementing. Therefore, I did not know in enough detail what a task would involve and how much time it would require. As the timetable deviated from the initial plan, the goals of the project changed priority; for example friction was identified as being more important than originally thought and made a core feature, while compound bodies were removed from the project.

Term 1

Time period	Initial plan	Activity
Week 1	Writing specification	Writing specification
Week 2		
Week 3	Evaluate Java and C++ frameworks and decide which language to use	Evaluated Java and C++ frameworks and chose C++
Week 4	Design test cases for instantaneous collisions	Started learning to use the C++ library. (got a square moving across the screen)
Week 5	Create interactive simulation viewer. Design and implement classes for instantaneous collisions	Continued to learn C++ and the library. Added circles and border lines
Week 6		Make them bounce off the sides; investigate fixed rate vs. variable rate updates
Week 7		Proper collision detection
Week 8	Design test cases for bodies in contact	Progress Report
Week 9	Design and implement classes for bodies in contact	Fix bugs in collision detection
Week 10		Start collision response with no rotation

I had originally planned to design test cases before starting development for any section. These were to be a (hopefully) comprehensive set of situations, each one demonstrating a different property needing to be tested – similar to unit testing. This quickly changed to creating test cases alongside development, however, when I discovered that the WGD library would allow me to easily load objects from a file once I had defined the attributes required. In hindsight, I do not think unit tests would have worked in any case; all components depend on one another too much.

I found that C++ and the WGD C++ library had a steeper learning curve than I expected, so by the end of the first term, I was slightly behind schedule.

Christmas Holiday

Time period	Initial plan	Activity
Week 1	Design and implement classes for bodies in contact.	Collision response with no rotation
Week 2		
Week 3		
Week 4		Rest

By the end of the Christmas holidays, I had a reasonably interesting looking simulation that performed well, but the bodies were incapable of rotating. They did, however, perform surprisingly well when placed in resting contact.

In my progress report, I had planned to complete instantaneous collisions by the end of the Christmas holidays, and then begin working on resting contacts in term 2. Within the first three weeks of the holiday I had in fact implemented both of these, but only for non-rotating bodies. I saw this as being twice of half the work; so roughly the same amount of work as I had planned to do. This allowed me a little time off during the festive period.

With a basic simulation working, I began to realise the limitations of a simulation without friction. The bodies would continue to slide around far longer than looked believable. I decided that after rotation had been implemented, if I wanted it to appear realistic, it would be more important to add friction than to add compound bodies. So the priorities of those two components in my plan were swapped – compound bodies would now be optional and friction would now be a project goal.

I had now changed the ordering and importance of the components, and even the components themselves now that collisions were split between rotating and non-rotating rather than instantaneous or resting. My initial plan was getting more and more unrecognisable, and sticking to the timetable did not seem viable – I did not feel capable of estimating how long any given task would take. I chose instead to work more solidly on the project, attempting to accomplish as many goals as possible, but with no pre-set targets for each section.

Term 2

Time period	Initial plan	Activity
Week 1	Design test cases for compound bodies.	User interface
Week 2	Design and implement classes for compound bodies.	Started to add rotation
Week 3		Got rotation working
Week 4		Try to fix jitter
Week 5	Bug fixing. Implement friction if ahead of schedule; catch up if behind schedule.	Try to fix jitter
Week 6		Try to fix jitter; friction
Week 7		Bug fixing
Week 8		Try to reduce jitter further; enforce polygon convexity

As resting contacts for non-rotating bodies had worked perfectly as a result of the instantaneous collision code, I had anticipated this continuing once rotation was added. Unfortunately, this was not the case – what I termed "jitter" was an obvious problem: the bodies would appear to vibrate and move about, rather than coming to a rest.

It was a much harder problem than I expected and many weeks were spent grappling with this problem. I tried many different approaches, but none of them seemed capable of completely removing the issue. Eventually I was forced to conclude that perhaps too much time had been spent on this one problem and that I should move on. Friction was the final feature to add and thankfully it was rather easier to add than I had anticipated; this allowed me to return to working on jitter for one final week.

One other task I attempted during this time was to integrate the physics system with the Warwick Game Design library. When starting this, I realised that I had not considered this in the original design of the system and that it would require a major refactoring effort to get working. I decided that given the limited time available, it was best to put this to one side and concentrate on other aspects of the project.

