23. Normal Mapping (Bump Mapping)

In this lesson, we will learn how to give a flat textured surface the appearance of depth. This technique is called normal mapping. We will actually be building directly from the last lesson, loading obj models. But instead of using the model from the last lesson, we will be using a more simple model which will be the ground, textured with grass.

The image to the left shows you the end result using normal mapping (left) compared to no normal mapping (right).

Introduction

Here we will learn how to impliment a technique called normal (or bump) mapping. What this does is give our flat texture the appearance of "bumps" or of depth when lighting is enabled.This technique works by using a normal specified at each pixel, instead of using the normal specified at each vertex, which is then "interpolated" across the surface. This technique itself is quite simple, but to impliment this technique, we must use a "space" called Texture, or Tangent space (similar to world, local view spaces).

Texture/Tangent Space

You may wonder why we are not able to just simply add the interpolated normal with the normal maps normal when implimenting bumps. This is because the Normal map's normals must be first transformed into the same space as the interpolated normal (vertex normal). Transforming the normal map into the interpolated normals space (objects world space) cannot be done by transforming the normal maps normal directly to the objects world space. This is because every vertex in the object may have different positions, normals, and texture coordinates. This means that each triangle might be facing different directions, and have different texture coordinates as others in the object, so we have to transform the normal maps normals to each triangles space, instead of the general world space.

Think about trying to apply a bump map to a cube. By default the normal maps normals are pointing towards the camera, and the front face of the cube is facing the camera. Now we put the normal map and cube into world space, which we will say rotates them 90 degrees. If we try to directly transform the normal map to the world space, we will see that the right side (previously the front side) of the cube has the correct bumps, since the normal map was also transformed 90 degrees, all the normal maps normals are now pointing right (previously pointing to the camera). However, the rest of the sides of the cube will also try to use the transformed normal map, which is now only pointing right. This will make all the normals on the cube point to the right, instead of directly out from the surface. Now you can see why we must use a separate space for each "face" on the object to impliment normal mapping correctly.

The axis' representing the texture space are called Normal, Bitangent, and Tangent (T, B, N). The Normal specifies the direction the face is facing, while Bitangent, and Tangent are like the U, V texture coordinates of the face.

Let's find out how to get the texture space. Of course there are other ways of doing this, but this is how we will do it.

Normal

The normal is the easy. We get the normal from the normal specified for each vertex, which is then interpolated across the face for the pixel shader, where we will do our normal mapping.

Tangent

The tangent is the V axis of the texture coordinates. We will create two vectors describing two of the faces edges (like we did when calculating normals in the obj loader lesson). Then we will create two 2d vectors describing two of the edges of the texture coordinates for that face. We will then find the tangent based on these edges. We will calculate the tangent for each vertex and store it in our vertex structure. Because of interpolation of the normal, The Normal and Tangent might not always be orthogonal in the pixel shader, So we must make sure that they are in the pixel shader. We can do this by "cutting off" any of the direction that the tangent points in the normals direction. We need to make sure there is a 45 degree angle between the tangent and normal.

Bitangent

Bitangent is sometimes refered to as Binormal, however, if you want to be "correct", Binormal in this case is wrong. Binormal has to do with NURBS and round surfaces or something. The Bitangent is the U axis of the texture coordinate. To make things more simple, we will create the bitangent for every pixel in the pixel shader (if bump mapping is on). We can easily create the bitangent by cross multiplying the tangent and normal (after we have made sure the tangent and normal are orthogonal).

The Normal Map

The normal map is an image whose color components (eg. RGB) for each pixel represent the normal for that pixel, Where rgb will be interpreted as xyz in code. When you look at normal maps, you will notice they are generally very blue, that is because the blue color component represents the z axis, and normals face away from the surface.

There are a couple different ways to make a normal map. One way to do it is use a photoshop plugin, which you can download from nvidia's website. This will take the texture you want to make a bump map for, and turn it into a bump map. Another way is to make a highly detailed mesh from a modeling application like maya or 3ds max, then use a plugin to turn that mesh into a normal map. This of course would be the best and most accurate way of doing it, however it would take a bit more time. I chose the simple route of using the photoshop plugin from nvidia.

When a normal map is created, its color components usually range from 0 to 255. Luckily, when we

load in the map in our pixel shader, this value is compressed to a value between 0 and 1. Now, We still have one more thing to do before using this normal. We need to make this value range from -1 to 1. We do this by multiplying the value by 2, then subtracting 1.

