Advanced Image Processing with DirectX 9 Pixel Shaders

Jason L. Mitchell, Marwan Y. Ansari, and Evan Hart
ATI Research

Introduction

With the introduction of the ps_2_0 pixel shader model in DirectX 9, we were able to significantly expand our ability to use consumer graphics hardware to perform image processing operations. This is due to the longer program length, the ability to sample more times from the input image(s), and the addition of floating-point internal data representation. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, we used the ps_1_4 pixel shader model in DirectX 8.1 to perform basic image processing techniques, such as simple *blurs*, *edge detection*, *transfer functions*, and *morphological operators* [Mitchell02]. In this chapter, we extend our image processing toolbox to include *color space conversion*, a better edge detection filter called the *Canny filter*, separable *Gaussian* and *median* filters, and a real-time implementation of the *Fast Fourier Transform*.

Review

As shown in our original image processing article in the *Direct3D ShaderX* book, post-processing of 3D frames is fundamental to producing a variety of interesting effects in game scenes. Image processing is performed on a GPU by using the source image as a texture and drawing a screen-aligned quadrilateral into the back buffer or another texture. A pixel shader is used to process the input image to produce the desired result in the render target.

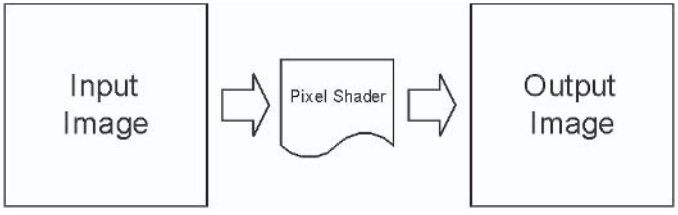


Figure 1: Using a pixel shader for image processing by rendering from one image to another

Image processing is especially powerful when the color of the destination pixel is the result of computations done on multiple pixels from the source image. In this case, we sample the source image multiple times and use the pixel shader to combine the data from the multiple samples (or *taps*) to produce a single output.

Color Space Conversion

Before we get into interesting multi-tap filters, we present a pair of shaders that can be used to convert between HSV and RGB color spaces. These shaders perform some relatively complex operations to convert between color spaces, even though they are only single-tap filters.

For those who may not be familiar with HSV space, it is a color space that is designed to be intuitive to artists who think of a color's tint, shade, and tone [Smith78]. Interpolation in this color space can be more aesthetically pleasing than interpolation in RGB space. Additionally, when comparing colors, it may be desirable to do so in HSV space. For example, in RGB space the color {100, 0, 0} is very different from the color {0, 0, 100}. However, their V components in HSV space are equal. Colors, represented by {*hue*, *saturation*, *value*} triples, are defined to lie within a hexagonal pyramid, as shown in Figure 2.

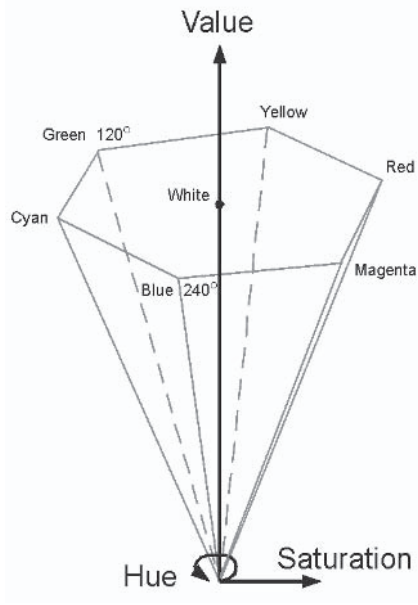


Figure 2: HSV color space

The *hue* of a color is represented by an angle between 0° and 360° around the central axis of the hexagonal cone. A color's *saturation* is the distance from the central (achromatic) axis, and its *value* is the distance along the axis. Both *saturation* and *value* are defined to be between 0 and 1.

We have translated the pseudocode RGB-to-HSV transformation from [Foley90] to the DirectX 9 High Level Shading Language (HLSL) and compiled it for the ps_2_0 target. If you are unfamiliar with HLSL, you can refer to the "Introduction to the DirectX 9 High Level Shading Language" article in *ShaderX²: Introductions & Tutorials with DirectX 9.* As described in [Smith78], you can see that the `RGB_to_HSV()` function in this shader first determines the minimum and maximum channels of the input RGB color. The max channel determines the *value* of the HSV color or how far along the achromatic central axis of the hexagonal cone the HSV color will be. The saturation is then computed as the difference between the max and min RGB channels divided by the max. Hue (the angle around the central achromatic axis) is then a function of the channel that had the max magnitude and thus determined the value.

```
float4 RGB_to_HSV (float4 color)
{
    float  r, g, b, delta;
    float  colorMax, colorMin;
    float  h=0, s=0, v=0;
    float4 hsv=0;

    r = color[0];
    g = color[1];
    b = color[2];

    colorMax = max (r,g);
    colorMax = max (colorMax,b);
    colorMin = min (r,g);
    colorMin = min (colorMin,b);
    v = colorMax;               // this is value

    if (colorMax != 0)
    {
        s = (colorMax - colorMin) / colorMax;
    }

    if (s != 0) // if not achromatic
    {
        delta = colorMax - colorMin;
        if (r == colorMax)
        {
            h = (g-b)/delta;
        }
        else if (g == colorMax)
        {
            h = 2.0 + (b-r) / delta;
        }
        else // b is max
        {
            h = 4.0 + (r-g)/delta;
        }

        h *= 60;

        if( h < 0)
        {
            h +=360;
        }

        hsv[0] = h / 360.0;    // moving h to be between 0 and 1.
        hsv[1] = s;
        hsv[2] = v;
    }

    return hsv;
}
```

The HSV-to-RGB transformation, also translated from [Foley90], is shown below in HLSL.

```
float4 HSV_to_RGB (float4 hsv)
{
    float4 color=0;
    float  f,p,q,t;
    float  h,s,v;
    float  r=0,g=0,b=0;
    float  i;

    if (hsv[1] == 0)
    {
        if (hsv[2] != 0)
        {
            color = hsv[2];
        }
    }
    else
    {
        h = hsv.x * 360.0;
        s = hsv.y;
        v = hsv.z;

        if (h == 360.0)
        {
            h=0;
        }

        h /=60;
        i = floor (h);
        f = h-i;
        p = v * (1.0 - s);
        q = v * (1.0 - (s * f));
        t = v * (1.0 - (s * (1.0 -f)));

        if (i == 0)
        {
            r = v;
            g = t;
            b = p;
        }
        else if (i == 1)
        {
            r = q;
            g = v;
            b = p;
        }
        else if (i == 2)
        {
            r = p;
            g = v;
```

```
            b = t;
        }
        else if (i == 3)
        {
            r = p;
            g = q;
            b = v;
        }
        else if (i == 4)
        {
            r = t;
            g = p;
            b = v;
        }
        else if (i == 5)
        {
            r = v;
            g = p;
            b = q;
        }

        color.r = r;
        color.g = g;
        color.b = b;
    }

    return color;
}
```

