Tutorial 1

In Tutorial 0 we installed the Tao DLLs onto our machine, and checked that they worked OK. Now it's time to start coding. In this tutorial, we're going to create a simple program making use of the Tao Framework to display objects in 3D using OpenGL. We will build upon this as we develop our 3D and Physics engine.

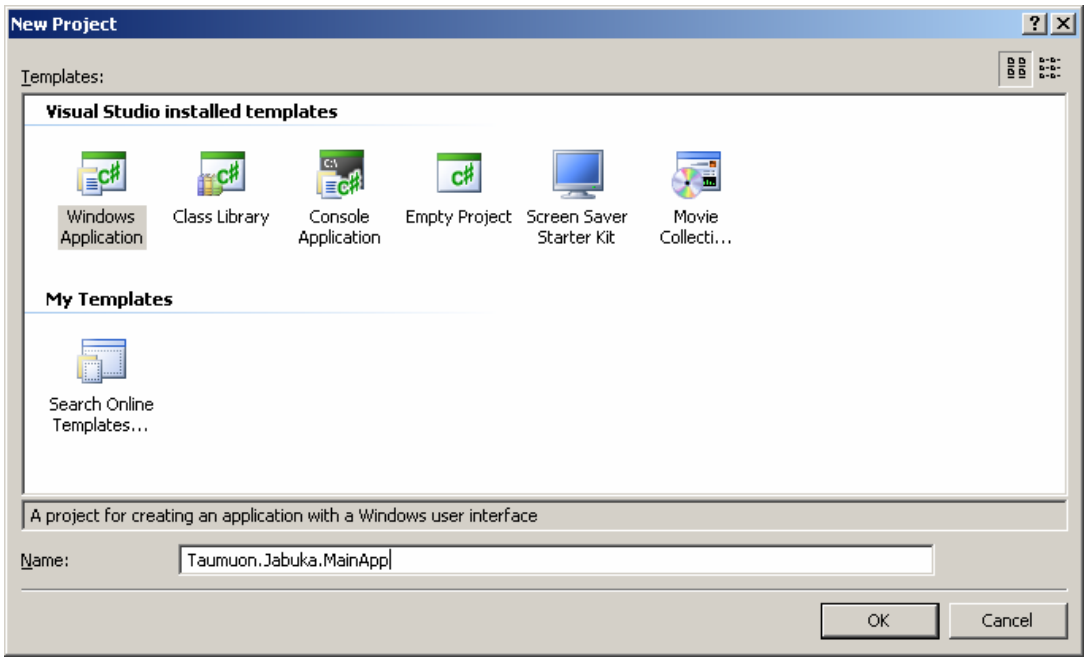
This tutorial will cover the basics of OO design, but we're not going to do any upfront design. Rather, we'll first create our application without introducing any objects, as in typical C OpenGL code, and gradually refactor it to be more object-oriented. This may seem to be a bit of a backward way around to do things, but I think that a good way to get your head around OO programming is through refactoring existing code.

It's easier to see how small changes to code can produce a good OO design, by moving methods and members so each class has the correct functionality, rather than trying to identify all classes and their functionality upfront (as taught in OO classes, by searching for nouns in the system documentation to be candidate classes). Also, refactoring is definitely a useful skill to add to your toolkit.

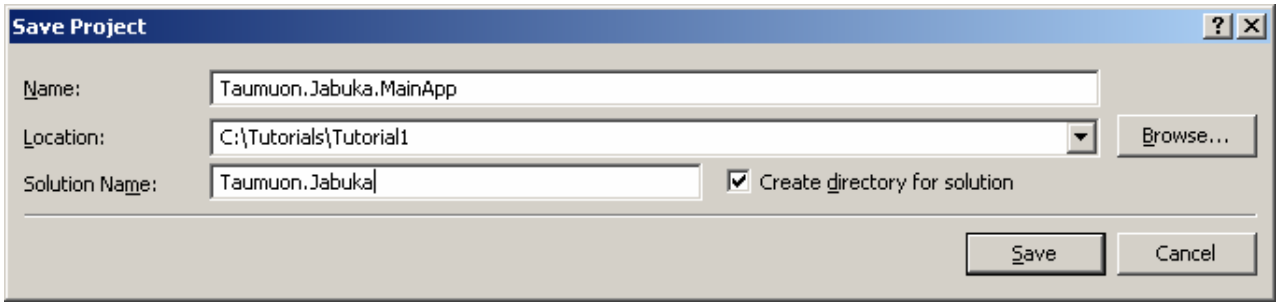
We're going to be making use of the Tao control, `SimpleOpenGlControl` that lives in the `Tao.Platform.Windows` namespace in our application. If you look at one of the NeHe examples which make use of this control in the examples folder of the source code, you'll see that there is a lot of hideous-looking window management code. We'll just use standards Windows events, which may not give the same performance, but it will keep the code simple. The aim is to keep the Windows event management and OpenGL drawing code factored into separate classes so that it will be simple to switch to GLUT, GLFW or SDL if we want to.

Creating the Solution

Create a new folder called Tutorials somewhere on your computer. Create a new Windows application called Taumuon.Jabuka.MainApp in a subfolder called Tutorial1 in the Tutorials folder that we've just created. 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.MainApp.sln As, name the project Taumuon.Jabuka.MainApp 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. Right click on Form1 in the Solution Explorer, and choose Rename. Type the new name as MainOpenGlForm, and select Yes if a dialog appears asking whether to rename all solution items.

Note the files that are part of the project. The Program.cs file is the application's entry point. This method creates a new instance of the `MainOpenGlForm` class, and passes it to the static `Application.Run()` method. `MainOpenGlForm` is a partial class, meaning that its definition is spread across more than one file. MainOpenGlForm.cs is where we'll be adding our own code, and MainOpenGlForm.Designer.cs is where Visual Studio adds code when we e.g. drop controls on the form.

Remove the references to System.Drawing, System.Deployment, 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 one for the `System` namespace, from the top of the Program.cs file:

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

And remove the following from the MainOpenGlForm.cs file:

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
```

We'll try and keep things uncluttered as much as possible, there's no harm in importing all of the namespaces that Visual Studio adds to our code, but I prefer to keep things simple and add namespaces and references in when necessary.

Note that all the code is contained in a namespace which has the same name as our project, `Taumuon.Jabuka.MainApp`. Namespaces generally take the form `Company.Product.<specific>` and the assembly name is generally 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 to it. Right click on the project to open up its properties, and in the Build tab set Treat Warnings as Errors to All.

Now, in the Build tab, check the XML documentation file: checkbox, and enter Taumuon.Jabuka.MainApp.xml as the output file. This will generate a warning if we omit any XML comments from any publicly visible items, and as we're treating warnings as errors, the project will fail to build. The XML file provides intellisense when shipped with the assembly, and can be built into compiled help documentation with a tool such as NDoc or Sandcastle. 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.

Now, as `MainOpenGlForm` is publicly visible, we need to provide an XML comment for it. Change the definition in MainOpenGlForm.cs to be:

```
public partial class MainOpenGlForm : Form
{
    /// <summary>
    /// The Application's main form.
    /// </summary>
    public MainOpenGlForm()
    {
        InitializeComponent();
    }
}
```

Add the same comment above the `MainOpenGlForm` definition in MainOpenGlForm.Designer.cs.

Getting the Tao OpenGL Control Working

We're going to be using the `Tao.Platform.Windows.SimpleOpenGlControl` control to do our drawing. The first thing to do is to make this control available via Visual Studio's toolbox, allowing us to drag and drop it onto our form.

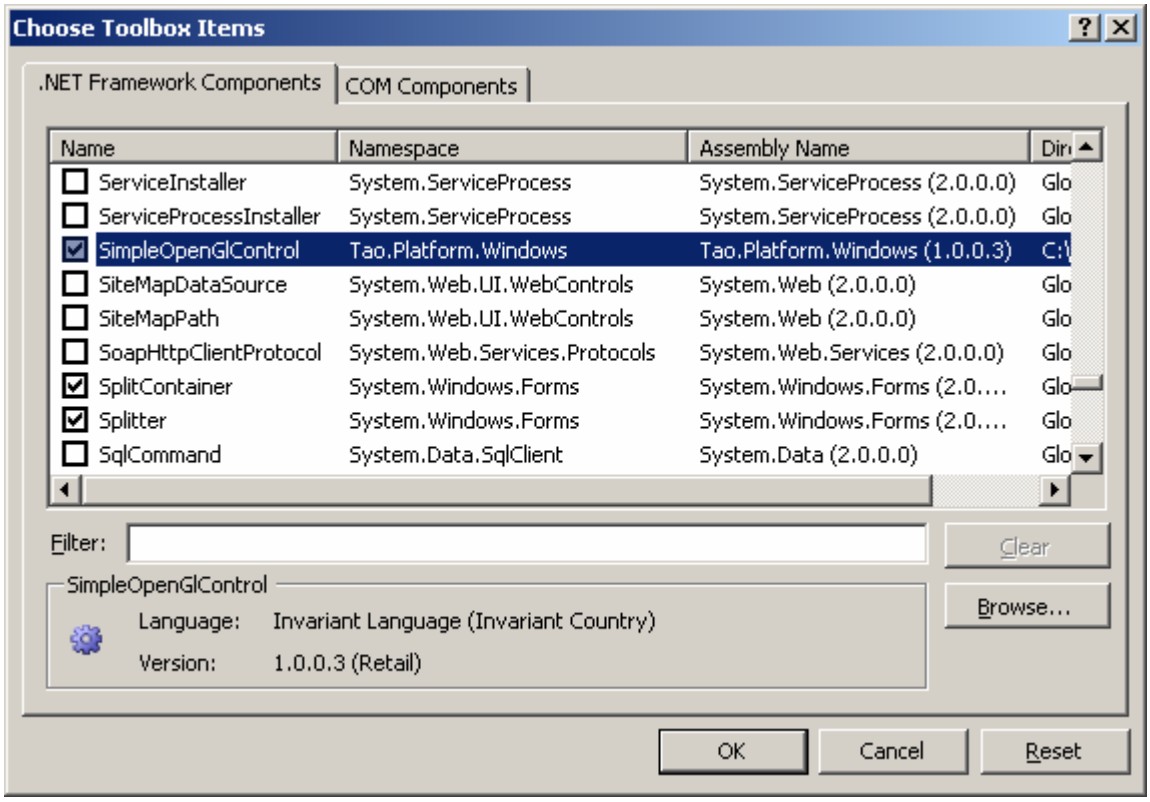
Make a folder in the Tutorial1 directory called Dependencies – this is where we'll place our compile time (assemblies that we have to reference to build our project against) and later on our runtime dependencies. Copy into there the Tao DLLs we're going to be using (Tao.OpenGl.dll, and Tao.Platform.Windows.dll). Note that you don't have to place your references into a folder such as this, but it's useful for cases such as checking in the correct version of all dependencies into a source control system.

Open `MainOpenGlForm` in design mode (right click on MainOpenGlForm.cs in the Solution Explorer, and select View Designer, or press Shift-F7 when displaying the code).

Now open up the Toolbox, under the General group right click on the Toolbox and select Choose Items…

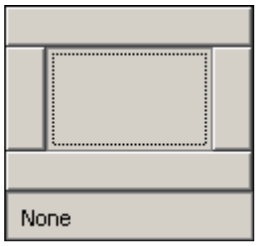
In the Choose Toolbox Items dialog, click the Browse… button and navigate to the Dependencies folder that we just created. Select the Tao.Platform.Windows.Dll.

You'll see that it's identified the SimpleOpenGlControl, and that it's selected:



Click on the control in the Toolbox, and drag it onto the form. This also adds a reference to Tao.Platform.Windows.dll to our project. This assembly, and any others referenced, will be copied from the Dependencies folder into the bin\release subfolder (for a release build) under the Taumuon.Jabuka.MainApp project folder (this is controlled by the Copy Local property, the properties are accessed by right clicking on the assembly in the References section of the Solution Explorer).

We'll change some properties on the control so that it fills the entire form. Right click on the control in the designer, and select Properties. Under the Appearance category, select the dropdown in the Dock property and click the middle of the 5 bars displayed to get the control to Fill the area:



If you try and run the application now, you'll get an error:

This is because the control needs to be initialised before it is used. We'll go ahead and do that now.

In the MainOpenGlForm.cs file, the constructor calls `InitializeComponent()`, the definition of which lives in the MainOpenGlForm.Designer.cs file. This method creates any user controls, sets the properties on the Form and those controls, and adds the controls to the Form's `Controls` collection.

We want to initialise the `SimpleOpenGlControl` after this initialisation is done. In MainOpenGlForm's constructor, following the call to `InitializeComponent()` add the following:

```
this.simpleOpenGlControl1.InitializeContexts();
```

The `this` keyword isn't really necessary in this case, in general it allows us to identify member variables if any local variables have the same name. In this case though, using the keyword allows Visual Studio's intellisense to pop up so to save some typing. The member `simpleOpenGlControl1` was added to the form when we dragged it onto the control – this can be found at the bottom of the MainOpenGlForm.Designer.cs file. The contents of MaiOpenGlForm.cs are shown below:

```
using System;
using System.Windows.Forms;

namespace Taumuon.Jabuka.MainApp
{
    public partial class MainOpenGlForm : Form
    {
        /// <summary>
        /// The Application's main form.
        /// </summary>
        public MainOpenGlForm()
        {
            InitializeComponent();
            this.simpleOpenGlControl1.InitializeContexts();
        }
    }
}
```

While we're at it, we'll go ahead and change the title bar of the form. The easiest way to do this is to select the drop down at the top of the Properties selector. Change the drop down to MainOpenGlForm and change the Text property to Tutorial 1.

You'll now be able to run the application without any errors, but it isn't behaving totally correctly – you can see that when running, the contents of the underlying window are displayed. We want to get it to redraw its contents when notified by Windows (this happens when, for example, the contents are temporarily obscured by another window moving over ours. When ours is displayed again Windows sends us an event allowing us to repaint our contents).

We'll go ahead and handle the Form's `Paint` event. Add a reference to Tao.OpenGl.dll to our project (right click on References in the Solution Explorer, and select Add Reference… select the Browse tab and navigate to the Dependencies folder that we created earlier. Select Tao.OpenGl.dll and click OK).

Add a `using` statement to the top of MainOpenGlForm.cs:

