DirectX Coursework Report

by Vladeta Stojanovic (0602920@live.abertay.ac.uk)

Coursework Overview

This coursework report describes the various methods and techniques used in the development of the DirectX Coursework Application, with a specific focus on using HLSL shader effects and Direct3D programming. The coursework application was developed over a period of 12 weeks, with no prior knowledge of the DirectX API. The authors previous experience in Win32 and OpenGL API programming proved to be helpful in implementing the desired DirectX framework.

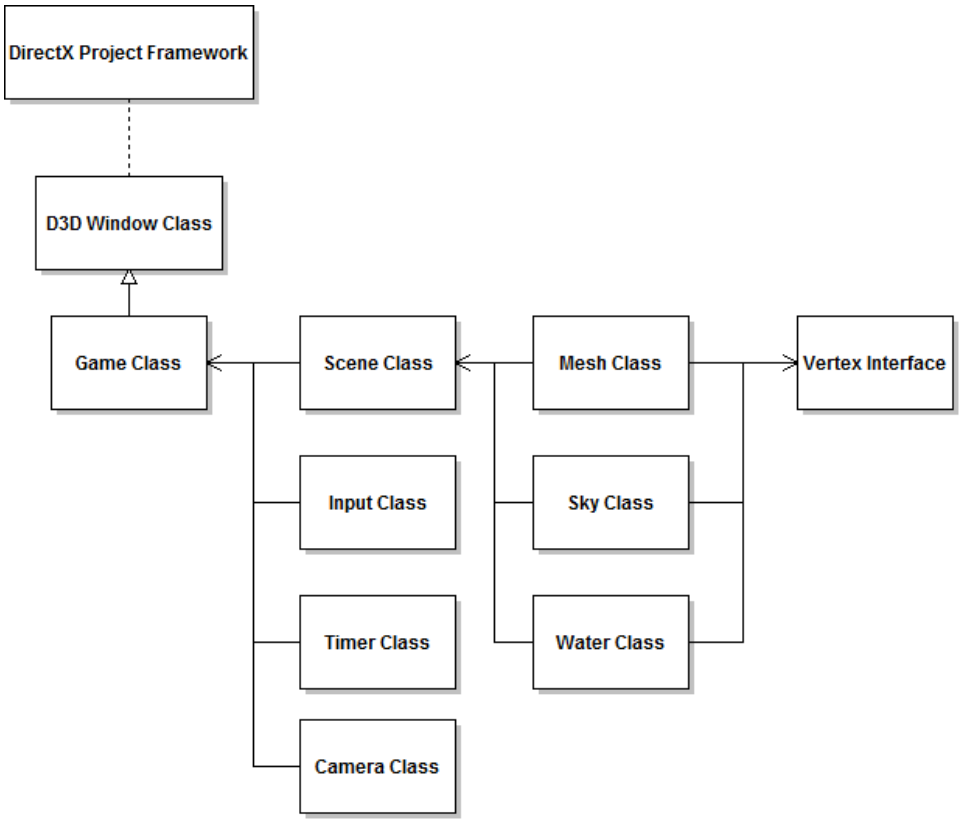
The coursework application makes use of the DirectX framework that was built in the 12 weeks, using many examples features in Frank D. Luna's “3D Game Programming with DirectX 9.0c: A Shader Approach” book, along with other examples provided by the module lecturer. The application framework was built using the DirectX 9.0c version of the DirectX API. DirectX 10 support was not implemented due to time constraints (though further work on projects that make use DirectX is planned, which will inevitably mean using DirectX 10 in the near future).

The main features of the coursework include exclusive use of the programmable graphics pipeline, using HLSL (High Level Shader Language). The aim of the coursework was to showcase some of the common 3D graphics rendering effects featured in today's games. These effects include per-pixel lighting using the Phong lighting model, environment mapping, normal mapping and using a combination of animated normal mapping and environment mapping to render a simple water effect.
Finally, the coursework application also makes use of parsing and rendering static .X mesh file format models, which allowed for a construction of a more complex 3D scene.

The 3D scene that was constructed for the coursework application shows an outdoor area with a pond, a rotating skull above the pond and a skysphere in the background. The reason why the skull object was chosen as the centre piece of the scene was because it is a fairly complex 3D mesh with cavities, meaning that it can be used to test various lighting and texturing techniques. In this case, the skull mesh is rendered using environment and diffuse map blending.