Other Color Spaces

It is worth noting that RGB and HSV are not the only color spaces of interest in computer graphics. For example, the original paper [Smith78] that introduced HSV also introduced a color space called HSL (for *hue*, *saturation*, and *lightness*), where L is often the same as the Luminance (Y) channel used in the YIQ color space. If you are interested in learning more about color spaces, [Smith78] and [Foley90] both provide excellent discussions.

Now that we have introduced some reasonably advanced single-tap image operations for converting between color spaces, we can discuss a few multi-tap filters that perform some sophisticated image processing operations.

Advanced Edge Detection

In *Direct3D ShaderX*, we discussed the Roberts and Sobel edge detection filters [Mitchell02]. Here, we expand upon those filters and introduce an implementation of the Canny edge detection filter [Canny86].

Step-by-Step Approach

As outlined in [Jain95], the Canny edge detection filter can be implemented by performing the following operations:

1. Apply a Gaussian blur.
2. Compute the partial derivatives at each texel.
3. Compute the magnitude and direction of the line ($\tan^{-1}$) at each point.
4. Sample the neighbors in the direction of the line and perform nonmaxima suppression.

Naturally, we implement this in a series of steps, each using a different shader to operate on the output from the preceding step. A Gaussian blur is the first shader that is run over the input image. This is done to eliminate any high frequency noise in the input image. Various filter kernel sizes can be used for this step.

The next step in the process is computation of the partial derivatives (P and Q) in the u and v directions, respectively:

Partial u Kernel

-1	1
-1	1

Partial v Kernel

-1	-1
1	1

Then the magnitude of the derivative is computed using the standard formula:

$$Magnitude = \sqrt{P^2 + Q^2}$$

Finally, the P and Q values are used to determine the direction of the edge at that texel using the standard equation:

$$\theta = \text{atan2}(Q, P)$$

Magnitude and θ are written out to an image so that the next shader can use them to complete the Canny filter operation. The edge direction, θ, is a signed quantity in the range of $-\pi$ to π and must be packed into the 0 to 1 range in order to prevent loss of data between rendering passes. In order to do this, we pack it by computing:

$$A = \text{abs}(\theta) / \pi$$

You've probably noticed that due to the absolute value, this function is not invertible, hence data is effectively lost. This does not present a problem for this particular application due to symmetries in the following step.

The final pass involves sampling the image to get the magnitude and the edge direction, θ, at the current location. The edge direction, θ, must now be unpacked into its proper range. Figure 3 shows a partitioning of all values of θ (in degrees) into four sectors.

The sectors are symmetric and map to the possible ways that a line can pass through a 3×3 set of pixels. In the previous step, we took the absolute value of θ and divided it by π to put it in the 0 to 1 range. Since we know that θ is already between 0 and 1 from the previous step, we are almost done. Since the partitioning is symmetric, it was an excellent way to reduce the number of comparisons needed to find the correct neighbors to sample. Normally, to complete the mapping, we would multiply A by 4 and be done. However, if you look closely at Figure 3 you can see that the sectors are centered around 0 and 180. In order to compensate for this, the proper equation is:

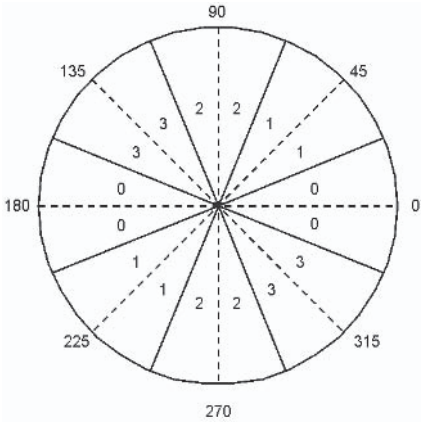


Figure 3: The 360 degrees of an angle partitioned into four sectors

$$\text{Sector} = \text{floor}((A - \pi/16) * 4)$$

Next, we compute the neighboring texel coordinates by checking which sector this edge goes through. Now that the neighbors have been sampled, we compare the current texel's magnitude to the magnitudes of its neighbors. If its magnitude is greater than both of its neighbors, then it *is* the local maximum and the value is kept. If its magnitude is less than either of its neighbors, then this texel's value is set to zero. This process is known as *nonmaxima suppression*, and its goal is to thin the areas of change so that only the greatest local changes are retained. As a final step, we can threshold the image in order to reduce the number of false edges that might be picked up by this process. The threshold is often set by the user when he or she finds the right balance between true and false edges.

As you can see in Figure 4, the Canny filter produces one-pixel-wide edges unlike more basic filters such as a Sobel edge filter.

Figure 4: One-pixel-wide edges from the Canny filter

Figure 5: Gradient magnitudes from the Sobel filter (see [Mitchell02])

Implementation Details

This shader is implemented in the VideoShader application on the companion CD (see the section 4\04 folder) using HLSL and can be compiled for the ps_2_0 target or higher. In this implementation, the samples are taken from the eight neighbors adjacent to the center of the filter. Looking at the HLSL code, you can see an array of float two-tuples called `sampleOffsets[]`. This array defines a set of 2D offsets from the center tap, which are used to determine the locations from which to sample the input image. The locations of these samples relative to the center tap are shown in Figure 6.