Updated Constant Buffer Structure

We need to update our constant buffer structure so that we can send a bool variable saying if bump mapping is enabled or not. Notice how we use the windows BOOL type instead of the standard bool type. This is explained in the code comments.

```
struct cbPerObject
{
    XMMATRIX  WVP;
    XMMATRIX World;

    //These will be used for the pixel shader
    XMFLOAT4 difColor;
    BOOL hasTexture;
    ///////////////**************new**************////////////////////
    //Because of HLSL structure packing, we will use windows BOOL
    //instead of bool because HLSL packs things into 4 bytes, and
    //bool is only one byte, where BOOL is 4 bytes
    BOOL hasNormMap;
    ///////////////**************new**************////////////////////
};
```

Updated Surface Material Structure

We have updated this structure to include the index of the image containing our normals in the texture array, and also to include a bool value which says if a normal map is used or not.

```
struct SurfaceMaterial
{
    std::wstring matName;
    XMFLOAT4 difColor;
    int texArrayIndex;
    ///////////////**************new**************////////////////////
    int normMapTexArrayIndex;
    bool hasNormMap;
    ///////////////**************new**************////////////////////
    bool hasTexture;
    bool transparent;
};
```

Updated Vertex Structure

We now need to add an element into our vertex structure which is the Tangent. We can then use the tangent and normal of the vertex to find the bitangent and then texture/tangent space, which we can use to transform the normals from our normal map correctly.

```
struct Vertex
{
    Vertex(){}
    Vertex(float x, float y, float z,
        float u, float v,
        float nx, float ny, float nz,
        float tx, float ty, float tz)
        : pos(x,y,z), texCoord(u, v), normal(nx, ny, nz),
        tangent(tx, ty, tz){}

    XMFLOAT3 pos;
    XMFLOAT2 texCoord;
    XMFLOAT3 normal;
    //////////////**********new**********////////////////////
    XMFLOAT3 tangent;
    XMFLOAT3 biTangent;
    //////////////**********new**********////////////////////
};

D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "NORMAL",     0, DXGI_FORMAT_R32G32B32_FLOAT,    0, 20, D3D11_INPUT_PER_VERTEX_DATA, 0},
    //////////////**********new**********////////////////////
    { "TANGENT", 0, DXGI_FORMAT_R32G32B32_FLOAT,    0, 32, D3D11_INPUT_PER_VERTEX_DATA, 0}
    //////////////**********new**********////////////////////
};
```

Updated LoadObjModel Function

We have updated this function to include the ability to load normal (bump) maps, and then to create the tangent for each vertex almost the same way we created the normal for each vertex. This is the entire function, I will go through the new parts next.

```
bool LoadObjModel(std::wstring filename,
    ID3D11Buffer** vertBuff,
    ID3D11Buffer** indexBuff,
    std::vector<int>& subsetIndexStart,
    std::vector<int>& subsetMaterialArray,
    std::vector<SurfaceMaterial>& material,
    int& subsetCount,
    bool isRHCoordSys,
    bool computeNormals)
{
    HRESULT hr = 0;

    std::wifstream fileIn (filename.c_str());    //Open file
    std::wstring meshMatLib;                     //String to hold our obj material library fi

    //Arrays to store our model's information
    std::vector<DWORD> indices;
    std::vector<XMFLOAT3> vertPos;
    std::vector<XMFLOAT3> vertNorm;
    std::vector<XMFLOAT2> vertTexCoord;
    std::vector<std::wstring> meshMaterials;

    //Vertex definition indices
    std::vector<int> vertPosIndex;
    std::vector<int> vertNormIndex;
    std::vector<int> vertTCIndex;

    //Make sure we have a default if no tex coords or normals are defined
    bool hasTexCoord = false;
    bool hasNorm = false;

    //Temp variables to store into vectors
    std::wstring meshMaterialsTemp;
```

Loading the Normal Map

Now when we search for maps to load in the obj models mtl file, we need to search for the line that starts with "map_bump", which is the normal map used to impliment the appearance of "bumps". This is almost the same exact thing as when we loaded in a diffuse texture, so not much new here.

```
//map_bump - bump map (we're usinga normal map though)
else if(checkChar == 'b')
{
    checkChar = fileIn.get();
    if(checkChar == 'u')
    {
        checkChar = fileIn.get();
        if(checkChar == 'm')
        {
            checkChar = fileIn.get();
            if(checkChar == 'p')
            {
                std::wstring fileNamePath;

                fileIn.get();    //Remove whitespace between map_bump and file

                //Get the file path - We read the pathname char by char since
                //pathnames can sometimes contain spaces, so we will read until
                //we find the file extension
                bool texFilePathEnd = false;
                while(!texFilePathEnd)
                {
                    checkChar = fileIn.get();

                    fileNamePath += checkChar;

                    if(checkChar == '.')
                    {
                        for(int i = 0; i < 3; ++i)
                            fileNamePath += fileIn.get();

                        texFilePathEnd = true;
                    }
```

