Assign. 9 – Normal Maps & Cube Maps

Overview

Now that you've got the basics up and running, it's time to get fancy. First, you'll be adding normal maps to your project, which will require a little extra data in your mesh class. Then you'll be adding a skybox, which will require the use of a special GPU resource: a cube map.

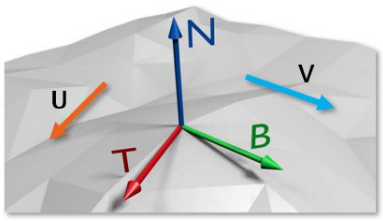
Task Overview

Here is the high-level overview of tasks, which are explained in more detail on the pages that follow:

- ❑ Normal Mapping
  - ❑ Update your vertex format, Vertex Shader and Mesh class to support a **tangent** vector
  - ❑ **Add normal map support** to your shaders
- ❑ Skybox
  - ❑ **Create a new set** of vertex and pixel shaders specifically for rendering a skybox
  - ❑ Create a **Sky class** that will handle all of the skybox initialization and drawing
  - ❑ **Draw** the sky after other solid geometry has been draw, using the proper render states
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

Task 1: Tangents

For normal mapping to work properly, each vertex will need a *tangent* vector in addition to its normal. This *tangent* vector points along the surface, but is oriented to the *u* direction of the uv texture mapping across the triangle. Once we have both a normal and a tangent available in our shaders, we can compute the third vector – the bi-tangent – on the fly.



Updating Your Vertex (C++ and HLSL)

Adding new data to your vertices requires updating your vertex format, which is probably stored in a file like Vertex.h. **Add a tangent** – an extra 3-component vector – to your Vertex struct.

Now that your vertices are bigger, you'll need to update your Vertex Shader's input struct to account for this. Adjust the VertexShaderInput struct to match the change you made to your C++ Vertex struct. Use "TANGENT" as the semantic for this data.

Order matters here so make sure the C++ and HLSL structs match exactly!

Calculating Tangents

You now have a place to store tangents for each vertex, but you don't have tangent data yet. While the .OBJ model format doesn't provide tangents, it does have everything we need to calculate them.

At the end of this document and on MyCourses, I've included a function to calculate tangents, given a set of vertices and indices. It uses the positions and uv coordinates of each vertex to figure out the direction of the uv mapping along the triangle's surface.

Add this function to your Mesh class. Ensure it's being called just before creating the actual Direct3D buffers, regardless of whether you're loading data from a file or just providing arrays of vertex & index data to the Mesh constructor.

Testing

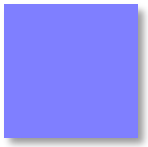
Ensure your project still runs before moving on. While you won't see any differences yet, you should use breakpoints to ensure the tangent calculation is being run when creating Mesh objects.

Task 2: Normal Mapping in HLSL

New Shaders or Old Shaders?

Decide if you want to simply include normal mapping in your existing lighting shaders, or if you want to have a separate set of Vertex and Pixel shaders specifically for normal mapping. Either is acceptable for this assignment but do note the following:

- If you only have *one set* of lighting shaders, all objects will need a normal map.
    - Objects that don't have a specific normal map could use a texture that effectively applies no change to the surface normal after the normal map calculation.
    - This "flat" normal map has the color (0.5f, 0.5f, 1.0f)
    - As integer values, the color is (127,127,255)
    - After unpacking this value, the tangent-space normal is approx. (0,0,1)
    - I've **included a copy of this file** if you'd like to go this route. Use the image I've included rather than making it yourself, as it has an extra flag to prevent the GPU from automatically adjusting for various color profiles (gamma correction).



- If you decide to make a *new set* of shaders, you'll be increasing the number of permutations that you need to work with.
    - Making a change in one will often necessitate the same change in all the others.
    - Smart usage of helper methods can absolutely simplify things here.
    - Even your light loop could be in a helper method, vastly simplifying each pixel shader.

Normal Mapping – New Structs?

