Tutorial 4

So far, in the last three tutorials we have created a very simple 3D application, where we've made use of semi-implicit Euler integration to get spheres to move under the influence of applied forces.

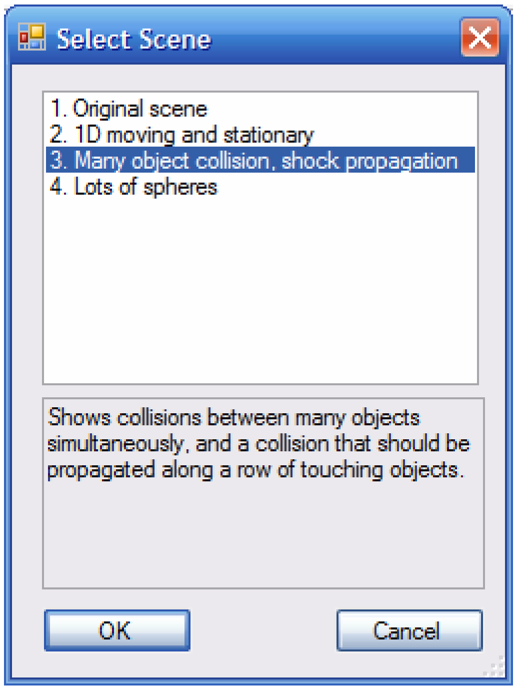
In this tutorial, we will write code to deal with collisions between spheres (sphere-sphere collision detection and response).

As there's much more code in this tutorial than in the previous tutorials, this tutorial won't be a walk-through in how to build up the code, but instead will explain the most interesting parts of the code.

Running the Example

When launching the example, you'll be presented with a dialog asking which scene to load. Choose the desired scene and hit the OK button. You can bring this dialog up at any time by hitting the Enter key.

Pressing the Space key takes a screenshot of the current frame, superimposes the time onto it, and saves it into C:\Screenshots\. Make sure that this folder exists, and that you have write access to it if you want to save images, otherwise you'll get an error message.

