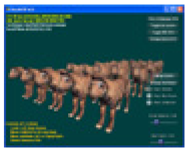
DepthOfField Sample

⊟ Collapse All

This sample shows several techniques for creating a depth-of-field effect in which objects are only in focus at a given distance from the camera and out of focus at other distances. Rendering objects out of focus adds realism to the scene. The methods it shows are reasonably cheap to perform on most hardware, and the depth of field post processing can easily be combined with other post process techniques such as image-based motion blur or high dynamic range (HDR) lighting.



Path

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

Background

In CMyD3DApplication::InitDeviceObjects it loads the geometry with **D3DXLoadMeshFromX** and calls **D3DXComputeNormals** to create mesh normals if there aren't any. It then calls **ID3DXMesh::OptimizeInplace** to optimize the mesh for the vertex cache. It then loads the textures using **D3DXCreateTextureFromFile**.

CMyD3DApplication::RestoreDeviceObjects does the following:

- Sets up the camera class. The camera class encapsulates handling mouse and keyboard input and updating the view matrix accordingly.
- Calls **D3DXCreateTexture** to create a D3DUSAGE_RENDERTARGET A8R8G8B8 texture with the same height and width as the back buffer. If that fails, it tries again but without the D3DUSAGE_RENDERTARGET flag.
- Creates a **ID3DXRenderToSurface** with **D3DXCreateRenderToSurface** to help render to the texture on cards that don't support render targets.
- Loads the D3DX effect file into a ID3DXEffectxxx, and caches the D3DXHANDLE to all the parameters that it will set every frame.
- Initializes effect parameters that don't change every frame, such as the material properties and render target texture, using **ID3DXBaseEffect::SetVector** and **ID3DXBaseEffect::SetTexture**.
- Finds the first valid technique in the effect for the current device.
- Creates a quad of vertices to be used in the post-processing pass. The quad has 6 sets of texture coords defined. Each set of texture coordinates is slightly shifted so when they are used in post process the pixel shader effect will be a blur. Note that the entire set of texture coordinates aren't used in every technique. How this quad is used will be explained below in more detail.
- Scales the effect's kernel TwelveKernelBase from pixel space (where 1.0 means 1 pixel) to texture coordinate space and stores the result into a separate array labeled TwelveKernel.

In CMyD3DApplication::FrameMove it simply calls FrameMove() on the camera class so it can move the view matrix according the user input over time.

CMyD3DApplication::Render essentially does 2 things:

1. It first renders the scene into a render target while it simultaneously calculates the blur factor and stores this in the alpha channel of the render target. An alpha value of 0 means that specific pixel is not blurry while an alpha value of 1 means that specific pixel is blurry.
2. It then uses this render target with depth information stored in the alpha channel and renders a full screen quad to execute a pixel shader for each pixel in the final surface.

In more detail, the first part of CMyD3DApplication::Render uses the technique WorldWithBlurFactor which uses the vs_1_1 vertex shader called WorldVertexShader. This vertex shader transforms vertex position into screen space, performs a simple N dot L lighting equation, copies texture coordinates, and computes the blur factor based on the depth. The blur factor is calculated with these equations:

```
fBlurFactor = dot(float4(vViewPosition, 1.0), vFocalPlane)*fHyperfocalDistance;
Output.Diffuse.a = fBlurFactor*fBlurFactor;
```

This equation calculates the distances from a point to a plane and scales the result by some factor. The hyperfocal distance is a factor that is the distance from clear to blurry. The blur factor is then squared and capped at 0.75f so that the center pixel is always weighted in the blurred image, thus reducing artifacts.

The technique WorldWithBlurFactor then uses a ps_1_1 pixel shader called WorldPixelShader which simply modulates the diffuse vertex color with the mesh's texture color. This could be done with texture blending if desired.

The second part of CMyD3DApplication::Render is where the blur actually happens. It uses one of five techniques to show off essentially 2 different pixel shader blur techniques.