Your *vertex input struct* should already have a tangent as part of its data (this was done back in Task 1). This is fine even for non-normal-map shaders, since they can simply ignore that data.

However, to implement normal mapping in a pixel shader, your vertex shader will need to pass extra data – the tangent – through the pipeline.

> *If you're working with a new set of shaders for this task, you'll need a variation of the existing VertexToPixel struct that specifically has room for tangent data. Make a copy of your existing VertexToPixel struct, which is probably now defined in a shader include file. Give this struct a clear name: something like `VertexToPixel_NormalMap` or `V2P_NormalMap` or similar.*

In either case, **update the appropriate struct** to also contain a suitable tangent.

Normal Mapping – Vertex Shader

First, ensure that your normal-map-specific Vertex Shader is properly using your normal-map-specific VertexToPixel struct as its **return type**.

Rather than passing tangent data through to the Pixel Shader unaltered, you'll need to ensure it is rotated in much the same way the normal is rotated. Perform **similar steps** to the incoming tangent data as you did to the normal, except *just use the world matrix* instead of the inverse transpose world. That's it for the vertex shader!

Normal Mapping – Pixel Shader

There is where the bulk of the normal mapping work occurs. The goal is to sample the normal from the normal map, unpack it to the correct range, transform it from tangent space to world space, and then use that result for all of your lighting equations.

As above, ensure that your normal-map-specific Pixel Shader is using your normal-map-specific VertexToPixel struct. You'll also need to define another **Texture2D** for the actual normal map.

Sample and Unpack Normal

Now that you have a normal map, you'll need to sample it, keep just the R/G/B components and then unpack the result. Remember that unpacking the normal means converting it from the [0 – 1] range (which is how colors are stored in textures) to the [-1 – 1] range, like so:

```
float3 unpackedNormal = NormalMap.Sample(BasicSampler, input.uv).rgb * 2 - 1;
unpackedNormal = normalize(unpackedNormal); // Don't forget to normalize!
```

Create the TBN Matrix

The next step is to create the rotation matrix you'll need to transform the unpacked normal from tangent space (the way it's stored in the normal map) to world space. This is necessary so the normal is properly rotated and compatible with our existing lighting equations.

You already have two of the vectors you need for this – the **normal** and the **tangent** – from the vertex shader. Make sure they're **both normalized** after reaching the pixel shader. Since they both most likely became un-normalized during interpolation, they're most likely not orthogonal – exactly 90 degrees apart – anymore. You can fix this with the Gram-Schmidt orthonormalize process (see below). Afterwards, **cross them** to get the third and final vector: the bi-tangent.

These three vectors essentially create a local X/Y/Z axis, where the normal (pointing out of the surface) is Z and the X & Y axes are oriented to match the uv texture mapping on the surface. Stack these vectors on top of each other to **form a 3x3 rotation matrix**.

```
// Feel free to adjust/simplify this code to fit with your existing shader(s)
// Simplifications include not re-normalizing the same vector more than once!
float3 N = normalize(input.normal);  // Must be normalized here or before
float3 T = normalize(input.tangent); // Must be normalized here or before
T = normalize(T - N * dot(T, N)); // Gram-Schmidt assumes T&N are normalized!
float3 B = cross(T, N);
float3x3 TBN = float3x3(T, B, N);
```

Transform the Unpacked Normal

The last step is to use this matrix to transform the normal you got out of the normal map. This is then the normal you use for the rest of the shader:

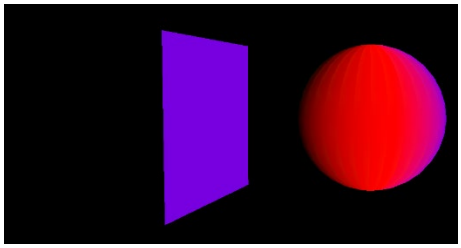
```
// Assumes that input.normal is the normal later in the shader
input.normal = mul(unpackedNormal, TBN); // Note multiplication order!
```

That's it for the shader!

