

Real-Time Depth of Field Simulation

Guennadi Riguer
ATI Technologies

Natalya Tatarchuk
ATI Research

John Isidoro
ATI Research

Introduction

Photorealistic rendering attempts to generate computer images with quality approaching that of real life images. Quite often, computer rendered images look almost photorealistic, but are missing something subtle - something that makes them look synthetic or too perfect. Depth of field is one of those very important visual components of real photography, which makes images look “real”. In “real-world” photography or cinematography, the physical properties of the camera cause some parts of the scene to be blurred, while maintaining sharpness in other areas. While blurriness sometimes can be thought of as an imperfection and undesirable artifact that distorts original images and hides some of the scene details, it can also be used as a tool to provide valuable visual clues and guide a viewer’s attention to important parts of the scene. Using depth of field effectively can improve photorealism and add an artistic touch to rendered images. Figure 1 shows a simple scene rendered with and without depth of field.

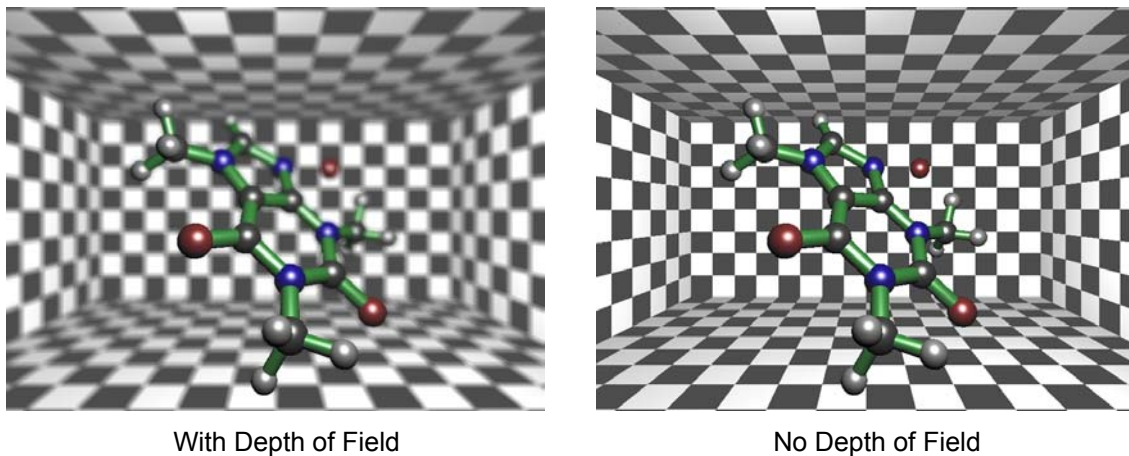


Figure 1

Recent developments in the field of programmable graphics hardware allow us to simulate complex visual effects such as depth of field in the real-time. This article presents two real-time implementations of depth of field effect using DirectX® 9 class hardware. The high level shading language (HLSL) from Microsoft is used to simplify shader development.

Camera Models and Depth of Field

Computer images rendered with conventional methods look too sharp and lack the defects and artifacts of real cameras. Without these defects it is hard to trick the eye into believing the images was captured by a real camera. Better camera models become even more important when computer-generated images have to be combined with ones produced by real camera. The visual discrepancy mostly comes from the difference between physical cameras and the camera models normally used in computer graphics. Computer graphics generally implicitly uses a pinhole camera model, while real cameras use lenses of finite dimensions.

Pinhole Camera Model

In the pinhole camera light rays scattered from objects pass through infinitely small pinhole lens. Only a single ray emanating from each point in the scene is allowed to pass through the pinhole. All rays going in other directions are ignored. Because only a single ray passes through the pinhole, only a single ray hits the imaging plane at any given point. This creates image that is always in focus. Figure 2 illustrates the pinhole camera in action.

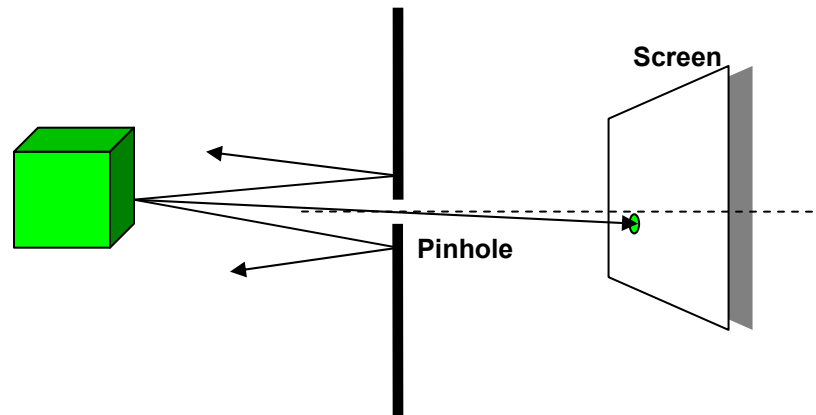


Figure 2. Pinhole camera.

Thin Lens Camera Model

In the real world, all lenses have finite dimensions and let through rays coming from multiple different directions. As a result, parts of the scene are sharp only if they are located at or near a specific focal distance. For a lens with focal length f , a sharp image of a given object is produced at the imaging plane offset from the lens by v , when the object is at the distance u from the lens. This is described by thin lens equation:

$$\frac{1}{u} + \frac{1}{v} = \frac{1}{f}$$

The distance from the image plane to the object in focus can be expressed as:

$$z_{\text{focus}} = u + v$$

Figure 3 demonstrates how the thin lens camera works.

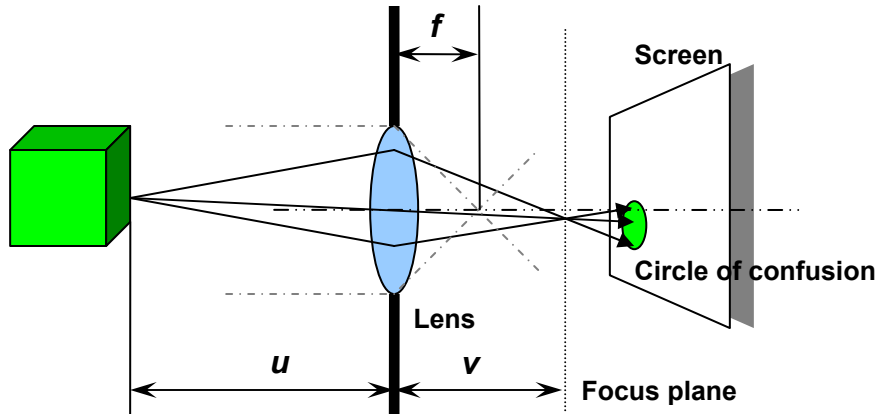


Figure 3. Thin lens camera.

Multiple rays scattered from a given point on an object will pass through the lens, forming the cone of light. If the object is in focus, all rays will converge at a single point on the image plane. However, if a given point on an object in the scene is not near the focal distance, the cone of light rays will intersect the image plane in an area shaped like a conic section. Typically, the conic section is approximated by a circle called the *circle of confusion*.

The circle of confusion diameter b depends on the distance of the plane of focus and lens aperture setting a (also known as f -stop). For a known focus distance and lens parameters, size of the circle of confusion can be calculated as:

$$b = \left| \frac{D \cdot f(z_{\text{focus}} - z)}{z_{\text{focus}}(z - f)} \right|, \text{ where } D \text{ is a lens diameter}$$

$$D = \frac{f}{a}$$

Any circle of confusion greater than the smallest point a human eye can resolve contributes to the blurriness of the image that we see as a depth of field.