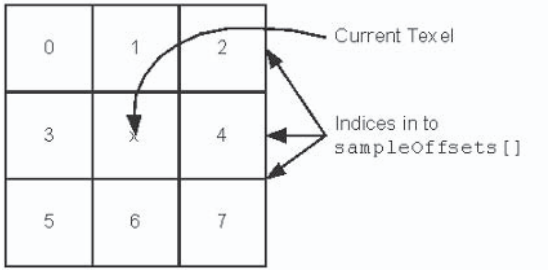


Figure 6: Locations of taps as defined in `sampleOffsets[]`

The four steps of the Canny edge detection filter described above have been collapsed into two rendering passes requiring the two shaders shown below. The first shader computes the gradients P and Q followed by the Magnitude and direction (θ). After packing θ into the 0 to 1 range, Magnitude and θ are written out to a temporary surface.

```
sampler InputImage;
float2 sampleOffsets[8] : register (c10);
```

```
struct PS_INPUT
{
    float2 texCoord:TEXCOORD0;
};

float4 main( PS_INPUT In ) : COLOR
{
    int i =0;
    float4 result;
    float  Magnitude, Theta;
    float  p=0,q=0;
    float  pKernel[4] = {-1, 1, -1, 1};
    float  qKernel[4] = {-1, -1, 1, 1};
    float2 texCoords[4];
    float3 texSamples[4];
    float  PI = 3.14159265358979323846264338327950;


    texCoords[0] = In.texCoord + sampleOffsets[1];
    texCoords[1] = In.texCoord + sampleOffsets[2];
    texCoords[2] = In.texCoord;
    texCoords[3] = In.texCoord + sampleOffsets[4];


    for(i=0; i <4; i++)
    {
        texSamples[i].xyz =  tex2D(InputImage, texCoords[i]);
        texSamples[i] = dot(texSamples[i], 0.33333333f);
        p += texSamples[i] * pKernel[i];
        q += texSamples[i] * qKernel[i];
    }

    p /= 2.0;
    q /= 2.0;

    Magnitude = sqrt((p*p) + (q*q));
    result = Magnitude;

    // Now we compute the direction of the
    // line to prep for Nonmaxima supression.
    //
    // Nonmaxima supression - If this texel isn't the Max,
    // make it 0 (hence, supress it)
    Theta = atan2(q,p); // result is -pi to pi

    result.a = (abs(Theta) / PI); // Now result is 0 to 1
                                  // Just so it can be written out.
    return result;
}
```

In the second pass of the Canny edge detector, Magnitude and θ are read back from the temporary surface. The edge direction, θ, is classified into one of four sectors and the neighbors along the proper direction are sampled using dependent

reads. The magnitudes of these neighbor samples along with a user-defined threshold are then used to determine whether this pixel is a local maximum or not, resulting in either 0 or 1 being output as the final result.

```
sampler InputImage;

float2 sampleOffsets[8] : register (c10);
float4 UserInput        : register (c24);

struct PS_INPUT
{
    float2 texCoord:TEXCOORD0;
};

float4 main( PS_INPUT In ) : COLOR
{
    int i =0;
    float4 result;
    float  Magnitude, Theta;
    float2 texCoords[4];
    float4 texSamples[3];
    float  PI = 3.14159265358979323846264338327950;


    // Tap the current texel and figure out line direction
    texSamples[0] = tex2D( InputImage, In.texCoord);
    Magnitude = texSamples[0].r;

    // Sample two neighbors that lie in the direction of the line
    // Then find out if this_texel has a greater Magnitude.
    Theta = texSamples[0].a;


    // Must unpack theta. Prior pass made Theta range between 0 and 1
    // But we really want it to be either 0,1,2, or 4. See [Jain95]
    // for more details.
    Theta = (Theta - PI/16) * 4 ; // Now theta is between 0 and 4
    Theta = floor(Theta); // Now theta is an INT.

    if( Theta == 0)
    {
        texCoords[1] = In.texCoord + sampleOffsets[4];
        texCoords[2] = In.texCoord + sampleOffsets[3];
    }
    else if(Theta == 1)
    {
        texCoords[1] = In.texCoord + sampleOffsets[2];
        texCoords[2] = In.texCoord + sampleOffsets[5];
    }
    else if(Theta == 2)
    {
        texCoords[1] = In.texCoord + sampleOffsets[1];
```

```
        texCoords[2] = In.texCoord + sampleOffsets[6];
    }
    else //if(Theta == 3)
    {
        texCoords[1] = In.texCoord + sampleOffsets[0];
        texCoords[2] = In.texCoord + sampleOffsets[7];
    }

    // Take other two samples
    // Remember they are in the direction of the edge
    for(i=1; i <3; i++)
    {
        texSamples[i].xyz =  tex2D( InputImage, texCoords[i]);
    }

    // Now it's time for Nonmaxima supression.
    // Nonmaxima supression - If this texel isn't the Max,
    // make it 0 (hence, supress it)
    // This keeps the edges nice and thin.
    result = Magnitude;
    if( Magnitude < texSamples[1].x || Magnitude < texSamples[2].x )
    {
        result =0;
    }

    // Threshold the result.
    if(result.x < UserInput.z)
    {
        result =0;
    }
    else
    {
        result = 1;
    }

    return result;
}
```

You can see in Figure 4 that this produces one-pixel-wide edges, which may be more desirable for some applications. You may see some gaps in the detected edges, and in some cases, it may be useful to apply a dilation operation to fill in these gaps [Mitchell02].

Separable Techniques

Certain filtering operations have inherent symmetry, which allows us to implement them more efficiently in a separable manner. That is, we can perform these 2D image processing operations with a sequence of 1D operations and obtain equivalent results with less computation. Conversely, we can implement a large separable filter kernel with the same amount of computation as a small, non-separable filter. This is particularly important when attempting to apply "blooms" to

final frames in high dynamic range space to simulate light scattering. In this final section of the chapter, we discuss three separable filtering operations: the Gaussian blur, a median filter approximation, and the Fast Fourier Transform.

Separable Gaussian

A very commonly used separable filter is the Gaussian filter, which can be used to perform blurring of 2D images. The 2D isotropic (i.e., circularly symmetric) Gaussian filter, $g2D(x, y)$, samples a circular neighborhood of pixels from the input image and computes their weighted average, according to the following equation:

$$g_{2D}(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

…where σ is the standard deviation of the Gaussian and x and y are the coordinates of image samples relative to the center of the filter. The standard deviation, σ, determines the size of the filter.

This means that we sample a local area of texels from the input image and weight them according to the above equation. For example, for a Gaussian with σ = 1, we compute the following filter kernel (after normalization).

0.0037	0.0146	0.0256	0.0146	0.0037
0.0146	0.0586	0.0952	0.0586	0.0146
0.0256	0.0952	0.1502	0.0952	0.0256
0.0146	0.0586	0.0952	0.0586	0.0146
0.0037	0.0146	0.0256	0.0146	0.0037

In theory, the Gaussian has infinite extent, but the contribution to the final result is insignificant for input texels outside of this 5×5 region.

