

# Tutorial 14: Dot3 Bump Mapping

by Victor Saar

## Content

In the last tutorials with lighting we always used it per-vertex with the help of the normal stored in each vertex. This leads us to a problem when the light comes close to the triangle, which is lit. Though the triangle should appear bright it is not, because the angle between the normal and the light vector is high. That is where dot3 bump mapping, also known as per-pixel lighting, is used to solve this problem. But that is not the only reason to use dot3 bump mapping. It also allows you to add high detail to surfaces or objects without increasing the number of polygons.

For per-pixel lighting we need per-pixel data, of course. Take a look at the image below. It shows a texture, which has a normal saved in each pixel. It is called normal map. The x component of the normal is stored in the red value. The y and z components are stored in the green and blue part of the color. There are different ways to create such a normal map. You can manually create one with help of a function I will explain later. Another way is to use a function, that is offered by the Direct3DX library. It is called `D3DXComputeNormalMap()` and creates the texture from a height map stored in another texture. In this tutorial we simply load the normal map from a file.

One problem of dot3 bump mapping is that the normal stored in the normal map are in texture space. That means if you rotate the object the normals remain unchanged. There are two ways of solving this problem. The first way is to transform the normals to world space, but this is only useful for static objects. The second way is to transform the light vector into texture space. The second way is more common and often used together with vertex and pixel shaders. You could also combine both ways. First you transform the normals to world space and if you rotate the object you transform the light vector.

In this tutorial the normals point in positive z direction, but the quad is oriented towards the negative z axis. The light vector has to be mirrored on the xy-plane. I use an already transformed light vector for this tutorial.

## application.cpp

All we have to do here is to add the render states for the first texture stage in `InitScene()`. The texture states for filtering are set according the settings you choose on startup. The two color arguments are the texture and the texture factor. The operation between them is a dot product, of course. Direct3D now performs a dot product between the normal stored in the texture and the light vector stored in the texture factor.

```
//normal
m_pDirect3DDevice->SetTextureStageState(0,D3DTSS_COLORARG1,D3DTA_TEXTURE);
//light vector
m_pDirect3DDevice->SetTextureStageState(0,D3DTSS_COLORARG2,D3DTA_TFACTOR);
//dot3
m_pDirect3DDevice->SetTextureStageState(0,
                                        D3DTSS_COLOROP,
                                        D3DTOP_DOTPRODUCT3);
```

The second thing to do in this file is to check if dot3 bump mapping is supported by the hardware else it isn't possible to see something.

```

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

if(!(m_DeviceCaps.TextureOpCaps & D3DTEXOPCAPS_DOTPRODUCT3))
{
    MessageBox(m_hWindow,
        "dot3 bump mapping not supported!",
        "CheckDeviceCaps()",
        MB_OK);
    m_bRunningD3D = false;
}
} //CheckDeviceCaps

```

## main.cpp

We start with a new function called `VectorToRGB()`. It converts a vector to a 32-bit color value. The coordinates of the vector range from `-1.0` to `1.0`. The function converts the vector to a range of `0` to `255` and saves the result in a `DWORD`. We will use it to store the light vector.

```

DWORD VectorToRGB(D3DXVECTOR3* NormalVector)
{
    DWORD dwR = (DWORD)(127 * NormalVector->x + 128);
    DWORD dwG = (DWORD)(127 * NormalVector->y + 128);
    DWORD dwB = (DWORD)(127 * NormalVector->z + 128);

    return (DWORD)(0xff000000 + (dwR << 16) + (dwG << 8) + dwB);
} //VectorToRGB

```

Here we have the new variables we will need. The texture, the light vector and the texture factor.

```

LPDIRECT3DTEXTURE9 pNormalMap = NULL; //normal map
D3DXVECTOR3 Light(-1.0f,0.0f,1.0f); //light vector
DWORD dwTFactor; //texture factor

```

Now we load the normal map from the file *normalmap.png*.

```

D3DXCreateTextureFromFileA(g_App.GetDevice(), "normalmap.png", &pNormalMap);

```

In the main loop we set the texture and the vertex buffer first. Then we convert the light vector to a color value and store the result in `dwTFactor`. Before we draw the quad we set the texture factor.

```

//set texture
g_App.GetDevice()->SetTexture(0,pNormalMap);
//set vertex buffer
g_App.GetDevice()->SetStreamSource(0,pQuadVB,0,sizeof(D3DVERTEX));

//light vector to DWORD
dwTFactor = VectorToRGB(&Light);
//set light vector as tex factor
g_App.GetDevice()->SetRenderState(D3DRS_TEXTUREFACTOR,dwTFactor);

//draw quad
g_App.GetDevice()->DrawPrimitive(D3DPT_TRIANGLESTRIP,0,2);

```