Conclusions

The goal of the project was to produce a 2D rigid body simulator and I consider it to be a success. Most of the initial objectives have been achieved, and the resulting product works well in the context it was designed for.

Technical Achievements

Of the initial features I hoped to include, compound bodies had to be dropped, but the remainder were fully implemented. A simulation viewer has been developed that is capable of dynamically creating and moving bodies. The system models friction and gravity, as well as using physically accurate concepts to simulate collisions. Objects within the simulation are capable of remaining in resting contact, stacked on top of one another without any visible problems.

Lessons Learnt

The most important lesson I will take away from this is that it always takes longer to implement a feature than you expect. Resting contacts, in particular, were a more challenging feature than I would have anticipated.

One area where I definitely should have done things differently is the class hierarchy. I started with a Body class, which was extended by Circle and Polygon classes to provide the necessary collision geometry. However, I was not at this stage considering how compound bodies might be implemented, or how other applications might use my engine as a library. Both of these cases would have been better served by having the Body class contain one or more Circle or Polygon objects; by the time I realised this, however, restructuring such a large section of the code would have taken too long to be practical. If I could have foreseen these issues, I would have designed it correctly the first time, eliminating the need for any time-consuming rewrites.

While there is no denying the usefulness of the WGD library in developing and debugging the project, I believe it is the major bottleneck when it comes to performance. If, instead of storing everything in the database, I used traditional programming techniques and allocated storage space within the rigid body class, I believe the engine would run significantly faster. This would come at a cost, however, as it would not be possible to easily access and modify data while running the program; something which made a tremendous difference when debugging.

Further Work

I fully intend to continue my work on the project until it can be easily integrated into the WGD library. Part of this integration work will include separating the rigid body class from the polygon and circle classes. Classes using the library will then extend the rigid body class and attach some collision geometry. This is exactly what is required to support compound bodies as a body will not be limited to only having one piece of collision geometry attached at a time.

Many of the algorithms used could be improved: to speed up development time in many areas, I implemented only a simple brute-force approach which could be easily improved upon. For example, the algorithm I use to test if two polygons are intersecting checks every edge of both polygons against every vertex of the other. For polygons with m and n vertices, this takes $O(mn)$ time, but it is possible to do the same check in $O(m\log n + n\log m)$ time [5]. Similarly, the broad-phase collision detection currently only consists of a bounding box test; this could probably be made faster by using a quadtree [8] to test only those bodies which are near one another.

Possible Extensions

In addition to the above, there are several larger extensions to the project that could constitute another project entirely.

The sequential resolution of collisions I have used is not without significant drawbacks. Another approach I have researched briefly (seen in the work of Catto [2] and Eberly [5]) involves solving collisions simultaneously through a system of linear equations. An evaluation of the advantages and disadvantages of each would require implementing both techniques (a significant amount of work) and comparing their accuracy and speed. I believe this would be very interesting and provide some incredibly useful information.

There are many different ways to constrain a body's movement with respect to another. They could be attached at a joint, allowing them to rotate relative to that point, or they could be attached by a rod, forcing two points to remain a specified distance apart. Springs are a softer constraint and should be easier to model, applying a larger force the further apart two points are. Together with additional constraints, such as the range of angles a joint can move through, this would be the perfect basis for a ragdoll simulator.

Another very interesting area to examine would be the dynamics of non-rigid bodies, which could to a certain extent be based on my work: modelling them as many tiny particles. With a pool of liquid, possible features to be modelled could include buoyancy, waves and splashing. Jelly-like substances also exhibit a wide range of interesting behaviours incapable of appearing in a rigid body system, like wobbling. There have been several games recently featuring gelatinous blobs as the player character (for example, *Gish* and *Loco Roco*), showing that this is an area with definite potential.

Finally, there is an obvious extension for the project into three dimensions. The challenges here would be magnified, but the rewards greater and more relevant to today's 3D games. Whereas in 2D, a body always rotates around the z-axis, objects in 3D space can rotate around any arbitrary axis. Thus, handling movement and collisions becomes more complex. Other aspects, such as the collision detection and creating a user interface, would also become more complicated. However, many of the lessons I have learnt in the course of development would be applicable to a 3D engine, and it would certainly be feasible to build a 3D engine using this project as a starting point.