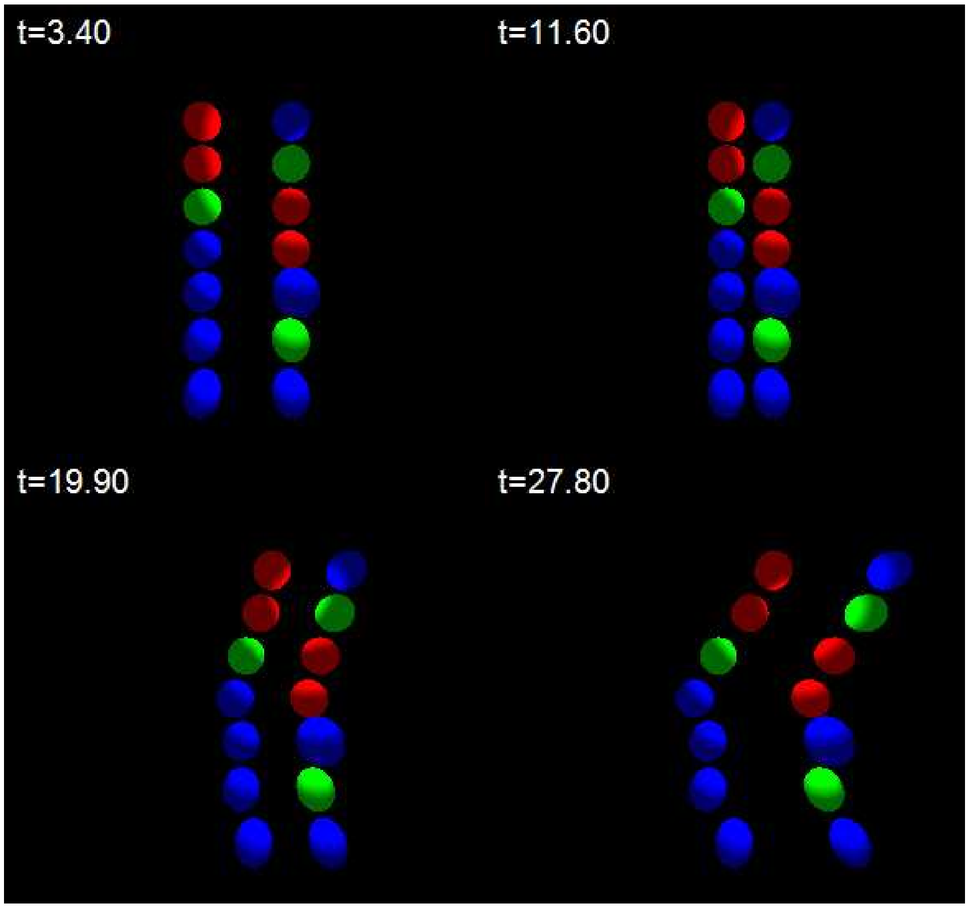

Select Scene
1. Original scene
2. 1D moving and stationary
3. Many object collision, shock propagation
4. Lots of spheres
OK
Cancel

Shows collisions between many objects simultaneously, and a collision that should be propagated along a row of touching objects.

In the following scenes, the green spheres have double the density of the blue spheres, and the red spheres are triple the density of the blue ones.

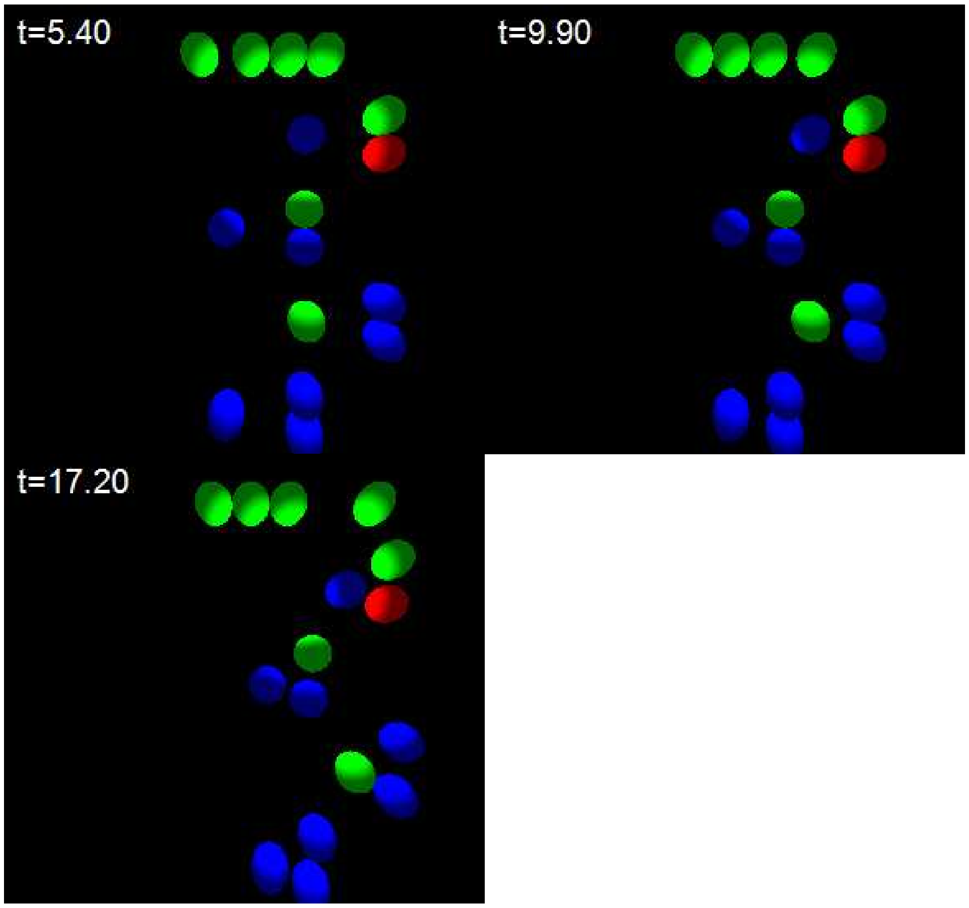
Scene 2

Shows a set of collisions between spheres of different densities.



