Adding Post Processing

Post-Processing is accomplished by rendering a 3D scene to a texture and then rendering that texture on a full screen quad using the Pixel Shader, which will apply one or more effects to the final scene. For our implementation, we will start by creating a new class that extends the base HMEffect class and implements a new IHMPostProcessable interface. This way, we can add post processors as another type of Effect the way we did our Shaders and let the effect manager and the component manager handle them the same way. Here is the new interface, which is handled the same way as all our others:

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

namespace HMEngine.HMComponents {
    public interface IHMPostProcessable : IHMComponent {
        void PostProcess(GraphicsDevice myDevice);
    }
}

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

public static void AddComponent(string name, IHMComponent component) {
    // Existing Code...
    if (component is IHMPostProcessable) {
        postprocessable.Add(name, (IHMPostProcessable)component);
    }
}

public override void Draw(GameTime gameTime) {
    // Exisiting Code...
    foreach (IHMPostProcessable postprocess in postprocessable.Values) {
        postprocess.PostProcess(GraphicsDevice);
    }

    base.Draw(gameTime);
}

// In HMEffectManager
using Microsoft.Xna.Framework.Graphics;

public static Texture2D ScreenTexture { get; internal set; }
```

The Post Processor Class

We will need to keep three resources as members of our new class, an array of vertices, a texture to copy the current back buffer to that we can use as input for our effect file, and an array of colors to use for copying the back buffer to that texture. Here is the whole class with explanations:

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

namespace HMEngine.HMEffects {
    public sealed class HMPostProcessor : HMEffect, IHMPostProcessable {
        private VertexPositionTexture[] verts;
```

Tutorial 8 - Adding Post Processing

```
        private Color[] data;
        private Texture2D ResolvedTexture { get; set; }
```

The three members I just mentioned are laid out first for filling in by LoadContent later.

```
        public HMPostProcessor(string asset) : base(asset) { }
```

The base constructor is called so that the asset can be correctly loaded by the content manager. We put the resolve target logic in its own function because we will call it from the draw method as well if the target texture is not the correct size, which can happen when we go fullscreen or stretch the window.

```
        public override void LoadContent(GraphicsDevice myDevice, ContentManager myLoader) {
            base.LoadContent(myDevice, myLoader);

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

            LoadResolveTarget(myDevice);
        }

        private void LoadResolveTarget(GraphicsDevice myDevice) {
            ResolvedTexture = new Texture2D(
                myDevice, myDevice.Viewport.Width,
                myDevice.Viewport.Height, false,
                myDevice.DisplayMode.Format
                );

            data = new Color[myDevice.Viewport.Width * myDevice.Viewport.Height];
        }

        // Unused in post processing
        public override void SetParameters(IHMRenderable renderable) { }

        public override void UnloadContent() {}
```

Our LoadContent function starts by calling the base class version to get the actual effect loaded. Then, it fills in the vertex list with the correct values to pass to the card later. We do this here because it is only necessary once and would be a pointless waste of resources to recreate each frame. The vertices in order are: top left corner of the screen, bottom left, bottom right, and top right.

We define them in this order so that we can use a triangle strip (the fastest type of primitive to draw) when we actually do the rendering in the draw function. This is a pretty important performance booster over using the triangle list that we have used in the past because we are going to be drawing this every frame on top of all of the other rendering we are doing:

```
        public void PostProcess(GraphicsDevice myDevice) {
            if (
                ResolvedTexture.Width != myDevice.Viewport.Width ||
                ResolvedTexture.Height != myDevice.Viewport.Height ||
                ResolvedTexture.Format != myDevice.DisplayMode.Format
                ) {
                LoadResolveTarget(myDevice);
            }

            myDevice.GetBackBufferData(data);
```

Tutorial 8 - Adding Post Processing

```
        ResolvedTexture.SetData(data);
        myDevice.Textures[0] = ResolvedTexture;

        foreach (EffectPass pass in Effect.CurrentTechnique.Passes) {
            pass.Apply();
            myDevice.DrawUserPrimitives(PrimitiveType.TriangleStrip, verts, 0, 2);
        }

        HMEffectManager.ScreenTexture = ResolvedTexture;
    }
```

PostProcess simply copies the back buffer, sets it to the device texture, and renders with the shader.

Our first Post Processor

The post processor we will use is a simple color inversion that will give a negative effect of the image that the card is already rendering. Keep in mind this is not a photo negative effect, as there is a bit more color shifting and math involved in producing one of those, but it will put forth a good start for how to create post processors for the engine.

I created this new PostInvert.fx effect file in the HMEngineContent/Shaders/PostProcessors folder:

```
sampler TextureSampler;

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

VS_INPUT NullVertexShader(VS_INPUT Input) {
    VS_INPUT Output;

    Output.Position = Input.Position;
    Output.Texcoord = Input.Texcoord;

    return Output;
}

float4 InvertTexture(VS_INPUT Input) : COLOR0{
    return (float4)1.0 - tex2D(TextureSampler, Input.Texcoord);
}

technique PostInvert {
    pass P0 {
        VertexShader = compile vs_2_0 NullVertexShader();
        PixelShader = compile ps_2_0 InvertTexture();
    }
}
```

The effect starts out with what I have chosen to call a null vertex shader. This simply passes the coordinates of the vertices given to it out to the pixel shader without any transformation at all, and since we are passing in the coordinates of the screen's corners, no transform is needed.

The InvertTexture function takes the input texture coordinate (which we also passed in as the corners) and does a simple algebraic inversion on them. This will output a negative effect to the screen.

Adding the post processor to the rendered output is as simple as putting two lines in the demo class:

Tutorial 8 - Adding Post Processing

```
private static readonly HMPostProcessor post =
    new HMPostProcessor("HMEngineContent/Shaders/PostProcessors/PostInvert");

HMEffectManager.AddEffect("PI", post);
```

Here's a shot of what it looks like in my version with the same camera angle as previous images: