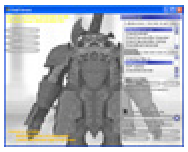
PostProcess Sample

Collapse All

This sample demonstrates some interesting image-processing effects that can be achieved interactively. Traditionally, image processing takes a significant amount of processor power on the host CPU, and is usually done offline. With pixel shaders, these effects can now be performed on the hardware more efficiently, allowing them to be applied in real time.

Note that the techniques shown here require pixel shader 2.0 and floating-point textures. Thus, not all cards support all of the postprocessing techniques.



Path

Source	*SDK root*\Samples\C++\Direct3D\PostProcess
Executable	*SDK root*\Samples\C++\Direct3D\Bin\x86 or x64\PostProcess.exe

Post Processing Effects

Image-processing techniques can improve the quality and realism of a scene, as well as achieve some specific effects, such as scene-wide blurring. In Direct3D, image processing is usually performed as a postprocess after the rendering is completed. An application first renders its scene onto a texture, then processes the texture with a pixel shader to produce a different image with the enhancement or alteration. The use of pixel shaders makes this operation very efficient, so it can be performed in real time.

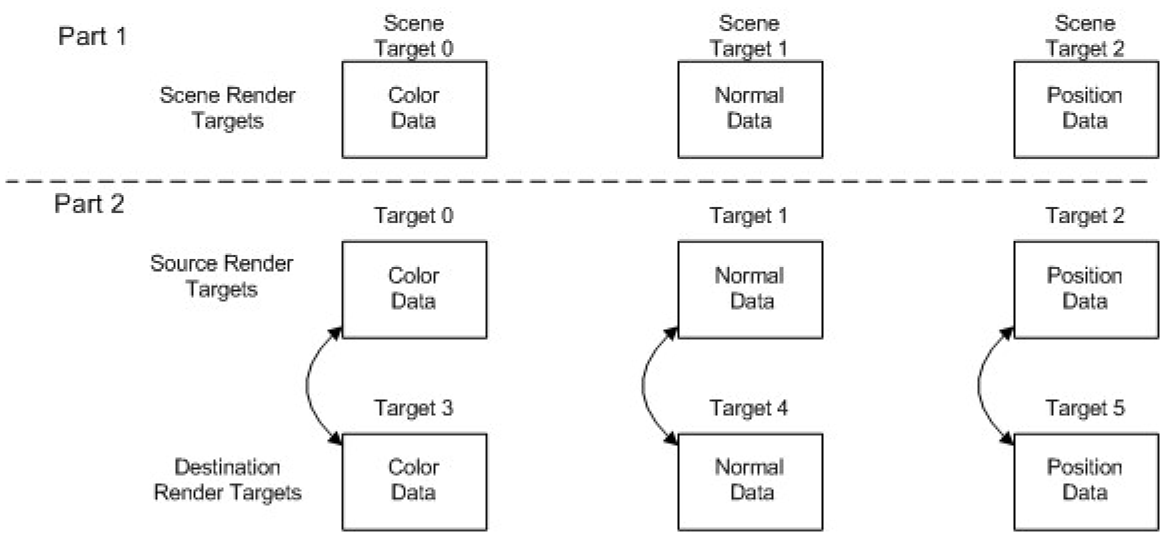
Postprocess effects can achieve interesting results particularly when a sequence of one or more postprocesses are applied to the original scene. Four such postprocess effects are included in this sample:

- Blur - Combine a pixel color with surrounding pixel colors to blur the focus.
- Bloom - Magnify bright spots in the image, as well as the area around the bright spots.
- Depth of Field - Blur a pixel based on the pixel's depth or z value. Pixels very far away or very close get blurred.
- Edge Glow - Use object normals to find an object's edge, and then use blur to help create a glow on the edges.

How the Sample Works

The scene contains a mesh object that can either be textured or environment-mapped, and a background in the form of a skybox cube. The sample can be divided into two parts: the first part renders scene data into three source render targets, and the second part uses pixel shaders to implement 2D effects into three destination render targets.

Part 1 of the sample renders per-pixel data into three scene render targets: 0, 1, and 2.

