Image Processing with 1.4 Pixel Shaders in Direct3D

Jason L. Mitchell

Introduction

With the ability to perform up to 16 ALU operations and 12 texture accesses in a single pass, Direct3D's pixel pipeline has begun to look like a powerful programmable 2D image processing machine. In this article, we will outline several techniques that utilize DirectX 8.1's 1.4 pixel shaders to perform 2D image processing effects such as *blurs*, *edge detection*, *transfer functions*, and *morphology*, and will point out applications of these 2D image operations to 3D game graphics. For an introduction to the structure and capabilities of 1.4 pixel shaders, please refer back to Wolfgang Engel's introductions to vertex and pixel shader programming.

As shown here and in "Non-Photorealistic Rendering with the Pixel and Vertex Shaders," post-processing of 3D frames and a general image-savvy mindset are fundamental to effectively utilizing 3D graphics processors to generate a variety of photorealistic and non-photorealistic effects [Saito90]. As shown in Figure 1, filtering an image with a GPU is done by using the image as a texture and drawing a screen-aligned quadrilateral into the *render target* (either the back buffer or another texture).

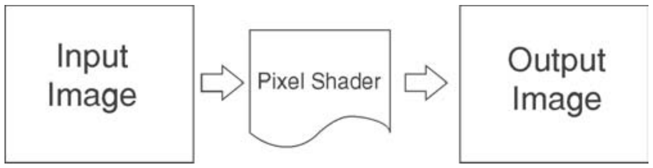

Figure 1: Using a pixel shader for image processing by rendering from one image to another

This results in the pixels of the render target being filled with the results of the pixel shader. With a very simple pixel shader that samples the source image and passes it directly to the destination without modification, this is equivalent to a Blt. In the simple Blt case, one source pixel determines the color of one destination pixel. In addition to merely copying the input image data, this operation can be useful for creating black-and-white or sepia tone images from the source image. We will illustrate these simple effects and more general transfer functions (like a heat signature) in the first part of this article.

Image processing becomes especially powerful when the color of the destination pixel is the result of computations done on multiple pixels sampled from the source image. In this case, we use a *filter kernel* to define how the different samples from the source image interact. Most of this chapter will be spent discussing the use of filter kernels to achieve different image processing effects, but first we will review basic 1.4 pixel shader syntax and the concept of a dependent texture read in the following section on simple transfer functions.

Simple Transfer Functions

For the purposes of this chapter, we define a *transfer function* to be a function (often one-dimensional) that is used to enhance or segment an input image. Transfer functions are often used in scientific visualization to enhance the understanding of complex datasets by making important details in the image clearly distinguishable. A transfer function might map a scalar such as heat, pressure, or density to a color to aid in understanding the phenomenon being visualized. This application of transfer functions is illustrated in "3D Textures and Pixel Shaders" by Evan Hart to apply pseudo-color to a dataset, which is a scalar density acquired from a medical imaging process.

Here, we show how to use transfer functions to stylize a whole rendered 3D frame or static 2D image. This allows us to take normal color images and cause them to become black and white or sepia toned, as was done for dramatic effect in recent films such as *Thirteen Days* and *Pleasantville*. We can also think of our frame buffer as containing heat and apply a heat signature transfer function to give an effect like that used in the films *Predator* and *Patriot Games*. The application of the following three transfer functions are shown in Color Plate 2 and illustrated at right for reference.

