Point Sprites and Particle Systems

Particle Systems

Many natural phenomena consist of many small particles that behave in a similar manner, stars in a star field, snow flakes, sparks etc. There are also many phenomena that can be described using small particles such as bullets from a gun.
Particle systems are used to model these phenomena.

Particle Systems and Point Sprites

There are different ways of modelling a particle system. One of these ways is by using point sprites.

Point Sprites

Particles can be modelled using small objects. These small objects can be mathematically modelled as points.
Since the days of DirectX 8.0 and on the methods used to implement point sprites have improved. Since we don't just want a single point primitive, that would be very small and uniform in size, as we often want to texture our sprites. Pre-DirectX 8.0 we had to use billboard sprites to simulate point sprites and to construct particle system. Post DirectX 8.0 we now have access to a special point primitive that is mainly applicable to particles systems.

Benefits of DirectX Point Primitives

- Can map textures to them
- They can change size
- We can define a point primitive using only a single point
- They are always rotated to face the camera

The MSDN Defines Point Sprites as:
Point sprites are generalizations of generic points that enable arbitrary shapes to be rendered as defined by textures.

Support for point sprites in Microsoft Direct3D for Microsoft DirectX 9.0 enables the high-performance rendering of points (particle systems).

Balancing Deception and Realism

Let's look at the snow storm a bit closer. In the real world, we know that a snow storm is made up of millions of snowflakes, falling at different speeds (velocities) in our field of view. Also in the real world, we know that (generally) no 2 snowflakes are the same. Our aim as coders is to try and code a snowstorm that, whilst being as realistic as possible, doesn't require a Silicon Graphics workstation to watch. In other words, it's a balancing act between appearance and deception. In this case, we can handle appearance by having hundreds (or even thousands) of snowflake particles, but we're going to have to use some

deception by making all our snowflakes identical. If we don't, we'll need a separate texture for every flake which, besides being pointless and impractical, the average user would never even notice!

Point Primitive Rendering Controls

Direct3D for DirectX 9.0 supports additional parameters to control the rendering of point sprites (point primitives). These parameters enable points to be of variable size and have a full texture map applied. The size of each point is determined by an application-specified size combined with a distance-based function computed by Direct3D. The application can specify point size either as per-vertex or by setting D3DRS_POINTSIZE, which applies to points without a per-vertex size. The point size is expressed in camera space units, with the exception of when the application is passing post-transformed flexible vertex format (FVF) vertices. In this case, the distance-based function is not applied and the point size is expressed in units of pixels on the render target.

The texture coordinates computed and used when rendering points depends on the setting of D3DRS_POINTSPRITEENABLE. When this value is set to TRUE, the texture coordinates are set so that each point displays the full texture. In general, this is only useful when points are significantly larger than one pixel. When D3DRS_POINTSPRITEENABLE is set to FALSE, each point's vertex texture coordinate is used for the entire point.

Point Size Computations

Point size is determined by D3DRS_POINTSCALEENABLE. If this value is set to FALSE, the application-specified point size is used as the screen-space (post-transformed) size. Vertices that are passed to Direct3D in screen space do not have point sizes computed; the specified point size is interpreted as screen-space size.

If D3DRS_POINTSCALEENABLE is TRUE, Direct3D computes the screen-space point size according to the following formula. The application-specified point size is expressed in camera space units.

S s = Vh * S i * sqrt(1/(A + B * D e + C *(D e2)))

In this formula, the input point size, S i, is either per-vertex or the value of the D3DRS_POINTSIZE render state. The point scale factors, D3DRS_POINTSCALE_A, D3DRS_POINTSCALE_B, and D3DRS_POINTSCALE_C, are represented by the points A, B, and C. The height of the viewport, V h, is the D3DVIEWPORT9 structure's Height member representing the viewport. D e, the distance from the eye to the position (the eye at the origin), is computed by taking the eye space position of the point (Xe, Ye, Ze) and performing the following operation.