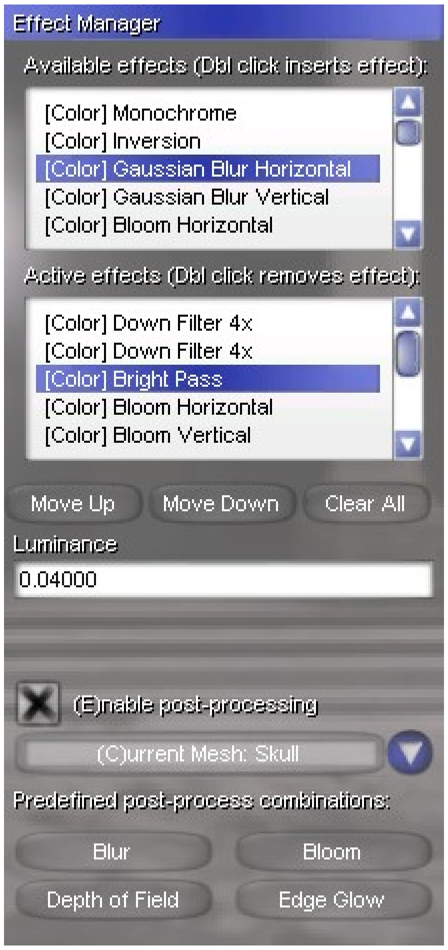


Part 2 of the sample may use source material from the three scene render targets and the source render targets (0,1,2), and renders a full-screen quad with a pixel shader into the destination render targets (3,4,5). The six render targets in Part 2 are organized into three swap chains indicated by the curved arrows in the figure. When a pixel shader completes, the destination render target is swapped with its respective source render target. As a result, this sample can be used to cascade postprocessed effects. For instance, [Color] Monochrome is a postprocess effect that samples the color source texture (target 3), converts it to grayscale, then outputs the result to the other texture in the swap chain for the color channel (target 6).

Not all postprocess effects require all three render targets. However, using three render targets provides more options for the postprocess effects. Most of the time, the color render target gets updated and replaced. However, allowing other textures to be replaced gives more flexibility. Creative postprocess effects that change the normal and position textures are possible with the sample's architecture. The number of postprocess effects that can be rendered is only limited by system resources.

Running the Sample

When the sample is run, the user sees a typical Direct3D sample window and a Select Post Process Effects dialog that looks like this:



Available effects (Dbl click inserts effect):

[Color] Monochrome
[Color] Inversion
[Color] Gaussian Blur Horizontal
[Color] Gaussian Blur Vertical
[Color] Bloom Horizontal

Active effects (Dbl click removes effect):

[Color] Down Filter 4x
[Color] Down Filter 4x
[Color] Bright Pass
[Color] Bloom Horizontal
[Color] Bloom Vertical

Luminance: 0.04000

[✓] (E)nable post-processing

(C)urrent Mesh: Skull

Predefined post-process combinations:

At the top of the dialog are two list controls. The first list displays the post-process effects that are available. The second list displays the active effects. After a scene is rendered, the active post-process effects are applied to the scene image in the order listed before displaying the result to the user. When the user modifies the active list, the visual change to the scene is immediately reflected.

Below the lists are buttons for changing the order the effects are applied. The button also clears the list. In the parameter section, some effects (such as bright pass) have parameters that the user can tweak to generate different visual appearances. The user can highlight an active post-process to adjust its parameters, if it has any.

At the bottom is the predefined post-process combinations that demonstrate how the different post-process effects can be used together to achieve even more interesting results.

The Effect Files

Each postprocess is implemented in one D3DX effect file. One technique, called PostProcess, exists in each file that is selected when its postprocess effect is needed. The technique can have one or more passes. Each pass specifies a pixel shader that does the actual work of image processing when invoked. The technique can have an integer annotation, named nRenderTarget, that specifies to which texture the pixel shader output should go. Assign it a value of 0, 1, or 2 to indicate that the pixel shader output is to be written to the color, normal, or position texture, respectively. If this annotation is missing, the output is assumed for the color texture.

The passes can optionally have two floating-point annotations named fScaleX and fScaleY. These two values specify the amount that the output image is scaled before written to the render target. After a postprocess scales the image to a size smaller than the render target texture, all subsequent postprocesses work with that image size until it is scaled again. This can have a significant performance boost, because the pixel shaders have fewer pixels to process. A common postprocessing practice is to scale down the image, apply postprocess effects to the sub-area of the texture, then scale up the result image to the original size. The performance gain from having fewer pixels to process usually far exceeds the penalty of performing the extra steps of scaling.