Testing

While you won't be able to test normal mapping itself until you load appropriate resources and set up your materials, you can test whether or not the tangent is making it to the pixel shader using one or more of the strategies outlined in previous assignments.

The world-space tangents of a rotating cube and sphere

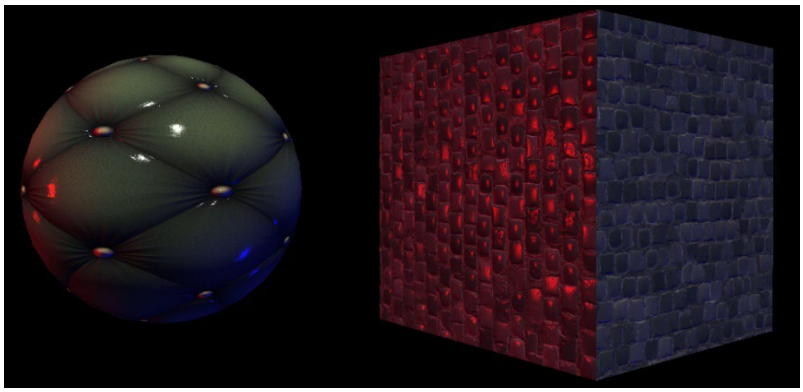


Task 3: Using Normal Maps

On MyCourses, I've included several sets of textures with corresponding normal maps. For this assignment, feel free to use the ones I've provided, find your own online or even try creating them yourself using the Normal Map filter in Photoshop.

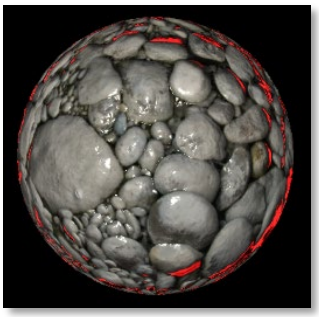
If you've updated your existing set of shaders for normal mapping, all of your entities will probably have normal maps. If you've created a second set of shaders for this, however, ensure that **at least one** of the materials in your project uses your new normal map shaders and has a proper normal map. Use that material on **at least one** of the entities on your screen.

Once you get this working, you should see your normal maps in action.



Normal Mapping Odds & Ends

Once you have normal maps working, you may notice a very strong specular reflection at glancing angles. In the first example to the right, the red light source is on the opposite side of the object, but we're still seeing reflections because some of the normals now face that way.



The math is doing what we've asked of it, but it doesn't take any sort of "self-shadowing" into account. We can mitigate this a bit by cutting the specular contribution if the diffuse is zero for the current pixel. See the second example to the right to see this fix in action.

If you're noticing this issue, use the following line in your shader (or helper methods) whenever you're calculating specular and diffuse for a light:

```
// Cut the specular if the diffuse contribution is zero
// - any() returns 1 if any component of the param is non-zero
// - In this case, diffuse is a single float value
//     - Meaning any() returns 1 if diffuse itself is non-zero
// - In other words:
//     - If the diffuse amount is 0, any(diffuse) returns 0
//     - If the diffuse amount is != 0, any(diffuse) returns 1
//     - So when diffuse is 0, specular becomes 0
spec *= any(diffuse);
```

Task 4: Sky Shaders

You'll need to create a new set of shaders for rendering the skybox, as it is not a standard game entity.

Sky Vertex Shader

Your sky vertex shader will need the following declarations:

- `VertexShaderInput` struct – this will be the same as your existing vertex shaders
- `VertexToPixel` struct – this will only need two members:
  - `float4 position  : SV_POSITION;`
  - `float3 sampleDir : DIRECTION;`
- `cbuffer` – this will only need two variables: a view matrix and a projection matrix

> *Note that if you're using a shader include file to hold common shader structs, you can reuse the same `VertexShaderInput` struct here but you'll need to declare an alternative `VertexToPixel` struct with a slightly different name (perhaps `VertexToPixel_Sky` or something similar).*

Update the declaration of the `main()` function to indicate it accepts and returns the proper structs. The actual body of the shader's `main()` function will need to perform the following steps:

- Create a variable of the appropriate type that will be returned at the end of the function
- Create a copy of the view matrix and set the translation portion of that copy to all zeros
  - `viewNoTranslation._14 = 0;`
  - `viewNoTranslation._24 = 0;`
  - `viewNoTranslation._34 = 0;`
- Apply *projection* & *updated view* to the input position, saving the result as the output position
- Ensure that the output depth of each vertex will be exactly 1.0 after the shader
  - This requires setting the output position's Z value equal to its W value
  - This ensures that, after the automatic perspective divide that occurs in the rasterizer, the depth will end up being 1.0
- Figure out the sample direction for this vertex from the center of the object
  - The sample direction ends up being equal to the input position, since each vertex's position is really just its offset from the origin
- Return the output variable

Sky Pixel Shader

Your sky pixel shader will need the following declarations:

- `VertexToPixel` struct – must match the one in your sky vertex shader (or shader include file)
- `TextureCube` variable to hold the cube map, bound to `register(t0)`
- `SamplerState` variable to hold the sampler options, bound to `register(s0)`

Update the shader's main() function declaration to accept an appropriate VertexToPixel parameter. The function itself needs only a single line of code: sample the cube map in the correct direction and return the result. Remember that the direction is coming from the Vertex Shader.

Task 5: The Sky Class

Create a new class in your project called "Sky" (or something similar). This is where pretty much all of your skybox-related C++ code will go, though you'll have some flexibility on a few of the specifics.

In the header, you'll need the following resources:

- `ComPtr<ID3D11SamplerState>` for sampler options
- `ComPtr<ID3D11ShaderResourceView>` for the cube map texture's SRV
- `ComPtr<ID3D11DepthStencilState>` for adjusting the depth buffer comparison type
- `ComPtr<ID3D11RasterizerState>` for rasterizer options (drawing the object's "inside")
- `shared_ptr<Mesh>` for the geometry to use when drawing the sky
- `shared_ptr<SimplePixelShader>` for the sky-specific pixel shader
- `shared_ptr<SimpleVertexShader>` for the sky-specific vertex shader

Constructor

At a minimum, the constructor will need to take a Mesh, an ID3D11SamplerState and the ID3D11Device. Since you're already loading meshes elsewhere, it makes sense to reuse one of them (the cube most likely) here. Same goes for the sampler state. Since you'll be creating some other resources *inside* this class, so you'll need access to the D3D device, too.

These are not the only things your constructor will need to do, however:

External Resources

All of the member variables should be set in the constructor, either by passing values in or creating them. However, where exactly you *load* your external resources (*textures* and *shaders*) is up to you:

- You could load the textures & shaders in Game.cpp and pass them in to the Sky constructor.
- Alternatively, you could simply pass the *file paths* to the Sky constructor and do all of the texture and shader loading there.
- (You could technically hardcode all of the file paths directly inside the Sky constructor, though that's not a great setup.)

When loading a cube map, you'll need to load 6 individual textures that happen to correspond to the 6 faces of a cube map and programmatically copy their contents to a blank cube map texture. Refer to the sample code at the end of this document (also provided on MyCourses) for this task. There you'll find a function you can copy/paste into your code: it takes 6 file paths, loads those textures and builds a cube map from them, returning a Shader Resource View you can use when drawing.

While this function could be a helper in Game.cpp, it would work even better inside the Sky class, where it can be called directly from the Sky's constructor. Use whichever setup you prefer.

Render States

You also need to create two render states for drawing the sky: a Rasterizer State and a Depth-Stencil State. For both, you'll need to fill out a description struct and call the associated Create() method from the D3D device object. These should both be created in the Sky constructor. Specifics below.

Rasterizer State

Create a `D3D11_RASTERIZER_DESC` variable and initialize it using the "`= {};`" trick. Recall that this sets every struct member to zero, which will be the correct value for most of the members. You'll need to manually set `FillMode` to `D3D11_FILL_SOLID` and set `CullMode` to `D3D11_CULL_FRONT`. This *culls* front faces (meaning you'll *draw* back faces – the "inside" of the object instead of the "outside").