Creating A New Material

When we create a new material, we need to make sure we set its defaults to avoid errors or unwanted results. So we set the normal maps index array to zero and the normal enabled bool value to false.

```
checkChar = fileIn.get();
if(checkChar == ' ')
{
    //New material, set its defaults
    SurfaceMaterial tempMat;
    material.push_back(tempMat);
    fileIn >> material[matCount].matName;
    material[matCount].transparent = false;
    material[matCount].hasTexture = false;
    ////////////////**************new**************////////////////////
    material[matCount].hasNormMap = false;
    material[matCount].normMapTexArrayIndex = 0;
    ////////////////**************new**************////////////////////
    material[matCount].texArrayIndex = 0;
    matCount++;
    kdset = false;
}
```

Computing the Tangent

Let's go down to where we compute the normals for our vertices. Since computing tangents is so similar to computing normals, we will do them together.

First we go through every triangle in our mesh and find the tangent for that triangle. I don't want to get into the math stuff, but if your interested, go to:

http://www.terathon.com/code/tangent.html

After we compute the tangent for each triangle, we average the tangents of all faces sharing the same vertices, the same way we did when we did normal averaging.

We could also have defined our bitangent here, however, because the normal is interpolated across the surface in the pixel shader, if we did that we would have to make sure the normal, tangent, and bitangent are all orthogonal, which would be a little more work than just making sure the tangent and normal are orthogonal and just crossing them to get the bitangent.

```
    ////////////////////Compute Normals/////////////////////////
    //If computeNormals was set to true then we will create our own
    //normals, if it was set to false we will use the obj files normals
    if(computeNormals)
    {
        std::vector<XMFLOAT3> tempNormal;

        //normalized and unnormalized normals
        XMFLOAT3 unnormalized = XMFLOAT3(0.0f, 0.0f, 0.0f);

        ///////////////*************new*************////////////////////
        //tangent stuff
        std::vector<XMFLOAT3> tempTangent;
        XMFLOAT3 tangent = XMFLOAT3(0.0f, 0.0f, 0.0f);
        float tcU1, tcV1, tcU2, tcV2;
        ///////////////*************new*************////////////////////

        //Used to get vectors (sides) from the position of the verts
        float vecX, vecY, vecZ;

        //Two edges of our triangle
        XMVECTOR edge1 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);
        XMVECTOR edge2 = XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f);

        //Compute face normals
        //And Tangents
        for(int i = 0; i < meshTriangles; ++i)
        {
            //Get the vector describing one edge of our triangle (edge 0,2)
            vecX = vertices[indices[(i*3)]].pos.x - vertices[indices[(i*3)+2]].pos.x;
            vecY = vertices[indices[(i*3)]].pos.y - vertices[indices[(i*3)+2]].pos.y;
            vecZ = vertices[indices[(i*3)]].pos.z - vertices[indices[(i*3)+2]].pos.z;
            edge1 = XMVectorSet(vecX, vecY, vecZ, 0.0f);    //Create our first edge
```

Loading the "ground.obj" Model

We are using a different model than the model from the last lesson, so we can change that here in the init scene function.

```
if(!LoadObjModel(L"ground.obj", &meshVertBuff, &meshIndexBuff, meshSubsetIndexStart, meshSubs
    return false;
```

Drawing the Model

Go down to the drawscene function where we draw our model. We need to make sure we send the bool variable saying if normal mapping is enabled or not to the shader, and if it is send the normal map. We do this for the non transparent and the transparent subsets of the model.