Bibliography

[1] BBC News. *In Depth: Millennium Bridge* [online].
http://news.bbc.co.uk/hi/english/static/in_depth/uk/2000/millennium_bridge/default.stm
[accessed 10 April 2008]

[2] Catto, E. *Iterative Dynamics with Temporal Coherence*.
Game Developer Conference Proceedings (2005).
Available: http://www.gphysics.com/files/IterativeDynamics.pdf
[accessed 26 February 2008]

[3] Catto, E. *Box2D website* [online].
Available: http://www.box2d.org/
[accessed 20 November 2007]

[4] Coumans, E. *Continuous Collision Detection and Physics*.
Sony Computer Entertainment, August 2005, Draft revision 5
Available: http://www.continuousphysics.com/BulletContinuousCollisionDetection.pdf
[accessed 15 March 2008]

[5] Eberly, D. *Game Physics*. Morgan Kaufmann, 2004.

[6] Eberly, D. *Intersection of Ellipses* [online].
Available: http://www.geometrictools.com/Documentation/IntersectionOfEllipses.pdf
[accessed 10 March 2008]

[7] Egan, K. *Techniques for Real-Time Rigid Body Simulation*.
Undergraduate Honors Thesis, Brown University, 2003.
Available: http://www.cs.brown.edu/research/pubs/theses/ugrad/2003/ktegan.pdf
[accessed 26 February 2008]

[8] Ericson, C. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[9] Erleben, K. *Module Based Design for Rigid Body Simulators*.
Technical Report DIKU 02/06, University of Copenhagen, 2002.
Available: http://www.diku.dk/publikationer/tekniske.rapporter/2002/02-06.pdf
[accessed 15 March 2008]

[10] Erleben, K. *Contact Graphs in Multibody Dynamics Simulation*.
Technical Report DIKU-TR-04/06, University of Copenhagen, 2004.
Available: http://www.diku.dk/publikationer/tekniske.rapporter/2004/04-06.pdf
[accessed 15 March 2008]

[11] Erleben, K. *Stable, Robust, and Versatile Multibody Dynamics Animation*.
PHD Thesis, University of Copenhagen, 2005.
Available: http://www.diku.dk/hjemmesider/ansatte/kenny/visionday05.pdf
[accessed 15 March 2008]

[12] Fiedler, G. *Fix your Timestep!* [online].
Available: http://www.gaffer.org/game-physics/fix-your-timestep/
[accessed 20 November 2007]

[13] Garstenauer, H. *A Unified Framework for Rigid Body Dynamics*.
Diploma Thesis, Johannes Kepler University Linz, 2006.
Available: http://www.gup.uni-linz.ac.at/~gk/Diplom/UFRBD-Helmut.pdf
[accessed 15 March 2008]

[14] Lembcke, S. *Chipmunk website* [online].
Available: http://wiki.slembcke.net/main/published/Chipmunk
[accessed 20 November 2007]

[15] Millington, I. *Game Physics Engine Development*. Morgan Kaufmann, 2007.

[16] Mirtich, B. *Timewarp Rigid Body Simulation*.
In Proceedings of the 27th annual conference on Computer graphics and interactive techniques (2000).

[17] Moravánszky, A; Terdiman, P. *Fast Contact Reduction for Dynamics Simulation*.
In *Game Programming Gems 4*, First Edition. Charles River Media Inc., 2004.

[18] Pope, N. *A definitive notation for behaviour in Empirical Modelling?*
CS405 Assignment, University of Warwick, 2007.
Available: http://www.dcs.warwick.ac.uk/~nick/documentation/NPope2007.pdf
[accessed 12 April 2008]

[19] Weber, J. *Farseer Physics Engine website* [online].
Available: http://www.codeplex.com/FarseerPhysics
[accessed 20 November 2007]

[20] Weber, J. *Farseer Physics Engine – Collision Detection* [online].
Available: http://www.codeplex.com/Project/Download/FileDownload.aspx?ProjectName=FarseerPhysics&DownloadId=24894
[accessed 15 March 2008]