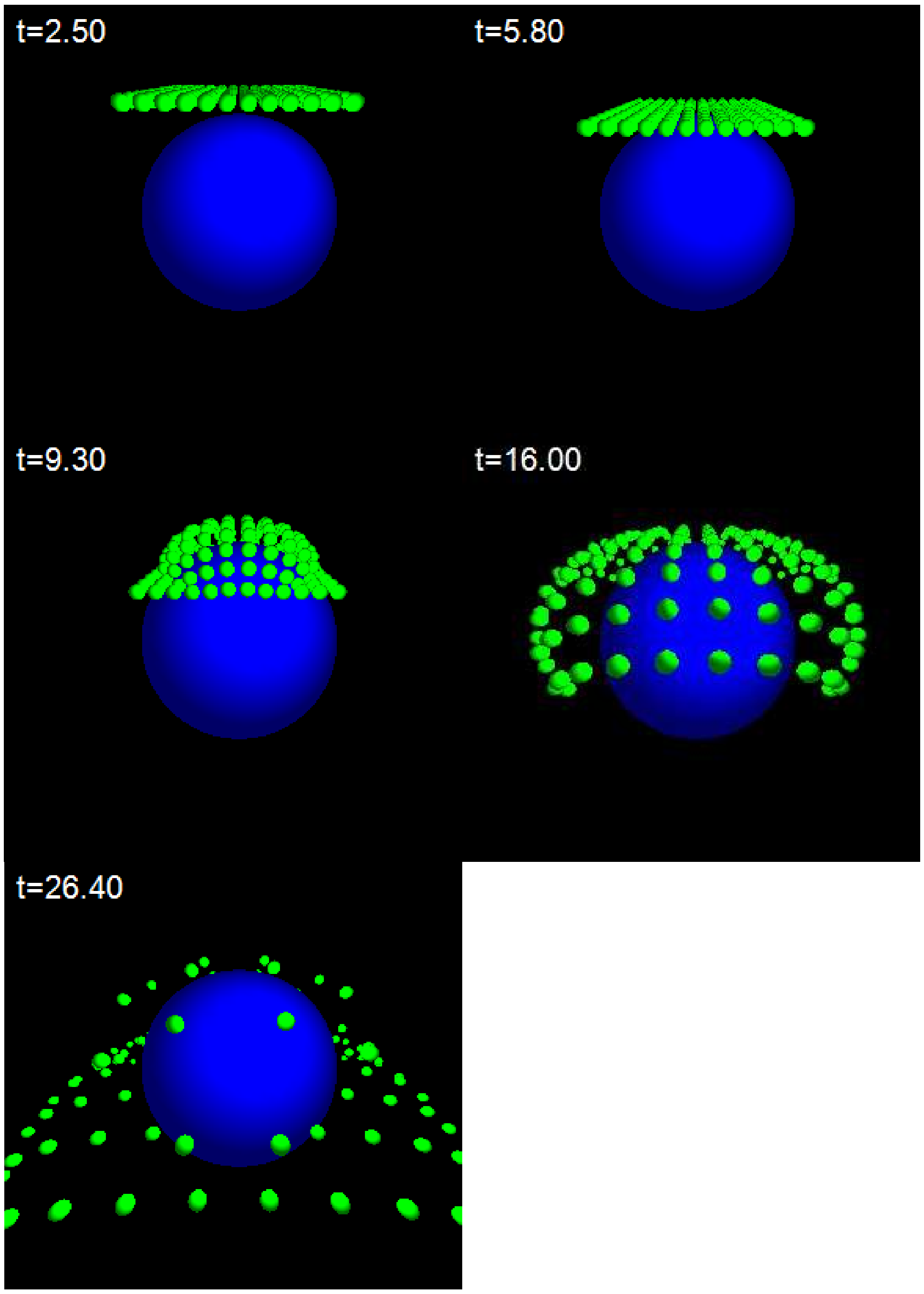
Scene 3

Shows shock propagation (i.e. Newton's Cradle) and collision between more than one object simultaneously (note that collisions between more than one object simultaneously don't currently transfer momentum correctly, and will be fixed in a later tutorial).



Scene 4

Collision of many small spheres with a large fixed sphere.



Changes to the Physics Component

We'll look at the changes in the Physics component, then the changes we made in the main app, and follow with a discussion on the problems I encountered whilst unit testing. Note that this isn't a tutorial on all of the maths and physics that you need to build a physics engine, the resources mentioned in tutorial 0 are more than adequate. What this tutorial aims to do is to show the use of the physics theory in practice, how the code is designed to support the physics in use in a simple game engine.

Overall picture of Collision Detection and Response.

The collision detection works by updating the location of all of the objects in the scene, in discrete intervals, as in the earlier tutorials, and then it checks whether any of the objects are colliding or intersecting.

The collision check is performed at the same discrete intervals that the simulation is updated at (i.e. it's not a swept sphere test – it's possible that if these intervals are too large we may miss any collisions). In the case of any spheres intersecting, the simulation is wound back to the start of the interval (any changes are discarded) and the spheres are updated with progressively smaller time steps until the time of collision is found. This is based on the algorithm in the Physics for Game Developers book (with some slight alterations I'll talk about later).

The collision response takes the form of an impulse method (an impulse is applied to both objects in a single time step – compare this with collision response using penalty methods, which inserts springs with a high spring constant between the intersecting objects).

Sphere-Sphere Collision detection