Overview of Depth of Field Techniques

A number of techniques can be used to simulate depth of field in rendered scenes. One technique used in offline rendering employs distributed ray tracing. For each image point, multiple rays are shot through the lens. Coming from a single point of the image plane these rays focus on the single point of the object if it is at the focal point. If the object is not in focus, the rays get scattered into environment, which contributes to blurring. Because the rays accurately sample the surrounding environment this method produces the most realistic depth of field effect, lacking many artifacts produced by other methods. The quality however comes at a cost, and this technique is unacceptable for real-time rendering.

Another method involves accumulation buffer. The accumulation buffer integrates images from multiple render passes. Each of the images is rendered from a slightly different position and direction within the virtual lens aperture. While less complex than ray tracing, this method is also quite expensive because images have to be rendered many times to achieve good visual results.

A cheaper and more reasonable alternative for real-time implementation is the post-processing method. Usually, this method involves two-pass rendering. On the first pass scene is rendered with some additional information such as depth. On the second pass some filter is run on the result of the first pass to blur the image. This article presents two variations of this general post-processing approach. Each version has some strengths and weaknesses and can produce high quality photorealistic depth of field effects on DirectX® 9 graphics hardware.

Depth of Field Implementation via Simulation of Circle of Confusion

The first implementation that we will present is an extension to the post-processing method proposed by Potmesil and Chakravarty in [Potmesil83]. On the first pass we render the scene, outputting the color as well as information necessary to blur the image. On the second pass, we filter the image from the first pass with a variable-sized filter kernel to simulate the circle of confusion. A blurriness factor computed on the first pass controls the size of the filter kernel used in the second pass. Special measures are taken to eliminate leaking of color of objects in focus onto backgrounds that have been blurred.

Pass One: Scene rendering

First, the whole scene is rendered by outputting depth and *blurriness factor*, which is used to describe how much each pixel should be blurred, in addition to the resulting scene rendering color. This can be accomplished by rendering the scene to the multiple buffers at one time. DirectX® 9 has a useful feature called Multiple Render Targets (MRT) that allows simultaneous shader output into the multiple renderable buffers. Using this feature gives us the ability to output all of the data channels (scene color, depth and blurriness factor) in our first pass. One of the MRT restrictions on some hardware is the requirement for all render surfaces to have the same bit depth, while

allowing use of different surface formats. Guided by this requirement we can pick the D3DFMT_A8R8G8B8 format for the scene color output and the two-channel texture format D3DFMT_G16R16 format for depth and blurriness factor. As shown in Figure 4, both formats are 32-bits per pixel, and provide us with enough space for the necessary information at the desired precision.

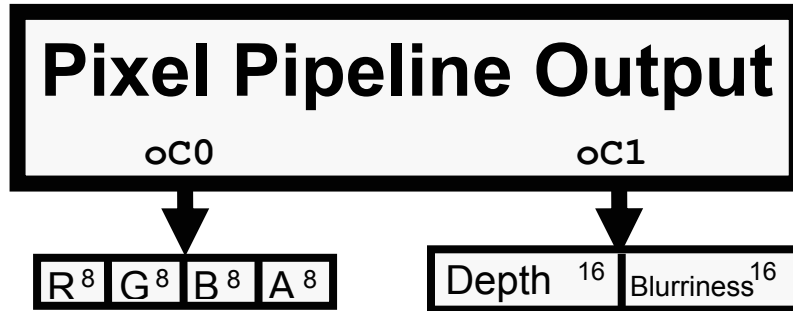


Figure 4. Pixel shader output for scene rendering pass.

Scene Rendering Vertex Shader

The vertex shader for the scene rendering pass is just a regular vertex shader with one little addition – it outputs scene depth in the camera space. This depth value is later used in the pixel shader to compute blurriness factor.

An example of a simple scene vertex shader is shown below:

```

////////////////////////////////////
float3 lightPos;    // light position in model space
float4 mtrlAmbient;
float4 mtrlDiffuse;
matrix matWorldViewProj;
matrix matWorldView;

////////////////////////////////////

struct VS_INPUT
{
    float4 vPos:        POSITION;
    float3 vNorm:       NORMAL;
    float2 vTexCoord:   TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 vPos:        POSITION;
    float4 vColor:      COLOR0;
    float  fDepth:      TEXCOORD0;
    float2 vTexCoord:   TEXCOORD1;
};
    
```

```
VS_OUTPUT scene_shader_vs(VS_INPUT v)
{
    VS_OUTPUT o = (VS_OUTPUT)0;
    float4 vPosWV;
    float3 vNorm;
    float3 vLightDir;

    // Transform position
    o.vPos = mul(v.vPos, matWorldViewProj);

    // Position in camera space
    vPosWV = mul(v.vPos, matWorldView);

    // Output depth in camera space
    o.fDepth = vPosWV.z;

    // Compute diffuse lighting
    vLightDir = normalize(lightPos - v.vPos);
    vNorm      = normalize(v.vNorm);
    o.vColor   = dot(vNorm, vLightDir) * mtrlDiffuse + mtrlAmbient;

    // Output texture coordinates
    o.vTexCoord = v.vTexCoord;

    return o;
}
```

Scene Rendering Pixel Shader

The pixel shader of the scene rendering pass needs to compute the blurriness factor and output it along with the scene depth and color. To abstract from the different display sizes and resolutions, the blurriness is defined to lie in the 0..1 range. A value of zero means the pixel is perfectly sharp, while a value of one corresponds to the pixel of the maximal circle of confusion size. The reason behind using 0..1 range is twofold. First, the blurriness is not expressed in terms of pixels and can scale with resolution during the post-processing step. Second, the values can be directly used as sample weights when eliminating “bleeding” artifacts.

For each pixel of a scene, this shader computes the circle of confusion size based on the formula provided in the preceding discussion of the thin lens model. Later in the process, the size of the circle of confusion is scaled by the factor corresponding to size of the circle in pixels for a given resolution and display size. As a last step, the blurriness value is divided by maximal desired circle of confusion size in pixels (variable `maxCoC`) and clamped to the 0..1 range. Sometimes it might be necessary to limit the circle of confusion size (through the variable `maxCoC`) to reasonable values (i.e. 10 pixels) to avoid sampling artifacts caused by an insufficient number of filter taps.

An the example of scene pixel shader that can be compiled to PS 2.0 shader model is shown below:

```

////////////////////////////////////

float focalLen;
float Dlens;
float Zfocus;
float maxCoC;
float scale;
sampler TexSampler;
float sceneRange;

////////////////////////////////////

struct PS_INPUT
{
    float4 vColor:    COLOR0;
    float  fDepth:    TEXCOORD0;
    float2 vTexCoord: TEXCOORD1;
};

struct PS_OUTPUT
{
    float4 vColor: COLOR0;
    float4 vDoF:   COLOR1;
};

////////////////////////////////////

PS_OUTPUT scene_shader_ps(PS_INPUT v)
{
    PS_OUTPUT o = (PS_OUTPUT)0;

    // Output color
    o.vColor = v.vColor * tex2D(TexSampler, v.vTexCoord);

    // Compute blur factor based on the CoC size scaled and
    // normalized to 0..1 range
    float pixCoC = abs(Dlens * focalLen * (Zfocus - v.fDepth) /
        (Zfocus * (v.fDepth - focalLen)));
    float blur = saturate(pixCoC * scale / maxCoC);

    // Depth/blurriness value scaled to 0..1 range
    o.vDoF = float4(v.fDepth / sceneRange, blur, 0, 0);

    return o;
}

```

Pass Two: Post-processing

During the post-processing pass, the results of the previous rendering are processed and the color image is blurred based on the blurriness factor computed in the first pass. Blurring is performed using a variable-sized filter representing the circle of confusion. To perform image filtering, a simple screen-aligned quadrilateral is drawn, textured with the results of the first pass. Figure 5 shows the quad's texture coordinates and screen positions for a render target of $W \times H$ dimensions. The quad corner positions are shifted by -0.5 pixels to properly align texels to pixels.