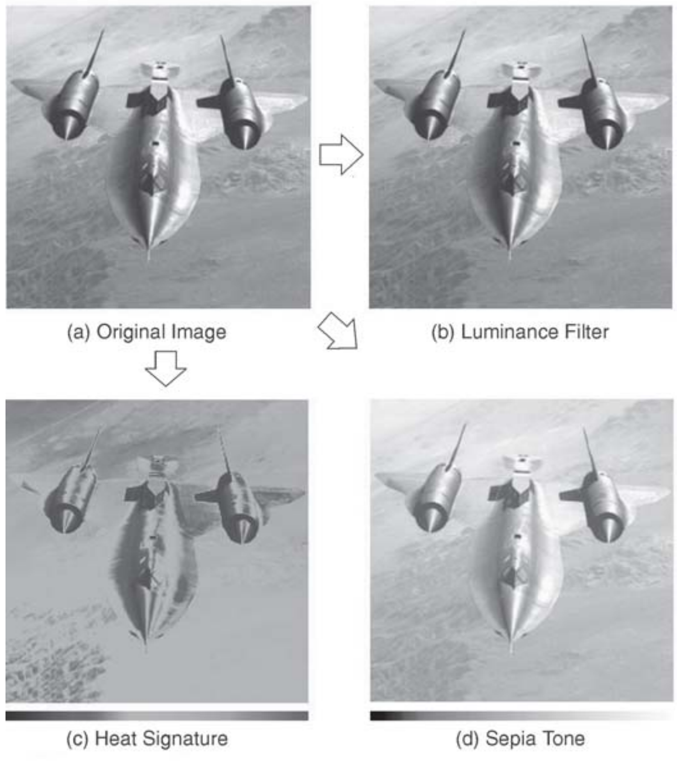

Figure 2: Three transfer functions

Black and White Transfer Function

As shown in Figure 2 and in Color Plate 2, we have applied three different transfer functions to the color input image of an SR-71 Blackbird. The first transfer function is a simple Luminance calculation which reads in the RGB color pixels from the source image and outputs them as RGB colors where R = G = B = Luminance. The Luminance operation is performed using a dot product operation to calculate Luminance = 0.3*red + 0.59*green + 0.11*blue. As shown in the pixel shader below, we define the constant {0.3f, 0.59f, 0.11f} with the def instruction and then sample texture 0 with the 0th set of texture coordinates using the texld instruction.

Listing 1: Simple Luminance shader for converting to black and white

```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0
dp3 r0, r0, c0
```

The Luminance is calculated with the dp3 instruction and the final pixel color is stored in register r0. This is a very simple transfer function that is expressible in the pixel shader assembly language. Often, it is convenient to use a lookup table to apply a more arbitrary function to the input pixels. The next transfer function will use this technique by doing a dependent read.

Sepia Tone Transfer Function

An example of a transfer function that is implemented by using a dependent read is a sepia tone effect. In this example, we apply a 1D sepia map to the luminance of the input image to make it look like an old photograph. Cut scenes in a 3D game set in the Old West would be a good candidate for this effect. The bottom right image in Color Plate 2 shows the sepia tone version of the original image. The colored stripe beneath the sepia image is the 1D texture used to map from Luminance to Sepia tone. As shown in the shader code below, the color from the original image is converted to Luminance and then used as a texture coordinate to sample the 1D sepia tone transfer function.

Sampling this 1D texture using texture coordinates that were computed earlier in the shader is known as a dependent read and is one of the most powerful concepts in pixel shading. Many of the shaders in following chapters will use dependent reads. Some examples include application of transfer functions as shown in "3D Textures and Pixel Shaders," perturbation refraction rays as shown in "Accurate Reflections and Refractions by Adjusting for Object Distance," and indexing reflection and refraction maps as shown in "Rippling Refractive and Reflective Water" in this book.

Listing 2: Simple transfer function for sepia or heat signature effects

```
ps.1.4
def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0
dp3 r0, r0, c0 // Convert to Luminance
phase
texld r5, r0   // Dependent read
mov r0, r5
```

Heat Signature

Another example of a transfer function that is implemented by using a dependent read is a heat signature effect. In this example, we use the same shader as the sepia effect but use the 1D texture shown in the bottom left of Color Plate 2. This gives the look of an image used to visualize heat. This further illustrates the power of dependent texture reads: The same shader can produce radically different results based on simple modification of the texture used in the dependent read.

In future versions of Direct3D, we look forward to applying tone-mapping transfer functions to real-time high dynamic range scenes with this same technique.

While we have been able to perform some useful operations mapping one pixel in the input image to one pixel in the output image, we can perform even more useful tasks by sampling multiple pixels from the input image. We will explore filters with this property in the remainder of this article.

Filter Kernels

Filter kernels are a combination of the locations that the input image is sampled relative to a reference point and any coefficients which modulate those image samples. A Gaussian blur, for example, is designed to sample the source image in a radially symmetric way and weight the samples according to a Gaussian distribution. The coefficients of a Gaussian blur sum to one. The simple box filter used in this chapter weights all samples evenly, with the coefficients also summing to one. Edge filters such as the Roberts and Sobel filters shown later in this chapter have positive and negative lobes. The coefficients of edge filters (including Roberts and Sobel) typically sum to zero.

