W-Buffering in Direct3D

Doug Rogers
NVIDIA Corporation
drogers@nvidia.com

"W-buffering is a depth-buffering alternative to z-buffering, and should be used in cases where z-buffering produces artifacts. W-buffering does a much better job of quantizing the depth buffer." [D3DIM.DOC]

W-buffering provides a linear representation of distance in the depth buffer. Z-buffering is non-linear and allocates more bits for surface that are close to the eyepoint and less bits farther away.

There are two ways to represent the W buffer, scaled integer and floating point. Two or three bytes can be specified as well.

There are two clipping planes used for W-buffering. These are W_{near} and W_{far}. W_{near}, the closest W value that is set to the device driver, is not used in the current implementation and is zero. W_{far} is the farthest W value that will be sent to the hardware.

W_{near} and W_{far} are initialized by setting the matrix operations in D3D using the following calls:

```
SetTransform( D3DTRANSFORMSTATE_WORLD,  &matWorld );
SetTransform( D3DTRANSFORMSTATE_VIEW,  &matView );
SetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );
```

There is no other way to initialize W_{near} and W_{far}. If you are performing your own transforms and lighting operations, and use W-buffering, you must still set W_{near} and W_{far} with SetTransform. W_{near} and W_{far} are calculated from the matrices this way:

```
dvWNear = m._44 - m._43 / m._33 * m._34;
dvWFar  = (m._44 - m._43) / (m._33 - m._34) * m._34 + m._44;
```

If you just want to set W_{near} and W_{far}, you can use this code which calls the SetTransform functions:

```
VOID D3DUtil_InitViewport( D3DVIEWPORT2& vp, DWORD dwWidth, DWORD dwHeight )
{
   ZeroMemory( &vp, sizeof(D3DVIEWPORT2) );
   vp.dwSize   = sizeof(D3DVIEWPORT2);
   vp.dwWidth  = dwWidth;
   vp.dwHeight = dwHeight;
   vp.dvMaxZ   = 1.0f;

   vp.dvClipX      = -1.0f;
   vp.dvClipWidth  = 2.0f;
   vp.dvClipY      = 1.0f;
   vp.dvClipHeight = 2.0f;
}

/*
  sets wNear and wFar
*/

HRESULT set_wbuffer_planes(LPDIRECT3DDEVICE3 lpDev, float dvWNear, float dvWFar)
{
  HRESULT res;
  D3DMATRIX matWorld;
  D3DMATRIX matView;
  D3DMATRIX matProj;

  D3DUtil_SetIdentityMatrix( matWorld );
  D3DUtil_SetIdentityMatrix( matView );
  D3DUtil_SetIdentityMatrix( matProj );

  if (dvWFar <= dvWNear)  return 1;
  res = lpDev->SetTransform( D3DTRANSFORMSTATE_WORLD,      &matWorld );
  if (res) return res;
  res = lpDev->SetTransform( D3DTRANSFORMSTATE_VIEW,       &matView );
  if (res) return res;

  matProj._43 = 0;
  matProj._34 = 1;
  matProj._44 = dvWNear; // not used
  matProj._33 = dvWNear / (dvWFar - dvWNear) + 1;

  res = lpDev->SetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );
  return res;
}
```

Values to use for W-buffering

The values that W ranges over is $[W_{near}, W_{far}]$. W_{near} must be greater than zero. W_{far} must be greater than W_{near}. Invert W and pass it in as *RHW*.

What the Device Driver Does

W-buffering values that are set to the hardware must be within the legal range that is representable. This is dependent upon the format.

Format	*Scale_factor*	$dvRW_{far} = \left(W_{far} / scale\_factor \right)$
16 bit floating point.	2^8 (256)	$\frac{W_{far}}{256}$
16 bit fixed point	1.0	W_{far}
24 bit floating point	2^{127} (1.7×10^{38})	1.0 or $\frac{W_{far}}{2^{127}}$
24 bit fixed point	1.0	W_{far}

W and W_{far}

The device driver calculates the w values that is passed to the hardware in the following way:

$$W_{scaled} = \frac{W}{W_{far}} * scale\_factor$$

Scale_factor scales *W* so W-buffering spans all representable W-buffer locations, maximizing the W-buffering precision.

Interpolation is performed over the inverse of *W*, which is passed to the TNT.

$$dvRW_{far} = \frac{W_{far}}{scale\_factor}$$

$$W_{TNT} = \frac{1}{W_{scaled}} = \frac{W_{far}}{W \cdot scale\_factor} = \frac{dvRW_{far}}{W}$$

$$\boxed{W_{TNT} = RHW \cdot dvRW_{far}}$$

where *scale_factor* is from the above table and *RHW* is the reciprocal of homogenous W.

Typically you want to keep W_{near} very small and W_{far} greater than one. You probably want to set W_{far} to many thousands or millions of units.

For 24 bit floating point W-buffering, we have the choice of scaling the incoming *W* or we can just store the value directly and lose seven bits of the mantissa. If we choose this, then we don't have to scale *RHW* at all, we can just load the W value in directly. If we want to scale the *W* values, the scaling contant $dvRW_{far}$ is $W_{far}/2^{127}$. There is no change in the application code if a 24 or 16 bit W-buffer is used.

Example 1:

The W values in your application range from 10 to 1000. If you are performing your own transformations:
$W_{near} = 10$
$W_{far} = 1000$

1) Set the projection matrix. set_wbuffer_planes(lpDev, W_{near}, W_{far}).

2) Set *RHW* in the vertex data, *vertex.rhw* = $\frac{1}{W}$.

Example 2:
The W values in your application range from 10 to 1000. D3D is performing the transformations.

1) Set the projection matrix. W_{near} and W_{far} are encoded in the projection matrix. This is function to set the projection matrix is from Microsoft's d3dframework.

```
HRESULT D3DUtil_SetProjectionMatrix( D3DMATRIX& mat, float fFOV, float fAspect, float
fNearPlane, float fFarPlane )
{
   if( (fFarPlane-fNearPlane) < 0.01f )
      return E_INVALIDARG;
   float fCos = (float)cos( fFOV/2 );
   float fSin = (float)sin( fFOV/2 );
   float Q    = (fFarPlane * fSin) / (fFarPlane - fNearPlane);
   ZeroMemory(&mat, sizeof(D3DMATRIX) );
   mat._11 = fCos * fAspect;
   mat._22 = fCos;
   mat._33 = Q;
   mat._34 = fSin;
   mat._43 = -Q * fNearPlane;
   return S_OK;
}
```

RHW will be set for you in the vertex data by the D3D transformations.