# Tutorial 3

In the last tutorial we used Test-Driven Development to get a sphere to move under the influence of a force. In this tutorial we will make some changes to our 3D engine to allow us to make use of this work.

First a quick recap. In the last tutorial we created a `Sphere` class in the Physics project, with an `Update()` method that performs Euler-Cromer numerical integration of the equations of motion, and two properties returning `Vector` instances, `Location` and `Velocity`. The `Vector` class that we built had properties to allow us to access its x, y and z components, and overloaded addition and multiplication operators.

Now, we want to get the `Sphere` in the MainApp to make use of the work we did in the last tutorial, so that it moves under the influence of gravity, instead of moving at a constant velocity.

## *Using the Physics Code*

To avoid confusion, I'm going to refer to the `Sphere` classes in the MainApp and Physics projects by prefixing their names with the last part of their namespaces, i.e. `MainApp.Sphere` and `Physics.Sphere`.

`MainApp.Sphere` has an `Update()` method that updates the location of the sphere (it just moves the sphere with a constant velocity). We want the `MainApp.Sphere`'s `Update()` method to do the same work as `Physics.Sphere`'s `Update()` method that we coded in the last tutorial. There are various ways we could go about this:

a) **One object for drawing and physics.** We'd copy and paste the code from `Physics.Sphere` into `MainApp.Sphere`. This is the simplest thing to do, but also the *least right*. For a start, if all of the classes in the system followed the same pattern of code duplication, we'd have to copy and paste any updates to the logic (bug fixes or performance enhancements) manually into all of these classes. This is pretty terrible. Also, it's not good design for the class to have two responsibilities; it doesn't make sense to have the `Sphere`'s drawing and physics code in the same class – we may want to make use of the `Update()` method to e.g. calculate object positions in creating PoVRay scripts – and the OpenGL drawing methods would be redundant.

b) **Inheritance.** Have `MainApp.Sphere` derive from `Physics.Sphere`. This solves the problems above – there is only one `Update()` method to maintain (and if this is shared between `Spheres` then this can be pushed into a base class). Commonly inheritance is used in OO design for code reuse. This isn't my preferred design because the `MainApp.Sphere` is tightly coupled to the `Physics.Sphere`. The `MainApp.Sphere` is only using the base sphere for code reuse, it's not overriding any of the base class methods, inheritance should be used for polymorphism, for *is-a* relationships. It may not be clear to see that a `MainApp.Sphere` *has-a* `Physics.Sphere` rather than *is-a* `Physics.Sphere` but hopefully this will become clearer below.

There's a good interview, with Erich Gamma of Design Patterns fame (and who stated that we should favour composition over inheritance) here:
http://www.artima.com/lejava/articles/designprinciples4.html

Deriving `MainApp.Sphere` from `Physics.Sphere` means that with C#'s single inheritance, deriving from a separate base class becomes more difficult (if we want to derive all of our drawable classes from a base class, it requires a rejiggling of the inheritance tree). Also, inheritance provides compile-time code reuse. If we introduce a `Physics.Sphere` making use of Runge-Kutta integration, we may end up having `Physics.SphereEuler`, `Physics.SphereRK` (we won't go down this route, this is just for sake of example), then changing the `MainApp.Sphere`'s integration method requires a recompile. However, making use of composition (calling on a member that implements `Physics.SphereX.Update()`, called via an interface), we can swap out the integration type at runtime.

c) **Composition.** Have `MainApp.Sphere` call `Physics.Sphere`. This is a step up from the previous example, the `MainApp.Sphere` would have a member holding the `Physics.Sphere`, and would call onto it; it would call onto the `Physics.Sphere` `Location` property. It's still not quite ideal; the `MainApp.Sphere`, through its interface, still has two tasks that it takes care of, drawing to the screen and updating its position (even if this actual work is delegated to the `Physics.Sphere`).

d) The final solution won't be too far from c) but with a jiggling around of the interfaces.