Framework Overview

Below is a high level UML overview of the constructed DirectX project framework:



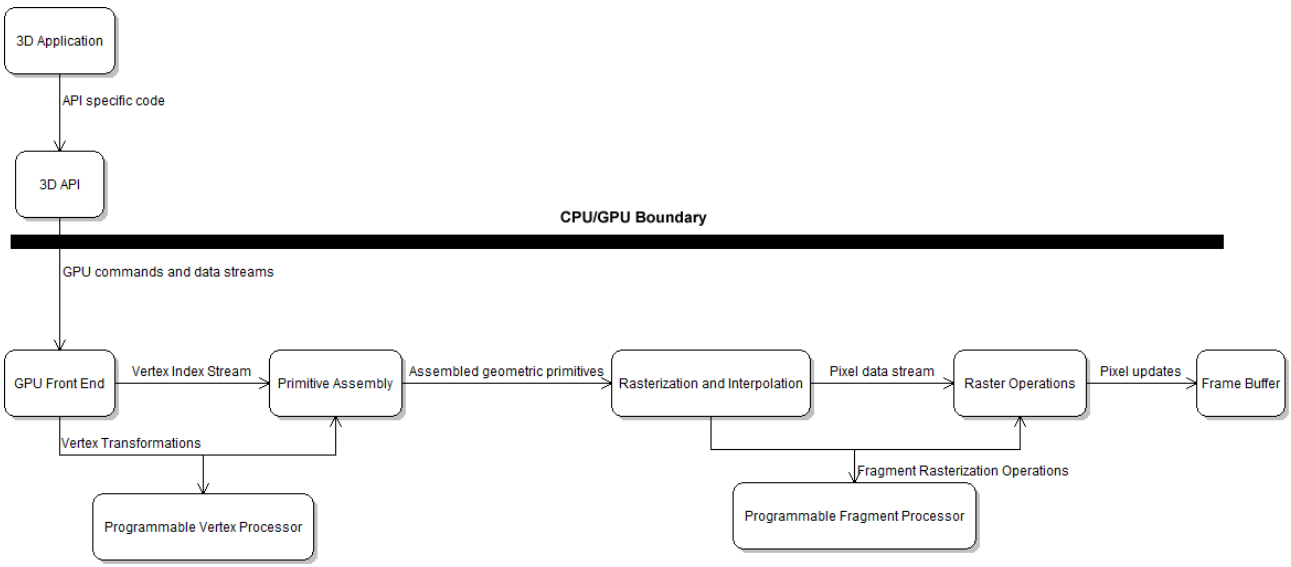
Direct3D and The Programmable Pipeline

The way rendering of 3D geometry in Direct3D is carried out with the use of vertex streaming. Since each geometric primitive is made up of a vertex (since a vertex is the most basic geometric primitive), a vertex deceleration structure is used to hold specific data that is used for the rendering of the given geometry. The data that's included in the vertex deceleration includes information such as the vertex position, colour, normal and UV texture coordinates. This information is manipulated during the rendering of the scene in order to present the viewer with the final rendered object on the screen. Along with the vertex structure, a vertex deceleration representing the given vertex data collection (structure to be more precise), is needed in order for Direct3D to know what vertices to stream for what geometry during the rendering of the scene.

Once the vertex structure and the appropriate vertex deceleration for that structure are in place, the given 3D geometry that implements the specific vertex type for it's rendering is approximated using using a triangle mesh. Triangle rasterization is the default geometry approximation method used by Direct3D.
The processing of the vertex data for each geometric primitive in the scene happens during the primitive assembly stage in the graphics pipeline. In the above text, the phrase “rendering” is mentioned. What is important to note here is that the term rendering is used to describe the entire process of rendering a geometric primitive to the screen via the programmable graphics pipeline.

