

Tutorial 16: Render-To-Texture

by Victor Saar

Content

Rendering to a texture is one of the advanced techniques in Direct3D. On the one hand it is simple, on the other hand it is powerful and enables numerous special effects. Examples are the glow effect, first introduced in Tron 2.0, environment mapping, shadow mapping, and many more. Rendering to a texture is simply an extension to rendering to a surface. There are just some additions needed. First the texture itself has to be created. Furthermore some other precautions have to be made. The second step is to render the appropriate scene to the texture. Finally the texture is used to render the final scene.

main.cpp

First the needed objects have to be declared. Of course we need a texture to render to. Furthermore to surfaces are needed. One is used to restore the back buffer after the rendering to the texture is done. The second is the render target for the texture. But later more to that. The first matrix is the projection matrix for the texture and the second is used to restore the old projection.

```
LPDIRECT3DTEXTURE9 pRenderTexture = NULL;
LPDIRECT3DSURFACE9 pRenderSurface = NULL, pBackBuffer = NULL;
D3DXMATRIX matProjection, matOldProjection;
```

Now the texture is created. The first two parameters are the resolution. Then comes the number of mip levels, in our case only one, and the usage. The usage parameter is very important and has to be set to `D3DUSAGE_RENDERTARGET`, which, of course, identifies the texture as a render target. The other parameters are the texture format, the memory pool and a pointer to a texture object. When using the texture as a render target the memory pool has to be `D3DPOOL_DEFAULT`.

```
g_App.GetDevice()->CreateTexture(256,
                                256,
                                1,
                                D3DUSAGE_RENDERTARGET,
                                D3DFMT_R5G6B5,
                                D3DPOOL_DEFAULT,
                                &pRenderTexture,
                                NULL);
```

In order to access the texture memory a surface object is needed, because a texture object in Direct3D always uses such a surface to store its data. To get the underlying surface we call the method `GetSurfaceLevel()` of the texture. The parameters are the index of the texture level to get and a pointer to a surface object.

```
pRenderTexture->GetSurfaceLevel(0, &pRenderSurface);
```

The next step is to create a projection matrix that fits to the texture dimensions, because the aspect ratio of the texture is not the same as the back buffers one. All other parameters remain the same and should be clear from recent tutorials.

```
D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI / 4.0f, 1, 1, 100);
```

Before we can start the render loop a pointer to the back buffer and a copy of its projection matrix have to be saved.

```
g_App.GetDevice()->GetTransform(D3DTS_PROJECTION,&matOldProjection);
g_App.GetDevice()->GetRenderTarget(0,&pBackBuffer);
```

The render loop is splitted into two parts. The first covers the rendering to the texture. Therefore the render target has to be set to the texture surface. Then the pyramid is rendered to this new target. Rendering to another surface does not differ from normal back buffer rendering, except one thing. The exception in this case is the call to `Present()`, which is skipped, because the content of the texture should not be seen on the screen. The surface has to be cleared first and `BeginScene()` and `EndScene()` are called as usual. To achieve proper rendering we have to set the appropriate projection matrix, which meets the requirements of the texture surface, as already mentioned. Otherwise the final image would be stretched.

```
//render-to-texture
//set new render target
g_App.GetDevice()->SetRenderTarget(0,pRenderSurface);
//clear texture
g_App.GetDevice()->Clear(0,
                        NULL,
                        D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        D3DCOLOR_XRGB(100,100,100),
                        1.0f,
                        0);
g_App.GetDevice()->BeginScene();

g_App.GetDevice()->SetTexture(0,pPyramideTexture);

D3DXMatrixRotationY(&matRotationY,fRotation);
D3DXMatrixTranslation(&matTranslation,0.0f,0.0f,5.0f);
g_App.GetDevice()->SetTransform(D3DTS_WORLD,
                               &(matRotationY * matTranslation));
//set projection matrix
g_App.GetDevice()->SetTransform(D3DTS_PROJECTION,&matProjection);

g_App.GetDevice()->SetStreamSource(0,pTriangleVB,0,sizeof(D3DVERTEX));
g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLELIST,0,4);

g_App.GetDevice()->EndScene();
```

The second part of the render loop covers the rendering of the final scene. The render target is reset to the back buffer and the projection matrix is restored. As the texture is ready to use now it is set in texture stage 0.

```
//render scene with texture
//set back buffer
g_App.GetDevice()->SetRenderTarget(0,pBackBuffer);
g_App.GetDevice()->Clear(0,
                        NULL,
                        D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                        D3DCOLOR_XRGB(0,0,0),
                        1.0f,
                        0);
g_App.GetDevice()->BeginScene();

//set rendered texture
g_App.GetDevice()->SetTexture(0,pRenderTexture);

g_App.GetDevice()->SetTransform(D3DTS_WORLD,&matTranslation);
//restore projection matrix
g_App.GetDevice()->SetTransform(D3DTS_PROJECTION,&matOldProjection);
```

```
g_App.GetDevice()->SetStreamSource(0,pQuadVB,0,sizeof(D3DVERTEX));  
g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLESTRIP,0,2);  
  
g_App.GetDevice()->EndScene();  
g_App.GetDevice()->Present(NULL,NULL,NULL,NULL);
```

At the end the surfaces need to be cleaned up by calling the `Release()` function.

```
pRenderSurface->Release();  
pRenderSurface = NULL;  
  
pBackBuffer->Release();  
pBackBuffer = NULL;
```

Rendering to a texture enables many possibilities, but there are also some restrictions you have to take care of. First the depth stencil surface must always be greater or equal to the size of the render target. Furthermore the format of the render target and the depth stencil surface must be compatible and the multisample type must be the same.