The simplest code to explain, and a good place to start, is for the case of determining whether one sphere is in collision with another. A `Sphere` class has been reintroduced into the `Physics` project, but it now lives in a `Collision` sub-folder (remember that a `Sphere` class was previously present containing the integration logic, but was renamed to `EulerRigidBody` in tutorial 3). Its purpose is to contain the logic for determining whether a sphere is in collision with another.

The `Sphere` has a member, of type `IEulerRigidBody`, that holds the physical properties of the sphere (its location and velocity), and a second member holding the radius of the sphere. The interface has been introduced to support unit testing (though that will be explained later).

The most interesting method on the `Sphere` class is `CheckForCollision()`. This method takes another `Sphere`, and uses the physical properties (location, velocity, radius) of the other sphere to detect whether the spheres are intersecting, colliding, or not touching. It returns a `CollisionType` enum that contains the results.

The simplest cases that the method checks for are that the spheres are either intersecting or not touching. It does this by calculating the magnitude of the vector that separates the centres of the two spheres, and checks whether this is larger than their combined radii.

As we're dealing with floating point values, all of the tests for equality are performed by using a static helper method, `MathUtils.IsEqualWithinTol()`, that checks whether the values are equal within a certain tolerance.

If the spheres are within touching distance, they may not be in contact with each other. The velocities need to be compared to see whether the spheres are moving towards each other or not. To do this, the velocities are decomposed into components parallel and perpendicular to the spheres' centres (using the `Vector.DotProduct()` operation). The magnitude of the relative velocity component that is parallel to the line between the sphere centres is then compared. If this component is zero, then the accelerations are compared to check whether the spheres are in resting contact (this isn't currently used but may be at some point in the future, maybe to support constraints).

Note that the `Vector` class has added methods to support subtraction, and for the dot product. The unit tests were kept up to-date with this new functionality.

