

Tutorial 1

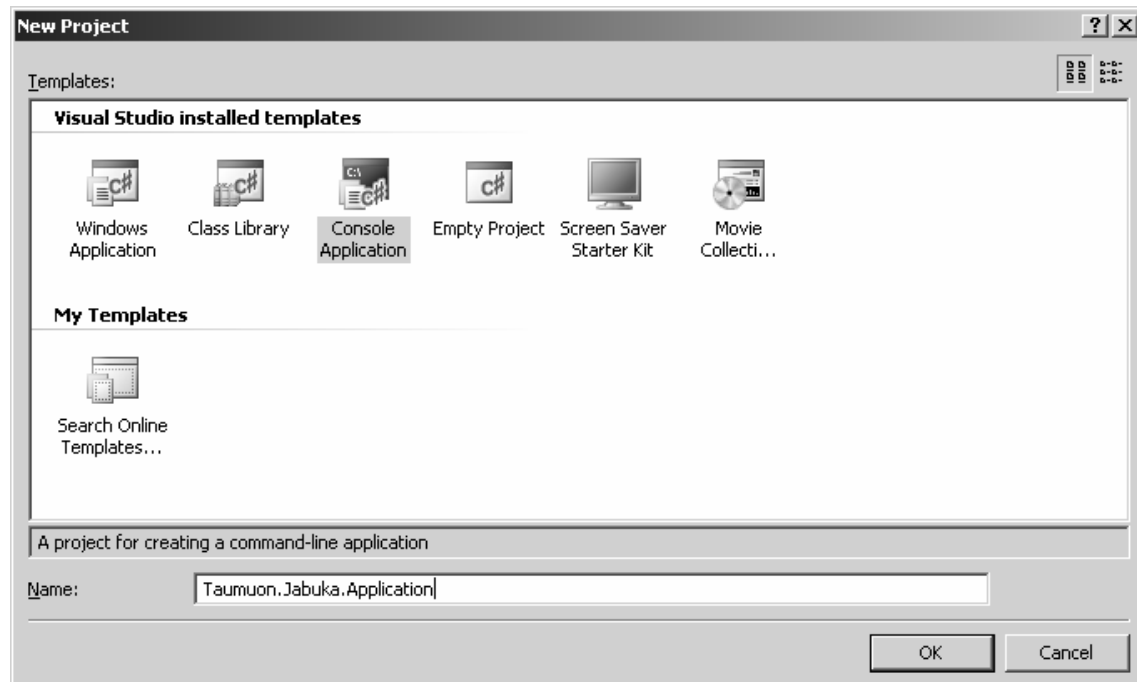
Now we've got the working Tao DLLs installed on our machine, it's time to start coding. First, we're going to create a simple program making use of the Tao framework, something we can build on as we develop our 3D and Physics engine.

This tutorial will cover the basics of OO design, but not through any upfront design! Rather, we'll arrive at an OO solution by creating our application without any objects, like typical C OpenGL tutorial code, and gradually refactor it to be more object-oriented. This may seem to be a bit of a backward way around to do things, but I think a good way to get your head around OO programming is through refactoring existing code. This is because it's easier to produce a good OO design through refactoring, rather than the method taught in OO classes (identify all nouns in the system upfront, as candidate classes) and this skill is definitely a useful one to add to your toolkit.

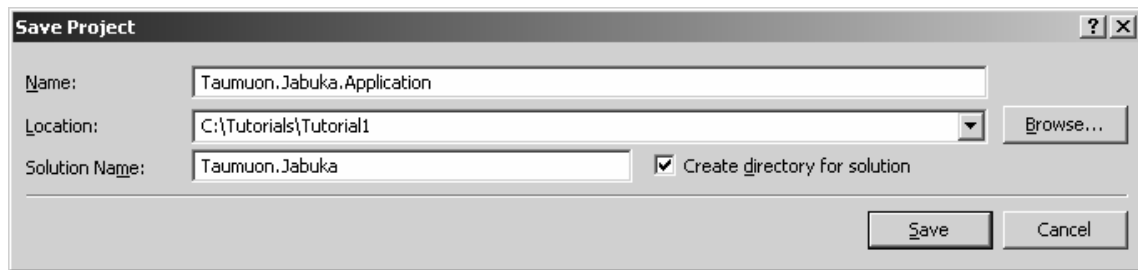
We're going to be using GLFW as our application framework – this handles window creation and user events for us (if you look at one of the NeHe examples which make use of the Tao Form control, `Tao.Platform.Windows.SimpleOpenGLControl`, in the Tao examples folder you'll see that there is a lot of hideous-looking window management code). I was originally using GLUT, but all the Tao builds I obtained since winter 2004 refuse to work with FreeGLUT. We won't worry too much about which application/GL framework we use (GLFW, Glut (if we get it working) or the Tao UserControl) as eventually all of our code should be structured so that it will be fairly trivial to easily switch between application frameworks.