## *Making the Changes*

Add a reference to the `Physics` project from the MainApp project, and a `using` statement into the top of MainApp.Sphere.cs for the `Physics` namespace:

```
using Physics = Taumuon.Jabuka.Physics;
```

Add a member variable, and a constructor that takes a `Physics.Sphere` and that sets the member variable to the `MainApp.Sphere`:

```
namespace Taumuon.Jabuka.MainApp
{
    class Sphere : IDrawable
    {
        #region Object Lifetime

        public Sphere(Physics.Sphere physicsSphere)
        {
            if (physicsSphere == null)
            {
                throw new ArgumentNullException("physicsSphere");
            }
            this.physicsSphere = physicsSphere;
        }
        #endregion Object Lifetime

        #region Member Variables

        private double sphereZLocation = 0.0;
        Physics.Sphere physicsSphere = null;

        #endregion Member Variables
        ...
    }
}
```

Now let's get the project to compile. In the MainApp project's `Program` class's `Main()` method, add the same `using` statement for the `Physics` namespace as above, and change the construction of the `MainApp.Sphere` to be:

```
            Physics.Vector initialLocation = new Physics.Vector(
                0.0, 0.0, 0.0);
            Physics.Vector velocity = new Physics.Vector(
                0.0, 0.0, 1.0);
            const double mass = 1.0;
            Physics.Sphere physicsSphere =
                new Physics.Sphere(initialLocation, velocity, mass);
            Sphere sphere = new Sphere(physicsSphere);
```

Add the following line into the `Update()` method:

```
        public void Update(double increment)
        {
            this.sphereZLocation += increment;
            this.physicsSphere.Update(increment);
        }
```

Change the `Draw()` method to obtain the z translation from the `physicsSphere` object:

```
            Gl.glTranslated(0.0, 0.0, physicsSphere.Location.Z);
```

This causes a `NullReferenceException` to be thrown, as the `Physics.Sphere` `Update()` method relies on a force being set. In the unit testing phase I was too focused on refactoring out the test code to common logic, and missed this obvious case. We should go ahead and change this. First we'll add a unit test for this case. Add the following test to `TestSphereNewtonLawsDimensionSpecific`:

```
        /// <summary>
        /// Tests that the Update method works if
        ///   ApplyForce is not called.
        /// </summary>
        [Test]
        public void TestSetNoForce()
        {
            Sphere sphere = CreateSphere(0.0, 0.0);
            sphere.Update(0.1);
        }
```

We get a red bar as this test fails due to the exception. We change the `Physics.Sphere`'s `Update()` method to be:

```
public void Update(double deltaTime)
{
    if (this.force == null)
    {
        this.force = new Vector(0.0, 0.0, 0.0);
    }
    Vector acceleration = this.force * (1.0 / mass);
    this.velocity += acceleration * deltaTime;
    this.location += this.velocity * deltaTime;
}
```

We get our expected green bar. This sets the force vector to a new zero vector if it has not already been set. Note that the `Update()` method is going to be called many times, on all of the physical objects in the system, so the extra code to check for the force being null on each call may be an unnecessary performance hit. It may offer better performance if this method can assume that the force has been set; it could be set to a zero value in the `Physics.Sphere`'s constructor, and allow `ApplyForce()` to overwrite the value (of course if we're constructing many objects and calling `ApplyForce()`, then we have a wasteful allocation of the unused zero vector). It may turn out when we profile the app that these micro optimisations are unimportant; our time may be better spent with managing which objects will have their `Update()` methods called at any given time, similar to a scene graph.

Now we can remove the redundant `sphereZLocation` member from `MainApp.Sphere`, and all references to it.

Our sphere is moving as before, down towards the bottom of the screen with a constant velocity, so it may look like we haven't achieved much since tutorial 1. However, we will now get the sphere to move with under acceleration.

## Adding in a Force

We should first change our view so that the sphere moves downwards according to normal convention; our sphere is moving downward towards the bottom of the screen as the value of z increases, but the normal convention is for x to be horizontal, y to be vertical, and (in a right-hand coordinate system, as is used in OpenGL), z to be positive coming out of the screen.