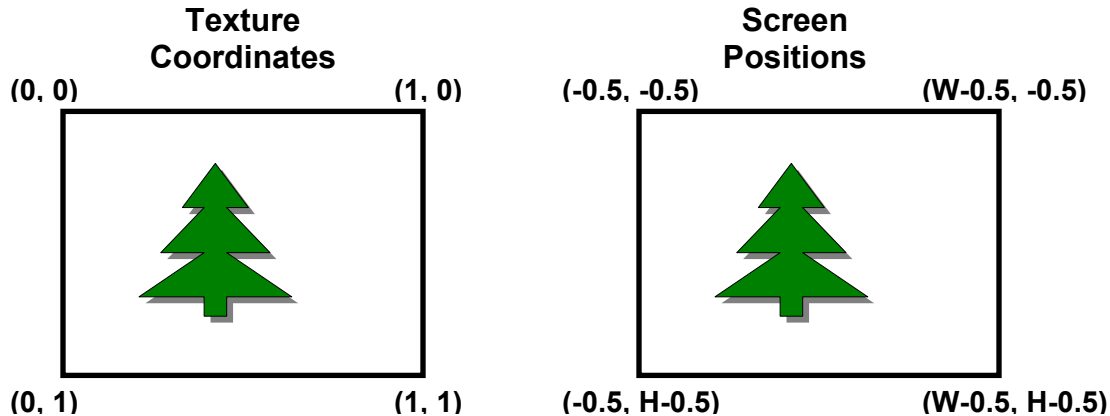


Figure 5. Texture coordinates and vertex positions for screen space quad.

This vertex shader is designed for vs_1_1 compilation target.

```
float4 viewportScale;
float4 viewportBias;

struct VS_INPUT {
    float4 vPos:    POSITION;
    float2 vTexCoord: TEXCOORD;
};

struct VS_OUTPUT {
    float4 vPos:    POSITION;
    float2 vTexCoord: TEXCOORD0;
};

VS_OUTPUT dof_filter_vs(VS_INPUT v)
{
    VS_OUTPUT o = (VS_OUTPUT)0;

    // Scale and bias viewport
    o.vPos = v.vPos * viewportScale + viewportBias;

    // Pass through the texture coordinates
    o.vTexCoord = v.vTexCoord;

    return o;
}
```

Post-processing Pixel Shader

The filter kernel in the post-processing step has 13 samples – a center sample and 12 outer samples, as shown in Figure 6. The number of samples was dictated by practical reasons of real-time implementation and represents the maximum number of samples that can be processed by a 2.0 pixel shader in a single pass.

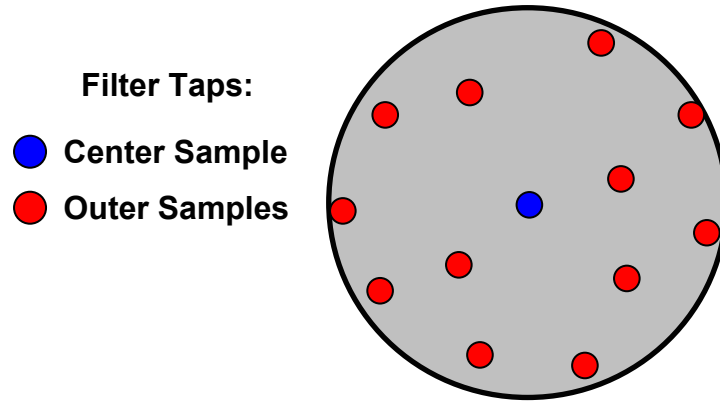


Figure 6. Depth of field filter kernel.

The center tap is aligned with the pixel being filtered, while the outer taps are sampled from nearby pixels. The filter uses stochastic sampling and the outer samples are aligned in the filter according to a Poisson disk distribution. Other sample patterns can be used to achieve specific artistic results as presented later in the section on lens Bokeh.

The filter size is computed per-pixel from the blurriness value of the center sample and the maximum allowable circle of confusion size. Figure 7 shows the relationship between blurriness and filter kernel size.

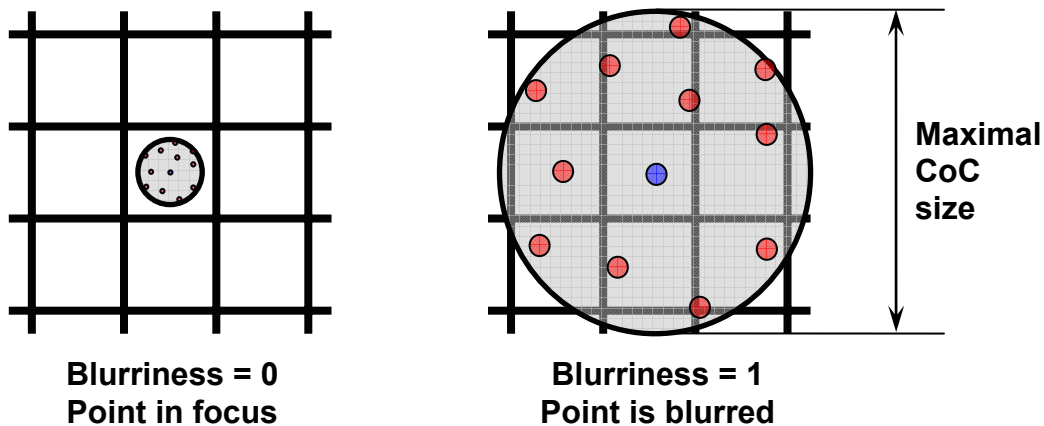


Figure 7. Relationship between blurriness and filter size.

The post-processing pixel shader computes filter sample positions based on 2D offsets stored in the `filterTaps` array and the size of the circle of confusion. The 2D offsets are locations of taps for the filter of one pixel in diameter. The following code shows how these values can be initialized in the program according to the render target resolution.