Creating the Solution

Create a new folder called Tutorials somewhere on your computer. Create a new console application called `Taumuon.Jabuka.Application` in a folder called `Tutorial1` in the tutorial folder created above. To do this with Visual Studio 2005 Express, either select `New Project` from the File menu, else from the Start Page select `Create: Project`.



Select File | Save Taumuon.Jabuka.Application.sln As, name the project Taumuon.Jabuka.Application and the solution as Taumuon.Jabuka, click the Browse button and navigate to your Tutorial1 folder, and save the project.



You can run the program if you'd like – select Debug | Start Without Debugging or press Ctrl-F5 to get a very unexciting console window appear.

Remove the references to `System.Data` and `System.Xml` from the Solution Explorer (open the References tree, right click on the references above and select Remove).

Remove the using statements apart from the `System` namespace from the top of the `program.cs` file:

```
using System.Collections.Generic;  
using System.Text;
```

We'll try and keep things uncluttered as much as possible, as there's a small chance that unused statements would mislead people maintaining the code. We'll add in references and using statements as and when we need them.

Change the namespace to `Taumuon.Jabuka.Application`. Namespaces generally take the form `Company.Project.<specific>` and the assembly name is given the same name as the namespace.

Project settings

I know that you're itching to get coding, but we'll change a couple of the project settings now to make our life easier as we go on.

The first thing we will change is for all compiler warnings to be treated as errors. This is because the compiler is very good at checking that we're not doing anything stupid, and we should make sure that we listen. Right click on the project to open up its properties, and Build tab set Treat Warnings as Errors to All.

Now, in the Build tab, check the XML documentation file: checkbox, and enter `Taumuon.Jabuka.Application.xml` as the output file. This will prevent the project from building (as we're treating warnings as errors) if we omit any XML comments from any publicly visible items. The XML file provides intellisense when shipped with the assembly, and can be built into NDoc documentation. It's good to do this now rather than later as it is more painful to visit months of code adding in documentation rather than writing the comments at the same time as the code.

Getting GLFW working

Make a folder in the base tutorial directory called **Dependencies** – this is where we'll place our compile time (assemblies that we have to reference to build our project against) and runtime dependencies. Copy into there `glfw.dll` and the Tao DLLs we're going to be using (`Tao.Glfw.dll`, `Tao.OpenGl.dll`, `Tao.OpenGl.Glu.dll`).

I'm going to base the code for the GLFW application setup and OpenGL framework management code on the Boing demo supplied with the Tao Framework.

Add a reference to `Tao.Glfw.dll`, by right clicking on **References** in the **Solution Explorer**, click on the **Browse** tab and navigate to the `Tutorials\Dependencies` folder and double click on the assembly. This assembly, and any others referenced, will be copied into the `bin\release` subfolder under the `Taumuon.Jabuka.Application` project folder (this is controlled by the **Copy Local** property by right clicking the assembly in the **References** section of the solution explorer).

We're now going to add in some code to actually call GLFW. This code won't do very much, apart from initialising and closing down the framework, just to check that everything is referenced correctly.

Add a using statement for the `Tao.Glfw` namespace at the top of the `Program.cs` file.

```
using Tao.Glfw;
```

Add the following two lines in the `Main()` method:

```
// Initialise GLFW
glfw.glfwInit();

// Close OpenGL window and terminate GLFW
glfw.glfwTerminate();
```

If you run the application now, you'll get an exception `DllNotFoundException`. When running, the application tries to locate the unmanaged DLL, `glfw.dll`. This should either be in the application runtime path, or in the `$windir\System32` folder.