```
using Tao.OpenGl;
```

Now, we'll override the Form's `OnPaint()` method, that handles the painting of the form, and in that method we'll just get the control to clear itself.

Add the following method into the `MainOpenGlForm` class, following the constructor:

```
/// <summary>
/// Handles painting of the form.
/// </summary>
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    // Clear Screen And Depth Buffer
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT |
        Gl.GL_DEPTH_BUFFER_BIT);
}
```

When overriding a control's method, we generally want the base control to do any work it wants to do before we go and do ours, in this case we want to call the `OnPaint()` method on the `Form` class that `MainOpenGlForm` derives from to allow it to fire any paint events. This is what the `base.OnPaint(e)` call does.

The Tao framework provides access to the OpenGL API; methods are provided as static methods on the `Gl` class, along with constants corresponding to those in the API. I won't go into too much detail about the `glClear()` method, as you could take a look to look at the opengl.org website, or another reference. But what `glClear()` basically does is to clear the painting area. The | operator is a bitwise or, and just combines the values of the bits stored in the color and depth bit, e.g. if the color bit had a value of 010 (i.e. 2), and the depth bit had a value of 001 (i.e. 1), then by or-ing the values we'd get a value of 011 (i.e. 3).

Now, when running the application we get a form with a totally black drawing surface. This shows that the painting is working correctly, even if all we're doing is clearing the drawing surface.

There is one last thing to do before we can start to actually draw any objects. We'll set up various OpenGL settings. Again, an OpenGL reference book should describe what these do in detail. Insert the following after the existing code in `MainOpenGlForm`'s constructor:

```
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

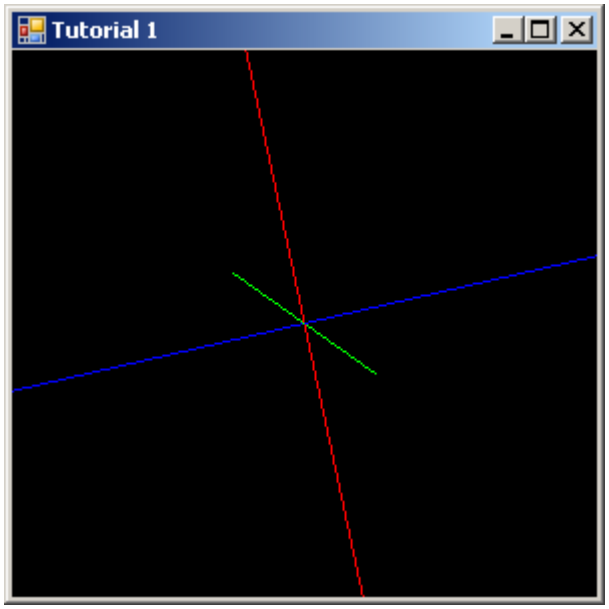
Now, let's go ahead and do some drawing. We will draw three perpendicular axes to the screen (so that you can see where the x, y and z axes lie), we will redraw the scene each time Windows informs our Form that it needs to be repainted. Let's add the following method to our `MainOpenGlForm` class, called `RenderScene()`, and call it in the `OnPaint()` method following the calling of the base class `OnPaint()` method. Remove the call to clear the buffers from the `OnPaint()` method as we'll do it as part of rendering the scene.

The following is our first taste of proper OpenGL. We first clear the scene, and then we store our current viewing transformation (this will become clearer later on when we start performing different transformations to more than one object).

```
/// <summary>
/// Draws a frame.
/// </summary>
private void RenderScene()
{
    // Clear Screen And Depth Buffer
    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();
    #endregion Draw Axes

    Gl.glPopMatrix();
    Gl.glFlush();
}

  /// <summary>
  /// Handles painting of the form.
  /// </summary>
```

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    RenderScene();
}
```

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 of different rotations. Note that when coding in OpenGL it's common practice to indent the code between the `glBegin()` and `glEnd()` methods.



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.

The code to do this was obtained from the OpenGL SuperBible book. We're going to override the Form's `OnResize()` method and change the OpenGL viewport to match the size of the screen. We also change the perspective view. A good OpenGL reference will explain this much better.

Now we add a private method to set the view:

```
private void SetView(int height, int width)
{
    // 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();
    }
```

We call `SetView()` in two places, first we override the `OnResize()` method as follows:

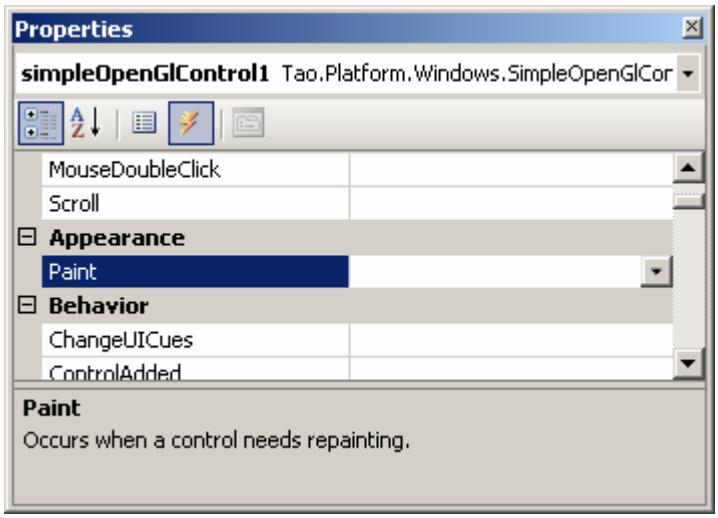
```
/// <summary>
/// Handles the resizing of the form.
/// </summary>
protected override void OnResize(EventArgs e)
{
    base.OnResize(e);
    SetView(this.Height, this.Width);
}
```

And we also call the method from the form's constructor:

```
/// <summary>
/// The Application's main form.
/// </summary>
public MainOpenGlForm()
{
    ... Our existing code is here!

    SetView(this.Height, this.Width);
}
```

Resize the window and notice that it turns black. What's happening here is that the `SimpleOpenGlControl` has its `ResizeRedraw` property to true, so it's attempting to draw itself automatically when it's resized, but its `Paint` event isn't being handled. The `SimpleOpenGlControl` that covers the surface of the form doesn't call any code to repaint its surface. The form isn't drawing itself as we haven't set its `ResizeRedraw` property. We could set this property, or call `Invalidate()` in the `OnPaint()` method, but the most elegant solution is to actually do the drawing in response to the `Control`'s, rather than the `Form`'s, `Paint` event.

In the designer, select the SimpleOpenGlControl, and select the event section of the Properties window. Double click on the Paint event to add a handler:




```
private void simpleOpenGlControl1_Paint(object sender,
                                        PaintEventArgs e)
{
    RenderScene();
}
```

Resize the window and you should see that the view is updated correctly. We can now remove `MainOpenGlForm`'s `OnPaint()` method.

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, it makes use of the OpenGL Utility Library (GLU). 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. The general case for `try-catch-finally` is as follows

- An exception is thrown in the `try` block, execution jumps straight into the `catch` block:
  - Either the `catch` block executes without an exception thrown (including the original being rethrown) after which the `finally` block is executed, then program execution continues in the code following the `try`-catch-finally block.
  - Or the `catch` block throws, so execution jumps straight into the `finally` block, after which the exception is thrown up to the caller.
- No exception is thrown, the `try` block completes followed by the `finally` block.

In our case, we haven't included the `catch` block as it would be empty – we want any exceptions to propagate up to the caller (we're not currently handling these at the application level, but we may move on to doing that).

Later on we may profile the code to see how inefficient creating and destroying the quadric every time that we render the scene is; it may be that it would be better to create a wrapper class that called `gluNewQuadric()` in its constructor and stored the `GLUquadric` in a member. If this class supported `IDisposable` to call `gluDeleteQuadric()` in its destructor then we could make use of the `using` statement rather than the `try-finally`. We will do this later in this tutorial.

Also, in the code above note that we have made use of the ternary operator (`? :`) when setting the `drawStyle` variable. This is just a shorthand way of writing:

```
int drawStyle;
if (drawWireFrame)
{
    drawStyle = Glu.GLU_LINE;
}
else
{
    drawStyle = Glu.GLU_FILL;
}
```

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 contained within one object; even though we do have some objects around (the `MainOpenGlForm` instance and the `SimpleOpenGlControl`) there's not much difference over a C program as all of the drawing functionality lives in the Form, and isn't factored into separate objects.

We want to make the example a bit more object-oriented, giving a firm foundation for doing further work. That's the essence of refactoring, to change the structure of existing code without changing its functionality. Although, refactoring typically goes hand-in-hand with adding new functionality, it's good practice while writing code to keep it well structured before it degenerates into something too complex to maintain. It's much easier to keep the code simple and easy to understand and maintain through small refactorings, than to salvage a hideous mess which is fragile to changes. We're going to go ahead and make some structural changes now.

There isn't that much code in MainOpenGlForm.cs, so I'll paste it here in its entirety prior to our refactoring:

```
using System;
using System.Windows.Forms;

using Tao.OpenGl;

namespace Taumuon.Jabuka.MainApp
{
    public partial class MainOpenGlForm : Form
    {
        /// <summary>
        /// The Application's main form.
        /// </summary>
        public MainOpenGlForm()
        {
            InitializeComponent();
            this.simpleOpenGlControl1.InitializeContexts();

            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);

            SetView(this.Height, this.Width);
        }

        /// <summary>
        /// Handles the resizing of the form.
        /// </summary>
        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
            SetView(this.Height, this.Width);
        }

        /// <summary>
        /// Draws a frame.
        /// </summary>
        private void RenderScene()
        {
            // Clear Screen And Depth Buffer
            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();
            #endregion Draw Axes

            #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

            Gl.glPopMatrix();
            Gl.glFlush();
        }

        private void SetView(int height, int width)
        {
            // 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();
        }

        private void simpleOpenGlControl1_Paint(object sender,
                                        PaintEventArgs e)
        {
            RenderScene();
        }
    }
}
```

Now, add a class to our project called `World`. Do this by right clicking on the project and selecting Add | Class, name the file that will be created 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>
    /// Holds and draws various objects.
    /// </summary>
    public class World
```

Add a member variable to the `MainOpenGlForm` class, as follows:

```
        #region Member Variables

        private World world = new World();

        #endregion
```

This creates a new instance of the `World` class, and assigns it our private member variable.

Add the following `using` statement to the top of the World.cs file:

```
using Tao.OpenGl;
```

Move the `SetView()` and `RenderScene()` methods from the `MainOpenGlForm` class into the `World` class, changing the access specifier to `internal` rather than `private`. The methods can't have the `private` specifier, as they would only be able to be called from other methods inside the `World` class. Being `internal` means that the methods can be called from any other types that live in the same assembly.

Change the form's constructor, and `OnResize()` methods to call these methods on our world instance:

```
        /// <summary>
        /// The Application's main form.
        /// </summary>
        public MainOpenGlForm()
        {
            ...
            world.SetView(this.Height, this.Width);
```

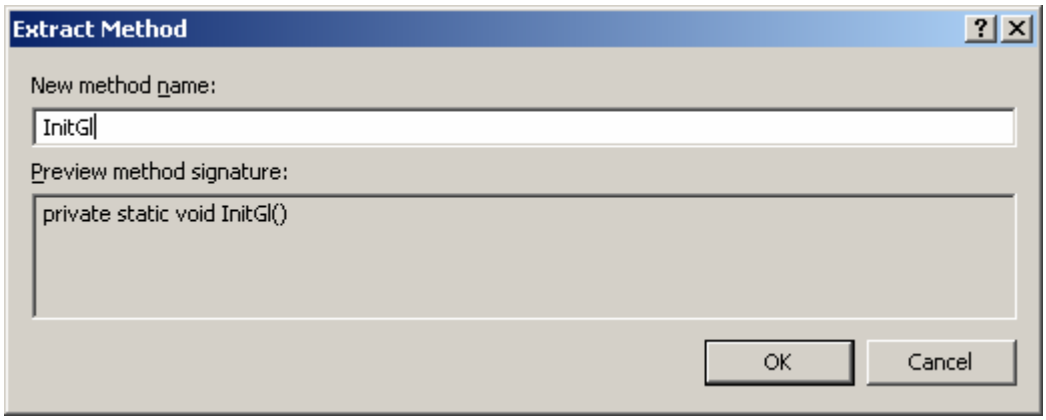
```
        }

        /// <summary>
        /// Handles the resizing of the form.
        /// </summary>
        protected override void OnResize(EventArgs e)
        {
            base.OnResize(e);
            world.SetView(this.Height, this.Width);
        }
```

The `MainOpenGlForm` class is now much cleaner. Now, it's only doing one job, handling the Form's event and routing the calls onto the `world` instance, the responsibility for rendering the scene has moved to the `World` class.

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:




```
        /// <summary>
        /// The Application's main form.
        /// </summary>
        public MainOpenGlForm()
        {
            InitializeComponent();
            this.simpleOpenGlControl1.InitializeContexts();

            InitGl();
            world.SetView(this.Height, this.Width);
        }

        private static void InitGl()
        {
            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);
        }
```

Now, move this method to the `World` class, make it internal and non-static, and in the `MainOpenGlForm` constructor call `InitGl` on the `world` instance, similar to what we just did above.

In my mind I've already got some more possible refactorings, 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 three axes along the scene's 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 three axes' y location. At the top of the `World` class, insert the following variables:

```
#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 our three axes, in the start of the `Draw Axes` region, insert the following:

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

And following the drawing of our three axes, 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()`.

And similarly for the drawing of the sphere, we add the same as the above with the following:

```
Gl.glTranslated(3.0, 3.0, 3.0 + sphereZLocation);
```

Run the application and you can see that nothing has changed. What we want now is for the location of the sphere and the three axes to be updated and then redrawn. We'll insert a new method into the `World` class:

```
internal void UpdateLocations()
{
    const double increment = 0.1;
    this.axesYLocation += increment;
    this.sphereZLocation += increment;
}
```

To get the objects to animate, we want the `UpdateLocations()` method to be called periodically. We'll go and add a timer to do this. Select `MainOpenGlForm` in the designer, and

drag a Timer from the Windows Forms section of the Toolbox onto the form. Change the timer's Enabled property to True, and its Interval property to 1.

Double click on the timer to add an event handler for its tick event, and in the handler add a call to update the locations of the objects, and to refresh the form. The `Refresh()` method gets the form to immediately redraw itself, and also the controls it owns (so the `SimpleOpenGlControl`'s `Paint` event gets fired):