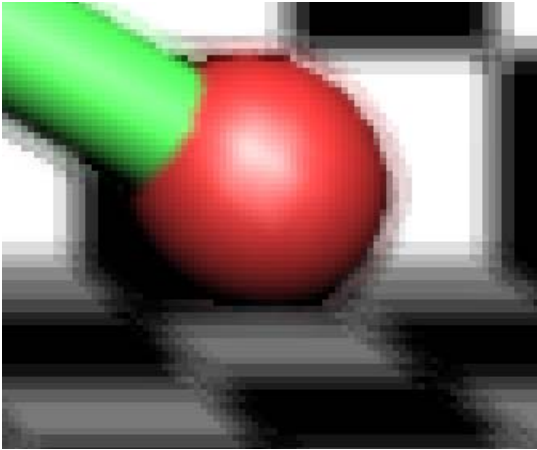
```
void SetupFilterKernel()
{
    // Scale tap offsets based on render target size
    FLOAT dx = 0.5f / (FLOAT)dwRTWidth;
    FLOAT dy = 0.5f / (FLOAT)dwRTHeight;

    D3DXVECTOR4 v[12];
    v[0] = D3DXVECTOR4(-0.326212f * dx, -0.40581f * dy, 0.0f, 0.0f);
    v[1] = D3DXVECTOR4(-0.840144f * dx, -0.07358f * dy, 0.0f, 0.0f);
    v[2] = D3DXVECTOR4(-0.695914f * dx, 0.457137f * dy, 0.0f, 0.0f);
    v[3] = D3DXVECTOR4(-0.203345f * dx, 0.620716f * dy, 0.0f, 0.0f);
    v[4] = D3DXVECTOR4(0.96234f * dx, -0.194983f * dy, 0.0f, 0.0f);
    v[5] = D3DXVECTOR4(0.473434f * dx, -0.480026f * dy, 0.0f, 0.0f);
    v[6] = D3DXVECTOR4(0.519456f * dx, 0.767022f * dy, 0.0f, 0.0f);
    v[7] = D3DXVECTOR4(0.185461f * dx, -0.893124f * dy, 0.0f, 0.0f);
    v[8] = D3DXVECTOR4(0.507431f * dx, 0.064425f * dy, 0.0f, 0.0f);
    v[9] = D3DXVECTOR4(0.89642f * dx, 0.412458f * dy, 0.0f, 0.0f);
    v[10] = D3DXVECTOR4(-0.32194f * dx, -0.932615f * dy, 0.0f, 0.0f);
    v[11] = D3DXVECTOR4(-0.791559f * dx, -0.59771f * dy, 0.0f, 0.0f);

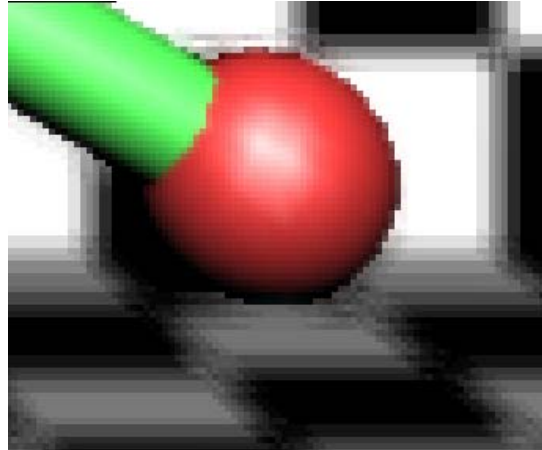
    // Set array of offsets
    pEffect->SetVectorArray("filterTaps", v, 12);
}
```

Once sample positions are computed, the filter averages color from its samples to derive the blurred color. When the blurriness value is close to zero, all samples come from the same pixel and no blurring happens. As the blurriness factor increases, the filter will start sampling from more and more neighboring pixels, thus increasingly blurring the image. All images are sampled with `D3DTEXF_LINEAR` filtering. Using linear filtering is not very accurate on the edges of objects where depth might abruptly change, however it produces better overall quality images in practice.

One of the problems commonly associated with all post-filtering methods is leaking of color from sharp objects onto the blurry backgrounds. This results in faint halos around sharp objects as can be seen on the left side of Figure 8. The color leaking happens because the filter for the blurry background will sample color from the sharp object in the vicinity due to the large filter size. To solve this problem, we will discard the outer samples that can contribute to leaking according to the following criteria: if the outer sample is in focus and it is in front of the blurry center sample, it should not contribute to the blurred color. This can introduce a minor popping effect when objects go in or out of focus. To combat sample popping, the outer sample blurriness factor is used as a sample weight to fade out its contribution gradually. The right side of Figure 8 shows a portion of a scene fragment with color leaking eliminated.



Leaking of sharp objects



Sharp objects without color leaking

Figure 8: Elimination of Color Leaking

Below, we show a depth of field pixel shader that implements the concepts discussed above. This shader can be compiled to the 2.0 pixel shader model.

```
////////////////////////////////////  
#define NUM_DOF_TAPS 12  
  
float maxCoC;  
float2 filterTaps[NUM_DOF_TAPS];  
  
////////////////////////////////////  
  
struct PS_INPUT  
{  
    float2 vTexCoord: TEXCOORD;  
};  
  
////////////////////////////////////  
  
float4 dof_filter_ps(PS_INPUT v) : COLOR  
{  
    // Start with center sample color  
    float4 colorSum = tex2D(SceneColorSampler, v.vTexCoord);  
    float totalContribution = 1.0f;  
    // Depth and blurriness values for center sample  
    float2 centerDepthBlur = tex2D(DepthBlurSampler, v.vTexCoord);  
  
    // Compute CoC size based on blurriness  
    float sizeCoC = centerDepthBlur.y * maxCoC;  
  
    // Run through all filter taps  
    for (int i = 0; i < NUM_DOF_TAPS; i++)  
    {  
        // Compute sample coordinates  
        float2 tapCoord = v.vTexCoord + filterTaps[i] * sizeCoC;  
  
        // Fetch filter tap sample  
        float4 tapColor = tex2D(SceneColorSampler, tapCoord);  
    }  
}
```

Real-Time Depth of Field Simulation

```
float2 tapDepthBlur = tex2D(DepthBlurSampler, tapCoord);

// Compute tap contribution based on depth and blurriness
float tapContribution =
    (tapDepthBlur.x > centerDepthBlur.x) ?
    1.0f : tapDepthBlur.y;

// Accumulate color and sample contribution
colorSum += tapColor * tapContribution;
totalContribution += tapContribution;
}

// Normalize color sum
float4 finalColor = colorSum / totalContribution;
return finalColor;
}
```

Now that we have discussed our implementation which models the circle of confusion with a variable-sized stochastic filter kernel, we will describe an implementation which is based on a separable Gaussian filter.

Depth of Field Rendering by Blurring with Separable Gaussian Filter

This separable Gaussian filter approach differs from the previous approach of simulating depth of field in two ways. First, it does not utilize multiple render targets for outputting depth information. Second, to simulate the blurring that occurs in depth of field, we apply a Gaussian filter during the post-processing stage instead of simulating the circle of confusion of a physical camera lens.

Implementation Overview

In this method, we first render the scene at full resolution to an offscreen buffer, outputting depth information for each pixel to the alpha channel of that buffer. We then downsample this fully-rendered scene into an image $\frac{1}{4}$ size ($\frac{1}{2}$ in x and $\frac{1}{2}$ in y) of the original. Next, we perform blurring of the downsampled scene in two passes by running the image through two passes of a separable Gaussian filter – first along the x axis and then along the y axis. On the final pass, we blend between the original full resolution rendering of our scene and the blurred post-processed image based on the distance of each pixel from the specified focal plane stored in the downsampled image. The intermediate filtering results are stored in 16-bit per channel integer format (D3DFMT_A16B16G16R16) for extra precision. We will now discuss this method in more detail, going step-by-step through the different rendering passes and shaders used.

Pass One: Scene rendering

During the scene rendering pass, we render the scene to the full resolution offscreen buffer, computing color information and a depth falloff value for each pixel. The depth falloff value will determine how much each pixel will be blurred during the subsequent post-processing stage. The distance from the focal plane is outputted to the alpha channel of the offscreen buffer.