Call the device's `CreateRasterizerState()` function to actually create the state.

Depth-Stencil State

Create a `D3D11_DEPTH_STENCIL_DESC` variable and initialize it as above. Manually set `DepthEnable` to `true` and `DepthFunc` to `D3D11_COMPARISON_LESS_EQUAL`. Normally, the depth buffer only accepts pixels whose depths are *less* than existing values in the depth buffer. This new depth function, however, ensures that pixels will be accepted if their depth values are less than *or equal to* existing values. Since our skybox pixels will have a depth of exactly 1.0, and the depth buffer starts each frame with depths of 1.0, this updated depth function allows our skybox to render properly.

Call the device's `CreateDepthStencilState()` function to actually create the state.

Destructor

You shouldn't need to manually clean up any of your smart pointers. However, if for some reason you're not using smart pointers for any objects, delete them manually here.

Sky::Draw( )

The only other function you'll need in the Sky class is one to actually draw the sky. You'll need to pass in the D3D Context object's ComPtr to perform the drawing commands, and a Camera object to get the necessary view and projection matrices.

Here are the basic steps:

1. Change the necessary render states
    - Use `context->RSSetState()` to set the rasterizer state.
    - Use `context->OMSetDepthStencilState()` to set the depth state. Use zero for the stencil ref parameter.

2. Prepare the sky-specific shaders for drawing
    - Activate both shaders using their `SetShader()` methods.
    - Set the appropriate SamplerState and SRV for the pixel shader.
    - Set the view and projection matrices for the vertex shader.
        - Don't forget `CopyAllBufferData()`!

3. Draw the mesh

4. Reset any render states you changed above
    - Passing in a null pointer (zero) instead of actual render state objects resets the states to their default options.

Task 6 is where you'll test your new Sky class.

Task 6: Putting it All Together

Over in your Game class, you'll need to initialize your Sky object (and load the cube map, if you're passing it into the Sky constructor) and then draw the sky.

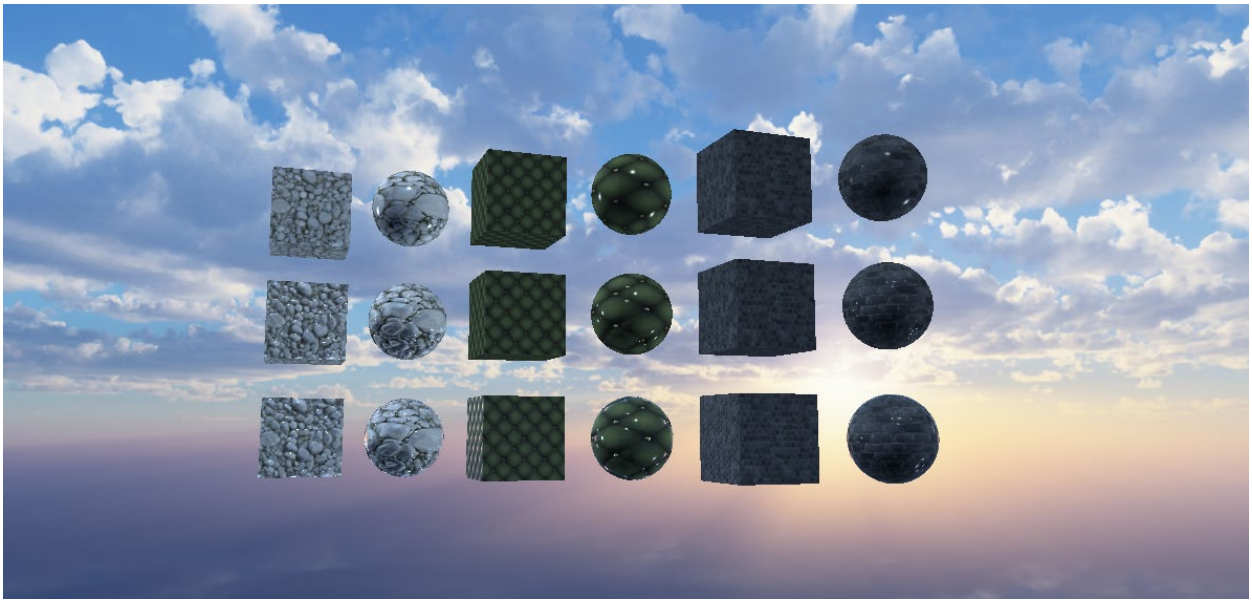
Decide which sky texture(s) you'll be using and ensure you're loading them correctly. Due to the size of all these files, *please don't include unused textures* in your project submissions.