There are two options when running our app, we can either copy `glfw.dll` into our `bin\release` and `bin\debug` folders before running our app, or copy all of the project output and dependent referenced assemblies from our project's `bin\release` folder into a `\Tutorials\Run` folder, along with any unmanaged references.

We could achieve the copying with a post build step, or if we move on to using build scripts to build our project, that could be one of the steps. For now, we'll just copy the `glfw.dll` manually into `bin\release` and `bin\debug` – we won't have to do this that often anyway, as we shouldn't have reason to clear `bin\` folders.

Now we can run our application again. Nothing earth-shattering will happen, but at least we know by initialising and terminating the GLFW framework that that everything's referenced correctly, so we're in a good state to add some functionality.

Add a reference to `Tao.OpenGl.dll`, and a using statement at the top of the file for `Tao.OpenGl` (necessary for the `GL_FALSE` constant).

Now we're going to add code to open a GLFW window, into which our rendering will take place. The following code returns from the `Main()` method and so terminates the application straight away if the window fails to open. Add the following code in-between the two method calls we just added above:

```
if(Glfw.glfwOpenWindow(400, 400, 0, 0, 0, 0, 16, 0,
                      Glfw.GLFW_WINDOW) == GL_FALSE)
{
    Glfw.glfwTerminate();
    return;
}
```

Run the program, but if you blink you'll miss it. Our window briefly appeared and disappeared, as execution reached the final `glfwTerminate()` call. You can pause the application briefly by inserting a sleep as follows:

```
System.Threading.Thread.Sleep(1000);
```

The sleep is where we'll actually have our main program loop. The program will run in a loop, drawing the scene, then maybe processing physics or AI, until the user exits the application by closing the window. This means that the frame rate will depend on the time spent performing those actions. There are probably more sophisticated ways to do structure the program execution (especially when running with multiple threads), but this is a simple start that will do us for now.

First, we'll get the window to display without disappearing:

```
bool isRunning = true;

// Main loop
while ( isRunning )
{
    // Swap buffers
    Glfw.glfwSwapBuffers();

    // Check if we are still running
    isRunning = (
        ( Glfw.glfwGetKey(Glfw.GLFW_KEY_ESC) ==
          Glfw.GLFW_RELEASE )
        && Glfw.glfwGetWindowParam(Glfw.GLFW_OPENED) ==
          GL_TRUE );
}
```

Insert the above code after the `glfwOpenWindow()` if block, and before the final `glfwTerminate()`. The while loop continues to loop, determining whether the window has been closed. Without the `glfwSwapBuffers()` call the application hangs, it probably ensures that the Windows messages are pumped.

Now that our application opens a window, and we can close the application by closing that window, there's not much point for the console window to appear. We'll stop the console window appearing now. Right click on the `Taumuon.Jabuka.Application` project and select `Properties`. On the `Application` tab, change the `Output Type` to be `Windows Application` rather than `Console Application`.

There is one last thing to do before we can start drawing. We'll set up various OpenGL settings. Again, an OpenGL reference book should describe what these do. Prior to the main loop insert the following.

```
// sync buffer swap to monitor vSync (if supported
// in hardware and drivers).
Glfw.glfwSwapInterval(1);

Gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
Gl.glShadeModel(Gl.GL_FLAT);
Gl.glEnable(Gl.GL_DEPTH_TEST);
Gl.glEnable(Gl.GL_CULL_FACE);
```

Starting to Draw

Now, let's go ahead and do some drawing. We will draw three perpendicular axes to the screen (so that you can see where the x, y and z axes lie), we will redraw the scene in each loop of our application. Let's add the following static method to our `Program` class, called `RenderScene()`, and call it in the main loop just before the call to `glfwSwapBuffers()`. The method is static as there isn't an instance of the program class created for the `Main()` method to call.