The `ICloneable` interface is supported to take copies of the sphere, for the overall collision detection and response routine, its use in the collision detection will be explained in the next section. In the implementation of the `Clone()` method, the protected `MemberwiseClone()` method performs a shallow copy of all of the members (in this case only the radius), and we then manually make a deep copy of the `EulerRigidBody`, as it also supports the `ICloneable` interface.

Collision Detection and Response.

The collision detection and response takes place in the `CollisionManager` class. The constructor of this class takes a list of spheres that are to take part in collision detection and response.

The `Update()` method first takes a clone of the spheres, using the `ICloneable` interface, as it will discard of any changes and use the original spheres' states in case of any intersections. Then the spheres have their `Update()` methods called from the `UpdateSpheres()` method. Next the `CheckForCollisions()` method is called. This compares every possible combination of `Sphere` pairs to check whether they are reporting a collision (the details of this method will be explained later). The results are stored in a dictionary of collision type, against the list of `Collision` instances that are returned (the `Collision` class holds the collision type, and the pair of colliding objects).

For any collisions, the `DoCollisionResponse()` method is called. This method decomposes the velocities into components parallel and perpendicular to the centres of the spheres, uses the formula (given on the EuclidianSpace website) that solves the equations of conservation of energy and momentum for the collision along the line

parallel to the centres (at
http://www.euclideanspace.com/physics/dynamics/collision/threed/index.htm
http://www.euclideanspace.com/physics/dynamics/collision/steve.htm).

The `CollisionManager.Update()` method then calls `UpdateOriginalSpheresFromClones()` to copy the properties from the updated spheres back onto the originals.

In the case of intersection, the original spheres that are found to intersect at the end of the time interval are passed to the method `GetTimeOfIntersection()`. This method divides the time by two, and determines whether the objects still intersect. If there is no intersection or collision, then that time is taken as the *floor*, and the remaining time is bisected to centre in on the exact time of collision.

Once the time of collision has been found, then spheres are updated at that time, then the `Update()` method is called again recursively with the remaining time of the interval (to allow it to cope with any further intersections in the time interval).

`CheckForCollisions()` finds the time of intersection within a certain number of iterations. If the intersection is not found in that time frame then collision response occurs at the location that the spheres were updated to in the iterations, so there's a chance that the spheres are still intersecting, but moving away from each other.

To support the above, the `CollisionManager.CheckForCollisions()` adds some more logic over the `Sphere.CheckForCollision()` method, if the spheres report an intersection a check is made to see whether they are stationary or moving apart, and if so then it is reported as not being a collision.

Note on Setting the Vector values

Note that the `Vector` objects are never replaced on the `EulerRigidBody` objects, and that the `EulerRigidBody` instances are never replaced on the `Spheres`; the `Vector`'s values are modified on the existing instance, rather than replacing the whole `Vector`. This is because even though the 3D engine is the only client of the physics engine, at this point I don't want to make any assumptions about how somebody is using the engine – they may embed a reference to the `Vector` that a sphere contains in another object (i.e. they use the reference to the `Vector` directly without always obtaining it from the `Sphere`'s `IEulerRigidBody`).

Obviously, if the `Vector` instance changed the user would have no way of knowing this unless we added a `LocationChangedEvent` and `VelocityChangedEvent`, or make it clear to a consumer of the class that the location and velocity should not be cached. I'll probably make changes to this code in the future; I may make the `Vector` a struct rather than a class to avoid too many object allocations from stressing the garbage collector, and if so I'd make it invariant.

Performance

The change made over the loop in Physics For Game Developers is that whereas it winds back all objects involved in the simulation to move them forwards at smaller time steps, we do that only with the objects that were actually intersecting at the end of the time interval.

I said that we wouldn't be doing any performance work yet, as we shouldn't be optimising prematurely, but conversely we shouldn't leave in code that we know is too slow. The program was running far too slowly on Scene 4 when any collisions occurred (on a Turion TL52 processor).