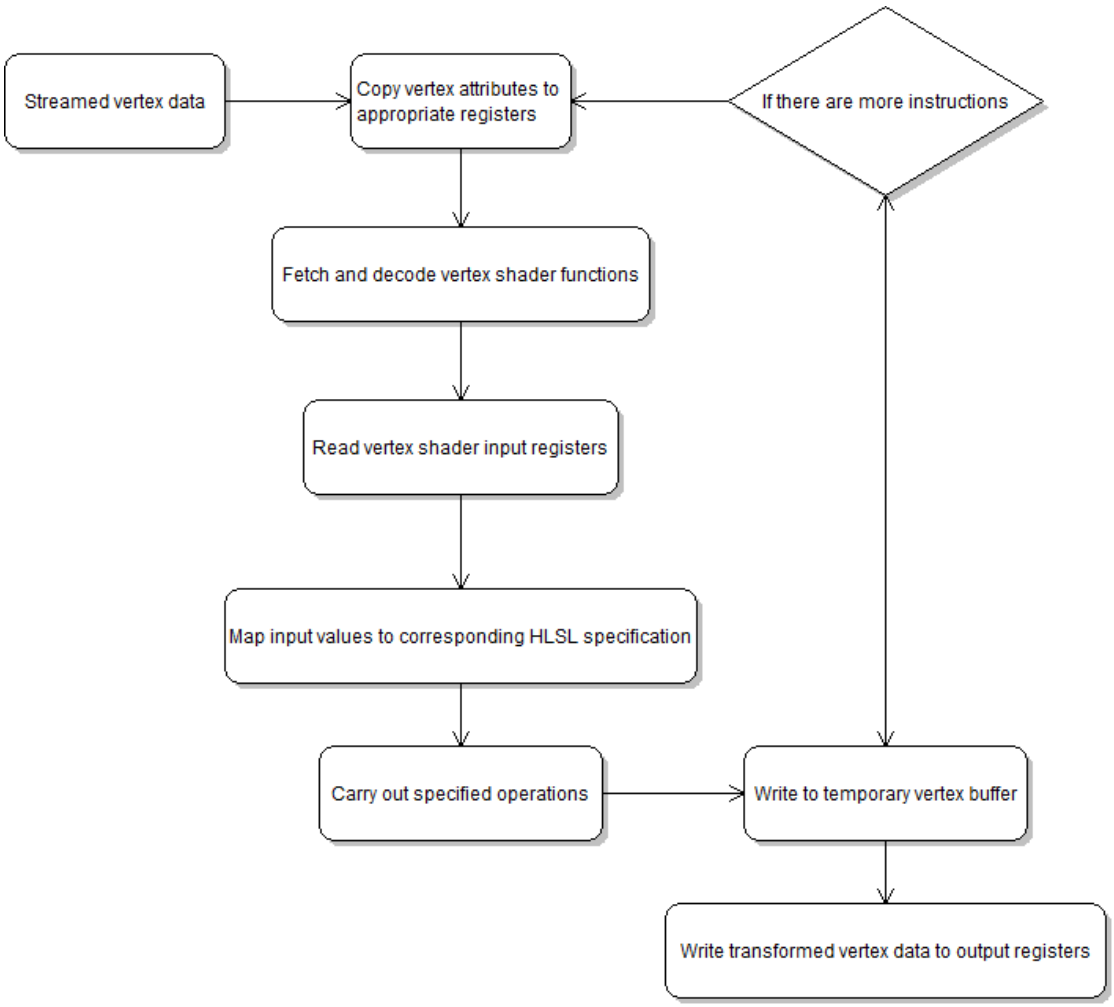
Unlike the fixed function pipeline, the programmable pipeline presents the developer with separate vertex and fragment processors, which are programmable. On the old fixed function pipeline, each of the computations stages, which included transformations, lighting, texturing and rasterization stages, would happen in a fixed manner where each of the operation methods were hard coded in the hardware of the graphics card. This approach greatly limited the possibilities of what could be accomplished in real time rendering, because programmers could not write their own custom operation methods for rendering of the scene geometry.

With the programmable graphics pipeline, the programmer is given much more flexibility in terms of being able to replace the old fixed function pipeline operation methods with custom written rendering techniques that would turn allow the given 3D scene to be rendered in various ways. The GPU (graphics processing unit) on all modern graphics cards supports the programmable pipeline programming model to a minimum level. Therefore, the programmers can write custom operations for rendering 3D scenes. These custom operations and techniques are called shaders.
Shaders are small programs that run on the GPU that perform vertex and fragment processing operations. The way these operations are performed is left for the programmer to decide, and allow the programmer to compute complex 3D rendering effects, that are computed either at the the primitive assembly or the rasterization stages of the programmable graphics pipeline. Below is an illustrated example of the programmable graphics pipeline model:



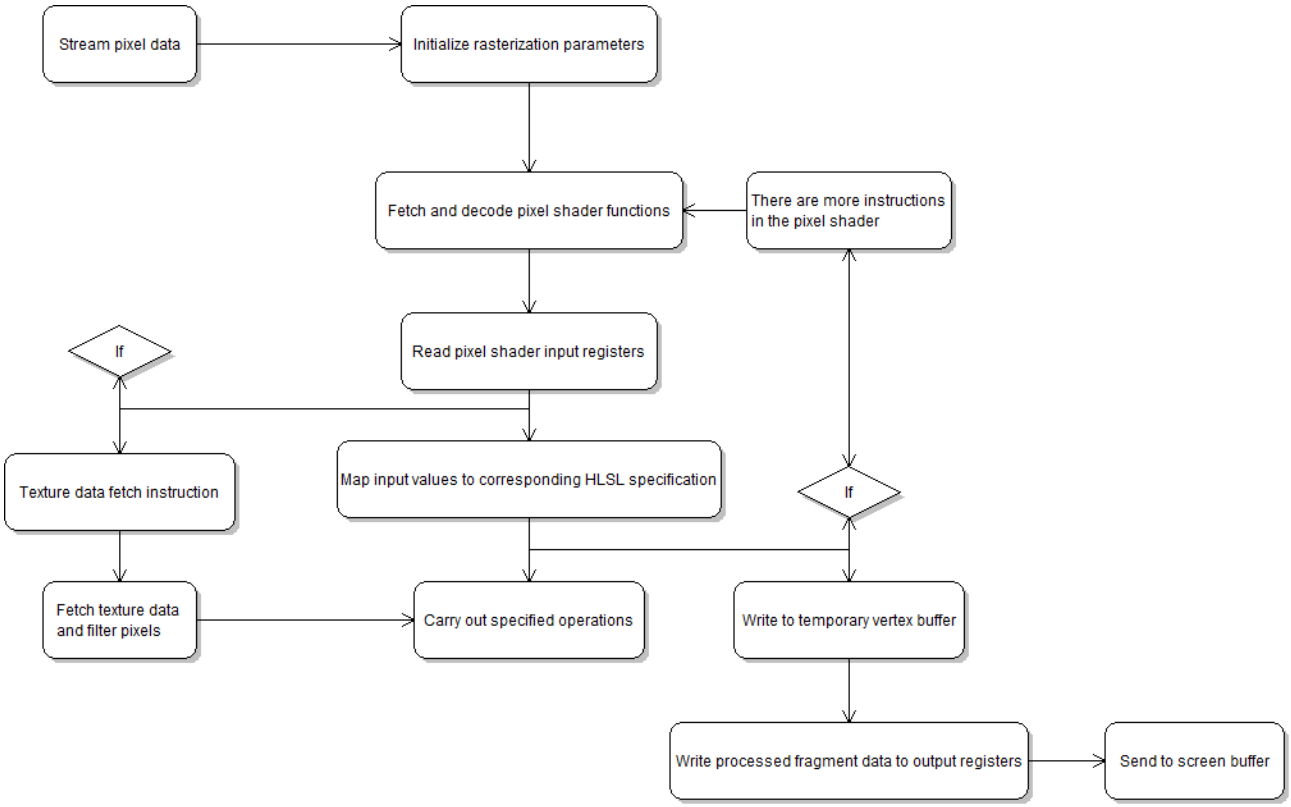
The programmable vertex processor is used for performing vertex operations on the streamed vertex data. The streamed vertex data is indexed so that vertex operations are not performed more then once (since a number of triangles can share the same vertex). The vertex data that is streamed includes a number of attributes (such as the vertex colour, position, normal, as mentioned in the above text).
The vertex processor fetches the streamed vertex data as read only data, as the vertex attributes are set and declared by the application. The vertex processor then performs the required vertex operations based on the code instructions in the vertex shader that is used. Once the vertex data has been processed, it is written to the output registers before being sent to the primitive assembly stage of the graphics pipeline.
The vertex processor has a number of registers which are used to store vertex data during the processing stages. The way the data is stored and how it is processed is described by the programmer in the vertex shader.

Below is an illustrated flow chart of the operational stages of the programmable vertex processor:



Like the programmable vertex processor, the programmable fragment processor performs various operations on the rasterized pixel data. The main stage of these operations the fragment processor is responsible for is texturing operations. The texturing operations access image data that is used by the primitive based on coordinates specified in the streamed vertex data. The image data along with the interpolated light and surface colour data is used to sample the final colour of the rasterized pixel before it gets written to the screen buffer. The way the fragment processor samples the image, lighting and surface colour data of the primitive that is being rendered, is controlled by a fragment shader (more commonly known as a pixel shader). Like the vertex shader, the pixel shader performs a number operations in a number of different ways based on the techniques specified by the programmer.

Below is an illustrated flow chart of the operational stages of the programmable fragment processor:



Overview of HLSL and Implementation of HLSL Shaders

High level Shader Language (HLSL) is the shader language used by Direct3D in order to allow programmers to write custom vertex and fragment programs for the GPU's programmable graphics pipeline. HLSL closely resembles C, however, it is compiled at run time and it's specifications changes with each new iteration of graphics hardware. The current standard specification that works on most of today's graphics hardware is the vertex and pixel shader version 2.0 specification.