Texture Coordinates for Filter Kernels

When rendering the output image, it is necessary to take care when setting the texture coordinates of the quadrilateral that is texture mapped with the input image. To achieve a one-to-one mapping from input texels to output pixels, the naïve 0.0 to 1.0 texture coordinates do not give the correct mapping. The correct coordinates are 0.0+   to 1.0+ , where   is equal to 0.5 / image_dimension. Obviously, for non-square images (*h*   *w*), this will result in a separate $_h$ = 0.5 / *h* and $_w$ = 0.5 / *w*. Filter samples that are offset from the one-to-one mapping must also take   into account. The function ComputeImagePlacement() in the DX81_ImageFilter sample application on the companion CD illustrates the correct method of setting the texture coordinates on the quadrilateral texture mapped with the input image. Obviously, this texture coordinate perturbation could be done in a vertex shader [James01], thus saving the cost of storing and transferring the additional texture coordinates for a multi-tap filter kernel; this is left out of the sample application for clarity.

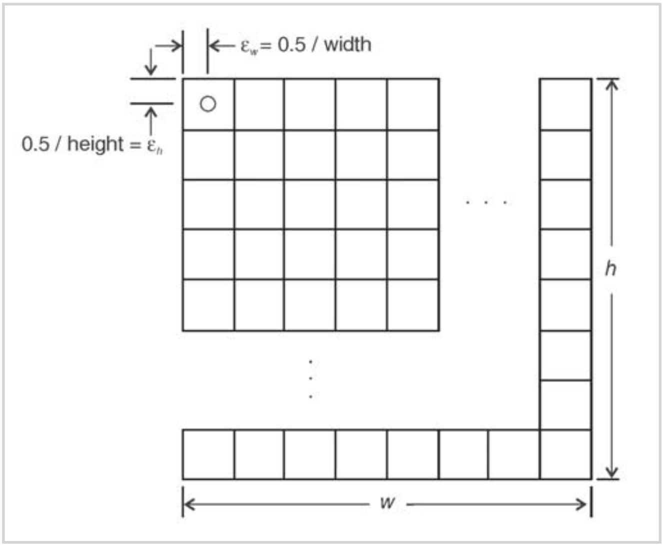

Figure 3: Correctly sampling textures for image processing

This may seem like a trivial detail and, indeed, getting this wrong may not introduce noticeable artifacts for some effects, but any ping-pong techniques that repeatedly render to and from a pair of textures can suffer massively from feedback artifacts if the texture coordinates are set up incorrectly. A ping-pong blur technique is implemented in the sample application.

Edge Detection

Edge detection is a very common operation performed in machine vision applications. In this section, we will illustrate two common edge detection filters and their applications to image processing. The use of image space methods for outlining of real-time non-photorealistic renderings will be explored further in "Non-Photorealistic Rendering with Pixel and Vertex Shaders" using the filters introduced here. A related gradient filter, the central-difference filter, is also discussed in "3D Textures and Pixel Shaders."

Roberts Cross Gradient Filters

One set of inexpensive filters that is commonly used for edge detection is the Roberts cross gradient. These 2×2 filters respond to diagonal edges in the input image and amount to doing a pair of 2-tap first-differences to approximate the image gradients in the diagonal directions.

–1	0
0	1

0	–1
1	0

Figure 4: Roberts cross gradient filters

Since the Roberts filters can be thought of as simply three taps (with one tap reused in both gradient calculations), it is easy to do the gradient magnitude calculation for both directions in the same pixel shader. The following pixel shader first computes luminance for all three taps, computes the luminance cross gradients, takes their absolute values, adds them together, and scales and inverts them before compositing them back over the original image.

Listing 3: Roberts cross gradient filter overlayed on original color image

```
// Roberts Cross Gradient Filter from color image
ps.1.4
Def c0, 0.30f, 0.59f, 0.11f, 1.0f
texld r0, t0   // Center Tap
texld r1, t1   // Down/Right
texld r2, t2   // Down/Left
dp3 r3, r0, c0
dp3 r1, r1, c0
dp3 r2, r1, c0
add r1, r3, -r1
add r2, r3, -r2
cmp r1, r1, r1, -r1
cmp r2, r2, r2, -r2
add_x4 r1, r1, r2
phase
mul r0.rgb, r0, 1-r1
+mov r0.a, c0.a
```

The result of this operation is black outlines layered over the original image in areas that have sharp luminance discontinuities, as shown in Figure 5.