Notice how we send the diffuse map through the 0th slot when sending it to the pixel shader, and how we send the normal map through the 1st slot. If we sent the normal map through the first slot, it would be stored in the diffuse variable in our effect file.

```
    for(int i = 0; i < meshSubsets; ++i)
    {
        //Set the grounds index buffer
        d3d11DevCon->IASetIndexBuffer( meshIndexBuff, DXGI_FORMAT_R32_UINT, 0);
        //Set the grounds vertex buffer
        d3d11DevCon->IASetVertexBuffers( 0, 1, &meshVertBuff, &stride, &offset );

        //Set the WVP matrix and send it to the constant buffer in effect file
        WVP = meshWorld * camView * camProjection;
        cbPerObj.WVP = XMMatrixTranspose(WVP);
        cbPerObj.World = XMMatrixTranspose(meshWorld);
        cbPerObj.difColor = material[meshSubsetTexture[i]].difColor;
        cbPerObj.hasTexture = material[meshSubsetTexture[i]].hasTexture;
        ////////////////**************new**************////////////////////
        cbPerObj.hasNormMap = material[meshSubsetTexture[i]].hasNormMap;
        ////////////////**************new**************////////////////////
        d3d11DevCon->UpdateSubresource( cbPerObjectBuffer, 0, NULL, &cbPerObj, 0, 0 );
        d3d11DevCon->VSSetConstantBuffers( 0, 1, &cbPerObjectBuffer );
        d3d11DevCon->PSSetConstantBuffers( 1, 1, &cbPerObjectBuffer );
        if(material[meshSubsetTexture[i]].hasTexture)
            d3d11DevCon->PSSetShaderResources( 0, 1, &meshSRV[material[meshSubsetTexture[
        ////////////////**************new**************////////////////////
        if(material[meshSubsetTexture[i]].hasNormMap)
            d3d11DevCon->PSSetShaderResources( 1, 1, &meshSRV[material[meshSubsetTexture[
        ////////////////**************new**************////////////////////
        d3d11DevCon->PSSetSamplers( 0, 1, &CubesTexSamplerState );

        d3d11DevCon->RSSetState(RSCullNone);
        int indexStart = meshSubsetIndexStart[i];
        int indexDrawAmount =  meshSubsetIndexStart[i+1] - meshSubsetIndexStart[i];
        if(!material[meshSubsetTexture[i]].transparent)
            d3d11DevCon->DrawIndexed( indexDrawAmount, indexStart, 0 );
    }
```

The Effect File

Now we need to fix up our effect file to do the actual normal mapping implimentation. First you can see we updated our constant buffer to hold the bool variable saying if bump mapping is enabled or not. Then we add a new 2d texture, which will be used to store the normal map data. After that we have an updated vertex structure output, which contains the tangent. We need to make sure we put the tangent into world space like we do with the normal.

```
struct Light
{
    float3 pos;
    float  range;
    float3 dir;
    float cone;
    float3 att;
    float4 ambient;
    float4 diffuse;
};

cbuffer cbPerFrame
{
    Light light;
};

cbuffer cbPerObject
{
    float4x4 WVP;
    float4x4 World;

    float4 difColor;
    bool hasTexture;
    bool hasNormMap;
};

Texture2D ObjTexture;
Texture2D ObjNormMap;
SamplerState ObjSamplerState;
TextureCube SkyMap;

struct VS_OUTPUT
{
```

The Pixel Shader

Let's take a look at our pixel shader. We have a new chunk of code, nice and packed together. First we check to make sure normal mapping is even enabled for this surface. If not, we skip this chunk completely. If normal mapping is enabled, we load in the normal for the current pixel from the normal map, same way as we load in the color for the current pixel. When we load in the value, it will be in a variable between 0 and 1, we need to change this so that it is between -1 and 1, which is what the next line does, times it by two and subtract one.

The next line makes sure the tangent and the normal are orthogonal. We do this by subtracting any direction that the tangent is pointing towards the normal, which will give them a 45 degree angle apart.

Now that they are orthogonal and perfect, we need to create the third axis of our texture space, the bitangent. We can easily create this by cross multiplying the normal and tangent.

Finally we store these three vectors into a 3 by 3 matrix, which defines the texture space for this specific pixel. We call this texSpace here.