The techniques UsePS20ThirteenLookups and UsePS20SevenLookups both use the ps_2_0 pixel shader called DepthOfFieldManySamples (but compile it using different parameters). The pixel shader is fairly simple: for a fixed number of samples it uses an array of values, called TwelveKernel, to offset the original UV texture coords and calls tex2D() with this new UV on a texture which is the render target created above. The kernel is defined to be two hexagons: one a single pixel out and the other two pixels out with close to "equal" spacing between the points in order to get the best blurring. For each sample it keeps a running sum of the color with this equation:

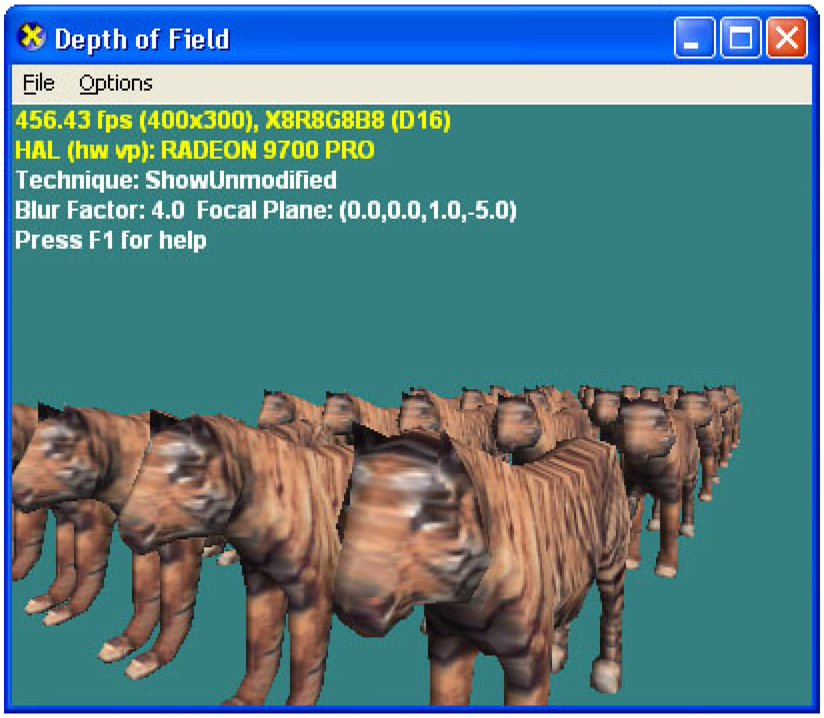
```
Blurred += lerp(Original.rgb, Current.rgb, saturate(Original.a*Current.a));
```

After all the samples have been taken, it returns this color divided by number of samples.

The other techniques UsePS20SixTexcoords,UsePS11FourTexcoordsNoRings, and UsePS11FourTexcoordsWithRings all are similar. Unlike the above, they don't use a kernel to offset the texture coordinates but instead use separate sets of texture coords that are precomputed. These texture coordinates are created in the application function called CMyD3DApplication::SetupQuad and each set is offset to create a blur when used in a very similar way as done above in DepthOfFieldManySamples. The advantage to this method is that it can be done with ps_1_1 hardware.