An extremely important property of the Gaussian is that it is separable. That is, it can be rearranged in the following manner:

$$g_{2D}(x, y) = \left(\frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}\right) \bullet \left(\frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{y^2}{2\sigma^2}}\right)$$

$$= g_{1D}(x) \bullet g_{1D}(y)$$

This means that we can implement a given Gaussian with a series of 1D filtering operations: one horizontal ($g_{1D}(x)$) and one vertical ($g_{1D}(y)$). This allows us to implement Gaussians with much larger kernels (larger σ) while performing the same amount of calculations that are required to implement a smaller non-separable filter kernel. This technique was used in our real-time implementation of Paul Debevec's *Rendering with Natural Light* animation as seen in Figure 7.

After rendering the scene in high dynamic range space, Debevec performed a number of large Gaussian blurs on his 2D rendered scene to obtain blooms on bright areas of the scene. In order to do this in real-time, we exploited the Gaussian's separability to perform the operation efficiently. In our case, we used σ = 7, which resulted in a 25×25 Gaussian.

Figure 7: Frame from real-time Rendering with Natural Light (See Color Plate 20.)

Due to the fact that we have only eight texture coordinate interpolators in the ps_2_0 pixel shader programming model, we must derive some of our texture coordinates in the pixel shader as deltas from the center tap location. To make the most efficient use of the hardware, we can perform as many reads from the input image as possible using non-dependent texture reads.

In our implementation, we divided our samples into three types: *inner taps*, *outer taps*, and the *center tap*. The center tap (**c**) and inner taps (**x**) shown in Figure 8 are performed using interpolated texture coordinates (and hence non-dependent texture reads).

Figure 8: Layout of 13 taps of separable Gaussian

The outer taps (**o**) shown in Figure 8 are sampled using texture coordinates computed in the pixel shader. That is, they are done with dependent reads. Note that the center tap (**c**) uses pick-nearest filtering and is aligned with the center of a specific texel in the input image. The other 12 taps all use bilinear filtering and are aligned so that they sample from two different texels in the input image. This Gaussian filter is implemented in HLSL in the following shader:

```
float4 hlsl_gaussian (float2 tapZero    : TEXCOORD0,
                      float2 tap12      : TEXCOORD1,
                      float2 tapMinus12 : TEXCOORD2,
                      float2 tap34      : TEXCOORD3,
                      float2 tapMinus34 : TEXCOORD4,
                      float2 tap56      : TEXCOORD5,
                      float2 tapMinus56 : TEXCOORD6 ) : COLOR
```

```
{
   float4 accum, Color[NUM_INNER_TAPS];
   Color[0] = tex2D(nearestImageSampler, tapZero);    // sample 0
   Color[1] = tex2D(linearImageSampler,  tap12);      // samples  1,  2
   Color[2] = tex2D(linearImageSampler,  tapMinus12); // samples -1, -2
   Color[3] = tex2D(linearImageSampler,  tap34);      // samples  3,  4
   Color[4] = tex2D(linearImageSampler,  tapMinus34); // samples -3, -4
   Color[5] = tex2D(linearImageSampler,  tap56);      // samples  5,  6
   Color[6] = tex2D(linearImageSampler,  tapMinus56); // samples -5, -6

   accum  = Color[0] * gTexelWeight[0]; // Weighted sum of samples
   accum += Color[1] * gTexelWeight[1];
   accum += Color[2] * gTexelWeight[1];
   accum += Color[3] * gTexelWeight[2];
   accum += Color[4] * gTexelWeight[2];
   accum += Color[5] * gTexelWeight[3];
   accum += Color[6] * gTexelWeight[3];

   float2 outerTaps[NUM_OUTER_TAPS];
   outerTaps[0] = tapZero *  gTexelOffset[0]; // coord for samp  7,   8
   outerTaps[1] = tapZero * -gTexelOffset[0]; // coord for samp -7,  -8
   outerTaps[2] = tapZero *  gTexelOffset[1]; // coord for samp  9,  10
   outerTaps[3] = tapZero * -gTexelOffset[1]; // coord for samp -9, -10
   outerTaps[4] = tapZero *  gTexelOffset[2]; // coord for samp  11, 12
   outerTaps[5] = tapZero * -gTexelOffset[2]; // coord for samp -11,-12

   // Sample the outer taps
   for (int i=0; i<NUM_OUTER_TAPS; i++)
   {
      Color[i] = tex2D (linearImageSampler,  outerTaps[i]);
   }

   accum += Color[0] * gTexelWeight[4]; // Accumulate outer taps
   accum += Color[1] * gTexelWeight[4];
   accum += Color[2] * gTexelWeight[5];
   accum += Color[3] * gTexelWeight[5];
   accum += Color[4] * gTexelWeight[6];
   accum += Color[5] * gTexelWeight[6];

   return accum;
}
```

Applying this shader twice in succession (with different input texture coordinates and the `gTexelOffset[]` table), we compute a 25×25 Gaussian blur and achieve the bloom effect that we are looking for.

Separable Median Filter Approximation

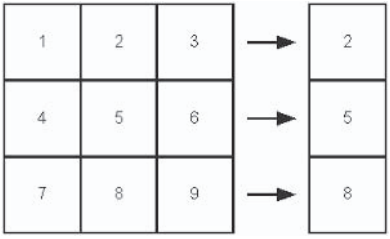
Another important filter in image processing is the median filter, the output of which is the median of the set of input data sampled by the filter kernel. For those who may not recall, the median of a set of values is the middle value after sorting or ranking the data. For example, if you have the following set of numbers: {9, 3,

6, 1, 2, 2, 8}, you can sort them to get {1, 2, 2, 3, 6, 8, 9} and select the middle value 3. Hence, the median of these values is 3. In image processing, a median filter is commonly used to remove "salt and pepper noise" from images prior to performing other image processing operations. It is good for this kind of operation because it is not unduly influenced by outliers in the input data (i.e., the noise) the way that a mean would be. Additionally, the output of a median filter is guaranteed to be a value that actually appears in the input image data; a mean does not have this property.