Lastly, we multiply the normal taken from the normal map with our texSpace matrix, and store it into our default normal variable, which we can now use to impliment lighting, and give the surface a cool 3d look!

```
float4 PS(VS_OUTPUT input) : SV_TARGET
{
    input.normal = normalize(input.normal);

    //Set diffuse color of material
    float4 diffuse = difColor;

    //If material has a diffuse texture map, set it now
    if(hasTexture == true)
        diffuse = ObjTexture.Sample( ObjSamplerState, input.TexCoord );

    //If material has a normal map, we can set it now
    if(hasNormMap == true)
    {
        //Load normal from normal map
        float4 normalMap = ObjNormMap.Sample( ObjSamplerState, input.TexCoord );

        //Change normal map range from [0, 1] to [-1, 1]
        normalMap = (2.0f*normalMap) - 1.0f;

        //Make sure tangent is completely orthogonal to normal
        input.tangent = normalize(input.tangent - dot(input.tangent, input.normal)*input.nor

        //Create the biTangent
        float3 biTangent = cross(input.normal, input.tangent);

        //Create the "Texture Space"
        float3x3 texSpace = float3x3(input.tangent, biTangent, input.normal);

        //Convert normal from normal map to texture space and store in input.normal
        input.normal = normalize(mul(normalMap, texSpace));
    }
```

That's it! In itself, bump mapping is a very simple idea. Its just the math gets a little confusing at parts. I'm sorry that I didn't thoroughly explain how to get the tangent, but you can look at the code.

Exercise:

1. Learn the math behind computing the tangent for each vertes ;)

2. Create your own normal map. Experiment with different plugins and programs to find which creates a normal map thats most fitting for your application.

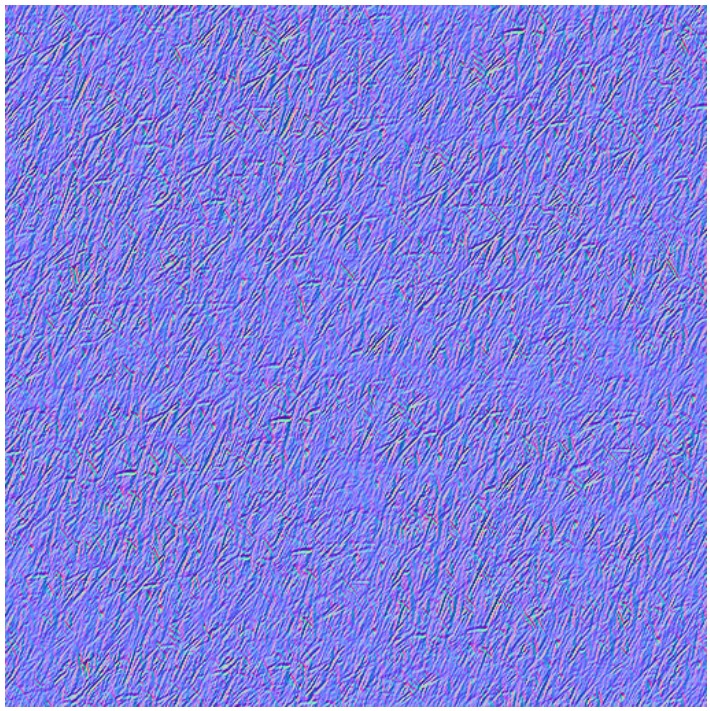
3. Try computing the bitangent at the same time as the tangent. HINT - You will need to make sure the bitangent and tangent are othogonal with the normal in the pixel shader.

4. Try computing the tangent and bitangent in the pixel shader instead of when loading your model. HINT - I believe you are able to do it using only the normal, although i'm not sure how accurate the

end result will be, its good to practice though!

grass.jpg

grassNormal.jpg



ground.obj

```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware
# File Created: 26.07.2011 13:47:43

mtllib ground.mtl

#
# object Window
#

v  -100.00000.0000 -100.0000
v  -100.00000.0000 100.0000
v  100.00000.0000 100.0000
v  100.00000.0000 -100.0000
# 4 vertices

vn 0.0000 1.0000 0.0000
# 1 vertex normals

vt 0.0000 0.0000 0.0000
vt 0.0000 30.0000 0.0000
vt 30.0000 30.0000 0.0000
vt 30.0000 0.0000 0.0000
# 4 texture coords

g ground
usemtl Grass
f 1/1/1 2/2/1 3/3/1
f 3/3/1 4/4/1 1/1/1
# 2 faces
```

ground.mtl

```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware
# File Created: 26.07.2011 13:47:43

newmtl Ground
    Ns 10.0000
    d 1.0000
    Tr 0.0000
    Tf 1.0000 1.0000 1.0000
    illum 2
    Ka 0.5882 0.5882 0.5882
    Kd 0.5882 0.5882 0.5882
    Ks 0.0000 0.0000 0.0000
    Ke 0.0000 0.0000 0.0000
    map_Ka grass.jpg
    map_Kd grass.jpg
    map_bump grassnormal.jpg
```