The following is our first taste of proper OpenGL. We first clear the scene, then we store our current viewing transformation (this will become clearer later on when we start transforming objects more than once).

```
/// <summary>
/// Draws a frame.
/// </summary>
private static void RenderScene()
{
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
              Gl.GL_DEPTH_BUFFER_BIT);

    Gl.glPushMatrix();

    // do our drawing here

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

    #region Draw Axes
    const float axisSize = 25.0f;

    // draw a line along the z-axis
    Gl.glColor3f(1.0f, 0.0f, 0.0f);
    Gl.glBegin(Gl.GL_LINES);
        Gl.glVertex3f(0.0f, 0.0f, -axisSize);
        Gl.glVertex3f(0.0f, 0.0f, axisSize);
    Gl.glEnd();

    // draw a line along the y-axis
    Gl.glColor3f(0.0f, 1.0f, 0.0f);
    Gl.glBegin(Gl.GL_LINES);
        Gl.glVertex3f(0.0f, -axisSize, 0.0f);
        Gl.glVertex3f(0.0f, axisSize, 0.0f);
    Gl.glEnd();

    // draw a line along the x-axis
    Gl.glColor3f(0.0f, 0.0f, 1.0f);
```

```
    Gl.glBegin(Gl.GL_LINES);
        Gl.glVertex3f(-axisSize, 0.0f, 0.0f);
        Gl.glVertex3f(axisSize, 0.0f, 0.0f);
    Gl.glEnd();
    Gl.glPopMatrix();
    Gl.glFlush();
    #endregion Draw Axes
}
```

We're now getting somewhere – we can see on the screen three different coloured axes. You can play about with the angles in the `GLRotate()` statement to see the effect.

Handling Window Resizing

There's still something to fix though. Try resizing the window – you can see that the window resizes but that as you resize it, it just shows more or less of the scene; this isn't what we want. We want our scene to be resized to always fit the window. To do this we add a callback that is called by GLFW whenever the window is resized.

GLFW provides a delegate for us to register our callback, `GLFWwindowresizefun()`. A delegate allows a reference to a method to be passed or stored (similar to function pointers in C++). The delegate provided is defined as having a signature of a void return type, and taking two parameters as ints. We can register whatever method we want to be called as long as it has the required signature.

We register the delegate using `glfwSetWindowSizeCallback()` as in the following code (insert this prior to your `glfwSwapInterval(1)` call above the main loop:

```
    glfwSetWindowSizeCallback(
        new GLFWwindowresizefun(Reshape));
```

And add the `Reshape()` method in as follows. The implementation is from OpenGL Superbible, I'm not too worried about how it works:

```
/// <summary>
///     Handles GLFW's window reshape callback.
/// </summary>
/// <param name="width">
///     New window width.
/// </param>
/// <param name="height">
///     New window height.
/// </param>
/// <remarks>
///     Implementation based on details in OpenGL Superbible
/// </remarks>
private static void Reshape(int width, int height)
{
    // Set viewport to window dimensions.
    Gl.glViewport(0, 0, width, height);

    // Reset projection matrix stack
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();

    const float nRange = 80.0f;

    // Prevent a divide by zero
    if (height == 0)
```

```
{
    height = 1;
}

// Establish clipping volume (left, right, bottom,
// top, near, far)
if (width <= height)
{
    Gl.glOrtho(-nRange, nRange, -nRange * height / width,
               nRange * height / width, -nRange, nRange);
}
else
{
    Gl.glOrtho(-nRange * width / height,
               nRange * width / height,
               -nRange, nRange, -nRange, nRange);
}

// reset modelview matrix stack
Gl.glMatrixMode(Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
}
```

Adding a Sphere

Now, that we've got the window behaving correctly, let's add another object to our scene. We'll add a sphere in, as it's a simple object that we'll be making much use of. This implementation is based on code from NeHe lesson 18.