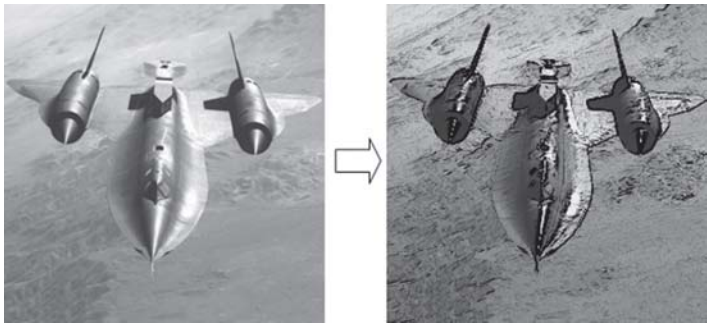
Figure 5: Input image and image with overlayed edges computed with Roberts filters

Sobel Filter

A more robust and popular edge detection filter is the Sobel filter [Sobel90]. The Sobel filters are 6-tap 3×3 filters which detect horizontal and vertical edges. Like the Roberts filters, the Sobel filters are just 90º rotations of each other. The filter is shown below:

$$\delta u = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \qquad \delta v = \begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Figure 6: The Sobel filter kernels

Because the Sobel filter for each direction, *u* and *v*, is a 6-tap filter and the 1.4 pixel shading model allows us to sample six textures, it is convenient to handle *u* and *v* gradients separately. This is not strictly necessary, as more taps can be sampled as dependent reads at texture coordinates which are derived from the other input texture coordinates. It should also be mentioned that it is possible to use the bilinear filtering capability of the hardware to approximate the double-weighting of the central samples. In order to keep this example manageable, however, we will avoid this and take a two-pass approach: one pass for | u| and one for | v|. The two partial gradient magnitude images can then be composited to form a resulting gradient magnitude image, which highlights the edges. A Luminance-only version of the original SR-71 Blackbird image is shown in Figure 7, along with | u|, | v|, and | u|+| v| images.

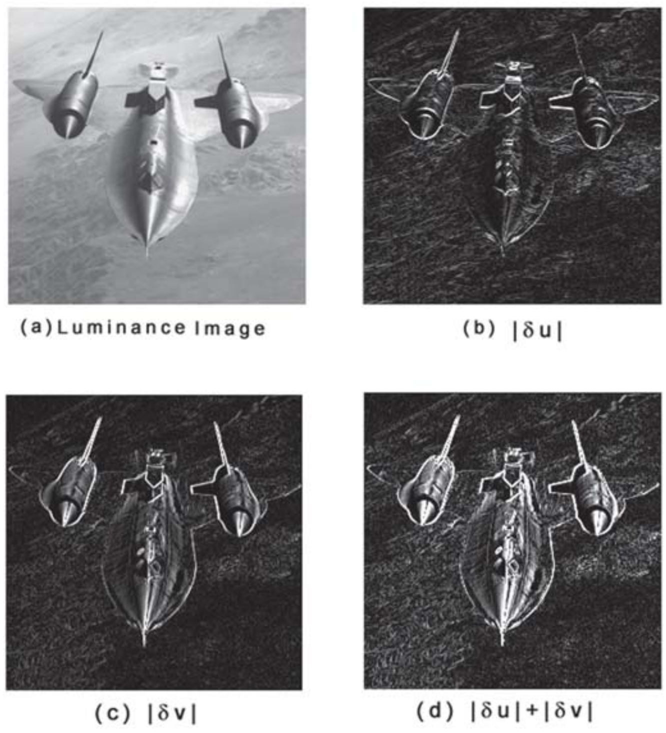

Figure 7: Input image and gradient magnitude images for u and v directions

A given Sobel filter is easily handled in one rendering pass with the following pixel shader that samples all six taps, accumulates the samples, and takes the absolute value of the results. For a non-photorealistic object and shadow outlining effect like the one shown in "Non-Photorealistic Rendering with Pixel and Vertex Shaders," these white edges can be thresholded and inverted to produce clean, black outlines. The results of one such effect are shown in Color Plate 6 (see "Non-Photorealistic Rendering").

