CPI 411 Graphics for Games

Lab XX (Blueprint Shader)

The main purpose of this exercise is to explore the technique of depth peeling and see how we can generate useful visuals of a 3D scene resembling a blueprint and examine its parts. The following are the files we will be making:

- `Shaders.fx:` Shader file to create the depth peeling effect.
- `EdgeMap.fx:` Shader file to create the edge map based on normal data.
- `Ztest.fx:` Shader file to create the edge map based on depth data.
- `Composite.fx:` Shader file to composite the depth maps together with a background.
- `LabXX.cs:` Sample game program to test the shaders.

A. Depth Peeling

Before we begin, set the project in VS and add the resource files we will use in our exercise.

- Make a new project, Lab0.cs
- Open the MonoGame Content Pipeline and add the `objects.fbx` and the `blueprint.jpg` files posted on the course website.
- Create the required shader files mentioned above.

To implement the shader, the main thing is to encode the depth in a depth map so that it can be used again to calculate the next depth map. We will call these different depth maps at every level 'depth layers'. After the first depth layer is created, we pass in the depth map again along with the scene and only render the fragments whose depth is greater than what is already recorded in the previous depth layer. This gives us the second depth layer, and we can continue in this fashion to render all subsequent layers. A stop condition can be achieved when there is no change between the previous depth layer and the newly rendered one, but for this exercise we will add our own controls to choose when to stop.

We will create the following variable and structures for the **Shaders.fx** file:

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

texture2D DepthTexture;

sampler depthMap = sampler_state
{
  Texture = <DepthTexture>;
  MipFilter = NONE;
  MinFilter = POINT;
  MagFilter = POINT;
};
```

```
struct VertexShaderInput
{
  float4 Position : POSITION0;
  float4 Position2D : TEXCOORD0;
  float4 Normal : NORMAL;
};

struct VertexShaderOutput
{
  float4 Position : POSITION0;
  float4 Position2D : TEXCOORD0;
  float3 Normal : TEXCOORD1;
};
```

Now we will make a helper function to help encode and decode our depth values using 24 of the 32 bits available in a typical float4 color variable. This can be extended to all 32 bits for greater precision, but the exercise will work with 24 as well. Note that without this step, the rendering will suffer from extreme perspective aliasing.

```
float4 EncodeFloatRGB(float f)
{
  float4 color;
  f *= 256;
  color.x = floor(f);
  f = (f - color.x) * 256;
  color.y = floor(f);
  color.z = f - color.y;
  color.xy *= 0.00390625; // *= 1.0/256
  color.a = 1;

  return color;
}

float DecodeFloatRGB(float4 color)
{
  const float3 byte_to_float = float3(1.0, 1.0 / 256, 1.0 / (256 * 256));
  return dot(color.xyz, byte_to_float);
}
```

We will also create the vertex shader output function. This will be common for all the peeling fragment shaders so we need only create it once. Note that we output the same Position value twice, once as a Texture Coordinate as well because we cannot alter these values in the fragment shader if it remains as a Position coordinate.

```
VertexShaderOutput DepthVertexShader(VertexShaderInput input)
{
  VertexShaderOutput output;

  output.Position = mul(mul(mul(input.Position, World), View), Projection);
  output.Normal = normalize(mul(input.Normal, WorldInverseTranspose).xyz);
```

```
    output.Position2D = output.Position;

    return output;
  }
```

The next part is creating the pixel shaders. We follow a similar process to the previous lab in which we studied depth mapping. Note how we use the `EncodeFloatRGB` function to store the depth into a float4 value instead of just the alpha value. We also use the `discard` function which is an inbuilt operation that tells the GPU to ignore the fragment's color when combining all the fragments that affect a pixel together. This is only done afterwards so we still have to return some value. We create the rendering of the initial depth layer as well as a rendering which stores its normal values as colors, both of which will be used later.

```
  float4 DepthMapPixelShader(VertexShaderOutput input) : COLOR0
  {
    float4 projTexCoord = input.Position2D / input.Position2D.w;
    projTexCoord.xy = 0.5 * projTexCoord.xy + float2(0.5, 0.5);
    projTexCoord.y = 1.0 - projTexCoord.y; // invert Y direction (because UV
map is opposite to y coordinate system)

    if (projTexCoord.x >= 0 && projTexCoord.x <= 1 && projTexCoord.y >= 0 &&
projTexCoord.y <= 1 && saturate(projTexCoord).x == projTexCoord.x &&
saturate(projTexCoord).y == projTexCoord.y)
    {

      float depth = projTexCoord.z;
      float4 color;
      color = (depth > 0) ? EncodeFloatRGB(depth) : EncodeFloatRGB(0);
      color.a = (depth > 0) ? 1 : 0; // culling

      return color;
    }

    else
    {
      discard;

      float4 color;
      color = EncodeFloatRGB(0);
      color.a = 0; // culling

      return color;
    }
  }

  float4 NormalPixelShader(VertexShaderOutput input) : COLOR0
  {
    float4 projTexCoord = input.Position2D / input.Position2D.w;
    projTexCoord.xy = 0.5 * projTexCoord.xy + float2(0.5, 0.5);
   projTexCoord.y = 1.0 - projTexCoord.y; // invert Y direction (because UV
map is opposite to y coordinate system)

    float depth = projTexCoord.z;
```

```
    float4 color;
    color.rgb = (normalize(input.Normal.xyz)) / 2.0f + 0.5f;
    color.a = (depth > 0) ? 1 : 0; // culling

    return color;
  }
```

We will now construct the peeling shaders. This is the most important part and the structure of these shaders resemble the depth map and normal map shaders we have done above, with one crucial difference. We are now going to also take in the previous depth layer, decode it using our helper function and compare the obtained depth value at the pixel where the projection of the current fragment would be located in the depth map, with the value of the actual depth of the current fragment. We will only render the new value if it is higher, otherwise we discard the fragment.

```
  float4 DepthPeelingPixelShader(VertexShaderOutput input) : COLOR0
  {
    float4 projTexCoord = input.Position2D / input.Position2D.w;
    projTexCoord.xy = 0.5 * projTexCoord.xy + float2(0.5, 0.5);
    projTexCoord.y = 1.0 - projTexCoord.y; // invert Y direction (because UV
map is opposite to y coordinate system)

    float depth = projTexCoord.z;

    float4 prevDepthLayer = tex2D(depthMap, projTexCoord.xy);
    float prevDepth = DecodeFloatRGB(prevDepthLayer);

    if (projTexCoord.x >= 0 && projTexCoord.x <= 1 && projTexCoord.y >= 0 &&
projTexCoord.y <= 1 && saturate(projTexCoord).x == projTexCoord.x &&
saturate(projTexCoord).y == projTexCoord.y)
    {
      if (depth >= 1.0f - 1.0f / 5000.0f)
      {
        float4 color;
        color.rgba = 0;

        return color;
      }

      else
      {
        if (depth <= prevDepth + 1.0f / 4096.5f)
        {
          discard;

          float4 color;
          color = EncodeFloatRGB(depth);

          return color;
        }

        else
        {
          float4 color;
```

```
          color = EncodeFloatRGB(depth);

          return color;
        }
      }
    }

    else
    {
      discard;

      float4 color;
      color = EncodeFloatRGB(depth);

      return color;
    }
  }

  float4 NormalPeelingPixelShader(VertexShaderOutput input) : COLOR0
  {
    float4 projTexCoord = input.Position2D / input.Position2D.w;
    projTexCoord.xy = 0.5 * projTexCoord.xy + float2(0.5, 0.5);
    projTexCoord.y = 1.0 - projTexCoord.y; // invert Y direction (because UV
map is opposite to y coordinate system)



    float depth = projTexCoord.z;

    float4 prevDepthLayer = tex2D(depthMap, projTexCoord.xy);
    float prevDepth = DecodeFloatRGB(prevDepthLayer);

    if (projTexCoord.x >= 0 && projTexCoord.x <= 1 && projTexCoord.y >= 0 &&
projTexCoord.y <= 1 && saturate(projTexCoord).x == projTexCoord.x &&
saturate(projTexCoord).y == projTexCoord.y)
    {
      if (depth <= prevDepth + 1.0f / 4096.5f)
      {
        discard;

        float4 color;
        color.rgb = 0;
        color.a = 0;

        return color;
      }

      else
      {
        float4 color;
        color.rgb = (normalize(input.Normal.xyz)) / 2.0f + 0.5f;
        color.a = 1;

        return color;
      }
```

```
  }
  else
  {
    discard;

    float4 color;
    color.rgb = 0;
    color.a = 0;

    return color;
  }
}
```

Finally, we add these vertex and fragment shaders together into four different techniques in our file.

