

Tutorial 17: Point Sprites

by Victor Saar

Content

With point sprites Direct3D offers an often underestimated feature. Though most people share the opinion that point sprites are too unflexible, especially when used for particle systems, they have advantages in memory usage and thus performance compared to usual approaches. This tutorial will just cover the basics of point sprites, but it's still a good base to work with. The name by itself already suggests some of the characteristics. All in all a point sprite is simply a screen-aligned element of variable size that is defined by a single point. This already leads to the advantages of point sprites since all you need is a single vertex for each point sprite instead of four for a textured quad. Using only one vertex, whose overall size is less than the size of usual textured verices, means the among of memory that needs to be transfered over the AGP bus decreases tremendously.

Direct3D creates point sprites the following way. From the given input point it creates four surrounding vertices using the size of the point sprite (s). This is shown in the image below. Texture coordinates can also be created, but more to that later.

The overall size of point sprites depends on several circumstances. You can either use fixed size (in screen-space units), independent from current view distance, or variable size (in camera-space units) point sprites computed by a user-defined distance-dependent function and an initial size. The size of a point sprite is either defined per-vertex or as a render state. Direct3D uses the following formula to compute the current size (S_s) of a point sprite:

$$S_s = V_h * S_i * \text{sqrt}(1/(A + B * D_e + C * (D_e^2)))$$

Input values for this formula are the height of the viewport (V_h), the initial point size (S_i), the distance from the eye to the position (D_e) and three user-defined values (A, B and C) which specify a function that is either static, linear or quadratic.

application.cpp

First of all we set the render states. Two of them need a floating point value is input.

```
float fPointSize = 1.0f, fPointScaleB = 1.0f;
```

The first render state set is `D3DRS_POINTSPRITEENABLE` which handles the texture coordinates. Setting this value to `true` causes Direct3D to create a full set of texture coordinates for each of the four vertices. With `false` each vertex will have the same texture coordinate, so using a texture in combination with this setting is useless. `D3DRS_POINTSCALEENABLE` decides how the final size of a point sprite is created. When using `false` Direct3D does not perform any size computations. The point size is interpreted as screen-space units. On the other hand `true` causes Direct3D to compute the size with the formula described above with point size in camera-space units.

```
m_pDirect3DDevice->SetRenderState(D3DRS_POINTSPRITEENABLE, true);  
m_pDirect3DDevice->SetRenderState(D3DRS_POINTSCALEENABLE, true);
```

The first of the following render states sets the global point size which is applied to all vertices not containing any size values. The second one defines the distance-dependent function that is

used to compute the point size. The values for parameter A and C need not be changed since their defaults (1.0f and 0.0f) meet our requirements. Furthermore additive blending is activated. This prevents overdraw of further away point sprites.

```
m_pDirect3DDevice->SetRenderState(D3DRS_POINTSIZE,
                                *((DWORD*)&fPointSize));
m_pDirect3DDevice->SetRenderState(D3DRS_POINTSCALE_B,
                                *((DWORD*)&fPointScaleB));

m_pDirect3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, true);
m_pDirect3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
m_pDirect3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

Since most graphic cards do not support per-vertex point size in hardware, we have to check the device capabilities first. This is only necessary when using hardware vertex processing since point sprites are fully supported in software. The first capability needed to check is the `D3DFVFCAPS_PSIZE` flag, which shows if per-vertex point size is supported. Then `MaxPointSize` is tested, because values smaller than or equal to 1.0f indicate that every point sprite is just a single point.

```
void CApplication::CheckDeviceCaps(void)
{
    m_pDirect3DDevice->GetDeviceCaps(&m_DeviceCaps);

    if(m_dwVertexProcessing == D3DCREATE_HARDWARE_VERTEXPROCESSING)
    {
        if(!(m_DeviceCaps.FVFCaps & D3DFVFCAPS_PSIZE))
        {
            MessageBox(m_hWindow,
                       "Per-vertex point size is not supported!",
                       "CheckDeviceCaps()",
                       MB_OK);
            m_bRunningD3D = false;
        }

        if(m_DeviceCaps.MaxPointSize <= 1.0f)
        {
            MessageBox(m_hWindow,
                       "Only single points supported!",
                       "CheckDeviceCaps()",
                       MB_OK);
            m_bRunningD3D = false;
        }
    }
} //CheckDeviceCaps
```

d3ddefs.h

Before we can start to render the point sprites we have to make an addition to the flexible vertex format. On the one hand the flag for per-vertex point size must be added to the FVF definition. On the other hand a floating point value must be inserted after the coordinates of the vertex.

```
//constants
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_PSIZE | D3DFVF_DIFFUSE)

//structures
struct D3DVERTEX
{
    float fX,
          fY,
          fZ,
          fSize;
```

```
DWORD dwColor;
};
```

main.cpp

The final render code is nothing special at all. The fourth parameter of each vertex defines the appropriate point size. I made the point sprites a bit different in size and color.

```
D3DVERTEX aPoints[] = {{0.0f, 1.0f, 10.0f, 0.1f, 0xffffffff},
                      {0.5f, 0.866f, 10.0f, 0.11f, 0xffeefeff},
                      {0.866f, 0.5f, 10.0f, 0.12f, 0xffdffff},
                      {1.0f, 0.0f, 10.0f, 0.13f, 0xffccccff},
                      {0.866f, -0.5f, 10.0f, 0.14f, 0xffbbbbff},
                      {0.5f, -0.866f, 10.0f, 0.15f, 0xffaaaaff},
                      {0.0f, -1.0f, 10.0f, 0.16f, 0xff9999ff},
                      {-0.5f, -0.866f, 10.0f, 0.17f, 0xff8888ff},
                      {-0.866f, -0.5f, 10.0f, 0.18f, 0xff7777ff},
                      {-1.0f, 0.0f, 10.0f, 0.19f, 0xff6666ff},
                      {-0.866f, 0.5f, 10.0f, 0.2f, 0xff5555ff},
                      {-0.5f, 0.866f, 10.0f, 0.21f, 0xff4444ff}};
```

Since `D3DRS_POINTSPRITEENABLE` is set to `true` a texture must be loaded. Otherwise you would only see a colored quad.

```
D3DXCreateTextureFromFile(g_App.GetDevice(), "texture.png", &pTex);
```

The render call is the easiest part. First, of course, the texture and vertex buffer has to be set. The only difference is the primitive type parameter, which is set to `D3DPT_POINTLIST` here.

```
g_App.GetDevice()->SetTexture(0, pTex);
g_App.GetDevice()->SetStreamSource(0, pVB, 0, sizeof(D3DVERTEX));
g_App.GetDevice()->DrawPrimitive(D3DPT_POINTLIST, 0, 12);
```