Scene Rendering Vertex Shader

We compute the depth falloff value and the distance from the focal plane in the vertex shader. First, we determine the distance of each vertex from the focal plane in view space. We output scaled to 0..1 range distance from the focal plane into the texture coordinate interpolator. This is illustrated in this vertex shader, compiled to vertex shader target vs_1_1:

```

////////////////////////////////////

float3 lightPos;    // light position in model space
float4 mtrlAmbient;
float4 mtrlDiffuse;
matrix matWorldViewProj;
matrix matWorldView;
float fFocalDistance;
float fFocalRange;

////////////////////////////////////

struct VS_INPUT
{
    float4 vPos:      POSITION;
    float3 vNorm:     NORMAL;
    float2 vTexCoord: TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 vPos:      POSITION;
    float4 vColor:    COLOR0;
    float  fBlur:     TEXCOORD0;
    float2 vTexCoord: TEXCOORD1;
};

////////////////////////////////////

VS_OUTPUT scene_shader_vs(VS_INPUT v)
{
    VS_OUTPUT o = (VS_OUTPUT)0;
    float4 vPosWV;
    float3 vNorm;
    float3 vLightDir;

    // Transform position

```

Real-Time Depth of Field Simulation

```
o.vPos = mul(v.vPos, matWorldViewProj);

// Position in camera space
vPosWV = mul(v.vPos, matWorldView);

// Normalized distance to focal plane in camera space,
// used as a measure of blurriness for depth of field
o.fBlur = saturate(abs(vPosWV.z - fFocalDistance) /
                  fFocalRange);

// Compute diffuse lighting
vLightDir = normalize(lightPos - v.vPos);
vNorm      = normalize(v.vNorm);
o.vColor   = dot(vNorm, vLightDir) * mtrlDiffuse + mtrlAmbient;

// Output texture coordinates
o.vTexCoord = v.vTexCoord;

return o;
}
```

Scene Rendering Pixel Shader

In the pixel shader we render our scene as desired. The alpha channel receives the blurriness value expressed as the distance from the focal plane. This pixel shader is designed to be compiled into ps_2_0 target.

```
////////////////////////////////////

sampler TexSampler;

////////////////////////////////////

struct PS_INPUT
{
    float4 vColor:    COLOR0;
    float  fBlur:     TEXCOORD0;
    float2 vTexCoord: TEXCOORD1;
};

////////////////////////////////////

float4 scene_shader_ps(PS_INPUT v) : COLOR
{
    float3 vColor;

    // Output color
    vColor = v.vColor * tex2D(TexSampler, v.vTexCoord);

    // Output blurriness in alpha
    return float4(vColor, v.fBlur);
}
```

Pass Two: Downsampling

To downsample the full resolution image we simply render a quad $\frac{1}{4}$ size of the original image while sampling from the original image and outputting it to the smaller offscreen buffer. The alpha channel of the downsampled image receives blurriness value computed as the scaled distance from the focus plane for each pixel. This information will be used during post-processing to control the amount of blurring applied to the downsampled image as well as to blend between a blurred image of the scene and the original rendering to simulate the effect of depth of field.

Downsampling Vertex Shader

In this simple vertex shader we transform the vertices into clip space and propagate incoming texture coordinates to the pixel shader. Note that at this point, the incoming model must be a screen-aligned quad of dimensions $\frac{1}{4}$ size of the original image.

```
matrix matWorldViewProj;

////////////////////////////////////////////////////////////////

struct VS_OUTPUT
{
    float4 vPos: POSITION;
    float2 vTex: TEXCOORD0;
};

////////////////////////////////////////////////////////////////

VS_OUTPUT main(float4 Pos: POSITION, float2 Tex: TEXCOORD0)
{
    VS_OUTPUT o = (VS_OUTPUT)0;

    // Output transformed vertex position:
    o.vPos = mul(matWorldViewProj, Pos);
    // Propagate texturecoordinate to the pixel shader
    o.vTex = Tex;

    return o;
}
```

Downsampling Pixel Shader

In the pixel shader for downsampling pass we sample the original scene rendering using texture coordinates from the smaller screen-aligned quad and store the results in an off-screen render target. This pixel shader can be compiled to ps_1_4 or above.

```
sampler renderTexture;

float4 main(float2 Tex: TEXCOORD0) : COLOR
{
    // Downsample rendered scene:
    return tex2D(renderTexture, Tex);
}
```

Post-processing for Simulation of Depth of Field

One of the most frequently used filters for performing smoothing of an image is the Gaussian filter (see Figure 11). Typically, the filter is applied in the following way:

$$F = \frac{\sum_{i=1}^n \sum_{j=1}^n P_{ij} C_{ij}}{S},$$

where F is the filtered value of the target pixel, P is a pixel in the 2D grid, C is a coefficient in the 2D Gaussian matrix, n is the vertical/horizontal dimensions of the matrix, and S is the sum of all values in the Gaussian matrix).

Once a suitable kernel has been calculated, Gaussian smoothing can be performed using standard convolution methods. The convolution can in fact be performed fairly quickly since the equation for the 2D isotropic Gaussian is separable into x and y components. Thus, the 2D convolution can be performed by first convolving with a 1D Gaussian in the x direction, and then convolving with another 1D Gaussian in the y direction. This allows us to apply a larger size filter to the input image in two successive passes of 1D filters. We will perform this operation by rendering into a temporary buffer and sampling a line (or a column, for y axis filtering) of texels in each of the passes.

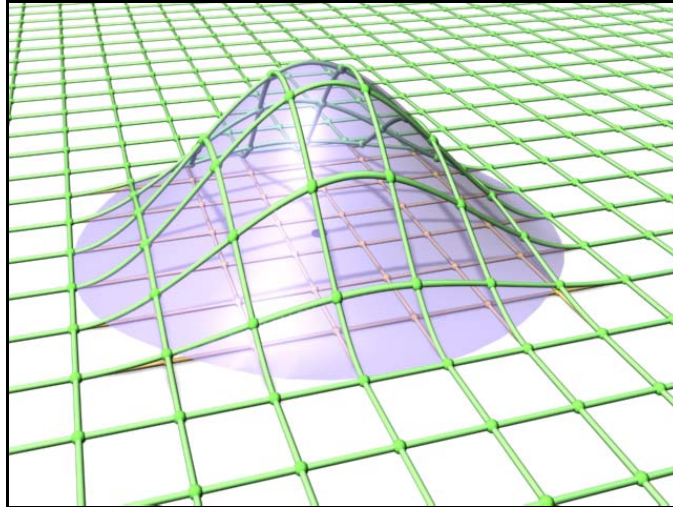


Figure 9: Gaussian filter kernel

The size of the downsampled buffer determines the size of texels used for controlling sampling points for the Gaussian filter taps. This can be precomputed as a constant to the shader ahead of time. Following is an example of how filter tap offset can be computed.

```
void SetupFilterKernel()
{
    // Scale tap offsets based on render target size
    FLOAT dx = 1.0f / (FLOAT)dwRTWidth;
    FLOAT dy = 1.0f / (FLOAT)dwRTHeight;

    D3DXVECTOR4 v[7];

    v[0] = D3DXVECTOR4(0.0f, 0.0f, 0.0f, 0.0f);
    v[1] = D3DXVECTOR4(1.3366f * dx, 0.0f, 0.0f, 0.0f);
    v[2] = D3DXVECTOR4(3.4295f * dx, 0.0f, 0.0f, 0.0f);
    v[3] = D3DXVECTOR4(5.4264f * dx, 0.0f, 0.0f, 0.0f);
    v[4] = D3DXVECTOR4(7.4359f * dx, 0.0f, 0.0f, 0.0f);
    v[5] = D3DXVECTOR4(9.4436f * dx, 0.0f, 0.0f, 0.0f);
    v[6] = D3DXVECTOR4(11.4401f * dx, 0.0f, 0.0f, 0.0f);

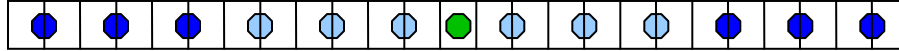
    // Set array of horizontal offsets for X-pass
    m_pEffect->SetVectorArray("horzTapOffs", v, 7);

    v[0] = D3DXVECTOR4(0.0f, 0.0f, 0.0f, 0.0f);
    v[1] = D3DXVECTOR4(0.0f, 1.3366f * dy, 0.0f, 0.0f);
    v[2] = D3DXVECTOR4(0.0f, 3.4295f * dy, 0.0f, 0.0f);
    v[3] = D3DXVECTOR4(0.0f, 5.4264f * dy, 0.0f, 0.0f);
    v[4] = D3DXVECTOR4(0.0f, 7.4359f * dy, 0.0f, 0.0f);
    v[5] = D3DXVECTOR4(0.0f, 9.4436f * dy, 0.0f, 0.0f);
    v[6] = D3DXVECTOR4(0.0f, 11.4401f * dy, 0.0f, 0.0f);

    // Set array of vertical offsets for Y-pass
    m_pEffect->SetVectorArray("vertTapOffs", v, 7);
}
```

Pass Three: Separable Gaussian Filtering in X Axis

First, we perform Gaussian filter blurring along the x axis of the downsampled image. For each pixel in the downsampled image, we sample n texture samples dynamically along the x axis in the following manner:



- Filter Taps:**
- **Center tap (nearest filtering)**
 - **Inner tap**
 - **Outer tap (dependent texture read)**

Figure 10: Samples for applying 1D Gaussian filter

The center sample and the inner taps of the filter are done with interpolated texture coordinates which are computed in the vertex shader. To compute the offsets for the first seven samples we use input texture coordinate and the precomputed tap offsets based on the image resolution.