The last question: is it better to draw the sky *before* the other game entities or *after* those entities?

It might initially make sense to draw the sky before the entities; you want the sky to be behind everything, right? But the depth buffer will take care of occlusion for us, so we'll get the same visual result regardless of the order that we draw solid (non-transparent) objects.

As it turns out, it's more efficient to draw the sky *after* all of your other solid (non-transparent) entities. This is because the rasterizer does an early check of the depth buffer for us, and automatically skips any pixels that fail the depth test (pixels that represent a surface "behind" something we've already drawn).

Another way to think about it: If we draw the sky *first*, we'll spend time changing every pixel of the screen. Then, as we draw other entities, we're covering up a bunch of the work we've already done. Wasteful! However, if we draw the sky *last*, we'll only end up running the sky pixel shader for pixels that will actually contribute to our final image. This difference is especially noticeable when we have lots of objects covering a large portion of the screen.



Cube Map Odds & Ends

Ambient Color

Previously, you chose an ambient color that matched your solid background color. Now that you have a skybox representing the world around your objects, your ambient color should match the general color of that cube map.

Reflections

We looked briefly at how to use a cube map to make objects look like they're reflecting the sky. While **not a requirement** for this assignment, feel free to experiment with this technique if you'd like! Follow the demo from the demo repo, check out the slides on Cube Maps or reach out for assistance.

This can be a bit of a rabbit hole; be sure you finish the other requirements first! Things to keep in mind:

- Unless you're trying to make a perfect mirror of the sky, you'll need to combine the reflection color and the surface's color in some way.
    - While you could just average them, this isn't how real reflections work.
    - Surfaces become more reflective as we look *across* them rather than *at* them. This means we need a scalar value that changes as our viewing angle to the surface changes.
        - Sounds like another dot product! The dot product between the surface's *normal* and the "to camera" *view* vector – known as "N dot V" – could work here, though it's not physically accurate.
        - A better option would be to use a Fresnel term, which can be quite accurately approximated using Schlick's Approximation. We often use 0.04 as R_0, which is a common, real-world value for reflections on non-metallic surfaces. For the cosine value, use the aforementioned N dot V.
    - When the two vectors point in the same direction, you have no reflection (just the surface color). When they are perpendicular, you only see reflection. In between, you interpolate the two colors according to the scalar value calculated above.
- Technically all real-world objects have reflections (they all reflect light; that's how we can see them). A shinier object has a more precise reflection while a duller object has a blurrier reflection. Our simple reflection model doesn't account for this, but we'll see one soon that does.

Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. Remember to follow the steps in the "Preparing a Visual Studio project for Upload" PDF on MyCourses to shrink the file size.

Seriously, this is important! The DirectX Toolkit can really bloat your project size, so be sure to remove the files as outline in the PDF mentioned above; it will be redownloaded from NuGet automatically the next time you open and build your project.