Add a reference to `Tao.OpenGl.Glu.dll`. After the drawing of the last axis, prior to the `popMatrix()` call in `RenderScene()`, add the following:

```
#region Draw Sphere
bool drawWireFrame = false;
int drawStyle = drawWireFrame ? Glu.GLU_LINE :
                                                         Glu.GLU_FILL;

double radius = 5.0;

// From NeHe lesson 18.
Glu.GLUquadric quadric = Glu.gluNewQuadric();
try
{
    Glu.gluQuadricDrawStyle(quadric, drawStyle);
    Glu.gluQuadricNormals(quadric, Glu.GLU_SMOOTH);
    Glu.gluSphere(quadric, radius, 40, 40);
}
finally
{
    Glu.gluDeleteQuadric(quadric);
}
#endregion Draw Sphere
```

The drawing of the sphere makes use of a quadric. The calls are enclosed in a try-finally block as we want to ensure that the quadric is deleted in the case of any errors. Later on we may profile the code to see how inefficient creating and destroying the quadric on every scene render is – it may be that we end up caching the quadric in an object and making it `IDisposable` to ensure that it's cleaned up.

You can see that the sphere is positioned over the centre of the axes. We can change the position of the sphere by enclosing the code to draw the sphere that we added above in the following calls.

```
#region Draw Sphere

// Store current view
Gl.glPushMatrix();
Gl.glTranslatef(3.0f, 3.0f, 3.0f);

.. our code to draw the sphere goes here

// Restore current view
Gl.glPopMatrix();

#endregion Draw Sphere
```

Some Refactoring

We've got a basic application working, with the ability to add and draw many objects on the screen. This example takes the form of many OpenGL tutorials you'll see on the web – it's simple, which is good to show the basic concepts, but a bit too simple to do further work on. It resembles a C program with all the methods being static, and not an object in sight. We want to make the example a bit more object-oriented, giving a firm foundation for doing further work. That's what we'll go ahead and do now.

Add a class called `World` in a new file, `World.cs`. (Do this by right clicking on the project and selecting **Add | Class**), name the class to `World.cs`.) Add the `public` specifier to the `World` class, this is to ensure that we get warnings for all missing XML comments. Add the following comment:

```
/// <summary>
/// Initialises GLFW, draws various objects.
/// </summary>
public class World
```

Add the following using statements at the top of the file:

```
using Tao.OpenGl;
using Tao.Glfw;
```

Add a public method called `Run()` (returning void and taking no parameters) and move the contents of `EntryPoint.Main()` into there. I'll let you add the comments, to these methods and others that I don't add throughout this tutorial. You can look at the downloaded code if you run out of ideas.

Move the `Reshape()` and `RenderScene()` methods into the `World` class, and remove the static modifiers from their declarations.

In the `Program` class's `Main()` method, instantiate a `World` class, and call `Run` on it:

```
World world = new World();
world.Run();
```

That's made the `Program` class much cleaner. Now, it's only doing one job – the responsibility for rendering the scene has moved to the `World` class. The `Program` class may

in the future be used for e.g. setting up the scene from XML, measuring the time to perform a specific run, or number of runs.

The `World` class is still doing two jobs, setting up GLFW and handling its events, and drawing the scene, so it's likely that we'll split this responsibility up later. For now though, it's not a big problem.

There's still a little more tidying to do. Select the lines

```
Gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
Gl.glShadeModel(GL.GL_FLAT);
Gl.glEnable(GL.GL_DEPTH_TEST);
Gl.glEnable(GL.GL_CULL_FACE);
```

And select the Refactor | Extract Method menu item, name the method in the dialog box as `InitGL`.

Now perform an Extract Method on the contents of the `while` loop, and give it the name `MainLoop`. The extract method hasn't been quite so successful this time – it passes the `isRunning` variable into the method, when all it needs to do is retrieve the value. Remove the `isRunning` parameter as follows:

```
/// <summary>
/// The main rendering loop. Draws the scene and handles
/// user input
/// </summary>
/// <returns>
/// true to indicate that program should continue,
/// false if the user has closed the window.
/// </returns>
private bool MainLoop()
{
    RenderScene();

    // Swap buffers
    Glfw.glfwSwapBuffers();

    // Check if we are still running
    bool isRunning = (
        (Glfw.glfwGetKey(Glfw.GLFW_KEY_ESC) ==
         Glfw.GLFW_RELEASE)
        && Glfw.glfwGetWindowParam(Glfw.GLFW_OPENED) ==
         Gl.GL_TRUE);
    return isRunning;
}
```

And remove the parameter from the call to the method in the `while` loop:

```
// Main loop
while (isRunning)
{
    isRunning = MainLoop();
}
```

In my mind I've already got some more possible refactorings, and the introduction of objects, but it's best to get some more functionality in first and look out for *code smells*, that is, where there is duplication of code or responsibility for performing an action is in the wrong place.