```
private void timer1_Tick(object sender, EventArgs e)
{
    world.UpdateLocations();
    this.Refresh();
}
```

That's it! We've now got smooth animation controlled via the Form's events. 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` class. 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 go ahead and add two classes to the project, `Axes` and `Sphere`. Remove the `using` statements for `System.Collections.Generic` and `System.Text` from the top of the added files.

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

Add a public `Draw()` method to both of our new classes. Into each, copy the drawing code specific to the object from `RenderScene()` including the preceeding `glPopMatrix()`, `glTranslated()`, and `dlPushMatrix()` 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 working program – in the worse case we could scrap our new work if we introduced a bug. This may seem overly cautious for this simple example, but it's a good habit to get into 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, just before the `Draw Axes` region, 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 holding their locations).

Add the following two calls into `World.UpdateLocation()`:

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

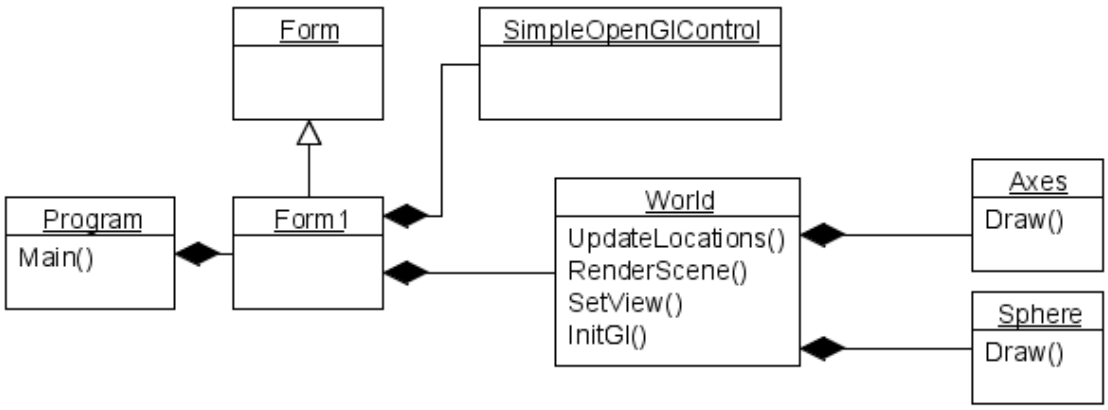
Now, run the program and it appears that there is only one set of objects again – we're successfully updating the positions of our newly-introduced objects to match the old copies, and so the old objects are redundant.

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 `UpdateLocations()` method. Delete the two `Draw` regions 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 diamonds connected to e.g. the `World` class represent composition – the lifetimes of the `Sphere` and `Axes` are controlled by the lifetime of the `World`. As C# is garbage collected, it's less important to know who 'owns' what, here composition means that the `World` and only the `World` class has references to the `Axes` and the `Sphere`, so as soon as the `World` is eligible for garbage collection, then so are they.

Notice that the `World` has two hardcoded members, the `Axes` and the `Sphere`. It has concrete knowledge of these types, and won't be able to draw, for example, cup cakes! 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. The timer isn't shown on here, but as one of `MainOpenGlForm`'s controls, it's lifetime and usage is as the `SimpleOpenGlControl`.



In the `World`'s `UpdateLocations()`, 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 two classes, so that we could, for example, change the internal representation of the locations to be stored in 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:

```
/// <summary>
/// The object updates its location by the given increment.
/// </summary>
public void Update(double increment)
```

The implementation for the method in the `Axes` class is:

```
axesYLocation += increment;
```

The `UpdateLocations()` method now can call:

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

And now the member variables contained in the `Axes` and `Sphere` classes can be made private. Now the `World` class interacts with the `Axes` and `Sphere` though the `Draw()` and `Update()` methods – it knows nothing of their internal representation. At the moment, the `World` class directly calls `Draw()` on a `Sphere` and `Axes` classes. We now want to get rid of this hardcoding; the world shouldn't be concerned with whether it's drawing spheres, or if it's drawing rabbits! All the `World` class wants to do is call `Draw()` and `Update()` on any object that we will store in the `World`.

To do this we'll make use of polymorphism; we want to call the same methods on an object, but it will do something different depending on its type (a sphere's `Draw()` method will draw a sphere to the screen, whereas a box's `Draw()` method will obviously draw a box). C# allows us to use an interface to achieve this. As an aside, in C++ you'd have to derive from an abstract base class rather than using an interface. You could still derive from an abstract class in C#, but it's prefereable to use interfaces as C# doesn't support multiple inheritance.

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.MainApp
{
    /// <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 at some point in the future 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 (obviously, there's no need to implement the interface as the required methods already exist in the classes):

```
    /// <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>();
```

Note that we're making use of generics here, so won't work with verions of .NET earlier than 2.0. However, it's not too difficult to change the type to an `ArrayList` if you're stuck with an older version of the compiler.

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 `UpdateLocations()` 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(increment)`. This is really tidy, the `World` class is much smaller, and its responsibilities have been delegated to the new classes that we've introduced.

The `World` class still isn't totally independent of the `Axes` and `Sphere` classes, as we've still got the hardcoded `Axes` and `Sphere` classes in the `World` constructor, but it's an easy job to construct this elsewhere.

We'll actually construct these in the `Program` class and pass them into the `World` class constructor, and then pass the `World` class into the form's constructor. We could construct these in the Form directly, but the Form should only be concerned with the window's event management. The definitions of the objects could be e.g. read from an XML file or retrieved from a database, so it makes sense to keep this functionality out of the `World` class. We'll go ahead and make the change for the `Program` class to create our objects.

Change the declaration of the member variable in `MainOpenGlForm` to be:

```
private World world = null;
```

And change the constructor to be:

```
/// <summary>
/// The Application's main form.
/// </summary>
public MainOpenGlForm(World world)
{
    if (null == world)
    {
        throw new ArgumentNullException("world");
    }

    this.world = world;

    InitializeComponent();
    this.simpleOpenGlControl1.InitializeContexts();

    world.InitGl();
    world.SetView(this.Height, this.Width);
}
```

And to get this to compile, change the `Application.Run()` call in the `Program` class's `Main()` method to be:

```
Application.Run(new MainOpenGlForm(new World()));
```

Now, everything is still running without any errors. Note that we throw an exception if null is passed into `MainOpenGlForm`'s constructor. We'll discuss the strategies (assertions versus

exceptions) for checking that a method's parameters are valid in another tutorial. Note that we set the world variable before the `InitializeComponent()` call, as that may cause `OnResize()` to be fired, which relies on the world member.

We can now move the creation of the drawable objects to be in the `Program` class. Change the member variable declaration in `World` to be:

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

And the `World`'s constructor to be:

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="drawableObjects">
/// List of objects that support OpenGl drawing.
/// </param>
public World(List<IDrawable> drawableObjects)
{
    if (null == drawableObjects)
    {
        throw new ArgumentNullException("drawableObjects");
    }
    this.drawableObjects = drawableObjects;
}
```

Now, add a `using` directive for `System.Collections.Generic` to the `Program` class, and change the class's `Main()` method to be:

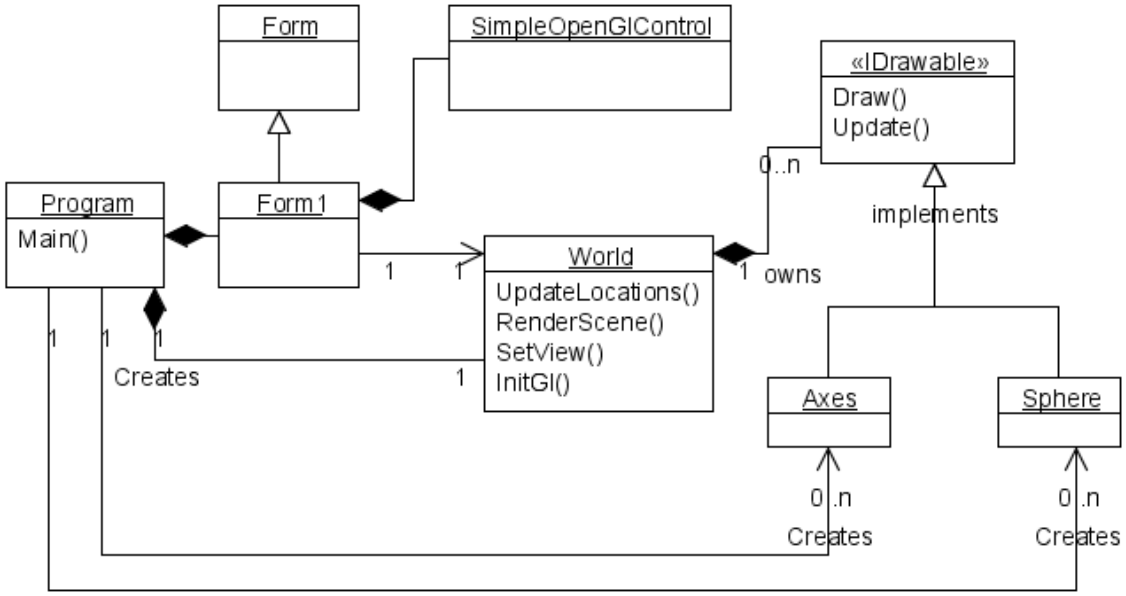
```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Axes axes = new Axes();
    Sphere sphere = new Sphere();
    List<IDrawable> drawableObjects = new List<IDrawable>();
    drawableObjects.Add(axes);
    drawableObjects.Add(sphere);
    World world = new World(drawableObjects);

    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainOpenGlForm(world));
}
```

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

```
/// <summary>
/// Holds a list of IDrawable objects, and draws them using
///     OpenGL calls.
/// Usage:
///   Construct with a list of IDrawable objects,
///   Call InitGl before making any calls.
///   Call SetView before doing any drawing, and when the
///       size of the viewport changes.
///   Call RenderScene to draw the objects to the screen.
///   Call UpdateLocations for the objects to update themselves.
```

```
    /// </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 lifetime of the `World` is directly dependent, and the lifetimes of the `Axes` and `Sphere` are indirectly dependent on the lifetime of the `Program`, but the composition shows that the lifetimes of the `Axes` and `Sphere` are directly linked to the lifetime of the `World`. The program creates the `Axes` and `Sphere`, hence the association, but it doesn't keep a reference to these objects, but the composition more shows us the intent of the relationship, it's possible that the `Program` could destroy the `World`, and the objects that it owns, and create a new one (for instance, when loading a new level in a game), and as the World is the only class to maintain references to the `Axes` and `Sphere`, I've represented the relationship as composition.

Using composition isn't really that useful to show when in a garbage collected environment, and I've not used any aggregation in the diagram, this is more of a philosophical/design difference: 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. We'll not do any more design or refactoring in this tutorial, but will perform some tidying up.

Checking For Errors

We'll make some changes to the code to get it to check for any OpenGL errors that occur. We'll register a callback for any errors that occur from the gluQuadric. The Tao.OpenGl assembly has a definition of a delegate that we can instantiate to be called if any errors occur.

Add the following bold lines to the previous sphere drawing code:

```
// From NeHe lesson 18.
Glu.GLUquadric quadric = Glu.gluNewQuadric();
try
{
    Glu.gluQuadricDrawStyle(quadric, drawStyle);
    Glu.gluQuadricCallback(quadric, Glu.GLU_ERROR, new
        Glu.QuadricErrorCallback(this.QuadricErrorCallback));
    Glu.gluQuadricNormals(quadric, Glu.GLU_SMOOTH);
    Glu.gluSphere(quadric, radius, 40, 40);
}
finally
{
    Glu.gluQuadricCallback(quadric, Glu.GLU_ERROR, null);
    Glu.gluDeleteQuadric(quadric);
}
```

Add the implementation of the callback method. We'll simply throw an exception in case of any errors that occur:

```
private void QuadricErrorCallback(int error)
{
    throw new ApplicationException("An error has occurred: "
        + error.ToString());
}
```

Similarly, at the end of the `World`'s `RenderScene()` method add the following:

```
int error = Gl.glGetError();
if (error != 0)
{
    throw new ApplicationException(
        "An error has occurred: " + error.ToString());
}
```

This error checking in place should give us some reassurance, but I was unable to get any of these methods to fire in practice. As is said in OpenGL Super Bible, an invalid call to an OpenGL command results in that command being silently ignored.

I changed the `Sphere`'s drawing code to delete the quadric immediately after allocating it, prior to the methods to set the normals and `gluSphere`, and there was still no error reported. However, running long enough resulted in a `FatalExecutionEngineException` being thrown. Similarly, changing the drawing code to allocate a million `GLUquadrics` without deleting them failed to result in any exceptions. Windows Vista ran and kept increasing the page file size until the OS became totally unresponsive.

Displaying the Frame Rate

We'll now add in a simple calculation of the frame rate, and will display it in the window's title bar.

Add a member variable to the `MainOpenGlForm` to hold the original form title, as we're going to append the frames per second to the form title:

```
private readonly string originalTitle;
```

Set this in the constructor, after the `InitializeComponent()` call, to ensure that the title has been set on the form from the property we set earlier in the designer:

```
            this.originalTitle = this.Text;
```

To calculate the frame count we need to know how many frames were displayed over a given time period. We set a time over which we want the frame count to be updated, say 5 seconds (5000 milliseconds).

Add the following member variables to the Form:

```
        private const int updateRateMilliseconds = 5000;
        private System.Diagnostics.Stopwatch stopwatch
            = new System.Diagnostics.Stopwatch();
        private int frameCount = 0;
```

At the end of the Form constructor start the stopwatch:

```
            stopwatch.Start();
```

Add a method to the form that will call the frame rate, and add a call to this method in the `SimpleOpenGlControl`'s `Paint` event handler:

```
        private void UpdateFrameRate()
        {
            frameCount++;
            if (stopwatch.ElapsedMilliseconds >
                updateRateMilliseconds)
            {
                long elapsedMilliseconds =
                    stopwatch.ElapsedMilliseconds;
                double framesPerSecond = frameCount * 1000.0
                    / elapsedMilliseconds;
                frameCount = 0;
                this.Text = string.Format("{0} ({1})",
                    this.originalTitle, framesPerSecond.ToString(
                    "F2", Application.CurrentCulture));
                stopwatch.Reset();
                stopwatch.Start();
            }
        }
```

IDisposable and Finalizers

We're now going to look at the performance benefit of having the Sphere's `GLUquadric` made into a member variable, rather than creating one in every `Draw()` call, and the changes that forces us to make to the code. We're going to look at performance properly later, we should create an easy to maintain application to profile before optimising, but the concepts behind disposing and finalizing are fundamental to .NET, so it makes sense to look at them now. First though, I'll describe a simple scene to measure the performance benefits.

Creating a Complex Scene

A later tutorial will introduce the concept of a `Scene`. For now, I'll describe the changes you can make to the tutorial to draw a large number of spheres, so that you can see the performance benefit of having the `GLUquadric` as a member.

You'll want to add two more x and y variables to go alongside the `sphereZLocation` variable. The sphere constructor should take 3 variables which will be used to set these members. The sphere's `Update()` method should have its body removed.

Before the sphere drawing code we set the sphere's colour given its location, and we change the `gluSphere()` call so that it draws with a smaller number of slices and stacks:

```
Gl.glTranslated(sphereXLocation, sphereYLocation,
    sphereZLocation);
double radius = 2.0;

float colorX = (200.0f - (float)sphereXLocation) / 200.0f;
float colorY = (200.0f - (float)sphereYLocation) / 200.0f;
float colorZ = (200.0f - (float)sphereZLocation) / 200.0f;
Gl.glColor3f(colorX, colorY, colorZ);
Glu.gluSphere(this.gluQuadric, radius, 4, 4);
```

The `Program` class's `Main()` method will create a million spheres:

```
for (int x = -100; x <= 100; x += 10)
{
    for (int y = -100; y <= 100; y += 10)
    {
        for (int z = -100; z <= 100; z += 10)
        {
            drawableObjects.Add(new Sphere(x, y, z));
        }
    }
}
```

In `World.cs` we can add a member to hold the current rotation, and add code into the `RenderScene()` method to ensure that the world rotates on every call:

```
viewAngle += 0.8f;
viewAngle = (viewAngle > 360.0f) ? viewAngle - 360.0f
    : viewAngle;
Gl.glRotatef(viewAngle, 1.0f, 0.2f, 0.0f);
```

Now running this, it's not very clear to see, but there are a million rotating spheres of different colours drawn on the screen. I obtained a frame rate of approximately 0.9 frames per second whilst running the application.

Making the GLUquadric a Member

The changes we make from now on are changes that will make their way into the final code. A `GLUquadric` member should be added to the sphere, and this should be set in the constructor:

```
public Sphere()
{
    bool drawWireFrame = false;
    int drawStyle = drawWireFrame ? Glu.GLU_LINE :
                                        Glu.GLU_FILL;
    this.gluQuadric = Glu.gluNewQuadric();
    Glu.gluQuadricDrawStyle(this.gluQuadric, drawStyle);
    Glu.gluQuadricCallback(this.gluQuadric, Glu.GLU_ERROR, new
        Glu.QuadricErrorCallback(this.QuadricErrorCallback));
    Glu.gluQuadricNormals(this.gluQuadric, Glu.GLU_SMOOTH);
```

```
        }
```

The `Draw()` method now uses this member variable to draw the sphere:

```
            Glu.gluSphere(this.gluQuadric, radius, 40, 40);

            //Glu.gluQuadricCallback(quadric, Glu.GLU_ERROR, null);
            //Glu.gluDeleteQuadric(quadric);
```

Now, running the program, I get double the frame rate at 1.8 frames per second. Note that we're not currently deleting our `gluQuadric`, and will leak memory. We need to ensure that the quadric gets cleared up. Note that even though we are doing this for our quadric, we will have to follow the same pattern of code for all unmanaged resources (for example, any display lists allocated).

Adding a Finalizer

The garbage collector, when it runs a garbage collection, determines which objects are no longer referenced and frees their managed memory. However, any unmanaged resources are not freed automatically, and will be leaked. The .NET framework does provide a mechanism for us to use to ensure that the memory is freed; the finalizer. On garbage collection, any objects with a finalizer are moved to a separate list, and on the next garbage collection the finalizer methods of these objects are called, to ensure that they can release any resources (i.e. it takes two garbage collections for the finalizers to run).

The syntax for a finalizer method in C# is the same as a C++ destructor, i.e. the method name preceeded by a tilde. We add one to our sphere as follows, and in that method we free our unmanaged resources:

```
        ~Sphere()
        {
            Glu.gluQuadricCallback(this.gluQuadric, Glu.GLU_ERROR, null);
            Glu.gluDeleteQuadric(this.gluQuadric);
        }
```

Now, the resources will be cleaned up once the sphere instances are no longer in use, in this case this will be when the program closes. However, it is not good practice to rely on finalization, we should clean up the resources as soon as they're not in use. Relying on finalizers we'd have to wait for two garbage collections for cleanup (as a garbage collection is only scheduled when memory is running low, it may be a long time between runs and in the meantime scarce unmanaged resources would not be reclaimed).

IDisposable

There is a standard method that is used in the framework, `IDisposable`, and a pattern of use. The interface is used throughout the framework, and is recognised in the `using` statement to clean up resources in case of exceptions (to avoid the `try-finally` pattern).

We ensure that our sphere implements the interface:

```
class Sphere : IDrawable, IDisposable
```

The dispose pattern for our sphere is as follows:

```
        public void Dispose()
        {
```

```
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        ~Sphere()
        {
            this.Dispose(false);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Dispose of any managed resources.
            }

            Glu.gluQuadricCallback(this.gluQuadric, Glu.GLU_ERROR, null);
            Glu.gluDeleteQuadric(this.gluQuadric);
        }
```

The `Dispose()` overload that takes no parameters is the implementation of the method provided in the interface. A virtual method is provided to allow derived classes to clean up any of their resources. The `Dispose()` method calls `SuppressFinalize()` to remove the object from the finalization queue, which ensures that its finalizer is not called.

The `Dispose(bool)` overload is called with a flag so we know whether the `Dispose()` method has been called, or whether the object is being finalized. On finalization, there are no guarantees on which order objects are being cleaned up, so only unmanaged resource should be freed in that case. Calling `Dispose()` multiple times should not result in any bad behaviour, but we will introduce changes to take this into account after some further cleanup.

Disposing of the Spheres

Before going on to optimise the use and cleanup of the GLUquadric, we need to ensure that the `Dispose()` method is called on all of our spheres. The `MainOpenGlForm` already has a `Dispose(bool)` method, in the MainOpenGlForm.Designer.cs file. Prior to calling the base class `Dispose()` method add the following:

```
            this.world.Dispose();
```

The world does not yet implement `IDisposable`, but it needs to be added to allow the form to get the world to clean up the resources that it owns. This shows the knock on effects of owning an `IDisposable` object; each member that owns one should itself implement `IDisposable`. Here's the implementation of the `World`'s dispose pattern:

```
        public void Dispose()
        {
            this.Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Dispose of any managed resources.
                foreach (IDrawable iDrawable in
                    this.drawableObjects)
```

```
            {
                if (iDrawable is IDisposable)
                {
                    ((IDisposable)iDrawable).Dispose();
                }

