Postprocessing in XNA

Prof. Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology
Giant warning

The code in these slides has not been tested.

There may be bugs and/or misconceptions.
Typical XNA setup code

```csharp
GraphicsDeviceManager graphics = new GraphicsDeviceManager(this);

ContentManager content = new ContentManager(Services);

GraphicsDevice device = graphics.GraphicsDevice;
```
Setup for postprocessing

SpriteBatch mySpriteBatch;
RenderTarget2D myRenderTarget;
Texture2D beforeProc;
Effect ppEffect;

ppEffect =
content.Load<Effect>("Content\Effects\CoolEffect");

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Creating the rendertarget

myRenderTarget = 
  new RenderTarget2D(device,
        device.Viewport.Width,
        device.Viewport.Height,
        1, // number of mipmap levels
        device.DisplayMode.Format
  // a SurfaceFormat)

Vector2 offset = new Vector2(0,1 / device.Viewport.Height);
Creating the rendertarget (advanced)

```csharp
myRenderTarget = new RenderTarget2D(device, 
device.Viewport.Width, 
device.Viewport.Height, 
1, // number of mipmap levels
device.DisplayMode.Format, // a SurfaceFormat
device.PresentationParameters.MultiSampleType,
device.PresentationParameters.MultiSampleQuality)
```

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
## SurfaceFormat enumeration

<table>
<thead>
<tr>
<th>Member name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha8</td>
<td>(Unsigned format) 8-bit alpha only.</td>
</tr>
<tr>
<td>Bgr233</td>
<td>(Unsigned format) 8-bit BGR texture format using 2 bits for blue, 3 bits for green, and 3 bits for red.</td>
</tr>
<tr>
<td>Bgr24</td>
<td>(Unsigned format) 24-bit BGR pixel format with 8 bits per channel.</td>
</tr>
<tr>
<td>Bgr32</td>
<td>(Unsigned format) 32-bit BGR pixel format, where 8 bits are reserved for each color.</td>
</tr>
<tr>
<td>Bgr444</td>
<td>(Unsigned format) 16-bit BGR pixel format using 4 bits for each color.</td>
</tr>
<tr>
<td>Bgr555</td>
<td>(Unsigned format) 16-bit BGR pixel format where 5 bits are reserved for each color.</td>
</tr>
<tr>
<td>Bgr565</td>
<td>(Unsigned format) 16-bit BGR pixel format with 5 bits for blue, 6 bits for green, and 5 bits for red.</td>
</tr>
<tr>
<td>Bgra1010102</td>
<td>(Unsigned format) 32-bit pixel format using 10 bits each for blue, green, and red, and 2 bits for alpha.</td>
</tr>
<tr>
<td>Bgra2338</td>
<td>(Unsigned format) 16-bit BGRA format using 2 bits for blue, 3 bits each for red and green, and 8 bits for alpha.</td>
</tr>
</tbody>
</table>

Direct3D/XNA SurfaceFormat Conversions

<table>
<thead>
<tr>
<th>Direct3D Surface Format</th>
<th>SurfaceFormat equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Floating Point</strong></td>
<td></td>
</tr>
<tr>
<td>Float32</td>
<td>D3DFMT_R32F</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_G32R32F</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_A32B32G32R32F</td>
</tr>
<tr>
<td>Float16</td>
<td>D3DFMT_R16F</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_G16R16F</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_A16B16G16R16F</td>
</tr>
<tr>
<td><strong>Unsigned Normalized</strong></td>
<td></td>
</tr>
<tr>
<td>64 bpp</td>
<td>D3DFMT_A16B16G16R16</td>
</tr>
<tr>
<td>32 bpp</td>
<td>D3DFMT_A8R8G8B8</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_X8R8G8B8</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_A8B8G8R8</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_X8B8G8R8</td>
</tr>
<tr>
<td></td>
<td>D3DFMT_A2R10G10B10</td>
</tr>
</tbody>
</table>

Rendering the preprocessed scene

device.SetRenderTarget(0, myRenderTarget);
// On Xbox 360, first argument must be set to zero
// since you only can set one Render Target on the 360

// PUT CODE TO DRAW STUFF HERE

device.ResolveRenderTarget(0);
// needed on Xbox 360 to copy eDRAM contents to main RAM

beforeProc = myRenderTarget.GetTexture();

// Set render target to the usual backbuffer
device.SetRenderTarget(0, null);

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Setting up the postprocessing effect

// Next sort of line only seems necessary if you have
// more than one technique
myEffect.CurrentTechnique = effect.Techniques[“BlurEffect”];

myEffect.Parameters[“offset”].SetValue(offset);

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Drawing the processed scene

device.Clear(Color.Black);
myEffect.Begin();
mySpriteBatch.Begin(SpriteBlendMode.None,
SpriteSortMode.Immediate, SpriteStateMode.None);
EffectPass pass = effect.CurrentTechnique.Passes[0]
pass.Begin();
mySpriteBatch.Draw(beforeProc, Vector2.Zero,
    Color.White);
pass.End();
mySpriteBatch.End();
myEffect.End();

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Just need a pixel shader

// CoolEffect.fx
sampler textureSampler;
float2 offset;

float4 threewayBlurPS(texCoord : TEXCOORD0) : COLOR0
{
    float4 color =
        (tex2D(textureSampler, texCoord) +
        tex2D(textureSampler, texCoord + offset) +
        tex2D(textureSampler, texCoord - offset)) / 3;
    return color;
}

technique BlurEffect {
    pass P0 {
        PixelShader = compile ps_2_0 threewayBlurPS();
    }
}

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Just need a pixel shader

sampler textureSampler

is sort of implicitly

sampler textureSampler : register(S0);

mySpriteBatch.Draw(beforeProc, Vector.Zero,
                   Color.White);

was sort of doing this somewhere:

device.Textures[0] = beforeProc;

Based on discussion on p. 277-281 of Chad Carter, “Microsoft XNA Unleashed,” 2008
Multiple textures

• In your C# code:

```csharp
graphics.GraphicsDevice.Textures[0] = firstTexture;
```

• In your shader code:

```csharp
sampler firstSampler : register(s0);
sampler secondSampler : register(s1);
```

Rendertarget semantics on Windows

• If not multisampling, a single area of video memory can be used for rendering or as a texture
  – Resolve is a no-op
  – Contents of rendertarget not lost
• If multisampling, need large area to render into and small area to copy into
  – Resolve copies, downsampling as it goes
  – Contents of both buffers not lost

Rendertarget semantics on Xbox 360

• Xenos GPU renders into only one physical rendertarget: 10 MB eDRAM
  – Cannot texture from eDRAM
  – Cannot render into main 512M RAM

• Must “resolve” to copy rendering in eDRAM back to main memory
  – Hardware designed to make this fast
  – Note from Shawn: “It is less obvious why the resolve call needs to clear the special memory, but apparently there is a performance gain from doing this: I don't pretend to understand why but I'm not going to complain as long as this keeps my Xbox running as fast as it does!”

Shawn Hargraves’ chart

<table>
<thead>
<tr>
<th></th>
<th>Rendertarget and texture share video memory (resolve is a no-op)</th>
<th>Changing rendertarget destroys existing contents</th>
<th>Resolve destroys existing contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows (not multisampled)</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Windows (multisampled)</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Xbox</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Shawn’s advice: “If you play it safe by always assuming your buffer contents will be lost when you call SetRenderTarget or Resolve, that code will work consistently on all platforms.”