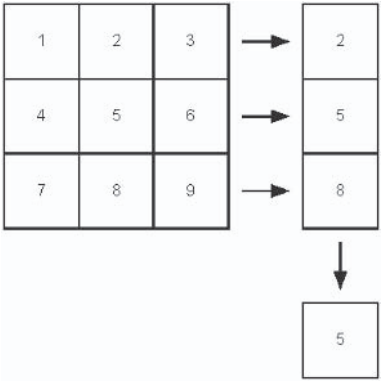
As it turns out, an approximation to a 2D median filter can be implemented efficiently in a separable manner [Gennert 03]. Say we have sampled a 3×3 region of our input image and the data are ranked in the following order:

1	2	3
4	5	6
7	8	9

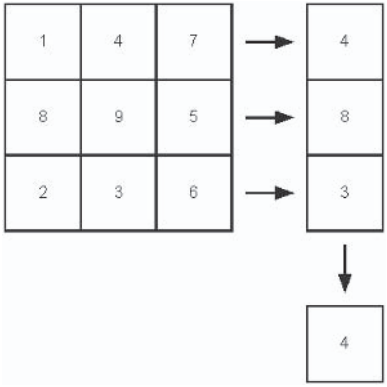
We can first take the median of the rows of the ranked data:



We can then take the median of these medians to get an approximation to the median of the whole 3×3 region:



From this, we obtain the data in the fifth-ranked image sample, which is the correct value. We say that this method is only an approximation to a true median filter because the true median will not be found if the ranked data is not so evenly distributed within the filter kernel. For example, if we have the following ranked data, we can get an incorrect median:



For a 3×3 filter kernel, however, the worst case that this separable median filter implementation will give you is the fourth or sixth rank instead of the fifth, which may be adequate for many applications.

We have implemented this separable approximation to a median filter with a two-pass rendering approach. The first pass finds the median of each 3×1 region of the image and outputs it to an intermediate buffer. The second pass performs the same operation on each 1×3 region of the intermediate buffer. The end result is equivalent to the separable median algorithm outlined above.

Median Filter HLSL Implementation

In our HLSL implementation of the separable median approximation, both passes will use the `FindMedian()` function, which takes three scalar inputs:

```
float FindMedian(float a, float b, float c)
{
    float median;

    if( a < b )
    {
        if( b < c)
        {
            median = b;
        }
        else
        {
            median = max(a,c);
        }
    }
    else
    {
        if( a < c)
        {
            median = a;
        }
        else
        {
```

```
            median = max(b,c);
        }
    }
    return median;
}
```

The first pass of the 3×3 median filter, shown below, takes three samples from the input image: the texel at the current location and the left and right neighbors. The median red, green, and blue values are found independently, and the result is written out to a temporary surface.

```
sampler InputImage;
float2 sampleOffsets[8];

struct PS_INPUT
{
    float2 texCoord:TEXCOORD0;
};

float4 main( PS_INPUT In ) : COLOR
{
    int i =0;
    float4 result;
    float2 texCoords[3];
    float3 texSamples[3];

    texCoords[0] = In.texCoord + sampleOffsets[3];
    texCoords[1] = In.texCoord;
    texCoords[2] = In.texCoord + sampleOffsets[4];

    // the left and right neighbors of this texel
    for(i=0; i <3; i++)
    {
        texSamples[i].xyz =  tex2D( InputImage, texCoords[i]);
    }

    result.r = FindMedian(texSamples[0].r,texSamples[1].r,texSamples[2].r);
    result.g = FindMedian(texSamples[0].g,texSamples[1].g,texSamples[2].g);
    result.b = FindMedian(texSamples[0].b,texSamples[1].b,texSamples[2].b);
    result.a = 0;

    return result;
}
```

In the second pass of the 3×3 median filter, the texel at the current location and the top and bottom neighbors are sampled. The median red, green, and blue values are found independently, and the final result of the shader is computed.

```
sampler InputImage;
float2 sampleOffsets[8];
struct PS_INPUT
{
    float2 texCoord:TEXCOORD0;
```

```
};

float4 main( PS_INPUT In ) : COLOR
{
    int i =0;
    float4 result;
    float2 texCoords[3];
    float3 texSamples[3];

    texCoords[0] = In.texCoord + sampleOffsets[1];
    texCoords[1] = In.texCoord;
    texCoords[2] = In.texCoord + sampleOffsets[6];

    // the top and bottom neighbors of this texel
    for(i=0; i <3; i++)
    {
        texSamples[i].xyz =  tex2D( InputImage, texCoords[i]);
    }

    result.r = FindMedian(texSamples[0].r,texSamples[1].r,texSamples[2].r);
    result.g = FindMedian(texSamples[0].g,texSamples[1].g,texSamples[2].g);
    result.b = FindMedian(texSamples[0].b,texSamples[1].b,texSamples[2].b);

    result.a = 0;

    return result;
}
```

Median Filter Results

To test the ability of this median filter approximation to remove salt and pepper noise, we have added noise to a test image and run the median filter over it twice to obtain the results shown in Figure 9.

(a) Original *(b) One median pass* *(c) Two median passes*

Figure 9: Median filter results

The original image (Figure 9a) has had some noise added to it. With only one pass of the median filter, much of the noise is removed (Figure 9b). Applying the median filter a second time eliminates the noise almost completely (Figure 9c).

Median-filtering the red, green, and blue channels of the image independently is a reasonably arbitrary decision that seems to work well for our data. You may find that another approach, such as converting to luminance and then determining the median luminance, works better for your data.

Fourier Transform

A very powerful concept in image processing is transformation of *spatial domain* images into the *frequency domain* via the Fourier Transform. All of the images that we have discussed so far have existed in the spatial domain. Using the Fourier Transform, we can transform them to the frequency domain, where the images are represented not by a 2D array of real-valued brightness values distributed spatially, but by a 2D array of complex coefficients that are used to weight a set of sine waves, which when added together would result in the source image. This set of sine waves is known as a Fourier series, named for its originator, Jean Baptiste Joseph Fourier. Fourier's assertion was that any periodic signal can be represented as the sum of a series of sine waves. This applies to any sort of signal, including images. The conversion from the spatial domain to the frequency domain is performed by a *Fourier Transform*. In the case of digital images consisting of discrete samples (pixels), we use a *Discrete Fourier Transform* (DFT). The equations for performing a DFT and its inverse on a two-dimensional image are shown below:

Fourier Transform

$$H(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} h(x,y)e^{-i2\pi(ux/M+vy/N)}$$

Inverse Fourier Transform

$$h(x,y) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1} H(u,v)e^{i2\pi(ux/M+vy/N)}$$

…where $h(x, y)$ is the value of the pixel located at location (x, y), $H(u, v)$ is the value of the image in frequency space at location (u, v), M is the width of the image in pixels, and N is the height of the image in pixels.