Sample Code: Tangent Calculation

```
// ------------------------------------------------------
// Calculates the tangents of the vertices in a mesh
// - Code originally adapted from: http://www.terathon.com/code/tangent.html
//   - Updated version found here: http://foundationsofgameenginedev.com/FGED2-sample.pdf
//   - See listing 7.4 in section 7.5 (page 9 of the PDF)
//
// - Note: For this code to work, your Vertex format must
//         contain an XMFLOAT3 called Tangent
//
// - Be sure to call this BEFORE creating your D3D vertex/index buffers
// ------------------------------------------------------
void Mesh::CalculateTangents(Vertex* verts, int numVerts, unsigned int* indices, int numIndices)
{
        // Reset tangents
        for (int i = 0; i < numVerts; i++)
        {
                verts[i].Tangent = XMFLOAT3(0, 0, 0);
        }

        // Calculate tangents one whole triangle at a time
        for (int i = 0; i < numIndices;)
        {
                // Grab indices and vertices of first triangle
                unsigned int i1 = indices[i++];
                unsigned int i2 = indices[i++];
                unsigned int i3 = indices[i++];
                Vertex* v1 = &verts[i1];
                Vertex* v2 = &verts[i2];
                Vertex* v3 = &verts[i3];

                // Calculate vectors relative to triangle positions
                float x1 = v2->Position.x - v1->Position.x;
                float y1 = v2->Position.y - v1->Position.y;
                float z1 = v2->Position.z - v1->Position.z;

                float x2 = v3->Position.x - v1->Position.x;
                float y2 = v3->Position.y - v1->Position.y;
                float z2 = v3->Position.z - v1->Position.z;

                // Do the same for vectors relative to triangle uv's
                float s1 = v2->UV.x - v1->UV.x;
                float t1 = v2->UV.y - v1->UV.y;

                float s2 = v3->UV.x - v1->UV.x;
                float t2 = v3->UV.y - v1->UV.y;

                // Create vectors for tangent calculation
                float r = 1.0f / (s1 * t2 - s2 * t1);

                float tx = (t2 * x1 - t1 * x2) * r;
                float ty = (t2 * y1 - t1 * y2) * r;
                float tz = (t2 * z1 - t1 * z2) * r;

                // Adjust tangents of each vert of the triangle
                v1->Tangent.x += tx;
                v1->Tangent.y += ty;
                v1->Tangent.z += tz;

                v2->Tangent.x += tx;
                v2->Tangent.y += ty;
                v2->Tangent.z += tz;

                v3->Tangent.x += tx;
                v3->Tangent.y += ty;
                v3->Tangent.z += tz;
        }
```

```
        // Ensure all of the tangents are orthogonal to the normals
        for (int i = 0; i < numVerts; i++)
        {
                // Grab the two vectors
                XMVECTOR normal = XMLoadFloat3(&verts[i].Normal);
                XMVECTOR tangent = XMLoadFloat3(&verts[i].Tangent);

                // Use Gram-Schmidt orthonormalize to ensure
                // the normal and tangent are exactly 90 degrees apart
                tangent = XMVector3Normalize(
                        tangent - normal * XMVector3Dot(normal, tangent));

                // Store the tangent
                XMStoreFloat3(&verts[i].Tangent, tangent);
        }
}
```

Sample Code: Creating a cube map from 6 textures

Note: This code assumes you're putting the function in Game.cpp, you've included WICTextureLoader.h and you have an ID3D11Device com pointer called "device". Make any adjustments necessary for your own implementation.

Header

```
// Helper for creating a cubemap from 6 individual textures
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> CreateCubemap(
        const wchar_t* right,
        const wchar_t* left,
        const wchar_t* up,
        const wchar_t* down,
        const wchar_t* front,
        const wchar_t* back);
```