That being said, let's introduce some movement into our application. We'll move the sphere along the z axis, and we'll move the axes along the y axis (hopefully this makes sense!).

Introducing Animation

We'll introduce two variables, one to hold the sphere z location, and one to hold the axes y location. Inside the `World` class, before the `Run()` method, insert the following:

```
#region Member Variables

double sphereZLocation = 0.0;
double axesYLocation = 0.0;

#endregion Member Variables
```

Now, we'll change the drawing of the objects to use these variables. In `RenderScene()`, just before the drawing of the x axis, at the start of the `Draw Axes` region, insert the following:

```
Gl.glPushMatrix();
Gl.glTranslated(0.0, axesYLocation, 0.0);
```

And following the drawing of the z axis, before the end of the `Draw Axes` region, insert:

```
Gl.glPopMatrix();
```

The `glPushMatrix()` call stores the current viewing transformation on the stack, so that we can make the `glTranslated()` call and return to our original state afterwards with the `glPopMatrix()`. We already have the `glPushMatrix()` and `glPopMatrix()` calls around the sphere drawing, we just change the `glTranslatef()` call to be `Gl.glTranslated(3.0, 3.0, 3.0 + sphereZLocation)`.

Run the application and you can see that nothing has changed. We actually want our main loop to change the location variables on each pass of the loop. In `MainLoop()`, after the swap buffers call and before the `isRunning` check, add the following:

```
const double increment = 0.1;
this.axesYLocation += increment;
this.sphereZLocation += increment;
```

You may want to change this increment to be a larger value on a slow system, and vice-versa.

Now we have the application doing just about everything we want for this tutorial, but we're not ready to introduce any Physics yet, we'll go ahead and do some more refactoring first.

More Refactoring

The first bit of code that we'll tackle is the `RenderScene()` method, it currently is hardcoded to draw two objects, making use of two member variables in the world. We could split out the drawing of the axes and the sphere into two separate methods (`DrawAxes()` and `DrawSphere()`), but this is a good time to introduce classes to represent these – we've got two similar methods both using similar data (the location of the object). We'll do them both together, so let's add two classes to the project, `Axes` and `Sphere`.

It's a fairly quick and easy job to just move the data and methods over to the new classes, but for demonstration purposes we'll take our time. We'll introduce the classes in baby steps, at no point we will move to far away from having a working program.

Add a `Draw()` method to both of our classes. Into each, copy the drawing code specific to the object from `RenderScene()` including the preceding `glPopMatrix()`, `glTranslated()`, and `glPushMatrix()` calls (each separate `Draw` region). We won't delete anything from `RenderScene()` for now, as we don't want to move too far away from a moving program – in the worse case we could scrap our new work if it had a hard-to-find bug in it. This may seem overly cautious for this simple example, but it's a good habit to get into for when performing large and/or complicated program changes.

Add the `using` directive for `Tao.OpenGl` at the top of the classes. Now, add the missing member variables to the classes, making them public, for example in the `Axes` class insert:

```
public double axesYLocation = 0.0;
```

You may be worried about these being public, but it's just a shortcut while we're refactoring, we'll soon be making these private. If you were concerned that you may forget to change these to private, you could always do so now and add properties to access them.

In the `World` class we'll add a couple of member variables for our new classes:

```
Axes axes = new Axes();  
Sphere sphere = new Sphere();
```

In the `RenderScene()` method, call the `Draw()` method of these two new classes:

```
axes.Draw();  
sphere.Draw();
```

Now, there are two copies of our axes and sphere being drawn, the ones we've just introduced obviously aren't moving (we're not changing their member variables).