```
technique DepthMap
{
  pass Pass1
  {
    VertexShader = compile vs_4_0 DepthVertexShader();
    PixelShader = compile ps_4_0 DepthMapPixelShader();
  }
}

technique NormalMap
{
  pass Pass1
  {
    VertexShader = compile vs_4_0 DepthVertexShader();
    PixelShader = compile ps_4_0 NormalPixelShader();
  }
}

technique DepthPeeling
{
  pass Pass1
  {
    VertexShader = compile vs_4_0 DepthVertexShader();
    PixelShader = compile ps_4_0 DepthPeelingPixelShader();
  }
}

technique NormalPeeling
{
  pass Pass1
  {
    VertexShader = compile vs_4_0 DepthVertexShader();
    PixelShader = compile ps_4_0 NormalPeelingPixelShader();
  }
}
```

B. Main Program (Game1 or LabXX.cs)

Now we will make the main program to test our depth peeling effect. First, add the following variables in the constructor of the main game class:

```
int depthLayerCount = 1;
RenderTarget2D depthRenderTarget;
RenderTarget2D normalRenderTarget;
RenderTarget2D compositeRenderTarget;
RenderTarget2D edgeMapRenderTarget;

RenderTarget2D depth2;
RenderTarget2D normal2;
RenderTarget2D edgeMap2RenderTarget;

RenderTarget2D layerBlendTarget;

Texture2D depthAndColorMap;
Texture2D testMap;
Texture2D depthAndColorMap2;
Texture2D testMap2;

Texture2D edgeMap;
Texture2D edgeMap2;
Texture2D compositeEdgeMap;

Texture2D layerBlend;
Texture2D bgTexture;

Effect edgeEffect;
Effect testEffect;
Effect compositeEffect;
```

Next, load the necessary content files into the program and create Render Targets for various stages of shading in the **LoadContent()** method. We will need at least two render targets for depth layers alone because we cannot read from and write to a texture at the same time, so we need to alternate between two textures.

```
font = Content.Load<SpriteFont>("Font");
model = Content.Load<Model>("objects");
effect = Content.Load<Effect>("Shaders");
edgeEffect = Content.Load<Effect>("EdgeMap");
testEffect = Content.Load<Effect>("ZTest");
compositeEffect = Content.Load<Effect>("Composite");
bgTexture = Content.Load<Texture2D>("blueprint");

PresentationParameters pp = GraphicsDevice.PresentationParameters;

depthRenderTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048, false,
SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

normalRenderTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048,
false, SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);
```

```
compositeRenderTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048,
false, SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

edgeMapRenderTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048,
false, SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

edgeMap2RenderTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048,
false, SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

layerBlendTarget = new RenderTarget2D(GraphicsDevice, 2048, 2048, false,
SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

depth2 = new RenderTarget2D(GraphicsDevice, 2048, 2048, false,
SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);

normal2 = new RenderTarget2D(GraphicsDevice, 2048, 2048, false,
SurfaceFormat.Color, DepthFormat.Depth24, 0,
RenderTargetUsage.PlatformContents);
```

Now let's add a statement to control the number of depth layers being peeled at a given time. We can use the 'D' key along with Left Shift to control this value, and we will use the **depthLayerCount** variable to achieve this. Add the following snippet into the **Update()** function (and make sure the previousKeyboardState is being recorded at the bottom):

```
if (Keyboard.GetState().IsKeyDown(Keys.D) &&
Keyboard.GetState().IsKeyDown(Keys.LeftShift)
&& !previousKeyboardState.IsKeyDown(Keys.D))
{
  depthLayerCount--;
}

else if (Keyboard.GetState().IsKeyDown(Keys.D)
&& !previousKeyboardState.IsKeyDown(Keys.D))
{
  depthLayerCount++;
}

previousKeyboardState = Keyboard.GetState();
```

Finally, we have to focus on our drawing methods. Let's make a series of draw functions that help separate the different renderings we must make. To visualize depth peeling, we will make two methods for the depth and normal renderings each. One will create the initial map, and the other will take that as input and perform the rendering for all subsequent layers using the four techniques in our shader file.

```
private void DrawDepthMap()
{
  effect.CurrentTechnique = effect.Techniques["DepthMap"];

  foreach (EffectPass pass in effect.CurrentTechnique.Passes)
  {
    foreach (ModelMesh mesh in model.Meshes)
    {
      foreach (ModelMeshPart part in mesh.MeshParts)
      {
        effect.Parameters["World"].SetValue(mesh.ParentBone.Transform);
        effect.Parameters["View"].SetValue(view);
        effect.Parameters["Projection"].SetValue(projection);

        Matrix worldInverseTransposeMatrix =
    Matrix.Transpose(Matrix.Invert(mesh.ParentBone.Transform));

    effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTranspose
    Matrix);

        pass.Apply();
        GraphicsDevice.SetVertexBuffer(part.VertexBuffer);
        GraphicsDevice.Indices = part.IndexBuffer;
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
            part.VertexOffset, part.StartIndex, part.PrimitiveCount);
      }
    }
  }
}

private void DrawNormalMap()
{
  effect.CurrentTechnique = effect.Techniques["NormalMap"];

  foreach (EffectPass pass in effect.CurrentTechnique.Passes)
  {
    foreach (ModelMesh mesh in model.Meshes)
    {
      foreach (ModelMeshPart part in mesh.MeshParts)
      {
        effect.Parameters["World"].SetValue(mesh.ParentBone.Transform);
        effect.Parameters["View"].SetValue(view);
        effect.Parameters["Projection"].SetValue(projection);
        Matrix worldInverseTransposeMatrix =
    Matrix.Transpose(Matrix.Invert(mesh.ParentBone.Transform));

    effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTranspose
    Matrix);
        effect.Parameters["DepthTexture"].SetValue(depthAndColorMap);
        pass.Apply();

        GraphicsDevice.SetVertexBuffer(part.VertexBuffer);
        GraphicsDevice.Indices = part.IndexBuffer;
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
```

```
                part.VertexOffset, part.StartIndex, part.PrimitiveCount);
      }
    }
  }
}

private void DrawDepthLayer()
{
  effect.CurrentTechnique = effect.Techniques["DepthPeeling"];

  foreach (EffectPass pass in effect.CurrentTechnique.Passes)
  {
    foreach (ModelMesh mesh in model.Meshes)
    {
      foreach (ModelMeshPart part in mesh.MeshParts)
      {
        effect.Parameters["World"].SetValue(mesh.ParentBone.Transform);
        effect.Parameters["View"].SetValue(view);
        effect.Parameters["Projection"].SetValue(projection);
        Matrix worldInverseTransposeMatrix =
    Matrix.Transpose(Matrix.Invert(mesh.ParentBone.Transform));

    effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTranspose
    Matrix);

        effect.Parameters["DepthTexture"].SetValue(depthAndColorMap2);
        pass.Apply();

