

Setting up an Object Framework

Creating a Component Based Object System

Since the XNA Framework uses a game architecture based on components, I think it would be a good idea to make our object system using components as well. An article on this approach is presented in Game Programming Gems 6, and will be implemented here, updated based on working within C# and the XNA framework.

The Basic Object

As is used in almost all game engines and scene graph implementations, our basic object class is going to consist of a position and a rotation. The difference between the normal method and the one we will implement here is that our objects will inherit from a number of interfaces and abstract classes that will extend their functionality in many ways. These will be simple interfaces that we will add to our object types depending on what we would like them to be able to do, such as be rendered or collide. To create our objects we will first need a base Component interface and a Renderable interface:

```
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMComponents {
    public interface IHMComponent { }

    public interface IHMRenderable : IHMComponent {
        void Render(GraphicsDevice myDevice);
    }
}
```

We will add more to the HMComponents namespace later on, but for now we will just leave it as a placeholder and add the complicated stuff when we need it. With our basic groundwork in place, we can put together our Object class:

```
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMObjects {
    public abstract class HMOBJECT : IHMRenderable {
        public Vector3 Position { get; set; }
        public Vector3 Scaling { get; set; }
        public Quaternion Rotation { get; set; }

        public abstract void Render(GraphicsDevice myDevice);
    }
}
```

You can see that we put the class together as an abstract class that just implements the public properties from IHMRenderable as auto-properties. This will keep us from having to repeat them in all of our objects later on.

Tutorial 2 - Setting up an Object Framework

Extending Our Component Interfaces

With the basic object setup in place, we need the components the object class is made up of to actually be used in some way by the engine, so we are going to make a component manager. To start out the `HMComponentManager` will just look like this:

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace HMEngine.HMComponents {
    internal sealed class HMComponentManager : DrawableGameComponent {
        private static readonly Dictionary<string, IHMComponent> masterList =
            new Dictionary<string, IHMComponent>();
        private static readonly Dictionary<string, IHMRenderable> renderable =
            new Dictionary<string, IHMRenderable>();

        internal HMComponentManager(Game game) : base(game) {}

        public static void AddComponent(string name, IHMComponent component) {
            masterList.Add(name, component);

            if(component is IHMRenderable) {
                renderable.Add(name, (IHMRenderable) component);
            }
        }

        public override void Draw(GameTime gameTime) {
            foreach (IHMRenderable render in renderable.Values) {
                render.Render(GraphicsDevice);
            }
        }
    }
}
```

The manager starts out with dictionaries of components and of renderable items indexed by a string name that we use in the Draw call to loop through and renderable items and pass the GraphicsDevice they need to perform their rendering. This method of storing lists of object types and looping through them will be used for all of our component types as we add them.

The Object Manager

To facilitate easy object management later on, we are going to make an object manager that will be a simple wrapper class for the component manager we just created. This is not technically needed to add objects to the engine, but will help with readability and understanding in our demo class.

```
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;

namespace HMEngine.HMObjects {
    public sealed class HMOBJECTManager : GameComponent {
        internal HMOBJECTManager(Game game) : base(game) { }

        public static void AddObject(string name, HMOBJECT obj) { HMComponentManager.AddComponent(name, obj); }
    }
}
```

Adding the two new managers to our engine is just as easy as adding the graphics manager was. We are simply going to add the game component to the Game.Components list in the constructor to `HMGame`.

The order is important here as we will later on want to make sure that the component manager operates after everything else, so make sure to always put it at the end of the list of GameComponents:

```
// In HMGame
using HMEngine.HMComponents;
using HMEngine.HMObjects;

public HMGame() {
    myGraphics = new GraphicsDeviceManager(this) {
        PreferredBackBufferWidth = width,
        PreferredBackBufferHeight = height
    };

    Components.Add(new HMOBJECTMANAGER(this));
    Components.Add(new HMComponentManager(this));

    // Window sizing code
}
```

Making Sure it all Works

To test that the whole system is working. We will add a simple test object to our codebase for now that clears the device with a new color inside its render function. This way we will be able to tell the whole system is being passed through correctly. Here is the sample object and how we add it to the scene in the demo:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMObjects {
    public class HMTTestObject : HMOBJECT {
        public override void Render(GraphicsDevice myDevice) {
            myDevice.Clear(Color.Orange);
        }
    }
}

// And the HMDemo class...
using HMEngine;
using HMEngine.HMObjects;

namespace HMDemo {
    internal static class HMDemo {
        private static readonly HMGame game = new HMGame(800, 600);
        private static readonly HMTTestObject test = new HMTTestObject();

        public static void Main() {
            HMOBJECTMANAGER.AddObject("test", test);
            game.Run();
        }
    }
}
```

As long as you got everything put together basically how I have done it here, you should be able to run the project now and see an orange surface being rendered instead of our default blue background. We will get to adding some objects that can actually be used to put a game together in the next tutorial.