In the pixel shader we sample the image for the center tap and first 6 inner taps, using nearest filtering for the center sample and bilinear sampling for the inner samples.

The pixel shader code derives the texture coordinates for the outer samples based on pre-computed deltas from the location of the center sample. The outer samples are fetched via dependent reads as texture coordinates are derived in the pixel shader itself.

All samples are weighted based on the predefined weight thresholds and blurriness values and added together. This results in a weighted sum of 25 texels from the source image, which is large enough to allow us to create a convincing blurring effect for simulating depth of field without violating the maximum number of instructions for 2.0 pixel shader.

Note that the output of this pass is directed to a separate off-screen buffer. At this point we have used three separate off-screen render targets: one to output results of the full scene rendering, one to output results of downsampling pass, and one to output results of Gaussian blurring.

Vertex Shader for X Axis of Separable Gaussian Filter

This vertex shader is designed for vs_1_1 compilation target.

```
float4 viewportScale;
float4 viewportBias;
// Offsets 0-3 used by vertex shader, 4-6 by pixel shader
float2 horzTapOffs[7];

struct VS_INPUT
{
    float4 vPos:        POSITION;
    float2 vTexCoord:  TEXCOORD;
};

struct VS_OUTPUT_TEX7
{
    float4 vPos:        POSITION;
    float2 vTap0:       TEXCOORD0;
    float2 vTap1:       TEXCOORD1;
    float2 vTap2:       TEXCOORD2;
    float2 vTap3:       TEXCOORD3;
    float2 vTap1Neg:    TEXCOORD4;
    float2 vTap2Neg:    TEXCOORD5;
    float2 vTap3Neg:    TEXCOORD6;
};

VS_OUTPUT_TEX7 filter_gaussian_x_vs(VS_INPUT v)
{
    VS_OUTPUT_TEX7 o = (VS_OUTPUT_TEX7)0;

    // Scale and bias viewport
    o.vPos = v.vPos * viewportScale + viewportBias;

    // Compute tap coordinates
    o.vTap0 = v.vTexCoord;
    o.vTap1 = v.vTexCoord + horzTapOffs[1];
    o.vTap2 = v.vTexCoord + horzTapOffs[2];
    o.vTap3 = v.vTexCoord + horzTapOffs[3];
    o.vTap1Neg = v.vTexCoord - horzTapOffs[1];
    o.vTap2Neg = v.vTexCoord - horzTapOffs[2];
    o.vTap3Neg = v.vTexCoord - horzTapOffs[3];

    return o;
}
```

Pixel Shader for X Axis of Separable Gaussian Filter

This pixel shader is fine-tuned to compile for ps_2_0 compilation target.

```
// Thresholds for computing sample weights
float4 vThresh0 = {0.1, 0.3, 0.5, -0.01};
float4 vThresh1 = {0.6, 0.7, 0.8, 0.9};

sampler renderTexture;
// Offsets 0-3 used by vertex shader, 4-6 by pixel shader
float2 horzTapOffs[7];

struct PS_INPUT_TEX7
{
    float2 vTap0:    TEXCOORD0;
    float2 vTap1:    TEXCOORD1;
    float2 vTap2:    TEXCOORD2;
    float2 vTap3:    TEXCOORD3;
    float2 vTap1Neg: TEXCOORD4;
    float2 vTap2Neg: TEXCOORD5;
    float2 vTap3Neg: TEXCOORD6;
};

float4 filter_gaussian_x_ps(PS_INPUT_TEX7 v) : COLOR
{
    // Samples
    float4 s0, s1, s2, s3, s4, s5, s6, vWeights4;
    float3 vWeights3, vColorSum;
    float fWeightSum;

    // Sample taps with coordinates from VS
    s0 = tex2D(renderTexture, v.vTap0);
    s1 = tex2D(renderTexture, v.vTap1);
    s2 = tex2D(renderTexture, v.vTap2);
    s3 = tex2D(renderTexture, v.vTap3);
    s4 = tex2D(renderTexture, v.vTap1Neg);
    s5 = tex2D(renderTexture, v.vTap2Neg);
    s6 = tex2D(renderTexture, v.vTap3Neg);

    // Compute weights for 4 first samples (including center tap)
    // by thresholding blurriness (in sample alpha)
    vWeights4.x = saturate(s1.a - vThresh0.x);
    vWeights4.y = saturate(s2.a - vThresh0.y);
    vWeights4.z = saturate(s3.a - vThresh0.x);
    vWeights4.w = saturate(s0.a - vThresh0.w);

    // Accumulate weighted samples
    vColorSum = s0 * vWeights4.x + s1 * vWeights4.y +
                s2 * vWeights4.z + s3 * vWeights4.w;

    // Sum weights using DOT
    fWeightSum = dot(vWeights4, 1);

    // Compute weights for 3 remaining samples
    vWeights3.x = saturate(s4.a - vThresh0.x);
    vWeights3.y = saturate(s5.a - vThresh0.y);
```

```

vWeights3.z = saturate(s6.a - vThresh0.z);

// Accumulate weighted samples
vColorSum += s4 * vWeights3.x + s4 * vWeights3.y +
             s6 * vWeights3.z;

// Sum weights using DOT
fWeightSum += dot(vWeights3, 1);

// Compute tex coords for other taps
float2 vTap4    = v.vTap0 + horzTapOffs[4];
float2 vTap5    = v.vTap0 + horzTapOffs[5];
float2 vTap6    = v.vTap0 + horzTapOffs[6];
float2 vTap4Neg = v.vTap0 - horzTapOffs[4];
float2 vTap5Neg = v.vTap0 - horzTapOffs[5];
float2 vTap6Neg = v.vTap0 - horzTapOffs[6];

// Sample the taps
s0 = tex2D(renderTexture, vTap4);
s1 = tex2D(renderTexture, vTap5);
s2 = tex2D(renderTexture, vTap6);
s3 = tex2D(renderTexture, vTap4Neg);
s4 = tex2D(renderTexture, vTap5Neg);
s5 = tex2D(renderTexture, vTap6Neg);

// Compute weights for 3 samples
vWeights3.x = saturate(s0.a - vThresh1.x);
vWeights3.y = saturate(s1.a - vThresh1.y);
vWeights3.z = saturate(s2.a - vThresh1.z);

// Accumulate weighted samples
vColorSum += s0 * vWeights3.x + s1 * vWeights3.y +
             s2 * vWeights3.z;

// Sum weights using DOT
fWeightSum += dot(vWeights3, 1);

// Compute weights for 3 samples
vWeights3.x = saturate(s3.a - vThresh1.x);
vWeights3.y = saturate(s4.a - vThresh1.y);
vWeights3.z = saturate(s5.a - vThresh1.z);

// Accumulate weighted samples
vColorSum += s3 * vWeights3.x + s4 * vWeights3.y +
             s5 * vWeights3.z;

// Sum weights using DOT
fWeightSum += dot(vWeights3, 1);

// Divide weighted sum of samples by sum of all weights
vColorSum /= fWeightSum;

// Color and weights sum output scaled (by 1/256)
// to fit values in 16 bit 0 to 1 range
return float4(vColorSum, fWeightSum) * 0.00390625;
}

```

Pass Four: Separable Gaussian Filtering in Y axis

In the next pass we perform a similar operation but blurring along the y axis. The input to this pass is the image that we just blurred along the x axis. The output of this pass is directed to an off-screen render target (`blurredXYTexture`), which is going to be used during final image compositing.