Remove the following three lines from the `World` class's `RenderScene()` method:

```
// initial viewing transformation
const float viewAngle = 103.0f;
Gl.glRotatef(viewAngle, 1.0f, 0.2f, 0.0f);
```

Add a line to the MainApp's `Program` class's `Main()` method, to add a force to the `Physics.Sphere` before passing it into the `MainApp.Sphere`'s constructor:

```
Physics.Sphere physicsSphere =
    new Physics.Sphere(initialLocation, velocity, mass);
Physics.Vector force = new Physics.Vector(
    0.0, -1.0, 0.0);
physicsSphere.ApplyForce(force);
Sphere sphere = new Sphere(physicsSphere);
```

That's better, now the sphere accelerates downwards, as positive y values go from the bottom to the top of the screen, negative acceleration is towards the bottom of the screen. To make the view a little more interesting, change the initial location vector in the same method to be:

```
Physics.Vector initialLocation = new Physics.Vector(
    20.0, 0.0, 0.0);
```

Also change the initial velocity to be upwards (positive y) and slightly towards the right of the screen (positive x), so that we can more clearly see that the sphere moves in a parabolic path:

```
Physics.Vector velocity = new Physics.Vector(
    2.0, 10.0, 0.0);
```

This currently doesn't have the desired effect; we need to change our drawing code to take into account the x and y location. Change the `glTranslated()` method call in `MainApp.Sphere.Draw()` to use the `physicalSphere`'s `X` and `Y` location properties:

```
Gl.glTranslated(physicsSphere.Location.X,
    physicsSphere.Location.Y,
    physicsSphere.Location.Z);
```

Now the sphere falls towards the bottom of the screen under gravity. The `MainApp.Sphere` uses composition to achieve this; it has a member of type `Physics.Sphere` and delegates to that object to obtain its position. We're not quite done yet though, we have to tidy up the interfaces, which will force us to do something similar for the `Axes` class.

## Changing the Interfaces

We'll now go ahead and tidy up how the `Update()` method is called. It doesn't make sense for an `IDrawable` object to also update its position. Remove the `Update()` method from the `IDrawable` interface. The `World` class now fails to compile, so add a new member to the class:

```
List<Physics.Sphere> physicalSpheres = null;
```

Add a parameter to the `World` constructor, and check that this parameter is not null before assigning it to the `physicalSpheres` member.

Change the code in `World.UpdateLocations()` to iterate over this list:

```
internal void UpdateLocations()
{
    const double increment = 0.1;
    foreach (Physics.Sphere physicalSphere in physicalSpheres)
    {
        physicalSphere.Update(increment);
    }
}
```

Change the MainApp's `Main()` method to create this list, and add the `Physics.Sphere` instance to the list after the creation of the `MainApp.Sphere` object:

```
Sphere sphere = new Sphere(physicsSphere);

List<Physics.Sphere> physicalSpheres =
    new List<Physics.Sphere>(1);
physicalSpheres.Add(physicsSphere);

List<IDrawable> drawableObjects = new List<IDrawable>(2);
drawableObjects.Add(axes);
drawableObjects.Add(sphere);
World world = new World(drawableObjects, physicalSpheres);
```

The `Update()` method on the `Axes` is now not called. Hopefully you can see that even though we don't have a `Physics.Axes` class, we could give the `Axes` class a `Physics.Sphere` class to take its location values from. This may seem like a hack, but actually, that's exactly what we're going to do! Rather than being a hack, what it really shows is that our `Physics.Sphere` class is misnamed.

## Rename Physics.Sphere

We can use the exact same class for working out the location of any physical object (and later, when we introduce rotations, then the only difference in types of objects will be the moment of inertia). There will be differences in calculating the collision detection, but they will probably be factored out into a different set of classes.

We will rename `Physics.Sphere` to be `EulerRigidBody`, as it is using Euler (well, Euler-Cromer) integration to calculate the parameters of a rigid body; this indicates that we would want to use different classes for different integration methods. Of course, this may change as we find out more about the system.

You may be wondering why we didn't give this name to the class in the first place, and may be worried that the name may change again. Well, with a bit of thought up front and we may have arrived to this name, but it's only now that we see how the class is going to be used with the information gained from refactoring our MainApp.

The idea behind Test-Driven Development is that we don't have to sweat worrying about small details up front; we can devote our entire focus to coding the immediate problem one step at a time. And the tests mean that we shouldn't be scared of change (as Kent Beck, eXtreme Programming means that we can *Embrace Change*).

We'll go ahead and actually change the name of `Physics.Sphere`. Right click on the Physics.Sphere.cs class in the Solution Explorer, and select Rename. Select the new name to be EulerRigidBody and go ahead and do the rename. We're still left with work to do in changing variables, and comments. I won't walk through this, but you can see these changes in the downloadable code file. Also, we rename all variables throughout the MainApp to be `eulerRigidBody` rather than `physicsSphere`.