For these equations, it is important to remember that these are complex numbers (i is the square root of –1). Additionally, from complex math:

$$e^{ix} = \cos(x) + i\sin(x) \text{ and } e^{-ix} = \cos(x) - i\sin(x)$$

GPU Implementation

A naïve implementation of these operations would be an extremely expensive processing step, $O(n^4)$ in big O notation. Fortunately, much research has gone into a class of algorithms known as *Fast Fourier Transforms* (FFTs). These algorithms refactor the transform equations above to reduce the complexity to $O(n * \log n)$. The initial algorithm described to accomplish this is referred to as "Decimation in Time" and was published in 1965 by Cooley and Tukey [Cooley65]. As it turns out, the decimation in time algorithm translates very naturally to multipass

rendering on graphics hardware with floating-point pixel processing pipelines. Our multipass rendering technique is based on the code listed in [Crane96].

The FFT uses two primary optimizations to minimize its computational complexity. The first optimization that the FFT makes is to exploit the transform's separability and break the two-dimensional transform into several one-dimensional transforms. This is done by performing a one-dimensional FFT across the rows of the image followed by a one-dimensional FFT along the columns of the resulting image. This greatly reduces the growth in complexity of the operation as the image size grows. The next optimization uses the fact that a Fourier Transform of size N can be rewritten as the sum of two Fourier Transforms of size N/2, eliminating redundant computations. This portion of the optimization reduces the cost of the one-dimensional transforms from $O(n^2)$ to $O(n * \log n)$.

The first thing to note when using a GPU to implement an FFT based on the decimation in time algorithm is that, to maintain most of its efficiency improvements, the algorithm must be implemented in multiple passes by rendering to floating-point temporary buffers. If the spatial domain image is color (i.e., has multiple channels), these temporary buffers need to be set up as multiple render targets, since the frequency domain representation of the image uses complex numbers, thus doubling the number of channels on the output.

For a *width* × *height* image, the decimation in time FFT algorithm takes $\log_2(width) + \log_2(height) + 2$ rendering passes to complete. For example, a 512×512 image takes 20 rendering passes, which renders at approximately 30 frames per second on today's fastest graphics processors. Because each step of the computation is based solely on the previous step, we are able to conserve memory and ping-pong between two floating-point renderable textures to implement the following steps of the decimation in time algorithm:

1. Horizontal scramble using *scramble map* to do dependent texture reads from the original image
2. $\log_2$ (*width*) butterfly passes
3. Vertical scramble using scramble map again
4. $\log_2$ (*height*) butterfly passes

Let's describe each of these steps in detail.

Scramble

The decimation in time algorithm starts with a phase referred to as a *scramble*. This phase reorders the data such that:

$$\text{data}[i] :=: \text{data}[\text{rev}(i)]$$

…where rev(i) is the bit reverse of i.

In other words, the data member at location i is swapped with the data member at the location at the bit-reversed address of i. The bit reverse of a given value is its mirror image written out in binary. For example, the bit reverse of 0111 is 1110. Figure 10 shows an example of a scramble of a 16-element image.

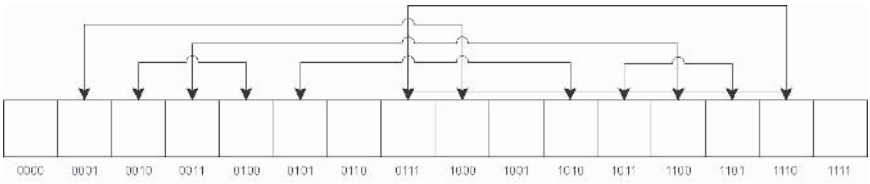


Figure 10: Simple scramble of 16×1 image

Values connected by arrows in Figure 10 are swapped during the scramble step. Obviously, symmetric values such as 0000, 0110, 1001, and 1111 are left in place. Since pixel shaders can't easily do such bit-twiddling of pixel addresses, the most effective way to perform the scramble step is via a dependent read from the input image using a specially authored *scramble map* stored in another texture to provide the bit-twiddled address from which to do the dependent read. The shader to perform such a dependent read for the horizontal scramble is shown below:

```
sampler  scramble    : register(s0);
sampler  sourceImage : register(s1);

struct PS_INPUT
{
    float1 scrambleLoc:TEXCOORD0;
    float2 imagePos:TEXCOORD1;
};

float4 main( PS_INPUT In ) : COLOR
{
    float2 fromPos;

    fromPos = In.imagePos;

    // scramble the x coordinate
    // fromPos.x gets assigned red channel of texture
    fromPos.x = tex1D(scramble, In.scrambleLoc);

    return tex2D(sourceImage, fromPos);
}
```

It is important to remember that the scramble map must contain enough bits to uniquely address each texel in the source image. Typically, this means the texture should be a 16-bit single channel texture, preferably an integer format such as `D3DFMT_L16`.

Butterflies

Once the image has been scrambled, a series of *butterfly* operations are applied to the image. In each butterfly pass, a pair of pixels is combined via a complex multiply and add. Due to the inability of graphics processors to write to random locations in memory, this operation must be done redundantly on both of the pixels in the pair, and therefore some of the ideal FFT efficiency gains are lost. The

locations of the paired pixels are encoded in a butterfly map. The butterfly map is as wide as the source image and has one row for each butterfly step. The code for applying horizontal butterflies is shown below.

```
//all textures sampled nearest
sampler  butterfly   : register(s0);
sampler  sourceImage : register(s1);

struct PS_INPUT
{
    float2 srcLocation:TEXCOORD0;
};

//constant to tell which pass is being used
float pass;  // pass = passNumber / log2(width)

float4 main( PS_INPUT In ) : COLOR
{
    float2 sampleCoord;
    float4 butterflyVal;
    float2 a;
    float2 b;
    float2 w;
    float temp;

    sampleCoord.x = srcLocation.x;
    sampleCoord.y = pass;

    butterflyVal = tex2D( butterfly, sampleCoord);
    w = butterflyVal.ba;

    //sample location A
    sampleCoord.x = butterflyVal.y;
    sampleCoord.y = srcLocation.y;
    a = tex2D( sourceImage, sampleCoord).ra;

    //sample location B
    sampleCoord.x = abs(butterflyVal.x);
    sampleCoord.y = srcLocation.y;
    b = tex2D( sourceImage, sampleCoord).ra;

    //multiply w*b (complex numbers)
    temp = w.x*b.x - w.y*b.y;
    b.y = w.y*b.x + w.x*b.y;
    b.x = temp;

    //perform a + w*b or a - w*b
    a = a + ((butterflyVal.x < 0.0) ? -b : b);

    //make it a 4 component output for good measure
    return a.xxxy;
}
```