Vertex Shader for Y Axis of Separable Gaussian Filter

In the vertex shader we again compute the first set of texture samples offsets to be used in the pixel shader for sampling the pre-blurred image. This vertex shader uses exactly the same approach as the vertex shader in the previous pass, but with different offset values. This particular vertex shader is designed for `vs_1_1` compilation target.

```

////////////////////////////////////
float4 viewportScale;
float4 viewportBias;
// Offsets 0-3 used by vertex shader, 4-6 by pixel shader
float2 vertTapOffs[7];

////////////////////////////////////

struct VS_INPUT
{
    float4 vPos:        POSITION;
    float2 vTexCoord:  TEXCOORD;
};

struct VS_OUTPUT_TEX7
{
    float4 vPos:        POSITION;
    float2 vTap0:       TEXCOORD0;
    float2 vTap1:       TEXCOORD1;
    float2 vTap2:       TEXCOORD2;
    float2 vTap3:       TEXCOORD3;
    float2 vTap1Neg:    TEXCOORD4;
    float2 vTap2Neg:    TEXCOORD5;
    float2 vTap3Neg:    TEXCOORD6;
};

////////////////////////////////////

VS_OUTPUT_TEX7 filter_gaussian_y_vs(VS_INPUT v)
{
    VS_OUTPUT_TEX7 o = (VS_OUTPUT_TEX7)0;

    // Scale and bias viewport
    o.vPos = v.vPos * viewportScale + viewportBias;

    // Compute tap coordinates
    o.vTap0   = v.vTexCoord;
    o.vTap1   = v.vTexCoord + vertTapOffs[1];

```

```

o.vTap2    = v.vTexCoord + vertTapOffs[2];
o.vTap3    = v.vTexCoord + vertTapOffs[3];
o.vTap1Neg = v.vTexCoord - vertTapOffs[1];
o.vTap2Neg = v.vTexCoord - vertTapOffs[2];
o.vTap3Neg = v.vTexCoord - vertTapOffs[3];

return o;
}

```

Pixel Shader for Y Axis of Separable Gaussian Filter

Similarly to processing the image in the previous pass, we again sample the first 7 samples along the *y* axis using interpolated texture offsets and combine these samples using appropriate kernel weights. Then we compute next 6 offset coordinates and sample the image using dependent texture reads. Finally we combine all weighted samples and output the value into an offscreen buffer. This pixel shader is compiled to ps_2_0 target.

```

////////////////////////////////////
float4 vWeights0 = {0.080, 0.075, 0.070, 0.100};
float4 vWeights1 = {0.065, 0.060, 0.055, 0.050};

sampler blurredXTexture;
// Offsets 0-3 used by vertex shader, 4-6 by pixel shader
float2 vertTapOffs[7];

////////////////////////////////////

struct PS_INPUT_TEX7
{
    float2 vTap0:    TEXCOORD0;
    float2 vTap1:    TEXCOORD1;
    float2 vTap2:    TEXCOORD2;
    float2 vTap3:    TEXCOORD3;
    float2 vTap1Neg: TEXCOORD4;
    float2 vTap2Neg: TEXCOORD5;
    float2 vTap3Neg: TEXCOORD6;
};

////////////////////////////////////

float4 filter_gaussian_y_ps(PS_INPUT_TEX7 v) : COLOR
{
    float4 s0, s1, s2, s3, s4, s5, s6;
    // Acumulated color and weights
    float4 vColorWeightSum;

    // Sample taps with coordinates from VS
    s0 = tex2D(blurredXTexture, v.vTap0);
    s1 = tex2D(blurredXTexture, v.vTap1);
    s2 = tex2D(blurredXTexture, v.vTap2);
    s3 = tex2D(blurredXTexture, v.vTap3);
    s4 = tex2D(blurredXTexture, v.vTap1Neg);
    s5 = tex2D(blurredXTexture, v.vTap2Neg);
    s6 = tex2D(blurredXTexture, v.vTap3Neg);
}

```

Real-Time Depth of Field Simulation

```
// Modulate sampled color values by the weights stored
// in the alpha channel of each sample
s0.rgb = s0.rgb * s0.a;
s1.rgb = s1.rgb * s1.a;
s2.rgb = s2.rgb * s2.a;
s3.rgb = s3.rgb * s3.a;
s4.rgb = s4.rgb * s4.a;
s5.rgb = s5.rgb * s5.a;
s6.rgb = s6.rgb * s6.a;

// Aggregate all samples weighting them with pre-defined
// kernel weights, weight sum in alpha
vColorWeightSum = s0 * vWeights0.w +
    (s1 + s4) * vWeights0.x +
    (s2 + s5) * vWeights0.y +
    (s3 + s6) * vWeights0.z;

// Compute tex coords for other taps
float2 vTap4    = v.vTap0 + vertTapOffs[4];
float2 vTap5    = v.vTap0 + vertTapOffs[5];
float2 vTap6    = v.vTap0 + vertTapOffs[6];
float2 vTap4Neg = v.vTap0 - vertTapOffs[4];
float2 vTap5Neg = v.vTap0 - vertTapOffs[5];
float2 vTap6Neg = v.vTap0 - vertTapOffs[6];

// Sample the taps
s0 = tex2D(blurredXTexture, vTap4);
s1 = tex2D(blurredXTexture, vTap5);
s2 = tex2D(blurredXTexture, vTap6);
s3 = tex2D(blurredXTexture, vTap4Neg);
s4 = tex2D(blurredXTexture, vTap5Neg);
s5 = tex2D(blurredXTexture, vTap6Neg);

// Modulate sampled color values by the weights stored
// in the alpha channel of each sample
s0.rgb = s0.rgb * s0.a;
s1.rgb = s1.rgb * s1.a;
s2.rgb = s2.rgb * s2.a;
s3.rgb = s3.rgb * s3.a;
s4.rgb = s4.rgb * s4.a;
s5.rgb = s5.rgb * s5.a;

// Aggregate all samples weighting them with pre-defined
// kernel weights, weight sum in alpha
vColorWeightSum += (s1 + s3) * vWeights1.x +
    (s1 + s4) * vWeights1.y +
    (s2 + s5) * vWeights1.z;

// Average combined sample for all samples in the kernel
vColorWeightSum.rgb /= vColorWeightSum.a;

// Account for scale factor applied in previous pass
// (blur along the X axis) to output values
// in 16 bit 0 to 1 range
return vColorWeightSum * 256.0;
}
```

Figure 11 shows the result of applying the 25×25 separable Gaussian to the downsampled image:



Figure 11 - 25×25 Gaussian Blurred image

Pass Five: Compositing the Final Output

In the final pass we create a composite image of the actual scene rendering with the Gaussian blurred image using the distance from the focal plane information that is stored in the alpha channel of the original image. The two offscreen render target are used to sample that information (in our example, `renderTexture` is used to sample from full scene rendering pass results, and `blurredXYTexture` contains results of applying Gaussian filtering to the downsampled image). All textures are sampled using interpolated texture coordinates.