Similarly, we'll rename all of our test classes from, e.g. `TestSphereNewtonLawsDimensionX` to be `TestEulerRigidBodyNewtonLawsDimensionX`. Throughout the tests we'll replace the name `Sphere` with `Body` in the code and the comments.

## Fixing the Axes

Now that's done, we'll fix our `Axes` class to make use of an `EulerRigidBody`, in the same way as we did for our `Sphere` above. The steps are to add an `EulerRigidBody` member to the `Axes` class, and set it in the constructor. Then, in the `Draw()` method, change the transform to take its `Location` from the `EulerRigidBody` member.

Remove the `Update()` method from the `Axes` class, and in the `Program` class's `Main()` method, construct an `EulerRigidBody` to pass into the `Axes` method as follows:

```
static void Main()
{
    Vector initialAxesLocation = new Vector(0.0, 0.0, 0.0);
    Vector axesVelocity = new Vector(0.0, 1.0, 0.0);
    const double axesMass = 1.0;
    EulerRigidBody axesRigidBody = new EulerRigidBody(
        initialAxesLocation, axesVelocity, axesMass);
    Axes axes = new Axes(axesRigidBody);

    ...

    List<EulerRigidBody> rigidBodies =
        new List<EulerRigidBody>(2);
    rigidBodies.Add(eulerRigidBody);
    rigidBodies.Add(axesRigidBody);
```

The set of axes are now moving as they were before the refactoring. If we wanted to we could simply get them to accelerate in a certain direction. We've just got a bit more tidying up to do to the application.

## Resetting the Scene

It's a bit annoying that we have to keep restarting the application to see the sphere move, so we'll reset the Sphere's and Axes positions whenever the user presses the space key.

Open up Form1 in the designer (right click and select View Designer, or use Shift-F7). Select simpleOpenGlControl1 in the Properties tab's combobox, and click on the Events toolbar icon to show the events for this control. Double click on the KeyDown event to add a handler to the Form1 class. Now add a method to the World class, as follows:

```
internal void ResetScene()
{
    foreach (EulerRigidBody rigidBody in rigidBodies)
    {
        rigidBody.Location.X = 0.0;
        rigidBody.Location.Y = 0.0;
        rigidBody.Location.Z = 0.0;
    }
}
```

Call this method on the world instance in our event handler:

```
private void simpleOpenGlControl1_KeyDown(object sender,
        KeyEventArgs e)
{
    world.ResetScene();
}
```

Press any key on the keyboard, and you can see that the object shoots off to the bottom of the screen. Its location is set to be zero, but its velocity is still high (as it has been accelerated). So we also reset the velocity.

```
        rigidBody.Velocity.X = 0.0;
        rigidBody.Velocity.Y = 0.0;
        rigidBody.Velocity.Z = 0.0;
```

Setting each of the vector's dimensions individually does look a bit poor. We should at least think about adding a setter to the EulerRigidBody's Location and Velocity properties, or at least adding a Set() method to the Vector class to allow us to set the x, y and z values in one go. But this code is only temporary; if we need to do something similar anywhere again in the future then we'll fix this.

We'll fix our KeyDown event handler to only fire if the Space key is pressed:

```
    if (e.KeyCode == Keys.Space)
    {
        world.ResetScene();
    }
```

### A Nasty Hack

Notice that the objects don't behave exactly the same after a reset, as in the `World`'s `Main()` method they're constructed with non-zero velocities. We could quickly hack this, and assign the proper velocities as we know the indices order that we added the rigid bodies to our List.

The objects themselves shouldn't have their position coded in, and neither should the World, as they are all supposed to be generic. We'll fix this properly at some point soon; we'll have some sort of scene builder object that will construct these objects .