In the effect file, there are six texture objects with samplers from which the postprocess pixel shader will sample. g_txSrcColor, g_txSrcNormal, and g_txSrcPosition contain the colors, normals, and camera space positions from the result of the previous postprocess. g_txSceneColor, g_txSceneNormal, and g_txScenePosition are the result from the scene renderer. For the first postprocess in the sequence, these two sets of textures will be identical.

Often, the postprocess pixel shader will need to sample a source texture multiple times for a single destination pixel, each at a different location near the destination pixel. When this is the case, the effect file declares a kernel which is an array of texel offsets (float2) from the destination pixel. The effect file then defines another equally-sized array of float2. This second array has a string annotation, named ConvertPixelsToTexels, that contains the name of the first kernel array. When the sample loads the effect file and sees this annotation, it will translate the offsets from pixel coordinates to texel coordinates, which are compatible with the texture sampling functions such as **tex2D(s, t) (DirectX HLSL)**. When the source texels are sampled, the offsets in the kernel are added to the texture coordinates to obtain the texel values near the destination pixel.

Some effects use parameters that the user can adjust. For instance, the Bright Pass effect has parameters that the user can tweak to give different visual appearances. Each PostProcess technique can define one or more annotations for this purpose. The maximum number of parameters supported by a single technique is four.

Annotation Name	Type	Description
Parameter0	string	Contains the name of the variable that the user can adjust.
Parameter0Def	float4	Default value for Parameter0. This is always a float4. If fewer than four floats are needed, use Parameter0Size to specify it.
Parameter0Size	int	Specifies how many components of Parameter0 are used. If this is 2, then only the first two components (x and y) are used.
Parameter0Desc	string	A string that describes this parameter.

The Sample Code

In the sample code, each postprocess effect type is represented by a CPostProcess class. This class encapsulates the effect files and provides a mechanism to activate the effect technique and change its parameters. At initialization, an array of CPostProcess objects is created, one for each postprocess effect file.

In the active postprocess effect list, each entry is represented by the CPProcInstance class. This class merely contains an index to the CPostProcess array to indicate the effect and the parameters to apply to this postprocess.

The Render method in the sample first renders the scene onto a set of render target textures. Three render targets are used simultaneously here for color, normal, and position. After that is complete, it calls PerformPostProcess, which handles every aspect of applying postprocesses to the scene image. After PerformPostProcess returns, the code calls **IDirect3DDevice9::StretchRect** to copy the postprocessing result to the device backbuffer so that the user can see the final image.

The postprocess work is done in two functions: PerformPostProcess and PerformSinglePostProcess.

The first thing that PerformPostProcess does is to set up a quad that defines how the rendering is done for the postprocess. Next, it creates a vertex buffer containing the quad created above. Then, it simply iterates through the active effect list and calls PerformSinglePostProcess for each entry in the list.

PerformSinglePostProcess applies a postprocess effect once to a render target texture. First, it initializes the textures and parameters for the effect. Then, it goes into a loop to render all passes of the technique. For each pass, it checks the scaling values and makes the necessary adjustment to the quad. After that, it renders the quad onto the render target texture with a **IDirect3DDevice9::DrawPrimitive** call. Finally, it swaps the render target texture with the corresponding source texture so that the next postprocess will have the correct textures from which to read and to which to write.

The following section lists the postprocesses included by this sample with a description of what each does.

