More Rendering Complexity, Effects and Cameras

Putting our Effect Architecture in Place

From here on in we will need effects to draw anything we need onto the screen. Some things like user interface components can be drawn pretty easily with the built in SpriteBatch in XNA, but for the more complicated objects or models we will want to use, an Effect framework is the best place to begin.

Simple, but a Good Starting Point

To start off with the simple ability to just get objects and models up on screen, we will implement a minimal effect system and not worry too much about the details yet. We will need to add a loadable interface that can be called by our component manager to load content in and have our effect class implement it. This will be handled by the component manager in exactly the same manner that our renderable component was:

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

namespace HMEngine.HMComponents {
    public interface IHMLoadable : IHMComponent {
        void LoadContent(GraphicsDevice myDevice, ContentManager myLoader);
        void UnloadContent();
    }
}

// In HMComponentManager
private static readonly Dictionary<string, IHMLoadable> loadable = new Dictionary<string, IHMLoadable>();

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

    if (component is IHMLoadable) {
        loadable.Add(name, (IHMLoadable)component);
    }

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

protected override void LoadContent() {
    foreach (IHMLoadable load in loadable.Values) {
        load.LoadContent(GraphicsDevice, Game.Content);
    }

    base.LoadContent();
}

protected override void UnloadContent() {
    foreach (IHMLoadable unloadcontent in loadable.Values) {
        unloadcontent.UnloadContent();
    }

    base.UnloadContent();
}
```

Tutorial 3 - More Rendering Complexity, Effects and Cameras

With that we can put together the effect class that will automatically have its assets loaded in for us just from implementing the interface:

```csharp
// HMEffect.cs
using HMEngine.HMComponents;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMEffects {
    public abstract class HMEffect : IHMLoadable {
        private readonly string myAsset;

        internal Effect Effect { get; private set; }

        protected HMEffect(string asset) { myAsset = asset; }

        public abstract void SetParameters(IHMRenderable renderable);

        public virtual void LoadContent(GraphicsDevice myDevice, ContentManager myLoader) {
            Effect = myLoader.Load<Effect>(myAsset);
        }

        public abstract void UnloadContent();
    }
}
```

The effect class so far doesn't actually accomplish much except loading the effect file and compiling it to an actual XNA Effect object. Once we know what sorts of parameters our effects will be using, we will add in a line or two to facilitate passing values from the engine to the effect as well. We will also be using this effect class as the base for two other classes, shaders and post processors.

Updating the Objects for Shaders and Adding a Manager

Since we now have a base to create our object shaders from, we need to update the objects to use them. I would normally place this functionality into the object manager, but since not all objects will have the same effect and eventually some may even have many, we will be sticking them into their own global manager and simply calling the effect that needs to be used for each object. The grouping of similar objects in the rendering pipeline will eventually happen automatically (either on the fly in the engine or in the scene editor tool we will build later). For now, we just need to get the shaders working. Here is the object shader class that inherits from our effect class from earlier and the effect manager:

```csharp
// HMShader.cs
namespace HMEngine.HMEffects {
    public sealed class HMShader : HMEffect {
        public HMShader(string asset) : base(asset) { }

        // We will add SetParamters later on
        public override void UnloadContent() { }
    }
}

// HMEffectManager.cs
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;

namespace HMEngine.HMEffects {
    public sealed class HMEffectManager : DrawableGameComponent {
        internal HMEffectManager(Game game) : base(game) { }
```

Tutorial 3 - More Rendering Complexity, Effects and Cameras

```
        public static HMEffect ActiveShader { get; private set; }

        public static void AddEffect(string effectLabel, HMEffect newEffect) {
            HMComponentManager.AddComponent(effectLabel, newEffect);
        }

        internal static void SetActiveShader(string effectLabel) {
            ActiveShader = HMComponentManager.GetComponent<HMShader>(effectLabel);
        }
    }
}
```

The effect uses the component manager to store the different shaders with their associated labels so we
can access them quickly by using the GetComponent function that we will add that looks like this:

```
// In HMComponentManager.cs
using System;

namespace HMEngine.HMComponents {
    internal sealed class HMComponentException : Exception {
        internal HMComponentException(string s) : base(s) { }
    }

    internal sealed class HMComponentManager : DrawableGameComponent {
        // Existing Code...

        internal static T GetComponent<T>(string name) {
            if (masterList.ContainsKey(name) && masterList[name] is T) {
                return (T)masterList[name];
            }

            throw new HMComponentException("No component with the name " + name + " exists");
        }

        // Existing Code...
    }
}
```

We need to add the effect manager as a component to HMGame as well to initialize it and load the
graphics content so everything is ready to be used when the objects start calling for it. We don't yet
have anything in the class overridden to be called by the Game, but once we have a default shader
written up for objects to use, we will want to have this here anyway, so we may as well add it in now so
we don't forget later on, also add in the GraphicsProfile.HiDef property in the GraphicsDeviceManager:

```
// In HMGame.cs
using HMEngine.HMEffects;

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

    Components.Add(new HMObjectManager(this));
    Components.Add(new HMEffectManager(this));
    Components.Add(new HMComponentManager(this));

    // Existing Code...
}
```

Tutorial 3 - More Rendering Complexity, Effects and Cameras

Now all that is left is to update the component manager class to correctly call the shader before rendering any items. We will need to put a few extra bits of rendering information in with the renderable interface as well so we can make sure the video card has everything it needs to get our objects on the screen. Here is what the new updated parts of the IHMRenderable interface and the HMComponentManager class look like:

```csharp
// IHMRenderable.cs
using Microsoft.Xna.Framework;

public interface IHMRenderable : IHMComponent {
    string Shader { get; set; }

    Vector3 Position { get; set; }
    Vector3 Scaling { get; set; }
    Quaternion Rotation { get; set; }

    void Render(GraphicsDevice myDevice);
}

// In HMObject
public string Shader { get; set; }

// In HMComponentManager.cs
using HMEngine.HMEffects;
using Microsoft.Xna.Framework.Graphics;

public override void Draw(GameTime gameTime) {
    foreach (IHMRenderable render in renderable.Values) {
        HMEffectManager.SetActiveShader(render.Shader);
        HMEffectManager.ActiveShader.SetParameters(render);

        foreach (EffectPass pass in HMEffectManager.ActiveShader.Effect.CurrentTechnique.Passes) {
            render.Render(GraphicsDevice, pass);
        }
    }

    base.Draw(gameTime);
}
```

The component manager draw call sets the device up to draw the type of vertices required by the current item, sets the active shader in our effect manager to the one assigned to the current renderable item and sets the shader parameters from the item as well. It then loops through the passes in the shader to actually perform the drawing. This is all done based on the Shader label that we will set on the objects in our demo code.

All of these additions to the render code here in the manager will allow us to leave out a lot of what would be duplicated code in each of our objects later on, starting with the quad.

The Textured Quad, Shader and All

We will now throw together a simple textured quad object that we can later utilize for a few other things (like billboarding or particle systems.) First of all, we will make an HMQuad class that inherits from HMObject. The class will have references to the location of the file used for its texture, an actual XNA Texture2D object to store the texture for setting on the video card prior to rendering the object, and a set of vertex and index buffers we can set our data up in. We will also want to add the

IHMLoadable interface to our base HMObject class so we can get the component manager to load the textures for us:

```csharp
// In HMObject
using Microsoft.Xna.Framework.Content;

public abstract class HMObject : IHMLoadable, IHMRenderable {
    // Existing Code...

    public abstract void LoadContent(GraphicsDevice myDevice, ContentManager myLoader);
    public abstract void UnloadContent();
}