Vertex Shader for Final Composite Pass

Real-Time Depth of Field Simulation

In this vertex shader, we simply transform the vertices and propagate the texture coordinate to the pixel shader. The vertex shader is designed to compile to vs_1_1 target.

```
////////////////////////////////////  
float4 viewportScale;  
float4 viewportBias;  
////////////////////////////////////  
struct VS_INPUT  
{  
    float4 vPos: POSITION;  
    float2 vTex: TEXCOORD;  
};  
struct VS_OUTPUT  
{  
    float4 vPos: POSITION;  
    float2 vTex: TEXCOORD0;  
};  
////////////////////////////////////  
VS_OUTPUT final_pass_vs(VS_INPUT v)  
{  
    VS_OUTPUT o = (VS_OUTPUT)0;  
    // Scale and bias viewport  
    o.vPos = v.vPos * viewportScale + viewportBias;  
    // Propagate texture coordinate to the pixel shader  
    o.vTex = v.vTex;  
    return o;  
}
```

Pixel Shader for Final Composite Pass

In this pixel shader we actually do the compositing of the final image. In the pixel shader we retrieve the depth falloff distance stored in the downsampled image's alpha channel. This focal plane distance is used as a blending weight to blend between the post-processed Gaussian-blurred image and the original full resolution scene rendering. This pixel shader is designed to compile to ps_1_4 target or above.

```
////////////////////////////////////  
sampler blurredXYTexture;  
sampler renderTexture;  
  
////////////////////////////////////  
float4 final_pass_ps(float2 Tex: TEXCOORD0) : COLOR  
{  
    // Sample Gaussian-blurred image  
    float4 vBlurred = tex2D(blurredXYTexture, Tex);  
  
    // Sample full-resolution scene rendering result  
    float4 vFullres = tex2D(renderTexture, Tex);  
  
    // Interpolate between original full-resolution and  
    // blurred images based on blurriness  
    float3 vColor = lerp(vFullres, vBlurred, vFullres.a);  
  
    return float4(vColor, 1.0);  
}
```

Figure 12 shows result of final compositing stage:



Figure 12: Final composite image for depth of field effect

The images for Figure 11 and 12 are taken from a screen saver from ATI RADEON 9700 demo suite, which you can download here:

<http://www.ati.com/developer/screensavers.html>

Bokeh

It has been noticed that different lenses with the same apertures and focal distances produce slightly different out-of-focus images. In photography the “quality” of an out-of-focus or blurred image is described by the Japanese term “*bokeh*”. While this term is mostly familiar to photographers, it is relatively new to computer graphics professionals.

The perfect lens should have no spherical aberration, and should focus incoming rays in a perfect cone of light behind the lens. In such a camera, if the image is not in focus, each blurred point is represented by a uniformly illuminated circle of confusion. All real lenses have some degree of spherical aberration, and always have non-uniform distribution of light in the light cone and thus in the circle of confusion. The lens’

diaphragm and number of shutter blades can also have some effect on the shape of the circle of confusion. The term “bokeh”, which is a Japanese phoneticization of the French word *bouquet*, describes this phenomenon and is a subjective factor, meaning that there is no objective way to measure people’s reaction to this phenomenon. What might be considered “bad” bokeh under certain circumstances, can be desirable for some artistic effects, and vice versa.

To simulate different lens bokeh, one can use filters with different distributions and weightings of filter taps. Figure 13 demonstrates part of the same scene processed with blur filters of the same size but with different filter taps distributions and weightings.

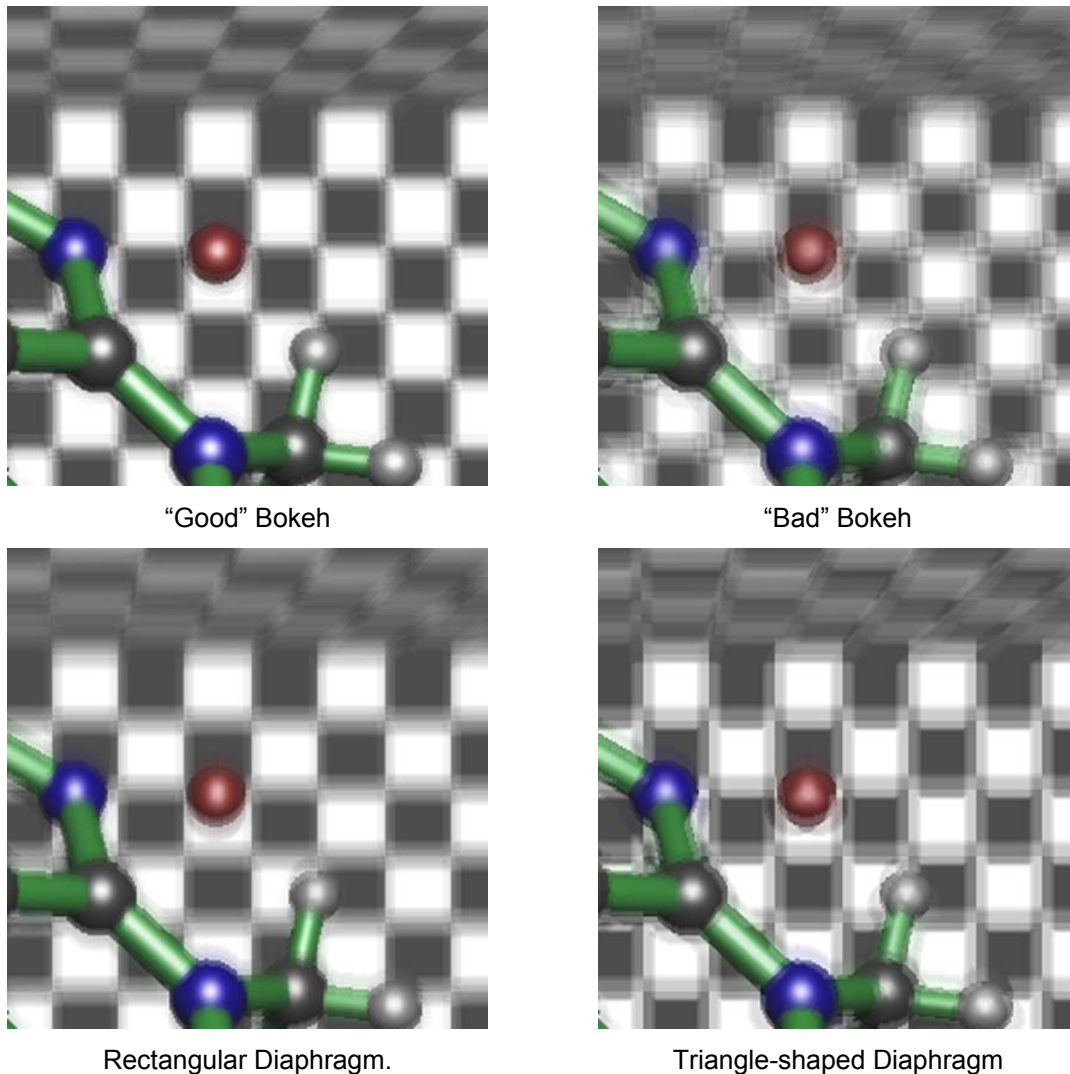


Figure 13 – Real-Time Lens Bokeh

Summary

This article presented two different real-time implementations of a depth of field effect using DirectX® 9 class programmable graphics hardware. Simulating depth of field is very important for creating convincing visual representations of our world on the computer screen. It can also be used in artistic ways to generate more cinematic looking visual effects.

Reference

1. [Potmesil83] ***Modeling motion blur in computer-generated images***, Michael Potmesil and Indranil Chakravarty, *Siggraph Proceedings of the 10th annual conference on Computer Graphics and interactive Techniques*, 1983