Post Processes

Name	Implementation File	Description
[Color] Monochrome	PP_ColorMonochrome.fx	Converts the color image to monochrome.
[Color] Gaussian Blur Horizontal	PP_ColorGBlurH.fx	Performs Gaussian blur horizontally. The pixel shader achieves this by sampling several texels to the left and right of the source texture coordinate and performing a weighted average among the samples.
[Color] Gaussian Blur Vertical	PP_ColorGBlurV.fx	Performs Gaussian blur vertically. The pixel shader achieves this by sampling several texels to the top and bottom of the source texture coordinate and performing a weighted average among the samples.
[Color] Bloom Horizontal	PP_ColorBloomH.fx	Performs Gaussian blur horizontally and magnifies the image at the same time.
[Color] Bloom vertical	PP_ColorBloomV.fx	Performs Gaussian blur vertically and magnifies the image at the same time.
[Color] Bright Pass	PP_ColorBrightPass.fx	Performs a bright pass on the image. The Luminance is adjustable.
[Color] Tone Mapping	PP_ColorToneMapping.fx	Performs tone mapping on the image. The Luminance is adjustable.
[Color] Edge Detection	PP_ColorEdgeDetect.fx	Performs edge detection based on the color texture. For each pixel shader pass, the shader samples the texel at texcoord and its four neighboring texels. Then, it computes the differences between the texel and each of its neighbors. Finally, the shader sums the four absolute values of the differences and outputs it. The end result is that if a pixel has significantly different color from its neighbors, the postprocess result pixel will look bright.
[Color] Down Filter 4x	PP_ColorDownFilter4.fx	Performs a down-filtering pass. The dimension to work with scales to 0.25 times the original. A down filter shrinks the image.
[Color] Up Filter 4x	PP_ColorUpFilter4.fx	Performs an up-filtering pass. The dimension to work with scales to 4 times the original. An up filter enlarges the image.
[Color] Combine	PP_ColorCombine.fx	Adds the result from previous postprocesses and the original scene together and outputs the sum.
[Color] Combine 4x	PP_ColorCombine4.fx	Adds the result from previous postprocesses and the original scene together and outputs the sum, as well as performing a 4x up-filtering pass.
[Normal] Edge Detection	PP_NormalEdgeDetect.fx	Performs an edge detection similar to [Color] Edge Detection except that this technique compares the normals of the neighboring texels instead of colors. An advantage of this method over the colored-based one is that texture patterns will not be treated as edges.
DOF Combine	PP_DofCombine.fx	Performs a linear interpolation between the result texel from previous postprocesses and the original image based on the z coordinate at the pixel. A vector parameter representing the focal plane is used to determine the weights from the two source textures.
Normal Map	PP_NormalMap.fx	Takes the normal (texel on the normal texture) at the pixel and outputs it as the color. This is simply provided as a way of visualizing normals in a scene.

Position Map	PP_PositionMap.fx	Takes the position (texel on the position texture) at the pixel and outputs the z value as the color. This is simply provided as a way of visualizing depth information in a scene.

Blur

Blur is a very common image-based effect, and it is also rather straightforward. First, a down filter is performed to reduce the number of pixels that will be drawn (and therefore improve performance). Then, one horizontal blur and one vertical blur are performed. If more blurring is desired, repeat the blur effects. Finally, an up filter is performed to get the result image to the original size.

Bloom

The visual effect that bloom achieves is that bright spots in a scene get magnified and brighten the area around those spots. In order to achieve this, two passes of down filter are first performed. Because bringing out the bright spots does not require much detail, two passes of down filter boost the performance without sacrificing quality. Next, a bright pass is done. This postprocess picks up the areas in the image that are above a certain threshold, makes them white, then makes the rest of the image black. After that, two alternating passes each of horizontal bloom and vertical bloom are performed. Finally, two passes of up filter are performed, then the result is combined with the original image. Note that the second up filter pass and the combine are done in a single effect, [Color] Combine 4x.

Depth of Field

The depth of field (DOF) effect blurs the pixel of the image based on the z distance at that pixel. The steps that are needed to perform this are identical to that for the blur effect, except that a DOF Combine effect is added to the end of the sequence. Before the DOF combine, there are two textures available from which to sample: the blurred image from previous postprocesses and the original unblurred image. The DOF Combine examines the z coordinate at each pixel and computes a color by linearly interpolating between the two textures.

Edge Glow

This effect is very similar to bloom in concept, which is to take an image, make a part of it colored and everywhere else black, then combine with the original image. In this case, the area where it is emphasized is around the edges in the scene. First, a normal-based edge detection is performed. This produces white lines along the edges in the image (where normals change dramatically between neighboring pixels). Then, a blur effect is performed, by down-filtering, blurring, and up-filtering. Finally, the result is combined with the original image to produce the effect of glow along the edges. Note that the up-filtering and the final combination are done in a single effect.