        GraphicsDevice.SetVertexBuffer(part.VertexBuffer);
        GraphicsDevice.Indices = part.IndexBuffer;
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                 part.VertexOffset, part.StartIndex, part.PrimitiveCount);
      }
    }
  }
}

private void DrawNormalLayer()
{
  effect.CurrentTechnique = effect.Techniques["NormalPeeling"];

  foreach (EffectPass pass in effect.CurrentTechnique.Passes)
  {

    foreach (ModelMesh mesh in model.Meshes)
    {
      foreach (ModelMeshPart part in mesh.MeshParts)
      {

         effect.Parameters["World"].SetValue(mesh.ParentBone.Transform);
         effect.Parameters["View"].SetValue(view);
         effect.Parameters["Projection"].SetValue(projection);
         Matrix worldInverseTransposeMatrix =
    Matrix.Transpose(Matrix.Invert(mesh.ParentBone.Transform));
```

```
      effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTranspose
      Matrix);

           effect.Parameters["DepthTexture"].SetValue(depthAndColorMap2);
           pass.Apply();

           GraphicsDevice.SetVertexBuffer(part.VertexBuffer);
           GraphicsDevice.Indices = part.IndexBuffer;
           GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                     part.VertexOffset, part.StartIndex, part.PrimitiveCount);
       }
      }
    }
  }
```

Finally, add the loop in the Draw() method that calls our drawing functions and generates the various renders that display the different depth layers:

```
  GraphicsDevice.Clear(Color.CornflowerBlue);
  GraphicsDevice.BlendState = BlendState.AlphaBlend;

  RasterizerState rasterizerState = new RasterizerState();
  rasterizerState.CullMode = CullMode.None;
  GraphicsDevice.RasterizerState = rasterizerState;

  GraphicsDevice.DepthStencilState = DepthStencilState.Default;

  GraphicsDevice.SetRenderTarget(compositeRenderTarget);
  GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Black, 1.0f, 0);

  GraphicsDevice.SetRenderTarget(depthRenderTarget);
  GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

  // *** Draw initial Depth Map
  DrawDepthMap();
  GraphicsDevice.SetRenderTarget(null);

  depthAndColorMap = (Texture2D)depthRenderTarget;
  depthAndColorMap2 = (Texture2D)depthRenderTarget;


  GraphicsDevice.SetRenderTarget(normalRenderTarget);
  GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Black, 1.0f, 0);

  // *** Draw initial Normal Map
  DrawNormalMap();
  GraphicsDevice.SetRenderTarget(null);

  testMap = (Texture2D)normalRenderTarget;
```

```
  testMap2 = (Texture2D)normalRenderTarget;

  for (int i = 1; i < depthLayerCount; i++)
  {
    if (i % 2 == 1)
    {
      GraphicsDevice.SetRenderTarget(depth2);
    }

    else
    {
      GraphicsDevice.SetRenderTarget(depthRenderTarget);
    }

    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

    // *** Draw next depth layer
    DrawDepthLayer();
    GraphicsDevice.SetRenderTarget(null);

    if (i % 2 == 1)
    {
      GraphicsDevice.SetRenderTarget(normal2);
    }

    else
    {
      GraphicsDevice.SetRenderTarget(normalRenderTarget);
    }

    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Black, 1.0f, 0);

    DrawNormalLayer();
    GraphicsDevice.SetRenderTarget(null);

    if (i % 2 == 1)
    {
      depthAndColorMap2 = (Texture2D)depth2;
      testMap2 = (Texture2D)normal2;
    }

    else
    {
      depthAndColorMap2 = (Texture2D)depthRenderTarget;
      testMap2 = (Texture2D)normalRenderTarget;
    }

    GraphicsDevice.SetRenderTarget(null);
  }
```

Add the following snippet to the Draw() method as well to visualize depth peeling. We will comment

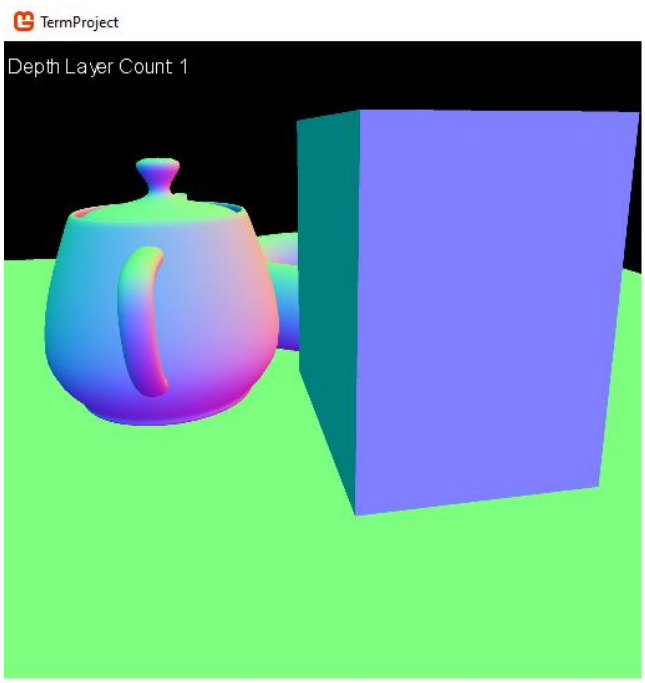
out this part later.

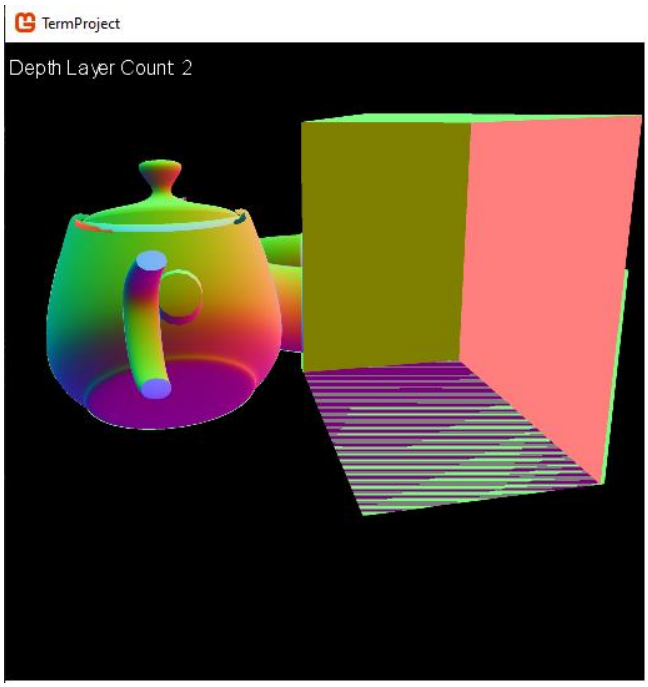
```
  GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

  using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
  {
    sprite.Begin();
    sprite.Draw(testMap2, new Vector2(0, 0), null, Color.White, 0, new
Vector2(0, 0), 0.25f, SpriteEffects.None, 1);
    sprite.End();
  }
```

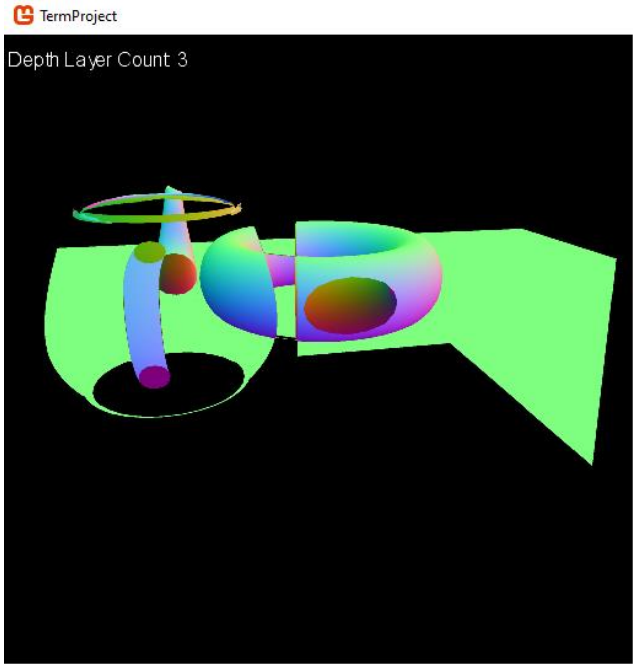
Finally, set all textures to null at the end of the draw cycle. You may also add statements using spriteBatch to visualize the current value of the depthLayerCount variable.