            }

        }
```

Creating a GluQuadricWrapper

Currently, our `Sphere` code has a finalizer to ensure that the GLUquadric is cleaned up in the case of users forgetting to dispose of it. It is best to contain all unmanaged objects in as thin a wrapper as possible. Making an object finalizable means that it is put on a list of finalizable objects which has to be walked on each garbage collection.

Additionally, in the case of the finalizer being called, any objects that the Sphere references would not be removed until finalization, increasing their lifetime. Any managed references that the Sphere owned would not be freed until finalization if it had a finalizer, whereas if it instead holds a separate finalizable object, then the managed references are cleaned up one garbage collection sooner.

Note that the Tao Framework takes a very purist view to the wrapping of the OpenGL API, and tries as much as possible to stay as close as it can to the API to make it easier for users of OpenGL on another platform to get up to speed. However, a library of these boilerplate wrapper classes would be beneficial.

Creating the Wrapper

We're going to create a wrapper class to manage the lifetime of our `gluQuadric`. Add a class to our project called `GluQuadricWrapper` and add a using statement for the Tao.OpenGl namespace. We will add a member `gluQuadric` that will be allocated in the constructor as is currently done in the Sphere, and implement the dispose pattern as above.

Now the lifetime of the `gluQuadric` is properly managed, but it's not much use yet – it's not actually available for us to use. The `gluSphere()` method takes a `gluQuadric`, and we have a `gluQuadric` wrapper that we want to supply to the method. We could simply add a property to the class to expose the `gluQuadric`, but it's better to encapsulate this away behind the `GluQuadricWrapper`'s interface; we don't want users to delete or otherwise corrupt our `gluQuadric` instance. Instead, we need to add methods onto the wrapper that external clients can call that in turn calls `gluSphere`. Add the following method to the wrapper:

```
        public void DrawSphere()
        {
            double radius = 5.0;

            // From NeHe lesson 18.
            Glu.gluSphere(this.gluQuadric, radius, 40, 40);
        }
```

Now we can replace all `gluQuadric` member in the `Sphere` with a `GluQuadricWrapper`, and call `DrawSphere()` from the `Sphere`'s `Draw()` method. We can remove the finalizer, `~Sphere()`, from the `Sphere` class now, but we still need the `IDisposable` implementation and we also keep the protected `Dispose(bool)` method. This means that a class that derives

from `Sphere` could add its own finalizer if it manages the lifetime of any of its own unmanaged resources. Here is the sphere's new dispose pattern:

```
public void Dispose()
{
    this.Dispose(true);
    GC.SuppressFinalize(this);
}

//~Sphere()
//{
//    this.Dispose(false);
//}

protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // Dispose of any managed resources.
        gluQuadricWrapper.Dispose();
    }
}
```

Guarding Against Multiple Dispose

Now, we want to ensure that it's safe to `Dispose()` of our `Sphere` and `GluQuadricWrapper` multiple times, and that trying to use a disposed object results in an exception.

We add a flag, `isDisposed`, to our `GluQuadricWrapper` and initially set it to false. We set it to true in our `Dispose(bool)` method:

```
protected virtual void Dispose(bool disposing)
{
    if (!isDisposed)
    {
        if (disposing)
        {
            // Dispose of any managed resources.
        }

        Glu.gluQuadricCallback(this.gluQuadric, Glu.GLU_ERROR,
            null);
        Glu.gluDeleteQuadric(this.gluQuadric);

        isDisposed = true;
    }
```

Now, we need to ensure that we can't use our `GluQuadricWrapper` after it has been disposed, we can do so by checking the flag in our `DrawSphere()` call and throwing an exception if the flag has been set:

```
public void DrawSphere()
{
    if (isDisposed)
    {
        throw new ObjectDisposedException("GluQuadricWrapper");
    }
    double radius = 5.0;

    // From NeHe lesson 18.
```

```
            Glu.gluSphere(this.gluQuadric, radius, 40, 40);
        }
```

Remaining Niggles

The `GluQuadricWrapper` is adequate for our purposes, but it can be made more general purpose. For a start, the `DrawSphere()` should take the parameters that are passed into gluSphere. The class should have methods to draw cylinders and discs, i.e. `DrawCylinder()` and `DrawDisc()`, and those methods should take the necessary parameters. Our constructor should also take the drawing style, the orientation, and we should also pass in any textures, if necessary.

Alternative to GluQuadricWrapper

Although the Tao Framework's goal is to maintain a purist approach to the OpenGL API, one way it could achieve the same interface but properly cleanup resources would be for there to be a change to the Tao Framework source, for it to return a `SafeHandle` derived class rather than the `GLUquadric` struct. Internally, the `GLUquadric` struct only has one member, an `IntPtr`, and this is populated from the unmanaged methods (as a struct is exposed we have some typesafety over using an `IntPtr`). However, having a class derived from `SafeHandle` would mean that we could use all of the same methods but we'd have the safety of knowing that we could dispose of the `SafeHandle`, and that it would finalize if a user forgot to.

The `SafeHandle` is quite heavyweight though – it ensures that our object is cleaned up even, for example, during AppDomain unloads and guards against security attacks, so it may be too heavyweight to use over our simple finalizable `GluQuadricWrapper`.

Another Optimisation

It is not a good idea to over optimise our app; we should favour readability and maintainability over obfuscating our application for performance. However, it is OK to optimise where performance is critical, and it's best to find those cases by profiling the application.

For a general wrapper such as the `GluQuadricWrapper`, we may not know all of the usage scenarios, and may wish to keep the number of runtime checks to a minimum which is in the same spirit as OpenGL. In this vein, it may be that we don't want to incur the cost of checking a boolean value on every `DrawSphere()` call (in real life scenarios it's unlikely that this is going to hurt performance much, if at all).

As said in the previous section, nulling out the `IntPtr` would result in an `AccessViolationException` being thrown, which is better than having a `FatalExecutionEngineException` thrown at an interderminate time, but not as informative as having an object disposed exception. The `IntPtr` is hidden inside the struct, but we could null out the struct with the following (after the `gluDeleteQuadric()` call):

```
int size = Marshal.SizeOf(typeof(IntPtr));
IntPtr buffer = Marshal.AllocHGlobal(size);
try
{
    // Null out buffer:
    Marshal.Copy(new byte[size], 0, buffer, size);
    this.gluQuadric =
        (Glu.GLUquadric)Marshal.PtrToStructure(buffer,
        typeof(Glu.GLUquadric));
}
```

```
        finally
        {
            Marshal.FreeHGlobal(buffer);
        }
```

Now, we could remove the `isDisposed` check in the `DrawSphere()` method (we'd still need the check in the `Dispose()` method as it's only safe to do this once). We won't actually do this as it is quite hacky, and it's not the best practice to allocate memory in the finalizer. It would be better for the wrapper to be internal to the Tao Framework, then it would be able to directly hold the `IntPtr` rather than a struct (as it would not be concerned with typesafety, all access would be via wrapper methods), and this further illustrates a motivation for having this type of wrapper provided by the Tao Framework.

Conclusion

We've been through quite a lot in this tutorial, from setting up an application to draw in 3D using OpenGL through refactoring to looking at object cleanup in .NET. The dispose pattern can be quite tricky to get correct, and it does have some subtleties. Joe Duffy's blog is an excellent resource on disposing and finalization (http://www.bluebytesoftware.com/blog/2005/04/08/DGUpdateDisposeFinalizationAndResourceManagement.aspx) In tutorial 2, we'll introduce some physics, and expand upon this design to get the physics and 3D coupled together.