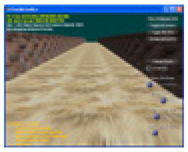
PixelMotionBlur Sample

Collapse All

Motion blur adds realism to the scene. It has the perceptual effect of creating high-speed motion. Instead of rendering the geometry multiple times with different alpha values to create a blur effect, this sample shows off a realistic image-based motion blur effect. While the scene is rendered, the shaders record the per-pixel velocity relative to the previous frame. This per-pixel velocity is then used in a post-process pass to blur the pixels in the final image.



Path

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

Background

In CMyD3DApplication::InitDeviceObjects, the sample:

1. Loads the geometry with **D3DXLoadMeshFromX**.
2. Calls **D3DXComputeNormals** to create mesh normals if there aren't any.
3. Calls **ID3DXMesh::OptimizeInplace** to optimize the mesh for the vertex cache.
4. Loads the textures using **D3DXCreateTextureFromFile**.
5. Sets up the structures to describe the scenes.
6. Sets up the camera.

In CMyD3DApplication::RestoreDeviceObjects, the sample:

- Uses **D3DXCreateTexture** to create a **D3DFMT_A8R8G8B8** render target that is the same size as the back buffer. This render target will be used by the pixel shader to render the scene's RGB color.
- Uses **D3DXCreateTexture** to create two more render targets that are the same size as the back buffer, but are instead floating point 16 bits per channel **D3DFMT_G16R16F**. These textures are used by the pixel shader to render the pixel's velocity. One of these textures will hold the per-pixel velocity of the previous frame and the other will hold the per-pixel velocity of the current frame. The usage of these render targets are explained in more detail below.
- Uses **D3DXCreateEffectFromFile** to load the .fx file into an **ID3DXEffect** and it caches the D3DXHANDLE to all the parameters that it will use every frame.
- Creates a quad of vertices to be used in the post-processing pass. The quad has a single set of texture coordinates defined. These texture coordinates are set up so that the texels of the render target will map directly to pixels. How this quad is used is explained in more detail below.
- Clears all of the render targets and sets some **ID3DXEffect** parameters to default values.

In CMyD3DApplication::FrameMove, the sample:

- Calls FrameMove on the camera class so it can move the view matrix according the user input over time.
- Swaps the pointers of the current frame's per-pixel velocity texture and the last frame's per-pixel velocity texture.
- Animates the objects in the scene, based on a simple function of time.
- Calls Sleep based on a user's setting. This simulates time spent doing complex graphics, artificial intelligence (AI), physics, networking, etc. This is useful in this sample to slow down the frame rate and thus have more motion between each frame.

The sample essentially does two things in CMyD3DApplication::Render:

1. Renders the scene's RGB into one render target, m_pFullScreenRenderTargetSurf, while it simultaneously renders the pixel's velocity into a second render target, m_pCurFrameVelocitySurf. This is possible only if the device caps report NumSimultaneousRTs greater than one; otherwise, this will need to be done in two passes.
2. Does a post-process pass. It renders a full-screen quad to execute a pixel shader for each pixel. The pixel shader uses the render target with velocity information to blur the color of the pixel using a filter, based on the direction of pixel's velocity.

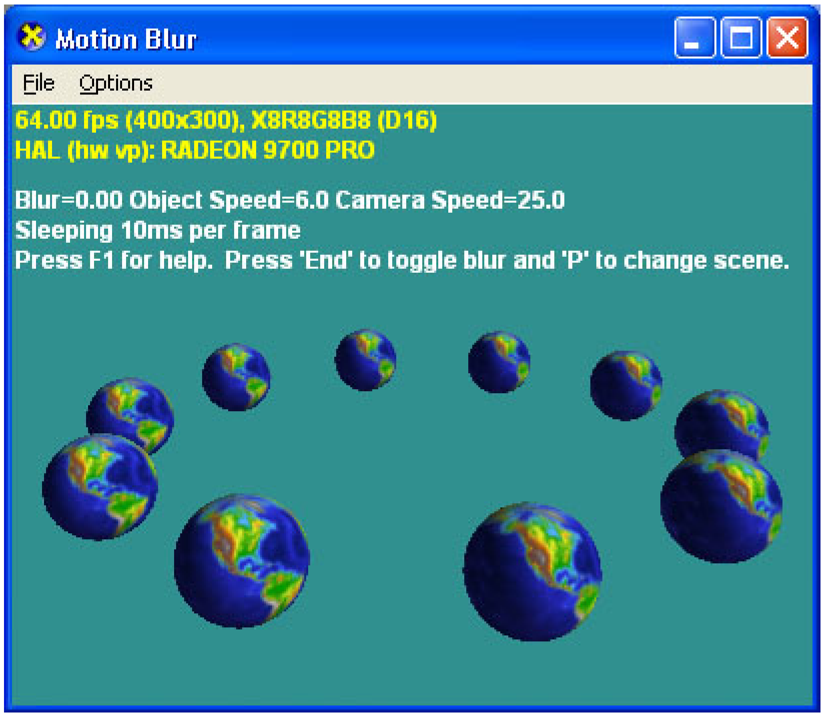
In more detail, the first part of CMyD3DApplication::Render uses the technique WorldWithVelocity, which uses a vs_2_0 vertex shader called WorldVertexShader. This vertex shader transforms vertex position into screen space, performs a simple N • L lighting equation, copies texture coordinates, and computes the velocity of the vertex by transforming the vertex by both the current and previous [world * view * projection] matrix and taking the difference. The difference will be the amount that this vertex has moved in screen space since the last frame. This per-vertex velocity is passed to the pixel shader by **TEXCOORD1**.