// The Quad Object
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace HMEngine.HMObjects {
    public class HMQuad : HMObject {
        private readonly string myAsset;

        public VertexPositionTexture[] Vertices { get; set; }
        public Texture2D Texture { get; set; }
        public short[] Indexes { get; set; }

        public HMQuad(string textureAsset) {
            myAsset = textureAsset;
            Scaling = new Vector3(1);
        }

        public override void LoadContent(GraphicsDevice myDevice, ContentManager myLoader) {
            Texture = myLoader.Load<Texture2D>(myAsset);

            Vertices = new[] {
                new VertexPositionTexture(new Vector3(-0.5f, 0.5f, 0), new Vector2(0, 0)),
                new VertexPositionTexture(new Vector3(0.5f, 0.5f, 0), new Vector2(1, 0)),
                new VertexPositionTexture(new Vector3(-0.5f, -0.5f, 0), new Vector2(0, 1)),
                new VertexPositionTexture(new Vector3(0.5f, -0.5f, 0), new Vector2(1, 1))
            };

            Indexes = new short[] {0, 1, 2, 1, 3, 2};
        }

        public override void Render(GraphicsDevice myDevice) {
            pass.Apply();

            myDevice.Textures[0] = Texture;
            myDevice.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, Vertices, 0, 4, Indexes, 0, 2);
        }

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

We start out in the constructor by storing the string to the texture asset for use later. The LoadContent function simply loads in the texture and sets up the vertex and index data. Rendering is accomplished by setting the texture on the video card and calling the built in indexed primitives rendering function. Since the setup of the render loop and the shader loops are all handled by code outside the object render function, we only have to worry about this specific object's rendering calls and we can let the engine handle the rest.

Tutorial 3 - More Rendering Complexity, Effects and Cameras

To get a quad on screen, we need to first add a texture to a content folder. I added a clouded skull I put together from some images on istockphoto to a Textures folder in the demo content project and set its asset name parameter to "hazymind". I also added a Content project called HMEngineContent and added it as a content reference in HMDemo. All that is needed now is to write the actual shader. Right click on the content project in the engine and add a Shaders folder. Then add an effect inside. Here is the shader that I wrote in TransformTexture.fx:

```
float4x4 World;
float4x4 View;
float4x4 Projection;

sampler TextureSampler;

struct VS_INPUT {
    float4 Position : POSITION0;
    float2 Texcoord : TEXCOORD0;
};

VS_INPUT Transform(VS_INPUT Input) {
    VS_INPUT Output;

    float4 worldPosition = mul(Input.Position, World);
    float4 viewPosition = mul(worldPosition, View);

    Output.Position = mul(viewPosition, Projection);
    Output.Texcoord = Input.Texcoord;

    return Output;
}

float4 Texture(VS_INPUT Input) : COLOR0 {
    return tex2D(TextureSampler, Input.Texcoord);
}

technique TransformTexture {
    pass P0 {
        VertexShader = compile vs_2_0 Transform();
        PixelShader = compile ps_2_0 Texture();
    }
}
```

The first lines set up the matrix we will use to transform the object to the screen and a sampler on the card to get texture values. The structs above the Transform and Texture functions are simply easier ways to manage input and output to them. The Transform function multiplies the input position by the matrices we will pass and hands the texture coordinates through unmodified. The Texture function uses the sampler to find correct pixel colors based on the coordinates from the vertex shader. For now, we will modify the shader parameter code to pass in identity matrices until we get the camera all set up:

```
// In HMShader
using HMEngine.HMComponents;
using Microsoft.Xna.Framework;

public override void SetParameters(IHMRenderable renderable) {
    if (null != Effect.Parameters["World"]) Effect.Parameters["World"].SetValue(Matrix.Identity);
    if (null != Effect.Parameters["View"]) Effect.Parameters["View"].SetValue(Matrix.Identity);
    if (null != Effect.Parameters["Projection"]) Effect.Parameters["Projection"].SetValue(Matrix.Identity);
}


// Updated HMDemo class
using HMEngine;
```