The shader performs an extremely simple operation to accomplish its goal. First, it fetches a texture to determine where on this line of the image to get two parameters a and b. This same texel contains a factor w that is combined with a and b to produce the final result. From these parameters, the algorithm can actually produce two of the results needed for the next pass (a' and b'), but since GPUs do not perform random writes to memory, the texture also includes a flag for which value to leave at this location. The following equation, a butterfly operation, shows the math used to convert a and b to a' and b'.

$$a' = a + wb$$
$$b' = a - wb$$

The shader only concerns itself with a single channel image and expects that the real component is fetched into the first component and the imaginary component is fetched into the fourth component. To handle more components, the shader does not need to change significantly, but it does need to use separate textures and multiple render targets to handle more than two channels simultaneously. The largest amount of magic is in the special butterfly texture. This texture contains the offsets of the a and b parameters to the function in its first two components and the real and imaginary parts of the w parameter in its last two components. Additionally, the second texture coordinate is given a sign to encode whether this execution of the shader should produce a' or b'. To ensure an accurate representation of all this with the ability to address a large texture, a 32-bit per-component floating-point texture is the safest choice.

After the scramble and butterfly passes are applied in the horizontal direction, the same operations are applied to the columns of the image to get the vertical FFT. The overall algorithm looks something like the following pseudocode:

```
// Horizontal scramble first
SetSurfaceAsTexture( surfaceA); //input image
SetRenderTarget( surfaceB);
LoadShader( HorizontalScramble);
SetTexture( ButterflyTexture[log2(width)]);
DrawQuad();

// Horizontal butterflies
LoadShader( HorizontalButterfly);
SetTexture( ButterflyTexture[log2(width)]);
for ( i = 0; i < log2( width); i++)
{
   SwapSurfacesAandB();
   SetShaderConstant( "pass", i/log2(width));
   DrawQuad();
}

// Vertical scramble
SwapSurfacesAandB();
LoadShader( VerticalScramble);
SetTexture( ButterflyTexture[log2(height)]);
DrawQuad();
```

```
// Vertical butterflies
LoadShader( VerticalButterfly);
SetTexture( ButterflyTexture[log2(height)]);
for ( i = 0; i < log2( height); i++)
{
   SwapSurfacesAandB();
   SetShaderConstant( "pass", i/log2(height));
   DrawQuad();
}
```

To transform back to the spatial domain, the same operations are performed on the data, except that as one final step the data has a scaling factor applied to bring it into the correct range.

Results

So, now that we know how to apply an FFT to an image using the graphics processor, what have we computed? What does this frequency domain representation look like, and what does it mean?

The output of the Fourier Transform consists not only of complex numbers but also typically spans a dynamic range far greater than what can be displayed directly in print or on a monitor. As a result, the log of the magnitude of the frequency is typically used when displaying the Fourier domain. The function used to visualize the Fourier domain in this article is given below:

$$f(x) = 0.1 * \log\left(1 + \sqrt{x.re^2 + x.i^2}\right)$$

Finally, the image is also shifted into what is referred to as normal form. This is done primarily as a way to simplify the interpretation of the data. The shift can be done on graphics hardware by setting the texture wrap mode to repeat and biasing the texture coordinates by (–0.5, –0.5). In this format, the lowest frequencies are all concentrated in the center of the frequency-domain image, and the frequencies are progressively higher, closer to the edges of the image.

Figure 11: Original image

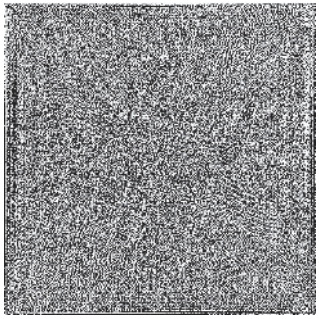


Figure 12: Fourier Transform (raw)

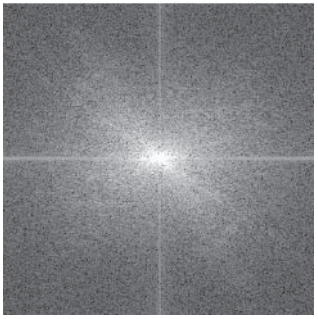


Figure 13: Fourier Transform in normal form

Utilizing the FFT

Besides just providing an interesting way to look at and analyze images, the frequency space representation allows certain operations to be performed more efficiently than they could be in the spatial domain.

First, removing high frequencies that contribute to aliasing can be most easily performed in frequency space. The simplest implementation of this simply crops the image in frequency space to remove the higher frequencies. This is the application of what is called the ideal filter, but its results tend to be anything but ideal on an image of finite size. The ideal filter really has an infinite width in the spatial domain, so when the cropped image is transformed back to the spatial domain, sharp edges will *ring* with ghosts propagating in the image. Other filters have been designed to work around such issues. One well-known filter for this sort of purpose is the Butterworth filter.

Additionally, frequency space can be used to apply extremely large convolutions to an image. Convolutions in image space are equivalent to multiplication in the frequency domain. So instead of having a multiply and add for each element of a convolution mask at each pixel, as would be required in the spatial domain, the operation takes only a multiply per pixel in the frequency domain. This is most useful on large non-separable filters like the Laplacian of Gaussians (LoG), which produces a second order derivative that can be used to find contours in images. In Figure 14, a LoG filter has been applied to the reference image used throughout the section. To apply the filter in the frequency domain, the image and the filter must first both be transformed into the frequency domain with the Fourier Transform. The filter must also be centered and padded with zeros so that it is the same size as the image to which it is being applied. Once in the frequency domain, the filter and image — both of which contain complex numbers — must undergo a complex multiplication. The result is next run through the Inverse Fourier Transform. Finally, the image must be translated similar to the way in which the frequency space images are translated to get the correct image. This last step appears to be often unmentioned in discussions of this operation, but failure to do it can lead to a fruitless bug hunt.

Figure 14: 17×17 Laplacian of Gaussian operation

Conclusion

In this chapter, we've added some sophisticated tools to our image processing toolbox, including HSV↔RGB color space conversion, the Canny edge detection filter, and separable implementations of a Gaussian blur, a median filter, and the decimation in time formulation of the Fast Fourier Transform. We hope that these implementations, presented here in the industry standard DirectX 9 High Level Shading Language, are easy for you to drop into your own image processing

applications. We also hope that they inspire you to create even more powerful image processing operations specific to your needs.