The technique WorldWithVelocity then uses a ps_2_0 pixel shader called WorldPixelShader, which modulates the diffuse vertex color with the mesh's texture color and writes this to COLOR0 and also writes the velocity of the pixel per frame in COLOR1. The render target for COLOR1 is **D3DFMT_G16R16F**, a floating point, 16-bits-per-channel texture. This render target records the amount the pixel has moved in screen space. The x direction change is the red channel, and the y direction change is in the green channel. The interpolation from per-vertex velocity to per-pixel velocity is done automatically because the velocity is passed to the pixel shader using a texture coordinate.

The second part of CMyD3DApplication::Render is where the motion blur actually happens. It uses the technique called PostProcessMotionBlur to do the post-process motion blur. The technique PostProcessMotionBlur uses a ps_2_0 pixel shader called PostProcessMotionBlurPS. This pixel shader does two texture look-ups to get the per-pixel velocity of the current frame (from CurFrameVelocityTexture) and of the last frame (from LastFrameVelocityTexture). If it just used the current frame's pixel velocity, then it wouldn't blur where the object was in the last frame because the current velocity at those pixels would be 0. Instead, you could filter neighboring pixels to see if any have a nonzero velocity. This requires a large number of texture look-ups which are limited and also may not work if the object has moved too far. So instead, the shader uses the larger of the current velocity or the last frame's velocity.

Once it has chosen the value to use as the pixel's velocity, it samples twelve times along the direction of the pixel's velocity. While sampling, it accumulates the color and divides by the number of samples to get the average color of all the samples and writes this as the final color for that pixel.

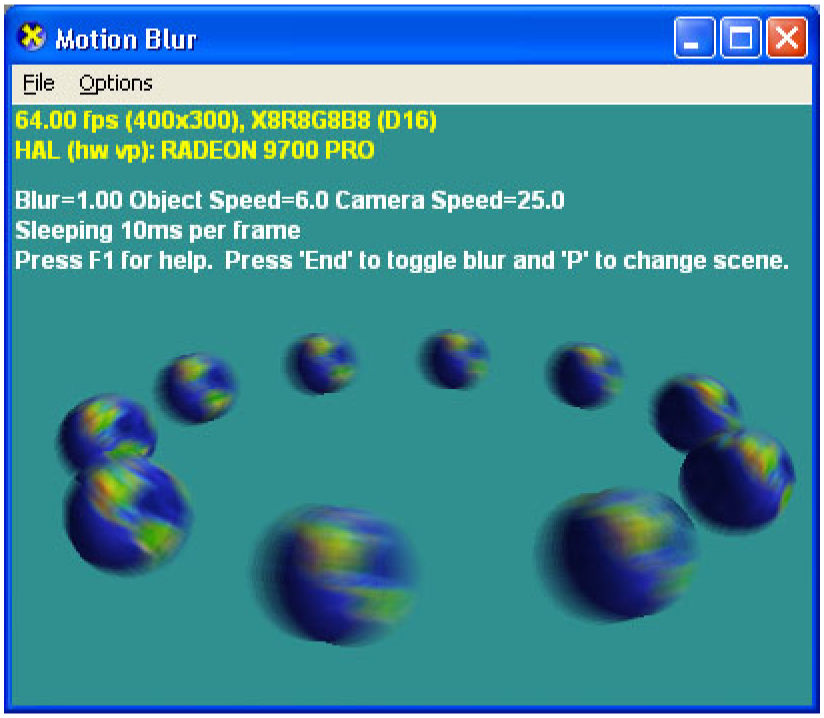
For example, here is an original unblurred scene:



64.00 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO

Blur=0.00 Object Speed=6.0 Camera Speed=25.0
Sleeping 10ms per frame
Press F1 for help. Press 'End' to toggle blur and 'P' to change scene.