Code

```
// ---------------------------------------------------------
// Loads six individual textures (the six faces of a cube map), then
// creates a blank cube map and copies each of the six textures to
// another face.  Afterwards, creates a shader resource view for
// the cube map and cleans up all of the temporary resources.
// ---------------------------------------------------------
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> Game::CreateCubemap(
        const wchar_t* right,
        const wchar_t* left,
        const wchar_t* up,
        const wchar_t* down,
        const wchar_t* front,
        const wchar_t* back)
{
        // Load the 6 textures into an array.
        // - We need references to the TEXTURES, not SHADER RESOURCE VIEWS!
        // - Explicitly NOT generating mipmaps, as we don't need them for the sky!
        // - Order matters here!  +X, -X, +Y, -Y, +Z, -Z
        Microsoft::WRL::ComPtr<ID3D11Texture2D> textures[6] = {};
        CreateWICTextureFromFile(device.Get(), right, (ID3D11Resource**)textures[0].GetAddressOf(), 0);
        CreateWICTextureFromFile(device.Get(), left, (ID3D11Resource**)textures[1].GetAddressOf(), 0);
        CreateWICTextureFromFile(device.Get(), up, (ID3D11Resource**)textures[2].GetAddressOf(), 0);
        CreateWICTextureFromFile(device.Get(), down, (ID3D11Resource**)textures[3].GetAddressOf(), 0);
        CreateWICTextureFromFile(device.Get(), front, (ID3D11Resource**)textures[4].GetAddressOf(), 0);
        CreateWICTextureFromFile(device.Get(), back, (ID3D11Resource**)textures[5].GetAddressOf(), 0);
```

```
	// We'll assume all of the textures are the same color format and resolution,
	// so get the description of the first texture
	D3D11_TEXTURE2D_DESC faceDesc = {};
	textures[0]->GetDesc(&faceDesc);

	// Describe the resource for the cube map, which is simply
	// a "texture 2d array" with the TEXTURECUBE flag set.
	// This is a special GPU resource format, NOT just a
	// C++ array of textures!!!
	D3D11_TEXTURE2D_DESC cubeDesc = {};
	cubeDesc.ArraySize = 6; // Cube map!
	cubeDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE; // We'll be using as a texture in a shader
	cubeDesc.CPUAccessFlags = 0; // No read back
	cubeDesc.Format = faceDesc.Format; // Match the loaded texture's color format
	cubeDesc.Width = faceDesc.Width;  // Match the size
	cubeDesc.Height = faceDesc.Height; // Match the size
	cubeDesc.MipLevels = 1; // Only need 1
	cubeDesc.MiscFlags = D3D11_RESOURCE_MISC_TEXTURECUBE; // A CUBE, not 6 separate textures
	cubeDesc.Usage = D3D11_USAGE_DEFAULT; // Standard usage
	cubeDesc.SampleDesc.Count = 1;
	cubeDesc.SampleDesc.Quality = 0;

	// Create the final texture resource to hold the cube map
	Microsoft::WRL::ComPtr<ID3D11Texture2D> cubeMapTexture;
	device->CreateTexture2D(&cubeDesc, 0, cubeMapTexture.GetAddressOf());

	// Loop through the individual face textures and copy them,
	// one at a time, to the cube map texure
	for (int i = 0; i < 6; i++)
	{
		// Calculate the subresource position to copy into
		unsigned int subresource = D3D11CalcSubresource(
			0,  // Which mip (zero, since there's only one)
			i,  // Which array element?
			1); // How many mip levels are in the texture?

		// Copy from one resource (texture) to another
		context->CopySubresourceRegion(
			cubeMapTexture.Get(), // Destination resource
			subresource,          // Dest subresource index (one of the array elements)
			0, 0, 0,              // XYZ location of copy
			textures[i].Get(),    // Source resource
			0,  // Source subresource index (we're assuming there's only one)
			0); // Source subresource "box" of data to copy (zero means the whole thing)
	}

	// At this point, all of the faces have been copied into the
	// cube map texture, so we can describe a shader resource view for it
	D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
	srvDesc.Format = cubeDesc.Format; // Same format as texture
	srvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURECUBE; // Treat this as a cube!
	srvDesc.TextureCube.MipLevels = 1;       // Only need access to 1 mip
	srvDesc.TextureCube.MostDetailedMip = 0; // Index of the first mip we want to see

	// Make the SRV
	Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> cubeSRV;
	device->CreateShaderResourceView(cubeMapTexture.Get(), &srvDesc, cubeSRV.GetAddressOf());

	// Send back the SRV, which is what we need for our shaders
	return cubeSRV;
}
```