```
  depthAndColorMap = null;
  testMap = null;
  depthAndColorMap2 = null;
  testMap2 = null;
  edgeMap = null;
  edgeMap2 = null;
  compositeEdgeMap = null;
  layerBlend = null;
```

The resulting render for the scenes upto the third layer are shown in the below images







C. Edge Map

Now that depth peeling is completed, we must implement the edge map algorithm followed in this paper (Nienhaus et al, "Edge Enhancement – An Algorithm for Realtime Non-Photorealistic Rendering", Journal of WSCG '03, 2003) in section 4. We will use the first technique that applies the two formulas for the normal map and the depth map respectively, and applies the values to the RGB and alpha values respectively. First, we will make the two shader files that obtain the edge maps from the normal layer and the depth layer and name them **EdgeMap.fx** and **Ztest.fx** accordingly.

For **EdgeMap.fx**, we use the following variables and structures. Note that we are now directly working

on textures and no longer need the original scene information.

```
  float4x4 MatrixTransform;
  texture2D modelTexture;
  float imageWidth;
  float imageHeight;

  sampler TextureSampler : register(s0) = sampler_state
  {
    Texture = <modelTexture>;
    ADDRESSU = CLAMP;
    ADDRESSV = CLAMP;
  };

  struct VS_OUTPUT
  {
    float4 Pos : POSITION;
    float2 UV0 : TEXCOORD0;
    float4 UV1 : TEXCOORD1;
  };
```

Then we apply the technique meant to be used on the normal map as described in the paper through the fragment shader as follows:

```
  VS_OUTPUT vtxSh(float4 inPos : POSITION, float2 inTex : TEXCOORD0)
  {
    VS_OUTPUT Out;
    Out.Pos = mul(inPos, MatrixTransform);
    Out.UV0 = inTex;
    Out.UV1 = float4(2 / imageWidth, 0, 0, 2 / imageHeight);

    return Out;
  };

  float4 pxlSh(VS_OUTPUT In) : COLOR
  {
    float4 tex = tex2D(TextureSampler, In.UV0);

    float4 texA = 2.0f * (tex2D(TextureSampler, In.UV0 - In.UV1.xy -
In.UV1.zw) - 0.5f);
    float4 texC = 2.0f * (tex2D(TextureSampler, In.UV0 + In.UV1.xy -
In.UV1.zw) - 0.5f);
    float4 texF = 2.0f * (tex2D(TextureSampler, In.UV0 - In.UV1.xy +
In.UV1.zw) - 0.5f);
    float4 texH = 2.0f * (tex2D(TextureSampler, In.UV0 + In.UV1.xy +
In.UV1.zw) - 0.5f);

    float4 color;

    color.rgb = 0.5 * (dot(texA.rgb, texH.rgb) + dot(texC.rgb, texF.rgb));
    color.a = 1.0f;
```

```
  return color;
};

technique EdgeDraw
{
  pass P0
  {
    VertexShader = compile vs_4_0 vtxSh();
    PixelShader = compile ps_4_0 pxlSh();
  }
};
```

We write a very similar shader in **Ztest.fx**,with the main difference being that we change the algorithm in the fragment shader to mimic the one used for the depth map in the paper. We will composite this into the alpha channel later. Note that since we are working with the depth map, we need our earlier `DecodeFloatRGB` helper function again.

```
float4x4 MatrixTransform;
texture2D depthTexture;
float imageWidth;
float imageHeight;

float DecodeFloatRGB(float4 color)
{
  const float3 byte_to_float = float3(1.0, 1.0 / 256, 1.0 / (256 * 256));
  return dot(color.xyz, byte_to_float);
}

sampler depthTextureSampler : register(s1) = sampler_state
{
  Texture = <depthTexture>;
  ADDRESSU = CLAMP;
  ADDRESSV = CLAMP;
};

struct VS_OUTPUT
{
  float4 Pos : POSITION;
  float2 UV0 : TEXCOORD0;
  float4 UV1 : TEXCOORD1;
};

VS_OUTPUT vtxSh(float4 inPos : POSITION, float2 inTex : TEXCOORD0)
{
  VS_OUTPUT Out;
  Out.Pos = mul(inPos, MatrixTransform);
  Out.UV0 = inTex;
  Out.UV1 = float4(2 / imageWidth, 0, 0, 2 / imageHeight);

  return Out;
};
```

```
  float4 zPxlSh(VS_OUTPUT In) : COLOR
  {
    float texA = DecodeFloatRGB(tex2D(depthTextureSampler, In.UV0 - In.UV1.xy
- In.UV1.zw));
    float texC = DecodeFloatRGB(tex2D(depthTextureSampler, In.UV0 + In.UV1.xy
- In.UV1.zw));
    float texF = DecodeFloatRGB(tex2D(depthTextureSampler, In.UV0 - In.UV1.xy
+ In.UV1.zw));
    float texH = DecodeFloatRGB(tex2D(depthTextureSampler, In.UV0 + In.UV1.xy
+ In.UV1.zw));

    float4 color;
    color.rgb = 1.0f - pow((1.0f - 0.5f * abs(texA - texH)), 2.0f) * pow((1.0f
- 0.5f * abs(texC - texF)), 2.0f);
    color.a = 1.0f;


    return color;
  };

  technique ZEdgeMap
  {
    pass P0
    {
      VertexShader = compile vs_4_0 vtxSh();
      PixelShader = compile ps_4_0 zPxlSh();
    }
  };
```

With the Edge Map shaders in place, we will modify our **Main Program** to visualize the generated edge maps for various depth layers. Don't forget to comment out the previous sprite rendering snippet for visualizing the depth peeling.

First, we will make our DrawEdgeMap() function:

```
  private void DrawEdgeMap()
  {
    edgeEffect.CurrentTechnique = edgeEffect.Techniques["EdgeDraw"];
    testEffect.CurrentTechnique = testEffect.Techniques["ZEdgeMap"];

    edgeEffect.Parameters["modelTexture"].SetValue(testMap2);
    testEffect.Parameters["depthTexture"].SetValue(depthAndColorMap2);

    edgeEffect.Parameters["imageWidth"].SetValue((float)testMap2.Width);
    testEffect.Parameters["imageWidth"].SetValue((float)testMap2.Width);


    edgeEffect.Parameters["imageHeight"].SetValue((float)testMap2.Height);
    testEffect.Parameters["imageHeight"].SetValue((float)testMap2.Height);