Listing 4: Sobel filter pixel shader

```
// Sobel Filter (use for U or V, perturbations control which)
ps.1.4
def c0, 0.1666f, 0.1666f, 0.1666f, 1.0f
tex1d r0, t0 // Expect samples in one of two layouts...doesn't matter which
tex1d r1, t1 //
tex1d r2, t2 // -t0    t3  -t0 -2t1 -t2
tex1d r3, t3 // -2t1 x 2t4 or  x
tex1d r4, t4 // -t2    t5  t3 2t4 t5
tex1d r5, t5 //
add r0, -r0, -r2
add r0, -r0, -r1_x2
add r0, r0, r3
add r0, r0, r4_x2
add r0, r0, r5
cmp r0, r0, r0, -r0
```

Mathematical Morphology

Morphological operators are used in image processing applications that deal with the extraction, enhancement, or manipulation of region boundaries, skeletons, or convex hulls. In some sense, one can think of mathematical morphology as *set theory for images*. In general, mathematical morphology applies an arbitrary morphological operator to a source image. In the following discussion, we apply a limited set of morphological operators since we effectively hard-code them into the pixel shaders themselves. As such, the generality of the operators is limited by the instruction count of the pixel shaders and the expressiveness of the language. Nevertheless, morphology is an important area of image processing, and one can see how this will become even more useful in future generations of hardware, which can access more samples from the input image and even express the morphological operator as an image in a texture map. Even for the relatively small number of taps available to us in Direct3D's 1.4 pixel shaders, we can apply some interesting morphological operators.

Mathematical morphology is typically applied to binary (1-bit) images, but it can just as easily be applied to scalar (grayscale) images. In many cases, the input image is created from thresholding another image, often one which has been edge-filtered. This binary image is then operated on by a morphological operator and is either dilated or eroded by the operator. In this chapter, we will use morphology to enhance the edges used in the non-photorealistic rendering application discussed in "Non-Photorealistic Rendering." The image processing sample application on the companion CD and the non-photorealistic rendering sample application on the CD can both perform dilation on outline images.

The Definition of Dilation

Figure 8 illustrates a dilation operation. The image on the left is the original binary image. Each small cell in this figure represents one pixel. The small two-by-two black square in the center is the operator. The white square specifies which pixel is the origin of the operator—in this case, the top left pixel of the two-by-two operator.

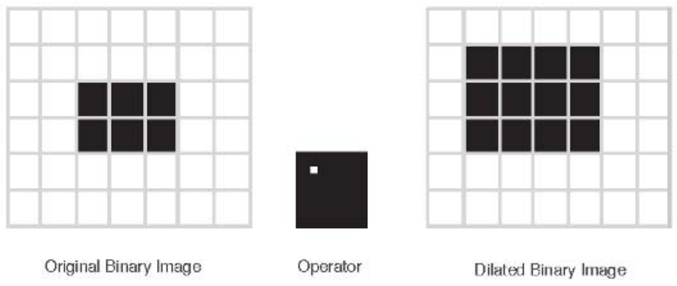

Figure 8: Dilation operation

To perform dilation of the original image, the operator is moved over the entire image. For every position of the operator, if any of the pixels of the operator cover a black pixel in the input image, then the pixel under the origin of the operator is set to black in the output image. This procedure results in the image on the right side of the figure where the black rectangle has been dilated up and to the left. The result of another dilation operation, with a 1×3 operator whose origin is centered vertically, is shown in Figure 9. Clearly, different dilation operators can enhance different features of an input image.