Add the following to the end of `ResetScene()`:

```
// TODO: hack! World shouldn't have knowledge
//  of objects initialisation!
rigidBodies[0].Location.X = 20.0;
rigidBodies[0].Velocity.X = 2.0;
rigidBodies[0].Velocity.Y = 10.0;

rigidBodies[1].Velocity.Y = 1.0;
```

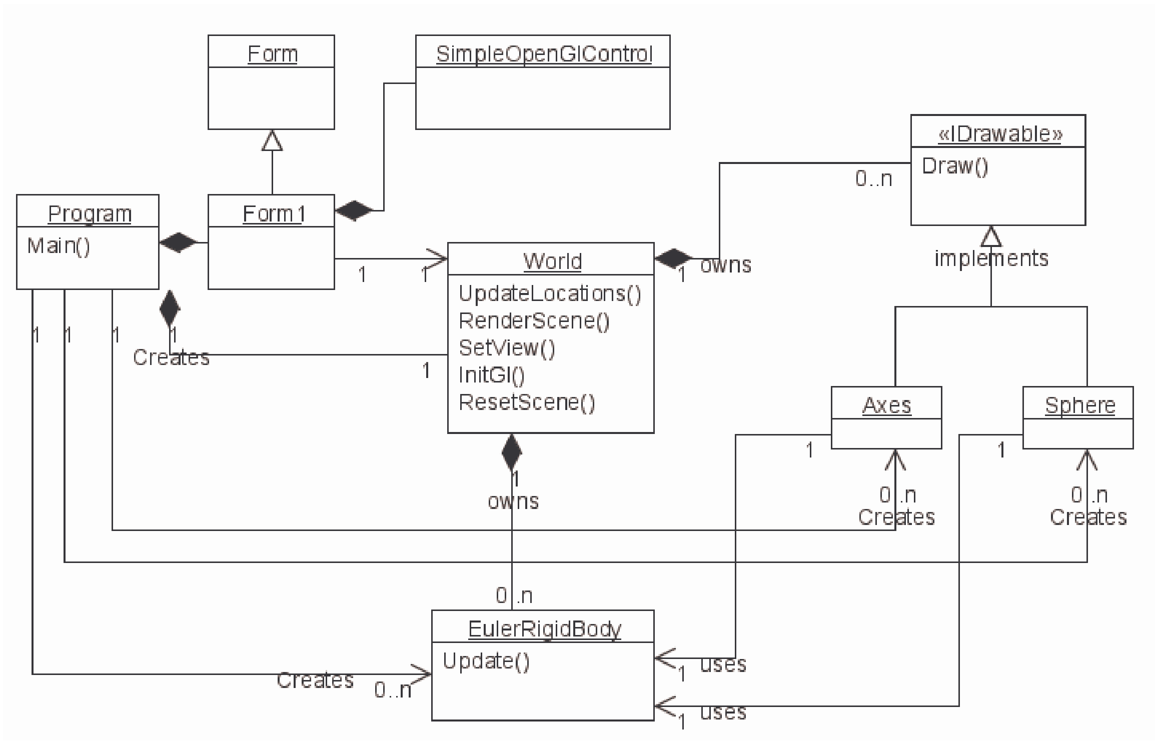
That's it, pressing the space key on the keyboard resets the objects to their initial states.

## Summary

In this tutorial, we got our sphere to move under a force, by making use of the work we did in the last tutorial. Along the way, we saw that our unit tests aren't infallible (we missed a simple case where an exception was thrown), we discussed inheritance .v. composition, and we renamed our `Physics.Sphere` to `EulerRigidBody`. The renaming came out of actually using our physics component for more than one object type. All testing was only done against a sphere, but if we'd done the testing also against the axes class (as maybe we should have done to be strictly TDD), then we would have came to the same solution.

We'll finish by looking at the UML diagram of our system so far. This shows the important classes in use (the `Vector` class isn't shown as it's too pervasive.) The Program class still creates the `World`, and all drawable and physical objects, and the `World` still has ownership for these other objects (this doesn't matter at the moment, as everything is garbage collected, but will become more important if these objects go on to implement `IDisposable`).

There's a few places we could go on from here, there's enough code here to, for example, create a simple particle system. Or we could go on to add in collision detection, or introduce rotations. But you'll have to wait until next time ;)


Form
SimpleOpenGlControl
«IDrawable»
Draw()
0..n
Program
Main()
Form1
World
UpdateLocations()
RenderScene()
SetView()
InitGl()
ResetScene()
owns
1
implements
Creates
Axes
Sphere
0..n
Creates
EulerRigidBody
Update()
uses
uses
Creates
0..n