Michael Schuld – mike@thehazymind.com – http://www.thehazymind.com

Rendering Real Models

Putting Together the Basic Model Object

Thanks to the content pipeline, the ease of loading and using meshes in our games is greatly simplified from the old methods. Using our current object class and a couple of the interfaces we have created for the object system, getting a model or two on screen becomes incredibly easy to do. Here is what a basic model class looks like:

```
using HMEngine.HMCameras;
using HMEngine.HMEffects;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMObjects {
    public class HMModel : HMObject {
        private readonly string myAsset;
        private Model myModel;

        public HMModel(string asset) {
            myAsset = asset;
            Scaling = new Vector3(1);
        }

        public override void LoadContent(GraphicsDevice myDevice, ContentManager myLoader) {
            myModel = myLoader.Load<Model>(myAsset);
        }

        public override void Render(GraphicsDevice myDevice, EffectPass pass) {
        }

        public override void UnloadContent() { }
    }
}
```

This basic class will take in a model asset name in the constructor and appropriately load the content the same way all of the other do (thanks to the IHMLoadable interface). The Render function for this one gets a bit complicated because we are using our own shader effects. If you wanted to simply use the default effect for all the models in game for a quick solution, you could do that as well, but I find it much easier to produce quality demos when you have full control over the rendering pipe, so we will be modifying the normal method of drawing our meshes to use our own shader manager. The rendering code itself looks pretty similar to that of the SDK demos, but we leave shader handling to the engine.

Render Method

The rendering this time is a bit more complicated than other objects, but not terrible. We have to loop through each mesh and part of the meshes and set the device texture and update the World matrix based on any bone transformations of the parent mesh. Beyond that it looks like always:

```
public override void Render(GraphicsDevice myDevice, EffectPass pass) {
```

```
    Matrix world =
        HMCameraManager.ActiveCamera.World *
        Matrix.CreateScale(Scaling) *
        Matrix.CreateFromQuaternion(Rotation) *
        Matrix.CreateTranslation(Position);

    foreach (ModelMesh mesh in myModel.Meshes) {
        if (null != HMEffectManager.ActiveShader.Effect.Parameters["World"]) {
            HMEffectManager.ActiveShader.Effect.Parameters["World"].SetValue(world * mesh.ParentBone.Transform);
        }

        // Each mesh is made of parts (grouped by texture, etc.)
        foreach (ModelMeshPart part in mesh.MeshParts) {
            // Change the device settings for each part to be rendered
            myDevice.SetVertexBuffer(part.VertexBuffer);
            myDevice.Indices = part.IndexBuffer;

            // Make sure we use the texture for the current part also
            myDevice.Textures[0] = ((BasicEffect)part.Effect).Texture;

            // Finally draw the actual triangles on the screen
            myDevice.DrawIndexedPrimitives(
                PrimitiveType.TriangleList, 0, 0, part.NumVertices, part.StartIndex, part.PrimitiveCount
            );
        }
    }
}
```

Checking that it Works

The best way to see if everything we added is working is to just stick a model in the scene and look around at it. For starters, we will need to add an existing .x or .fbx model to the Content folders in the demo. Make sure you copy all the textures that go along with the model you use into the same folder as the mesh. They don't need to be added into the actual project folders within XNA Game Studio, but they do need to be in the same folder as the model file so the framework can find and compile them together with the mesh. After you have added the correct files, simply go to the demo class and place a new model object into the scene and set up the shader and scaling you may want; I added a colored sphere:

```
// In HMDemo
private static readonly HMModel model = new HMModel("Content/Models/sphere");

// In HMDemo.Main
model.Shader = "TT";

quad.Position = new Vector3(1, 0, 0);
model.Position = new Vector3(-1, 0, 0);

// Existing Objects
HMObjectManager.AddObject("sphere", model);
```

That's all. What were you expecting, a ton of work? Remember, the guys on the XNA Team are making sure that using the framework is as painless as possible, and as far as I am concerned, being able to set custom shaders on loaded models with that little code is pretty painless. My sphere loaded in simply as a plain model with no lighting at all, but that is because it was created without any materials or textures.

We will go about making our own shader to handle the lighting and such similar to the BasicEffect included in the framework soon to get our hands a little dirty inside the rendering pipeline.