D e = sqrt (Xe2 + Y e2 + Z e2)

The maximum point size, Pmax, is determined by taking the smaller of either the D3DCAPS9 structure's MaxPointSize member or the D3DRS_POINTSIZE_MAX render state. The minimum point size, Pmin, is determined by querying the value of D3DRS_POINTSIZE_MIN. Thus the final screen-space point size, S, is determined in the following manner.
If Ss > Pmax, then S = P max
if S < Pmin, then S = P min
Otherwise, S = S s
Point Rendering

A screen-space point, P = (X, Y, Z, W), of screen-space size S is rasterized as a quadrilateral of the following four vertices.

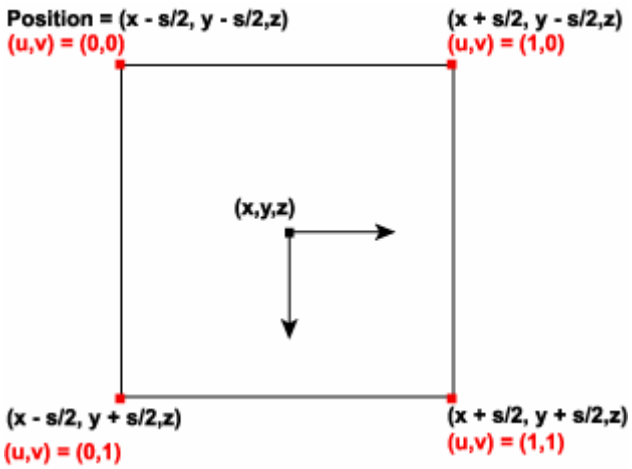
((X + S/2, Y + S/2, Z, W), (X + S/2, Y - S/2, Z, W), (X - S/2, Y- S/2, Z, W), (X - S/2, Y + S/2, Z, W))

The vertex colour attributes are duplicated at each vertex; thus each point is always rendered with constant colours.

The assignment of texture indices is controlled by the D3DRS_POINTSPRITEENABLE render state setting. If D3DRS_POINTSPRITEENABLE is set to FALSE, then the vertex texture coordinates are duplicated at each vertex. If D3DRS_POINTSPRITEENABLE is set to TRUE, then the texture coordinates at the four vertices are set to the following values.

(0.F, 0.F), (0.F, 1.F), (1.F, 0.F), (1.F, 1.F)

This is shown in the following diagram.



When clipping is enabled, points are clipped in the following manner. If the vertex exceeds the range of depth values - MinZ and MaxZ of the D3DVIEWPORT9 structure - into which a scene is to be rendered, the point exists outside of the view frustum and is not rendered. If the point, taking into account the point size, is completely outside the viewport in X and Y, then the point is not rendered; the remaining points are rendered. It is possible for the point position to be outside the viewport in X or Y and still be partially visible.

Points may or may not be correctly clipped to user-defined clip planes. If D3DPMISCCAPS_CLIPPLANESCALEDPOINTS is not set in the D3DCAPS9 structure's PrimitiveMiscCaps member, points are clipped to user-defined clip planes based only on the vertex position, ignoring the point size. In this case, scaled points are fully rendered when the vertex position is inside the clip planes, and discarded when the vertex position is outside a clip plane. Applications can prevent potential artifacts by adding a border geometry to clip planes that is as large as the maximum point size.

If the D3DPMISCCAPS_CLIPPLANESCALEDPOINTS bit is set, then the scaled points are correctly clipped to user-defined clip planes.

Hardware vertex processing may or may not support point size. For example, if a device is created with D3DCREATE_HARDWARE_VERTEXPROCESSING on a hardware abstraction layer (HAL) device (D3DDEVTYPE_HAL) that has the D3DCAPS9 structure's MaxPointSize member set to 1.0 or 0.0, then all points are a single pixel. To render pixel point sprites less than 1.0, you must use either FVF TL (transformed and lit) vertices or software vertex processing (D3DCREATE_SOFTWARE_VERTEXPROCESSING), in which case the Direct3D run time emulates the point sprite rendering.

Putting into Code
First of all we set the render states.
Two of them need a floating point value as 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
```

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
{
floatfX,
fY,
fZ,
fSize;
DWORDdwColor;
};
```

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,0xffeeeeff},{0.866f,0.5f,10.
0f,0.12f,0xffddddff},
{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);
```