Tutorial 3 - More Rendering Complexity, Effects and Cameras

```
using HMEngine.HMObjects;
using HMEngine.HMEffects;
using Microsoft.Xna.Framework;

namespace HMDemo {
    internal static class HMDemo {
        private static readonly HMGame game = new HMGame(800, 600);
        private static readonly HMQuad quad = new HMQuad("Content/Textures/hazymind");
        private static readonly HMShader shader = new HMShader("HMEngineContent/Shaders/TransformTexture");

        public static void Main() {
            HMEffectManager.AddEffect("TT", shader);
            quad.Shader = "TT";

            HMObjectManager.AddObject("quad", quad);
            game.Run();
        }
    }
}
```

Adding in our Basic Camera

The camera implementation that we will use for the engine will rely on Quaternions. There are many reasons for choosing this method over vectors or simply storing the transformation information directly in matrices. For one, quaternions prevent Gimble Lock, which basically causes certain rotations to be impossible if two axes of rotation are ever alligned (think about those spinning rides at the fair that have three circles inside each other and you strapped in the middle.) The reason for storing the rotations and transformations in quaternions instead of matrices is basically for ease of understanding how the camera works. It also makes our code for the different types of transformations short and easy to read. The camera only needs the addition of an updatable interface (like the loadable one we just added):

```
// New Updatable Interface
using Microsoft.Xna.Framework;

namespace HMEngine.HMComponents {
    public interface IHMUpdatable : IHMComponent {
        void Update(GameTime gameTime);
    }
}

// In HMComponentManager
using Microsoft.Xna.Framework.Content;

private static readonly Dictionary<string, IHMUpdatable> updatable = new Dictionary<string, IHMUpdatable>();

public static void AddComponent(string name, IHMComponent component) {
    // Exisiting Code...

    if (component is IHMUpdatable) {
        updatable.Add(name, (IHMUpdatable)component);
    }
}

public override void Update(GameTime gameTime) {
    foreach (IHMUpdatable update in updatable.Values) {
        update.Update(gameTime);
    }

    base.Update(gameTime);
}


// HMCamera.cs
using HMEngine.HMComponents;
```

Tutorial 3 - More Rendering Complexity, Effects and Cameras

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

namespace HMEngine.HMCameras {
    public sealed class HMCamera : IHMUpdatable {
        private Viewport myViewport;

        public Vector3 Position { get; set; }
        public Quaternion Rotation { get; set; }
        public Matrix World { get; private set; }
        public Matrix View { get; private set; }
        public Matrix Projection { get; private set; }
        public float FieldOfView { get; set; }
        public Vector3 Target { get; set; }

        public Viewport Viewport {
            get { return myViewport; }
            set {
                myViewport = value;
                myViewport.MinDepth = 1.0f;
                myViewport.MaxDepth = 1000.0f;
            }
        }

        public HMCamera(Viewport newViewport) {
            Position = new Vector3(0, 0, 1);
            Rotation = new Quaternion(0, 0, 0, 1);
            FieldOfView = MathHelper.Pi / 3.0f;
            Target = new Vector3(0, 0, 0);

            Viewport = newViewport;
        }

        public void Update(GameTime gameTime) { UpdateMatrices(); }

        private void UpdateMatrices() {
            World = Matrix.Identity;

            View = Matrix.Invert(
                Matrix.CreateFromQuaternion(Rotation) *
                Matrix.CreateTranslation(Position)
                );

            Projection = Matrix.CreatePerspectiveFieldOfView(
                FieldOfView,
                Viewport.AspectRatio,
                Viewport.MinDepth, Viewport.MaxDepth
                );
        }
    }
}
```

This basic camera will simply take in a viewport (that the GraphicsComponent can be asked for) and will create a projection and view matrix based on its position and rotation and the width and height values in the viewport. Update will be called automatically by the Game class which will keep the matrices current every time we make a change the camera's position or rotation with motion functions later on. We could actually remove the Update method altogether and trust the modifying methods we will add later to always update the matrices, but it is convenient to do so here and we may need update for more complicated behaviors later on. To get everything working with the new camera, we will add a manager for it as well:

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

Tutorial 3 - More Rendering Complexity, Effects and Cameras