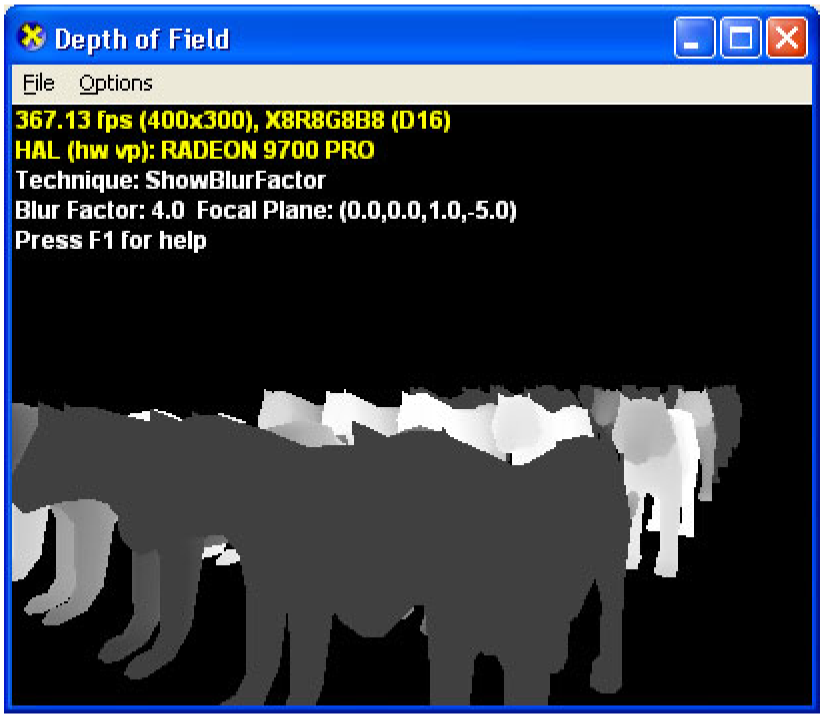
The difference between UsePS11FourTexcoordsNoRings, and UsePS11FourTexcoordsWithRings is that UsePS11FourTexcoordsWithRings linearly interpolates between the center pixel's RGB and the 3 jitter pixel's RGB based on the alpha value (the blur factor) of the center pixel. If one of the jitter pixels is not blurry while the center pixel is blurry, then the jitter pixel is incorrectly averaged in. However, UsePS11FourTexcoordsNoRings linearly interpolates between the center pixel's RGB and the 3 jitter pixel's RGB based on the value of center pixel's alpha multiplied by the jitter pixel alpha. Since 0 means no blur, this means that if one of the jitter pixels is not blurry while the center pixel is blurry then 0 * 1 = 0 so that the jitter pixel is not averaged in, causing a better result without artifacts.

For example, here is an original unblurred scene:



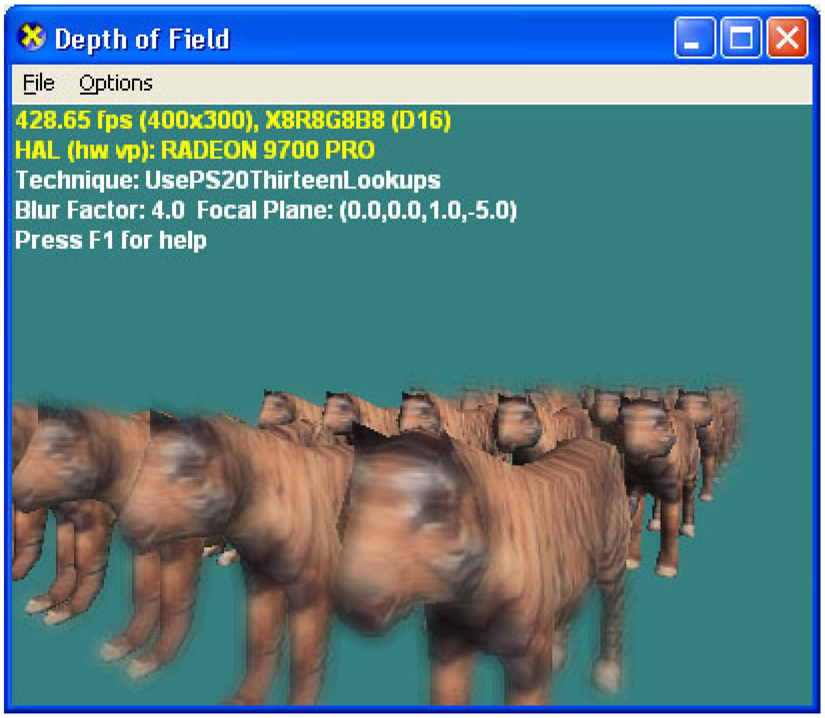
```
456.43 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO
Technique: ShowUnmodified
Blur Factor: 4.0  Focal Plane: (0.0,0.0,1.0,-5.0)
Press F1 for help
```

By pressing I, the sample will visually show you the blur factor of the scene. White means that pixel is not blurry (alpha=0.0f), and black means that pixel is blurry (alpha=1.0f).



```
367.13 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO
Technique: ShowBlurFactor
Blur Factor: 4.0  Focal Plane: (0.0,0.0,1.0,-5.0)
Press F1 for help
```

Here's what the final scene looks like:



```
428.65 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO
Technique: UsePS20ThirteenLookups
Blur Factor: 4.0  Focal Plane: (0.0,0.0,1.0,-5.0)
Press F1 for help
```