In the `MainLoop()` method, after the two lines where we increment the `World`'s members, add the following:

```
axes.axesYLocation += increment;  
sphere.sphereZLocation += increment;
```

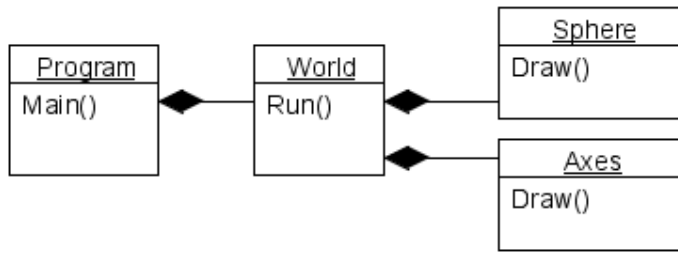
Now, run the program and it appears that there is only one set of objects again – we're successfully updating the positions of our newly-introduced objects to match the old copies, and so the old objects are redundant.

From the `World` class, remove the two member variables that hold the axes and sphere locations, and remove the code that increments these variables from the `MainLoop()` method. Delete the regions of code that draw our objects from `RenderScene()`. Doesn't that feel much neater? The `World` class is no longer cluttered with object-specific drawing code.

Introducing Interfaces

Here's an UML diagram of our classes so far (the filled in box represents composition – the lifetimes of the `Sphere` and `Axes` are controlled by the lifetime of the `World`).

Notice that the `World` has two hardcoded members, the `Axes` and the `Sphere`. It won't be much more work to make the `World` totally generic, to allow us to pass in any objects we want and to get the `World` to draw them.



In `MainLoop()`, rather than updating the two location member variables directly, we're going to delegate to the classes to update their positions themselves. There are two motivations behind this – the first is that we want to encapsulate the inner workings of the class, now the internal representation of the location could be a `Vector` class, and we wouldn't care. The other motivation is to get the interface for the two classes the same, as you'll see below. Onto the `Axes` and `Sphere` classes, add a method:

```
public void Update(double increment)
```

The implementation for the `Axes` class is:

```
axesYLocation += increment;
```

The `MainLoop()` method now can call:

```
axes.Update(increment);
sphere.Update(increment);
```

And the member variables contained in the `Axes` and `Sphere` classes can be made private. Now the `World` class interacts with the `Axes` and `Sphere` through the `Draw()` and `Update()` methods – it knows nothing of their internal representation. Rather than directly calling e.g. `Draw()` on the axes and the sphere, we want to call `Draw()` on any object that we will store in the `World`. This is the perfect use of polymorphism, and `C#` allows us to use an interface to achieve this (in `C++` you'd have to derive from an abstract base class). Add a new item to the file (`Project | Add | New Item | Code File`), and name it `IDrawable.cs`. Insert the following implementation:

```
namespace Taumuon.Jabuka.Application
{
    /// <summary>
    /// Allows objects to be drawn.
    /// </summary>
    public interface IDrawable
    {
        /// <summary>
        /// Draws an object using OpenGL calls.
        /// </summary>
        void Draw();
        /// <summary>
        /// Lets the object update its position.
        /// </summary>
        /// <param name="increment">
```

```
    /// Time increment - larger numbers move object faster.
    /// </param>
    void Update(double increment);
}
}
```

Note that the `Update()` method doesn't really belong here, the `Drawable` in the name suggests that the class has the capability to be drawn (similar to an `IDisposable` class having the capability of being disposed). The updating of the location will be handled by some sort of physics interface, but we won't worry about this for now – it's not too much hassle to move things around, and it's probably better to defer it until we have more knowledge of the system, rather than second guessing ourselves. Now, derive the `Axes` and `Sphere` classes from this new interface:

```
/// <summary>
/// Draws a sphere on the screen.
/// </summary>
public class Sphere : IDrawable
```

Now, in the `World` class, add the following member:

```
List<IDrawable> drawableObjects = new List<IDrawable>();
```

Add a constructor, and populate this list by removing the `axes` and `sphere` member variables – we'll create them as local variables in the constructor and add them directly to the list:

```
/// <summary>
/// Constructor.
/// </summary>
public World()
{
    Axes axes = new Axes();
    Sphere sphere = new Sphere();
    drawableObjects.Add(axes);
    drawableObjects.Add(sphere);
}
```