The `CheckForCollisions()` method has to do n^2 checks on every call (where n is the number of colliding objects). The code in Physics For Game Developers winds forward every object in the scene and checks for intersections, iterating until the collision is found. With 100 objects in the scene, if it takes 5 time steps to find the collision, it has to update 100 objects 5 times, and then do the relevant collision checks. The number of checks with 100 objects is $5 * 100^2 = 50000$ checks.

The loop was modified to only wind forward and checks collisions between the intersecting objects. Now, if only 4 of the objects are intersecting in a time interval, we have to update 4 objects 5 times, and we have $5 * 4^2 = 80$ collision checks. This keeps the simulation running smooth during collisions. Part of this is due to the reduced object allocations; the `CollisionManager.Update()` method creates many temporary objects, and puts a high load on the garbage collector. We're not properly looking at performance work yet, but this does illustrate that rather than jumping straight into a profiler to try to improve performance of the actual work being done (e.g. maybe by implementing the `Vector` as a struct rather than a class amongst other things), it's obviously better to make sure that we're using efficient algorithms in the first place.

Main application.

The main app has undergone a couple of changes since the last tutorial. The interface `IScene` has been introduced, that allows the `World` class to hold any different scene. The main app uses the `SceneChooser` dialog to choose the scene to display.

The criteria to be a scene is that a class should support the `IScene` interface, and take a default constructor. A `SceneAttribute` decorates the scene with the name and description to display in the scene chooser. The program uses reflection to search for all types that are decorated with the `Scene` attribute along with the other requirements. This is only a step away from providing a plug-in architecture where the user can provide the assemblies that contain the scenes to be loaded.

Note that rather than using the attribute, we could get away with using reflection to search for objects implementing `IScene`, and maybe display their types to the user. However, apart from the fact that we wouldn't get a description and friendly scene name, we may not necessarily want all `IScene` objects to be visible (it may be, for instance, that we have some objects using the decorator design pattern that we wouldn't want the user to initialise). We could of course use an additional interface, named something along the lines of `ISceneDescriptor`, but this is why attributes are available in .NET.

The OpenGl perspective mode has been enabled, along with `GL_COLOR_MATERIAL`. The color material flag sets the material properties to ensure that shading works when using lights. As we're using `gluSphere` to create our sphere, we don't have to worry about setting any surface normals; that method does it for us.

Testing

The coding and testing followed Test-Driven Development (TDD), as was shown in tutorial 2. The unit testing of the collision manager was simple and trivial, up to a point.

The the `Sphere`'s `CheckForCollision()` method was used to drive many of the changes to the Vector class, adding the Sphere class, testing its constructor, and EulerRigidBody property etc.

Things became more involved when it came to testing and implementing the `CollisionManager`.

Problems with the Testing Approach

I approached coding the tests as usual, writing the tests first and then the supporting code. As can be seen in the algorithm for the collision detection and response, the first thing that happens is that the spheres are cloned, so I implemented test code to test that `CollisionManager.Update()` calls `Clone()` on its spheres.

Unfortunately, this means that I ended up testing the internals of the class rather than testing just its public interface. In other words, we shouldn't care whether the method clones its spheres or not on intermediate steps (it could instead of cloning, keep a copy of the spheres original sphere and location, and restore the values from these). The only thing that the test should care about is that calling the method results in the returned spheres being in the expected state (they have the same references to the `EulerRigidBody`s and `Vector`s, and their physical properties have been correctly updated).

This immediately makes this method fragile to change; it's obviously wasteful for the whole sphere to be cloned, when its only the `Velocity` and `Location` that change throughout the `CollisionManager.Update()` method, making changes to the `Update()` method to fix this will immediately mean that the test method will need changes. I'll discuss the problems with my testing approach in more detail after looking at what the tests that are present are actually doing.