Sample Application

The image processing techniques presented in this chapter were developed using live and recorded video fed to Direct3D via the Microsoft Video Mixing Renderer (VMR). The sample app, VideoShader, demonstrates the use of Direct3D and the VMR, with the above filters and several others implemented using HLSL. Source for the sample application and all of the shaders is available on the companion CD as well as the ATI Developer Relations web site (www.ati.com/developer). The latest version of VideoShader is available at http://www2.ati.com/misc/demos/ATI-9700-VideoShader-Demo-v1.2.exe.

Acknowledgments

Thanks to John Isidoro of Boston University and ATI Research for the separable Gaussian filter implementation. Thanks to Michael Gennert of Worcester Polytechnic Institute and David Gosselin of ATI Research for discussions that resulted in the implementation of the separable median filter approximation.

References

[Canny86] Canny, John, "A Computational Approach to Edge Detection," IEEE PAMI 8(6) 679-698, November 1986.

[Cooley65] Cooley, J. W. and O. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, 19, 297-301, 1965.

[Crane96] Crane, Randy, *A Simplified Approach to Image Processing: Classical and Modern Techniques in C*, Prentice Hall, 1996.

[Foley90] James Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Second Ed., Addison-Wesley, 1990.

[Gennert03] Gennert, Michael, personal communication, 2003.

[Jain95] Jain, Ramesh and Rangachar Kasturi, et al., *Machine Vision*, McGraw Hill, 1995.

[Mitchell02] Mitchell, Jason L., "Image Processing with 1.4 Pixel Shaders in Direct3D," *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wolfgang Engel, ed., Wordware Publishing, 2002, pp. 258-269.

[Smith78] Smith, Alvy Ray, "Color Gamut Transform Pairs," SIGGRAPH '78, pp. 12-19.

Night Vision: Frame Buffer Post-processing with ps.1.1 Hardware

Guillaume Werle
Montecristo Games

Introduction

A few years ago, when hardware-accelerated rendering was starting to be a common feature in every game engine, players complained that all games were somehow looking quite the same.

Now that programmable hardware is available, the entire rendering process can be configured. This means that any game with a creative graphic programmer or a skilled technical artist can have its own graphic touch and look that is different from the others.

Frame buffer post-processing is one of the easiest ways to achieve a unique look. Many resources on these topics are available on the Internet nowadays, but most of them make use of ps.1.4 hardware. In this article I describe how to use texture-dependent reads on ps.1.1 class hardware to achieve the following effect (see Figures 1 and 2).

Figure 1: Scene from the Raw Confessions demo (models and textures by Christophe Romagnoli and Guillaume Nichols)

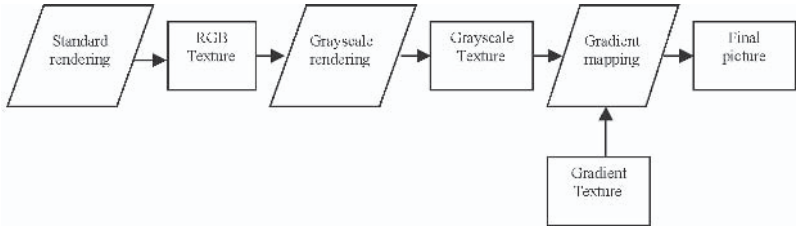
Figure 2: Scene from the Raw Confessions demo (models and textures by Christophe Romagnoli and Guillaume Nichols)

Description

Texture-dependent reads are definitely harder to use when targeting ps.1.1 hardware. The rendering process is split into several passes to take care of this issue.

Here's a quick description of the required steps to achieve this effect:

1. Render the scene in a texture.
2. Convert to grayscale while rendering in another render texture.
3. Use the luminance value of each pixel as an index into a gradient texture and render in the frame buffer.



Technical Brief on Render Texture

Instead of rendering directly in the frame buffer, the rendering must be done in a texture. Create a texture with the same size, color format, and depth format as your frame buffer, and then use the ID3DXRenderToSurface interface provided with the D3DX library to map that the `BeginScene()` and the `EndScene()` calls.

NOTES

- Render textures' dimensions don't need to be aligned on a power of two if the caps D3DPTEXTURECAPS_NONPOW2CONDITIONAL is set.
- Don't use the `D3DXCreateTexture()` function to create your render texture; this function will round the dimensions to the nearest power of two, even if it's not needed.

Converting to Grayscale

The luminance value of a color can be computed using a dot product.

$$\text{Luminance} = \text{Red} * 0.3 + \text{Green} * 0.59 + \text{Blue} * 0.1$$

The following pixel shader applies this formula to output the luminance value in every color channel:

```
ps.1.1
tex t0              // rgb texture
// c0 = (0.3, 0.59, 0.1, 1.0)
dp3 r0, t0, c0    // r0 = t0.r * 0.3 + t0.g * 0.59 + t0.b * 0.1
```

Quad Rendering

Once the scene is stored in the texture, a quad is used for rendering into the frame buffer.

The Microsoft Direct3D texture sampling rules say that texels are fetched in the top-left corner. For example, when enabling bilinear filtering, if you sample a texel at the coordinates (0,0) and if the addressing mode is set to warp, the resulting color will be a mix of the four corners of the texture.

Knowing this, an offset of a half-texel size (0.5 / TextureSize) must be added to the texture coordinates.

Color Remapping

This is the last step of the effect. The pixel shader in charge of the color remapping uses the `texreg2gb dest, src` instruction. This opcode is able to interpret the green and blue color components of the source register — the grayscale texture — as texture coordinates to sample the destination register — the gradient texture.

The gradient texture is a simple 1D texture. Figure 3 shows the gradient used to produce Figures 1 and 2.

Figure 3: 1D gradient texture

This code snippet shows the whole pixel shader :

```
ps.1.1

// t0 grayscale texture
// t1 gradient

tex       t0        // grayscale texture
texreg2gb  t1, t0    // sample t1 at the coordinates (t0.g, t0.b)

mov r0, t1          // output
```

Enhancement

Those shaders leave a great deal of room for visual improvements and experimentation; for example, a blur filter can be applied while converting the picture to grayscale, and some extra textures can be blended with the gradient remapping results. Figures 1 and 2 use this technique to achieve the scanline screening effects.

Final Words

I implemented this shader for a demo scene application called Raw Confession. The demo can be found on the companion CD or downloaded from my web page: http://cocoon.planet-d.net/raw/!Raw_Beta.zip.

The corresponding RenderMonkey workspace can be found on the companion CD as well.

Special thanks to Bertrand Carre and David Levy for proofreading this article.