```csharp
namespace HMEngine.HMCameras {
    public sealed class HMCameraManager : DrawableGameComponent {
        internal HMCameraManager(Game game) : base(game) { }
        private static HMCamera DefaultCamera { get; set; }
        public static HMCamera ActiveCamera { get; private set; }

        protected override void LoadContent() {
            DefaultCamera = new HMCamera(GraphicsDevice.Viewport);

            AddCamera("HMEngine.HMCameras.DefaultCamera", DefaultCamera);
            SetActiveCamera("HMEngine.HMCameras.DefaultCamera");

            base.LoadContent();
        }

        public static void AddCamera(string name, HMCamera camera) {
            HMComponentManager.AddComponent(name, camera);
        }

        public static void SetActiveCamera(string name) {
            ActiveCamera = HMComponentManager.GetComponent<HMCamera>(name);
        }
    }
}
```

To use the new camera manager, we will add it as a component to the HMGame just as we did with our other managers. This will make sure to create the default camera for us and update the cameras' matrices when the shaders are set up each frame so we can be sure to always have an updated view no matter where in the engine we call the cameras from. We also need to make sure that when we resize the window the camera knows it:

```csharp
// In HMGame
using HMEngine.HMCameras;

public HMGame() {
                                            Components.Add(new HMCameraManager(this));
    // Other managers

    // Window sizing code
}

private void Window_ClientSizeChanged(object sender, System.EventArgs e) {
    myGraphics.PreferredBackBufferWidth = Window.ClientBounds.Width;
    myGraphics.PreferredBackBufferHeight = Window.ClientBounds.Height;

    HMCameraManager.ActiveCamera.Viewport = GraphicsDevice.Viewport;
}
```

The Viewport is only being set on the active camera for now, but we will add more logic around viewports when we get to implementing split screen for multiplayer games. Now that we have a default camera set up in the manager, we need to get it passing its matrices into the shaders so everything is being rendered how it should be relative to the active camera at the time of calling the render function. To do this, I am just going to change the function in the shader setting its parameters:

Tutorial 3 - More Rendering Complexity, Effects and Cameras

```csharp
// New HMShader
using HMEngine.HMCameras;
using Microsoft.Xna.Framework;

namespace HMEngine.HMEffects {
    public sealed class HMShader : HMEffect {
        public HMShader(string asset) : base(asset) { }

        public override void SetParameters(IHMRenderable renderable) {
            Matrix world =
                HMCameraManager.ActiveCamera.World *
                Matrix.CreateScale(renderable.Scaling) *
                Matrix.CreateFromQuaternion(renderable.Rotation) *
                Matrix.CreateTranslation(renderable.Position);

            if (null != Effect.Parameters["World"]) {
                Effect.Parameters["World"].SetValue(world);
            }

            if (null != Effect.Parameters["View"]) {
                Effect.Parameters["View"].SetValue(HMCameraManager.ActiveCamera.View);
            }

            if (null != Effect.Parameters["Projection"]) {
                Effect.Parameters["Projection"].SetValue(HMCameraManager.ActiveCamera.Projection);
            }
        }

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

The SetParameters function will contain a lot more variables to be passed later on when we implement more complicated lighting and animations in our shaders, but for now it will do to just pass these few variables.

Running the demo now should show the same quad as before, only this time it will be using the real camera that we have just created. You will notice that resizing the window now doesn't simply stretch the quad anymore (and that it is actually a square finally). This is because we are passing the correct viewport values to the shader through the projection matrix. Without much work, you can see how easy it would be to set up multiple cameras in a scene in the world and switch back and forth between them.