Mocking

The purpose of the mock objects is to test a method while isolating it from all of its dependencies. For instance, in the `TestIteratesToTimeOfCollision` tests, we don't want to use real spheres, with `EulerRigidBody`s, as that makes the tests dependent on all of the logic in the `EulerRigidBody`. For a start, we'd have to hard code the starting location and velocity, and the time steps to ensure that the collision occurred on an easy-to-check boundary (e.g. 0.5 or 0.625 – a whole number of time steps as we subdivide the time for the collision detection).

But worse than this, the method would then be dependent on this logic. If we changed the integration method throughout from Euler, to Runge-Kutta, then all of the collision times would change. Using mock objects isolates us from this situation.

Mock objects either implement interfaces that the class depends on, or derive from those classes. In both of these cases this makes the code more open (obviously, code that only derives on interfaces is better than having code coupled to explicit classes).

The mocks are setup to supply the expected inputs to the method when called (e.g. we can set the mock up with a list of velocities to return from the `Velocity` property, when called). In addition, the mocks store the data that was passed into the methods, to enable the tests to check that the `CollisionManager` made the correct calls.

There is a generic type, `IMockClonesFactory<T>`, used to test both `Cloneable Spheres` and `RigidBodies`. This allows the mocks to be set up to either return clones in the case where we're testing for iteration, or for other tests when we want to set specific clones to be returned on the mocks.

It may be that using a mocking tool, such as NMock or Rhino would help with the testing, leading to less explicit setup code for the mocks. Some mocking tools allows us to derive from non-virtual methods, and much of the mocking code and testing code is present to support testing the `Vectors` values; it would definitely be overkill to introduce an `IVector` interface for testing.

More Criticisms of My Approach to the Testing

The first few of the `CollisionManager` tests are testing the internal implementation of the method. Instead of writing tests to test the internals of the method, I should have only ensured that the tests exercised all code paths through those methods. In testing, we want the interface to be well tested, but we purposely want much of what happens behind the interface to be free to change. Unit testing is grey-box testing, as we ensure that all of the code paths in the method are tested, but that should only to be to test that the method is performing everything that its specification demands of it. We shouldn't explicitly be testing the internals of the method. Testing the specification of the method should exercise all code paths (as all code paths are present to support the method's specification), but we shouldn't be *explicitly* testing those code paths, only the results of those code paths. In this case, I should have ensured that the test exercised the logic that cloned the spheres on `Update()`, but shouldn't have explicitly tested for that. We should be free to make any optimisations or refactorings as necessary, and the tests should still pass as long as the code's external behaviour remains unchanged.

The first few tests that were present test the internal implementation of the methods. This was getting too awkward, so I stopped testing and implemented the actual

`CollisionManager` code, returning afterwards to add the final tests in the class (just adding these tests were invaluable, they spotted bugs that weren't apparent through running the code, and goes to show that any testing is better than none).

I could have changed the tests to do as they should but I thought that showing how I went down the wrong road can be more illustrative than showing testing done well, it also shows that even experienced testers can sometimes get things wrong.

I should have started with a simple test to check the post and pre states for the mock spheres in a simple situation (no collision), moving gradually to more complex situations (a collisions halfway through the time step, then a collision at ¾ of a time step, then a collision that can't be exactly found in our 8 iterations, moving through to the case where one pair of spheres collides at a certain time, and another pair in the same scene collides at a later time, but all within the same time step).

One thing that may have helped in the testing would be for me to introduce more classes; I was scared of having a proliferation of classes, but instead I've ended up with a hard-to-test and hard-to-maintain *God class*. There are a lot of different behaviours in the `CollisionManager`, and maybe if my approach to testing was better, that behaviour would be separated into different classes. That would allow those various aspects of the overall behaviour to be more easily individually tested, tweaked, or swapped entirely.

One example is that to change our `CollisionManager.Update()` to iterate until the exact time of collision was found, rather than 8 steps. That would require changes in `CollisionManager.CheckForCollisions()`, and to check that there were no intersections at the start of the `Update()` method. It would obviously be better if this behaviour were encapsulated in a separate class.

This whole area will definitely be revisited in the future.