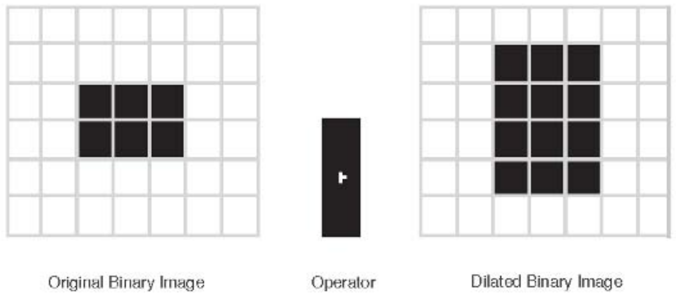

Figure 9: Dilation operation

A Dilation Shader

Once we have the binary input image, we apply the morphological operator in exactly the same way as the filters applied earlier, but the shader performs a different operation on the samples which lie under the kernel/operator. In the case of dilation, we use the cmp instruction to perform a threshold on the summation of the inverted samples. If the summation of the inverted samples exceeds a threshold, then part of the operator must cover some black pixels and we set the pixel to black. The source code for a 4-tap dilation operation is shown below:

Listing 5: Dilation shader

```
// Simple 4-tap dilation operator
ps.1.4
def c0, -0.2f, 1.0f, 0.0f, 0.0f
texld r0, t0              // Origin Tap
texld r1, t1
texld r2, t2
texld r3, t3

add r0, 1-r0, 1-r1        // Sum the inverted samples
add r0, r0, 1-r3
add r0, r0, 1-r2

add r0, r0, c0.r          // Subtract threshold
cmp r0, r0, c0.b, c0.g    // Set based on comparison with zero
```

Executing this shader on an image of object outlines thickens the outlines, as shown in Figure 10. Obviously, we could feed the dilated back through a dilation operation to thicken the lines further using a technique similar to the ping-pong blur in the image processing sample application on the companion CD. Application of dilation to non-photorealistic rendering is discussed further in "Non-Photorealistic Rendering."

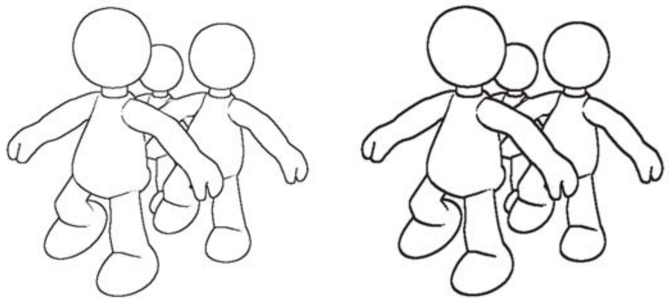
Figure 10:- An edge image which has been dilated to thicken the edge lines

The Definition of Erosion

Where a dilation operation can "thicken" images, an erosion operation thins the structures present in an input binary image. This is often useful to clean up noise or thin down structures in an image for pattern recognition.

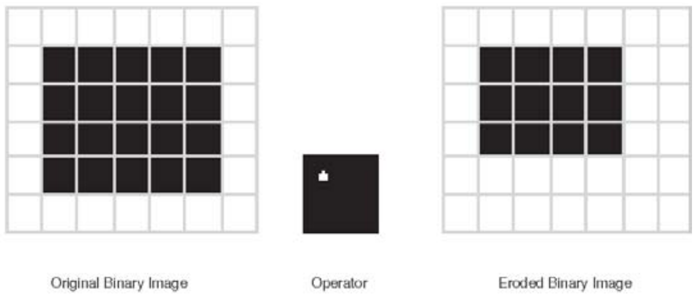

Figure 11: Erosion operation

An Erosion Shader

The erosion shader is very similar to the dilation shader. The 4-tap shader below applies a 2x2 box operator like the one shown in Figure 11. If any of the pixels under the box are white, then the pixel under the origin (upper left) is set to white.

Listing 6: Erosion shader

```
// Simple 4-tap erosion operator
ps.1.4
def c0, -0.05f, 1.0f, 0.0f, 0.0f
texld r0, t0              // Origin Tap
texld r1, t1              //
texld r2, t2              //
texld r3, t3              //

add r0, r0, r1            // Sum the samples
add r0, r0, r2
add r0, r0, r3

add r0, r0, c0.r          // Subtract threshold
cmp r0, r0, c0.g, c0.b    // If any pixels were white, set to white
```

Applying this operator to the original edge image, for example, thins down the lines and causes breaks in some places.