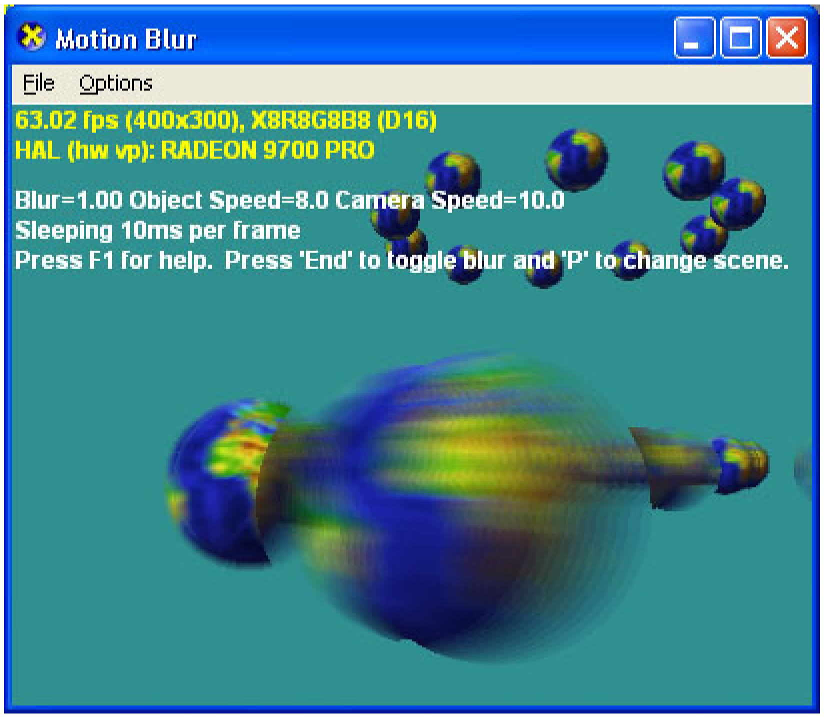
And here's what a blurred scene looks like:



```
64.00 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO

Blur=1.00 Object Speed=6.0 Camera Speed=25.0
Sleeping 10ms per frame
Press F1 for help.  Press 'End' to toggle blur and 'P' to change scene.
```

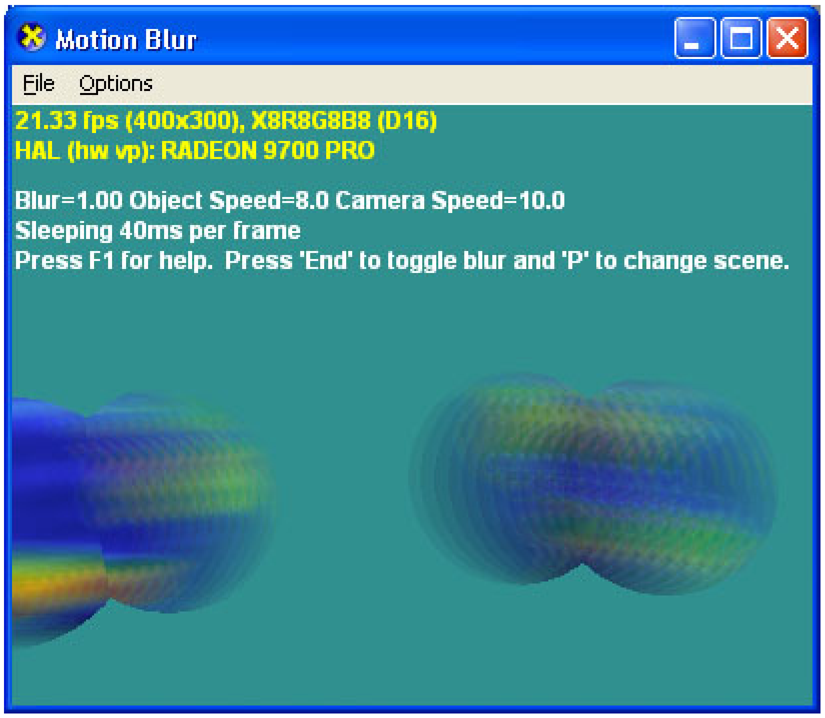
What are the pitfalls of this technique? Because the final post-process pixel shader is blurring the whole scene at once without knowledge of object edges, artifacts will arise if objects are occluded because the blur kerne l from one object will sample color from another nearby object. For example:



```
63.02 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO

Blur=1.00 Object Speed=8.0 Camera Speed=10.0
Sleeping 10ms per frame
Press F1 for help.  Press 'End' to toggle blur and 'P' to change scene.
```

Also, it will be noticeable if the pixels in an object have moved too far since the last frame.



```
21.33 fps (400x300), X8R8G8B8 (D16)
HAL (hw vp): RADEON 9700 PRO

Blur=1.00 Object Speed=8.0 Camera Speed=10.0
Sleeping 40ms per frame
Press F1 for help.  Press 'End' to toggle blur and 'P' to change scene.
```

To correct this, you can either increase the frame rate so that the pixels do not move as far between frames, or reduce the blur factor.