    Matrix projection = Matrix.CreateOrthographicOffCenter(0, 800, 600, 0, 0,
```

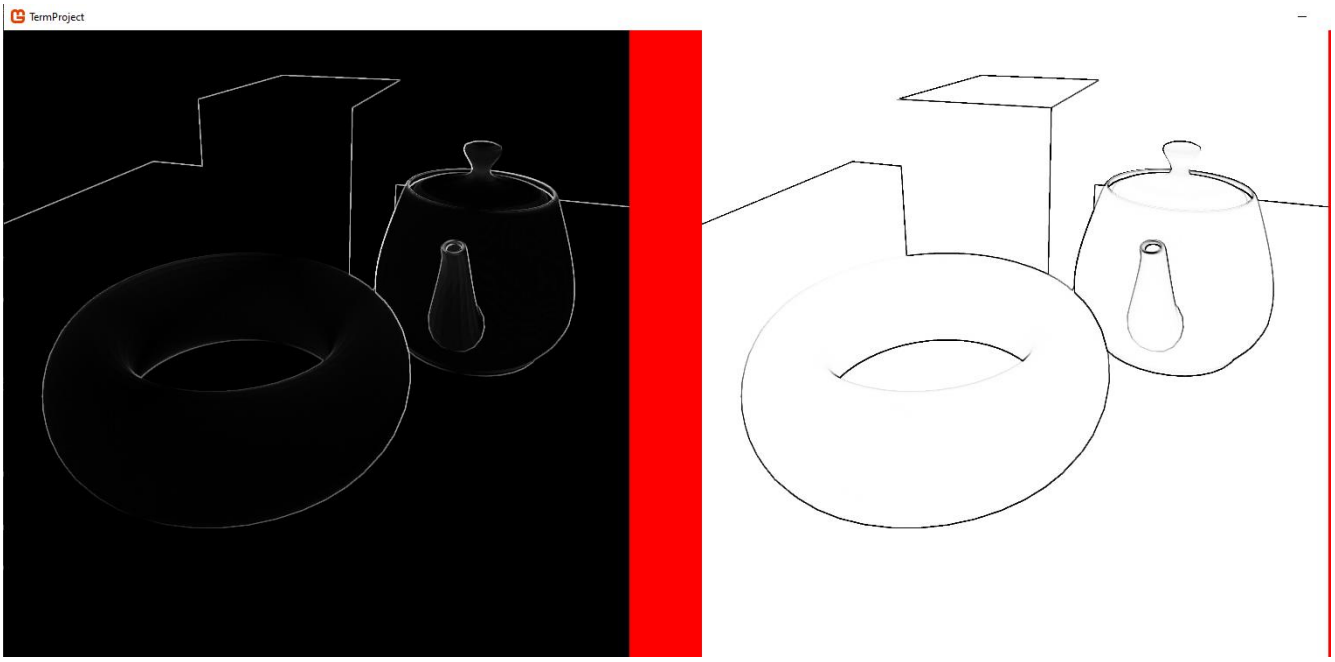
```
1);
    Matrix halfPixelOffset = Matrix.CreateTranslation(-0.5f, -0.5f, 0);

    edgeEffect.Parameters["MatrixTransform"].SetValue(halfPixelOffset *
projection);
    testEffect.Parameters["MatrixTransform"].SetValue(halfPixelOffset *
projection);

    GraphicsDevice.SetRenderTarget(null);
    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Red, 1.0f, 0);

    edgeMap = testMap2;
    edgeMap2 = testMap2;

    // *** We will comment the following code later, this is for visualization

    using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
    {
      sprite.Begin(0, null, null, null, null, testEffect);
      sprite.Draw(testMap2, Vector2.Zero, null, Color.White, 0, Vector2.Zero,
0.175f, SpriteEffects.None, 0);
      sprite.End();

      sprite.Begin(0, null, null, null, null, edgeEffect);
      sprite.Draw(edgeMap2, new Vector2(400, 0), null, Color.White, 0,
Vector2.Zero, 0.175f, SpriteEffects.None, 0);
      sprite.End();

    // ***
    }
  }
```

With the function in place, we can call it from the main Draw() method. We will call it once initially, and once again in the depth peeling loop

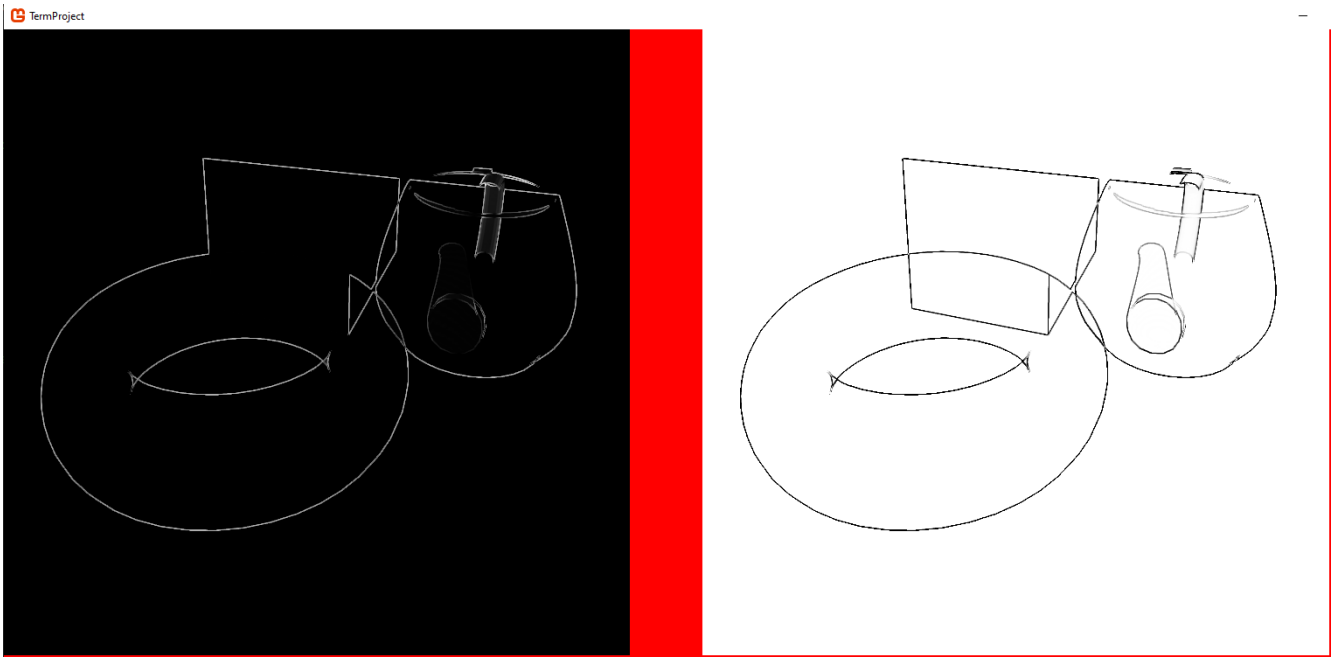
```
  ...
  DrawNormalMap();

  GraphicsDevice.SetRenderTarget(null);

  testMap = (Texture2D)normalRenderTarget;
  testMap2 = (Texture2D)normalRenderTarget;

  DrawEdgeMap();
  GraphicsDevice.SetRenderTarget(null);
  ...

  // *** Inside depth peeling loop

    ...
    if (i % 2 == 1)
```

```
    {
      depthAndColorMap2 = (Texture2D)depth2;
      testMap2 = (Texture2D)normal2;
    }

    else
    {
      depthAndColorMap2 = (Texture2D)depthRenderTarget;
      testMap2 = (Texture2D)normalRenderTarget;
    }

    DrawEdgeMap();

    GraphicsDevice.SetRenderTarget(null);
  }
```

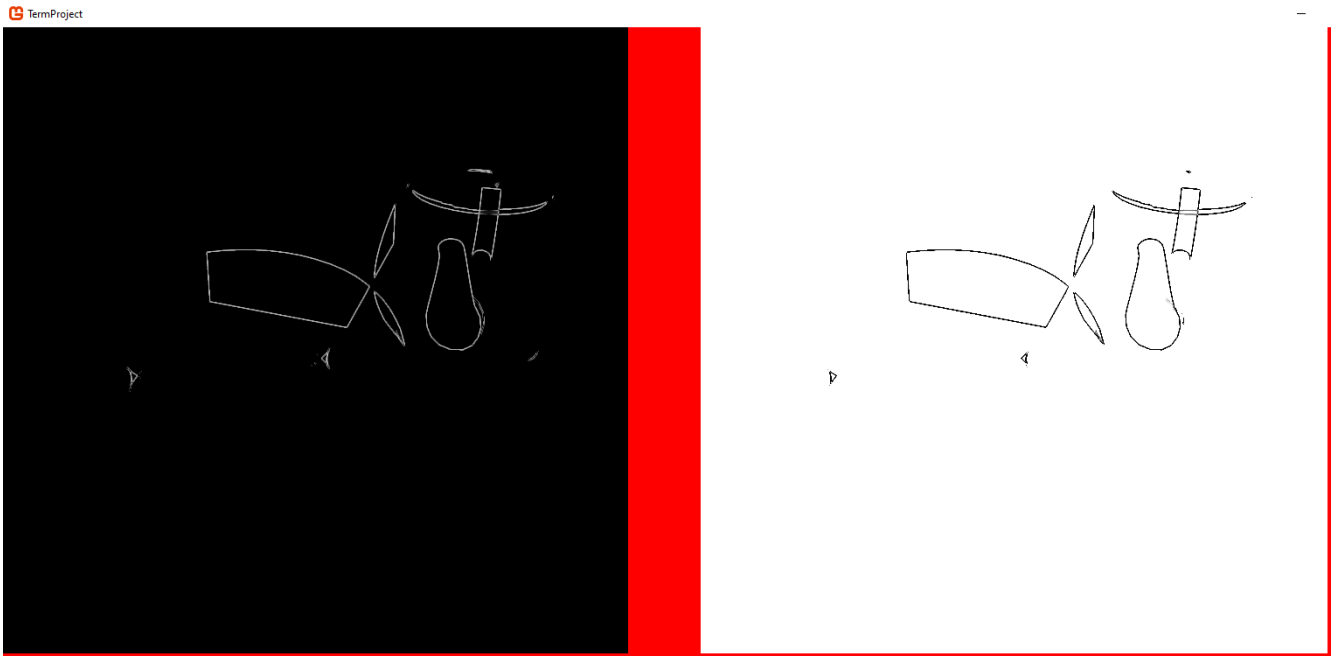
With that in place, we will be able to visualize the edge maps side-by-side. The first three layers are displayed below. Note how different edges are captured on the left side of the cube by the two algorithms and the two different input maps.



Layer 1



Layer 2



Layer 3

D. Compositing

Now we will write our final shader to help put the two edge maps together and create our final blueprint render. First, we must comment out the previous visualization code in the DrawEdgeMap() function and

replace it with this snippet to render it to our render targets:

```
    ...
    GraphicsDevice.SetRenderTarget(edgeMapRenderTarget);
    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

    using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
    {
      sprite.Begin(0, null, null, null, null, testEffect);
      sprite.Draw(testMap2, Vector2.Zero, null, Color.White, 0, Vector2.Zero,
0.175f, SpriteEffects.None, 0);
      sprite.End();
    }

    edgeMap = (Texture2D)edgeMapRenderTarget;

    GraphicsDevice.SetRenderTarget(edgeMap2RenderTarget);
    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

    using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
    {
      sprite.Begin(0, null, null, null, null, edgeEffect);
      sprite.Draw(testMap2, Vector2.Zero, null, Color.White, 0, Vector2.Zero,
0.175f, SpriteEffects.None, 0);
      sprite.End();
    }

    edgeMap2 = (Texture2D)edgeMap2RenderTarget;
  }
```

Next, we will write our shader file named **Composite.fx** in which we will include two fragment shader programs for two techniques. The first will take the normal edge map values and put it into our final rendering's RGB values as well as the depth edge map into the alpha channel. The second will blend this result with a blueprint background so it looks draw on blueprint grid paper. This part can be reworked into backgrounds of your choice.

The variables and structures needed resemble our previous edge map shaders, once again we need the `DecodeFloatRGB` helper function:

```
  float4x4 MatrixTransform;
  texture2D normalEdgeTexture;
  texture2D depthEdgeTexture;
  float imageWidth;
  float imageHeight;

  float DecodeFloatRGB(float4 color)
  {
    const float3 byte_to_float = float3(1.0, 1.0 / 256, 1.0 / (256 * 256));
```

```
  return dot(color.xyz, byte_to_float);
}

sampler depthEdgeTextureSampler : register(s0) = sampler_state
{
  Texture = <depthEdgeTexture>;
  ADDRESSU = CLAMP;
  ADDRESSV = CLAMP;
};

sampler normalEdgeTextureSampler : register(s1) = sampler_state
{
  Texture = <normalEdgeTexture>;
  ADDRESSU = CLAMP;
  ADDRESSV = CLAMP;
};

struct VS_OUTPUT
{
  float4 Pos : POSITION;
  float2 UV0 : TEXCOORD0;
  float4 UV1 : TEXCOORD1;
};

VS_OUTPUT vtxSh(float4 inPos : POSITION, float2 inTex : TEXCOORD0)
{
  VS_OUTPUT Out;
  Out.Pos = mul(inPos, MatrixTransform);
  Out.UV0 = inTex;
  Out.UV1 = float4(2 / imageWidth, 0, 0, 2 / imageHeight);

  return Out;
};
```

Next, we will add the fragment shaders and our two techniques. Note that the finalFragmentComposite() fragment shader can be rewritten in different ways to combine with different types of backgrounds so play around with it to generate your own results.

```
float4 pxlSh(VS_OUTPUT In) : COLOR
{
  float4 ntex = tex2D(normalEdgeTextureSampler, In.UV0);
  float4 ztex = tex2D(depthEdgeTextureSampler, In.UV0);

  float4 color;

  color.rgb = 0.6 * (1.0 - ntex.rgb);

  if (color.r <= 0.01f && color.g <= 0.01f && color.b <= 0.01f)
  {
    color.a = 0.0f;
  }
```

```
    else
    {
      color.a = DecodeFloatRGB(ztex);
    }

    return color;
  };

  float4 finalFragmentComposite(VS_OUTPUT In) : COLOR
  {
    float4 bgtex = tex2D(normalEdgeTextureSampler, In.UV0);
    float4 fgtex = tex2D(depthEdgeTextureSampler, In.UV0);

    float4 color;

    color.rgb = fgtex.rgb;
    color.rgb = (1.0f - color.rgb) * bgtex.rgb;
    color.a = fgtex.a;

    return color;
  };

  technique ComposeEdgeMaps
  {
    pass P0
    {
      VertexShader = compile vs_4_0 vtxSh();
      PixelShader = compile ps_4_0 pxlSh();
    }
  };

  technique DrawBG
  {
    pass P0
    {
      VertexShader = compile vs_4_0 vtxSh();
      PixelShader = compile ps_4_0 finalFragmentComposite();
    }
  };
```

Then, we can make our DrawComposite() and DrawBG() functions using these shader techniques. We will render them directly to render targets this time to obtain our final result.

```
  private void DrawComposite()
  {
    compositeEffect.CurrentTechnique =
compositeEffect.Techniques["ComposeEdgeMaps"];

    compositeEffect.Parameters["normalEdgeTexture"].SetValue(edgeMap2);
    compositeEffect.Parameters["depthEdgeTexture"].SetValue(edgeMap);

    compositeEffect.Parameters["imageWidth"].SetValue((float)edgeMap.Width);
```

```
    compositeEffect.Parameters["imageHeight"].SetValue((float)edgeMap.Height);

    Matrix projection = Matrix.CreateOrthographicOffCenter(0, 800, 600, 0, 0,
1);
    Matrix halfPixelOffset = Matrix.CreateTranslation(-0.5f, -0.5f, 0);

    compositeEffect.Parameters["MatrixTransform"].SetValue(halfPixelOffset *
projection);
    compositeEffect.CurrentTechnique.Passes[0].Apply();

    GraphicsDevice.SetRenderTarget(compositeRenderTarget);

    using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
    {
      sprite.Begin(0, null, null, null, null, compositeEffect);
      sprite.Draw(edgeMap2, Vector2.Zero, null, Color.White, 0, Vector2.Zero,
0.6f, SpriteEffects.None, 0);
      sprite.End();
    }
  }


  private void DrawBG()
  {
    compositeEffect.CurrentTechnique = compositeEffect.Techniques["DrawBG"];

    compositeEffect.Parameters["normalEdgeTexture"].SetValue(bgTexture);
    compositeEffect.Parameters["depthEdgeTexture"].SetValue(compositeEdgeMap);

    compositeEffect.Parameters["imageWidth"].SetValue((float)bgTexture.Width);

compositeEffect.Parameters["imageHeight"].SetValue((float)bgTexture.Height);

    Matrix projection = Matrix.CreateOrthographicOffCenter(0, 800, 600, 0, 0,
1);
    Matrix halfPixelOffset = Matrix.CreateTranslation(-0.5f, -0.5f, 0);

    compositeEffect.Parameters["MatrixTransform"].SetValue(halfPixelOffset *
projection);

    compositeEffect.CurrentTechnique.Passes[0].Apply();

    GraphicsDevice.SetRenderTarget(null);

    GraphicsDevice.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.White, 1.0f, 0);

    layerBlend = compositeEdgeMap;

    using (SpriteBatch sprite = new SpriteBatch(GraphicsDevice))
    {
      sprite.Begin(0, null, null, null, null, compositeEffect);
      // *** Try changing the main texture passed to the shader to see what
results you will get
```

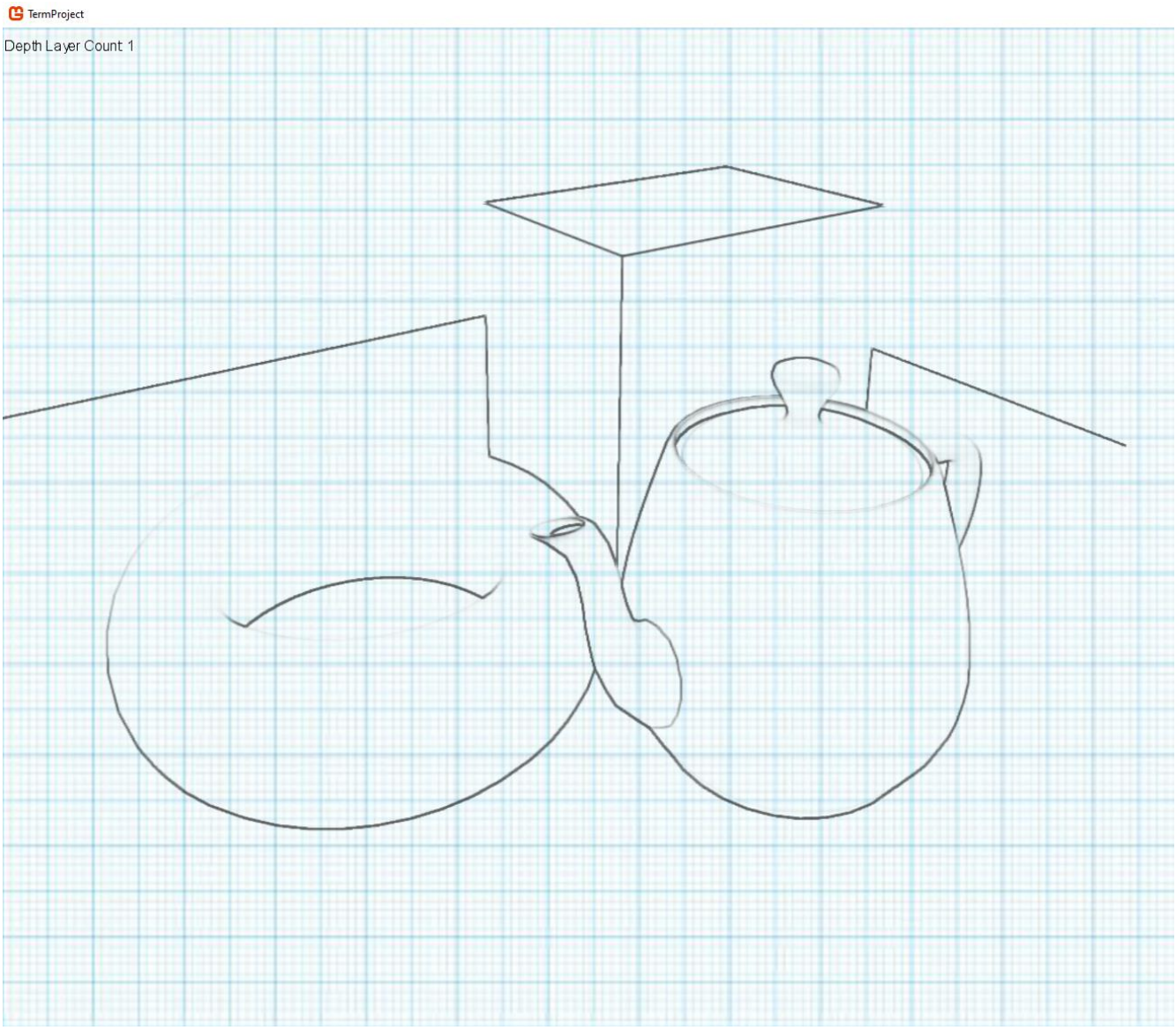
```
      //sprite.Draw(layerBlend, Vector2.Zero, null, Color.White, 0,
Vector2.Zero, 0.28f, SpriteEffects.None, 0);
      sprite.Draw(bgTexture, Vector2.Zero, null, Color.White, 0, Vector2.Zero,
1.5f, SpriteEffects.None, 0);
      sprite.End();
    }
  }
```

Finally, we can call the functions in our main Draw() method. Just like last time, we will call the DrawComposite() method once initially, and then once again in the loop for depth peeling. This time, we keep drawing on top of the render target without clearing so that we can get as many layers as we need composited on top of each other, before calling our DrawBG() function to add the background.

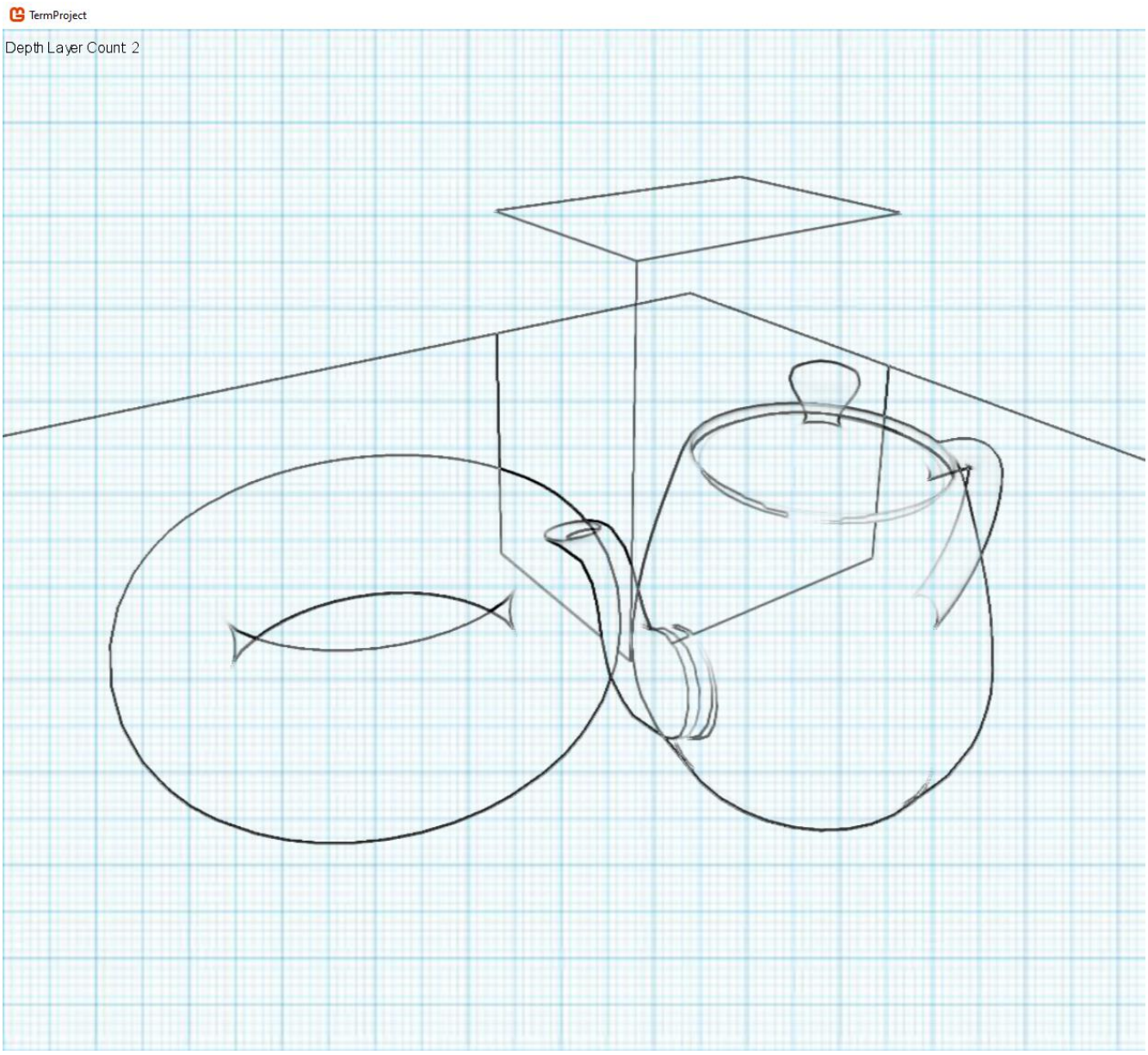
```
  ...
  DrawNormalMap();

  GraphicsDevice.SetRenderTarget(null);

  testMap = (Texture2D)normalRenderTarget;
  testMap2 = (Texture2D)normalRenderTarget;

  DrawEdgeMap();
  GraphicsDevice.SetRenderTarget(null);
  DrawComposite();


  // *** Inside depth peeling loop

    ...
    if (i % 2 == 1)
    {
      depthAndColorMap2 = (Texture2D)depth2;
      testMap2 = (Texture2D)normal2;
    }

    else
    {
      depthAndColorMap2 = (Texture2D)depthRenderTarget;
      testMap2 = (Texture2D)normalRenderTarget;
    }

    DrawEdgeMap();

    GraphicsDevice.SetRenderTarget(null);

    DrawComposite();
  }

  // *** Outside the loop, before setting the textures to null

  compositeEdgeMap = (Texture2D)compositeRenderTarget;
  DrawBG();
```

```
depthAndColorMap = null;
testMap = null;
depthAndColorMap2 = null;
testMap2 = null;
edgeMap = null;
edgeMap2 = null;
compositeEdgeMap = null;
layerBlend = null;
```

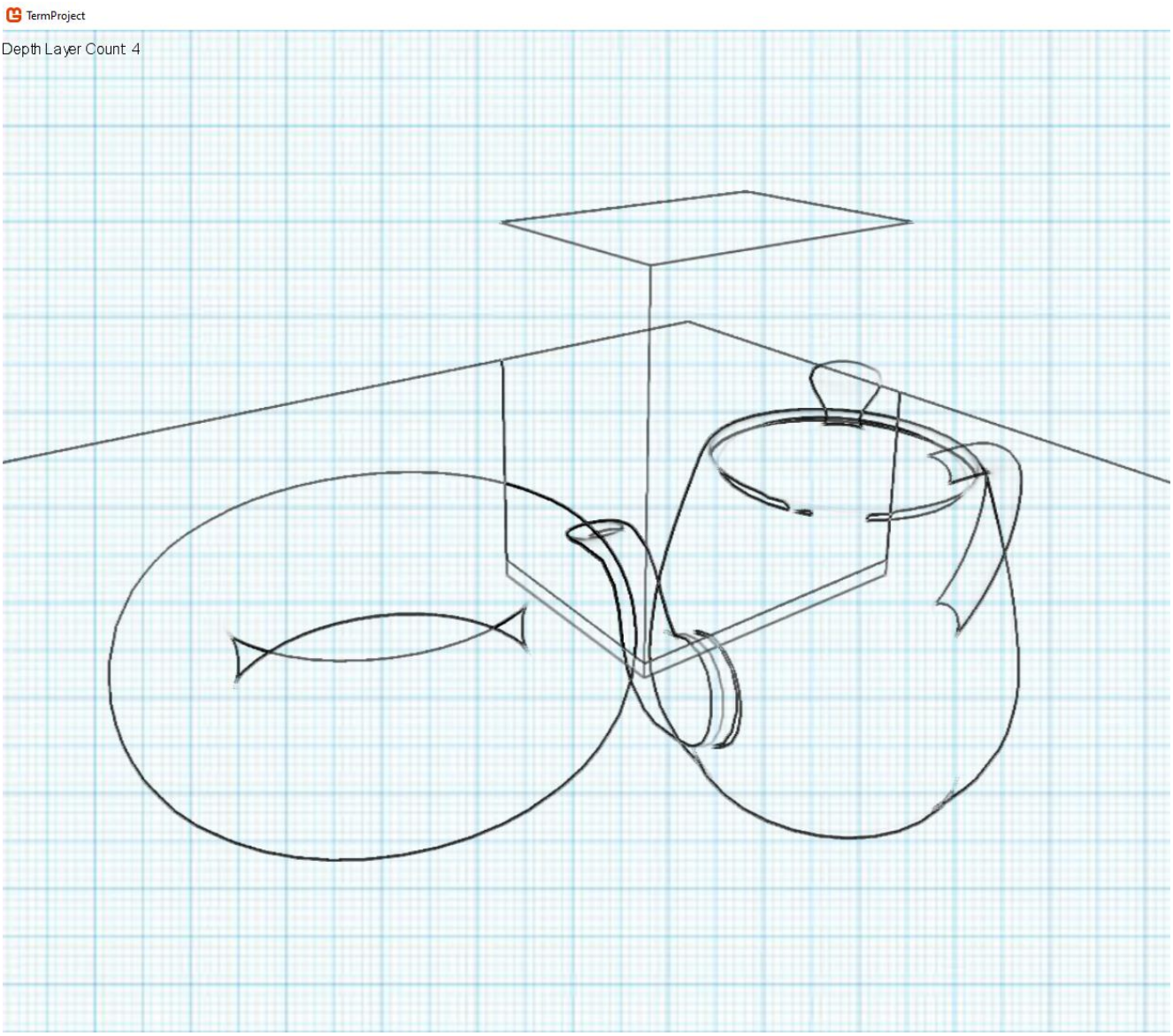
With that, you should be able to get your final blueprint render that would look something like this. Shown below is the final render with different numbers of depth layers blended together.



One layer Blueprint



Two Layer Blueprint



Four Layer Blueprint

E. Main Exercise

Now you can play with the blueprint shader and come up with your own renders. Some exercises to be implemented on your own are:

- Write a function to produce a render with all important layers (stop the rendering when the depth peeling no longer produces any new colors).
- Change the background to resemble more traditional blueprints, with the blue paper and white ink.
- There is a lot of lost edge information when combining the edge maps in the RGB and alpha channels as suggested, try combining them in your own way to capture more edges and come up with a more aesthetic result.

***** IMPORTANT *****

Complete the exercise in E section, and submit a zipped file including the solution (.sln) file and the project folders to course online site. The submission item is located in the "**Quiz and Lab**" section. Each lab has **10 points**. If you complete the exercise in class time, the full points will be assigned. The late submission is accepted just before the next class with 2 points reductions, because the solution is demonstrated in the next class.