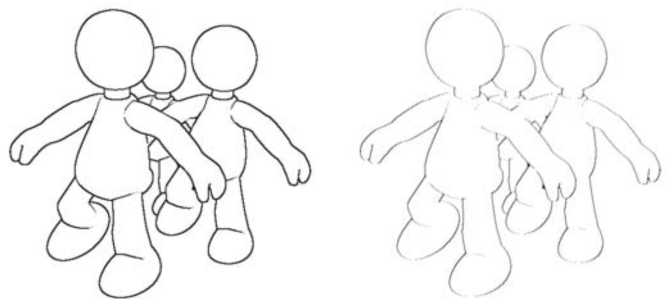

Figure 12: Original and eroded outlines

This erosion operator can also get rid of salt-and-pepper noise in a binary image. To illustrate this, we have added some small black areas (mostly smaller than 2x2 pixels) to the edge image and applied the erosion shader. The stray black pixels are all eliminated, and the original edges are thinned.

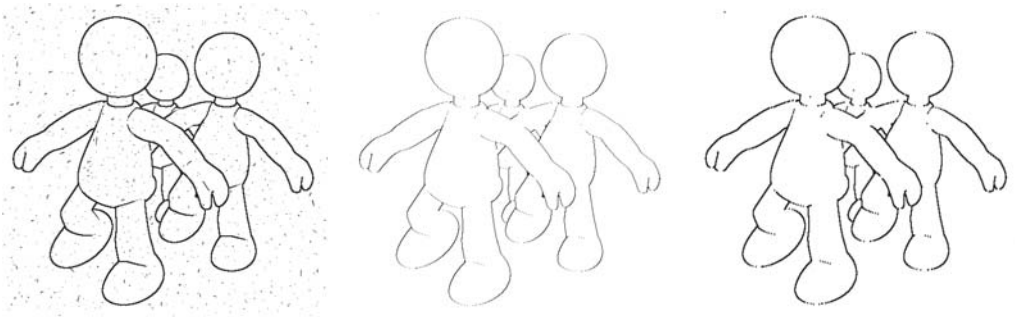

Figure 13: Noisy outline image, eroded noisy image, and opening of noisy image

Erosion can be followed by a dilation to eliminate salt-and-pepper noise while preserving larger structures in the input image. This is called the *opening* of the input image. Dilation followed by erosion is called the *closing* of the input image. Naturally, there is much more to mathematical morphology than just dilation and erosion, but these two operations form the building blocks of more advanced morphology techniques, including opening and closing.

Conclusion

We have provided an introduction to the concept of image processing in this chapter. This includes application of analytical and texture-based transfer functions, blurring and edge detection filters, and mathematical morphology. These filters have applications in data visualization, depth-of-field, non-photorealistic rendering, and general 3D frame post-processing. This chapter really only scratches the surface of image processing with pixel shaders but will hopefully serve as a reference for developers interested in post-processing their 3D scenes. For more on applying these filters to non-photorealistic rendering, see "Non-Photorealistic Rendering." For more on image processing in general, an excellent introduction to image processing concepts including those discussed here can be found in [Gonzalez92].

Hardware Support

At publication, only boards based on the ATI RADEON™ 8500 graphics processor support the 1.4 pixel shading model necessary to perform the image filters illustrated here. Any future hardware that supports the 1.4 pixel shader model or future pixel shader models such as ps.2.0 will be able to execute these shaders.

Sample Application

The DX81_ImageFilter sample on the companion CD illustrates image processing in Direct3D and implements all of the filtering examples shown here. Most of the shaders used require support for the 1.4 pixel shader model. Due to the simplicity of this application, it is reasonable to run this application on the reference rasterizer if you don't have a board that supports 1.4 pixel shaders. Rendered output images are also shown on the CD.

References

[Gonzalez92] Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing* (Addison-Wesley, 1992).

[James01] Greg James, "Operations for Hardware-Accelerated Procedural Texture Animation," *Game Programming Gems 2*, Mark DeLoura, Ed. (Charles River Media, 2001), pp. 497-509.

[Saito90] Takafumi Saito and Tokiichiro Takahashi, "Comprehensible Rendering of 3-D Shapes," SIGGRAPH Proceedings, 1990, pp. 197-206.

[Sobel90] Irvin Sobel, "An isotropic 3×3 image gradient operator," *Machine Vision for Three-Dimensional Scenes*, H. Freeman, Ed. (Academic Press, 1990), pp. 376-379.