In the `MainLoop()` method, rather than calling `Update()` on the two objects directly, you can just iterate over the contents of the list and `Update()` them, as follows:

```
foreach (IDrawable drawableObject in drawableObjects)
{
    drawableObject.Update(increment);
}
```

And similarly in the `RenderScene()` method – just iterate over the objects as in the `foreach` loop above, and call `Draw()` rather than `Update()`. This is really tidy, the `World` class is much smaller, and its responsibilities have been diverted to the new classes that we've introduced. We've still got the hardcoded `Axes` and `Sphere` classes in the `World` constructor, but it's an easy job to construct this in the `Program` class (maybe from XML), or deserialise it in the `World` class. We'll go ahead and make the change for the `Program` class to create our objects. Change the member variable declaration in `World` to be:

```
List<IDrawable> drawableObjects = null;
```

And the `World`'s constructor to be:

```
/// <summary>
/// Constructor.
/// </summary>
public World(List<IDrawable> drawableObjects)
{
    System.Diagnostics.Debug.Assert(null != drawableObjects);
    this.drawableObjects = drawableObjects;
}
```

Don't worry about the `Assert()` method for now, we'll discuss the strategies (assertions versus exceptions) for methods' arguments in another tutorial. Now, finally, add a `using` directive for `System.Collections.Generic` to the `Program` class, and change the class's `Main()` method to be:

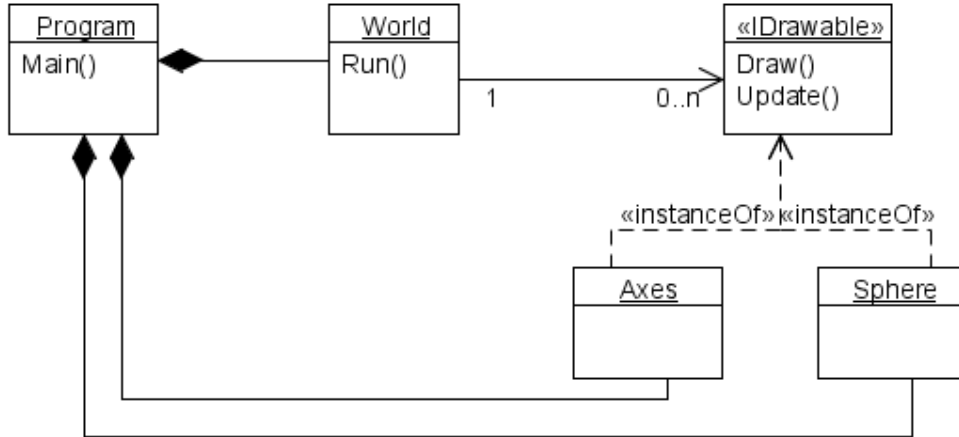
```
static void Main( string[] args )
{
    Axes axes = new Axes();
    Sphere sphere = new Sphere();
    List<IDrawable> drawableObjects = new List<IDrawable>();
    drawableObjects.Add(axes);
    drawableObjects.Add(sphere);

    World world = new World(drawableObjects);
    world.Run();
}
```

Finally, change the `World`'s comment to be more descriptive:

```
/// <summary>
/// Initialises GLFW, opens a window and holds a list of
/// IDrawable objects, drawing them.
/// Usage:
/// Construct with IDrawable list, call Run() and loop runs
/// drawing objects. Run() returns when the user has closed
/// the GLFW window.
/// </summary>
public class World
```

Now, this is much better, the `World` only talks to the objects it contains through the `IDrawable` interface – it's trivial for us to create another object and get it to be drawn by the `World`. Now, the lifetimes of the `World`, `Axes` and `Sphere` are obviously dependent on the lifetime of the `Program` (hence the composition). The `World` has associations to objects implementing the `IDrawable` interface (I've used association rather than aggregation in the diagram as there's not any difference in the implementation. There's a discussion of this to be found at: <http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf>)



There are another couple of little niggles with the code – the storing of the z location in the sphere, and the y location in the axis. It’s easy to imagine introducing some sort of `Vector` class, but I’m reluctant to do it without introducing some unit tests.

In the next tutorial, we’ll introduce some physics, and expand upon this design to get the physics and 3D coupled together.