A HLSL shader is written in the form of an effect file, with the generic .fx file extension. A HLSL shader effect file contains HLSL code for both vertex and fragment data operations. HLSL vertex and pixel shaders can target a range of hardware, allowing a single HLSL shader to run on a number of different graphics hardware configurations, where each different hardware configuration supports different versions of the vertex and pixel shader specification. However, it is up to the programmer to write specific versions of each vertex and pixel shader that are going to be compiled to a different specification, in order to run on different graphics hardware. In terms of the coursework application, all vertex and pixel shaders that were written support only the vertex and pixel shader 2.0 specification.

The coursework application framework makes extensive use of HLSL shaders. To use a shader with the coursework application, it must first be declared in the appropriate class or interface. All of the main shader effects in the coursework application (with the exception of the water and sky shader effects) are declared in the Scene class (found in scene.h):

```
ID3DXEffect*      mFX0;
ID3DXEffect*      mFX1;
```

The above two variables act as pointers to the ID3DXEffect interface. Since DirectX is COM based (component object model), all of the libraries used by D3D come in the form of interfaces. The above two effect declerations are used to perform enivronment and normal mapping in the 3D scene.

Each effect pointer also needs to allow the application using the effect file to have access to the variables that are declared in the in the shader file. The use of effect file handles is used in this case. Effect file handles serve as pointers to externed variables contained in the HLSL effect file. The application uses these effect handles to compile the effect file during the run time of the application, as well as to pass certain paramater values to the shader in order for the shader to perform the necessary operations on either the vertex or pixel data. These paramaters that are passed into the shader can include things like the global world, view and projection matrices, as well as sampled textures used by the application.

```
D3DXHANDLE   mhFXTech0; //Environment mapping shader technique
D3DXHANDLE   mhFXTech1; //NM + diffuse shader technique

D3DXHANDLE   mhWVP;
D3DXHANDLE   mhWorldInvTrans;
D3DXHANDLE   mhEyePos;
D3DXHANDLE   mhEyePosW;
D3DXHANDLE   mhWorld;
D3DXHANDLE   mhWVPfx1;
D3DXHANDLE   mhWorldInv;
D3DXHANDLE   mhMtrlfx1;
D3DXHANDLE   mhLightfx1;
D3DXHANDLE   mhEyePosWfx1;
D3DXHANDLE   mhTexfx1;
D3DXHANDLE   mhMtrl;
D3DXHANDLE   mhEnvMap;
D3DXHANDLE   mhReflectivity;
D3DXHANDLE   mhNormalMap;
D3DXHANDLE   mhTex;
D3DXHANDLE   mhLight;
```

Each of the declared handles for use with the two shaders in the scene relate to a corresponding declared variable in the application framework.

Once the handles are set up, next the shader has to be built. This process involves compiling the shader before the running the shader code, assigning the declared application effect file handles to the corresponding shader handles, and making sure the effect compiles without any errors.

The following function found in scene.cpp does this:

```
void Scene::BuildFX()
{
	// Create the FX from a .fx file.
	ID3DXBuffer* errors = 0;
	HR(D3DXCreateEffectFromFile(gd3dDevice, "shaders/EnvMap.c",
	   0, 0, D3DXSHADER_DEBUG, 0, &mFX0, &errors));

	HR(D3DXCreateEffectFromFile(gd3dDevice, "shaders/NormalMap.c",
	   0, 0, D3DXSHADER_DEBUG, 0, &mFX1, &errors));

	if(errors)
	{
		MessageBox(0, (char*)errors->GetBufferPointer(), 0, 0);
	}

	//Obtain handles ofr environment map shader
	mhFXTech0		= mFX0->GetTechniqueByName("EnvMapTech");
	mhWVP		= mFX0->GetParameterByName(0, "gWVP");
	mhWorldInvTrans		= mFX0->GetParameterByName(0, "gWorldInvTrans");
	mhMtrl		= mFX0->GetParameterByName(0, "gMtrl");
	mhLight		= mFX0->GetParameterByName(0, "gLight");
	mhEyePos		= mFX0->GetParameterByName(0, "gEyePosW");
	mhWorld		= mFX0->GetParameterByName(0, "gWorld");
	mhTex		= mFX0->GetParameterByName(0, "gTex");
	mhEnvMap		= mFX0->GetParameterByName(0, "gEnvMap");
	mhReflectivity		= mFX0->GetParameterByName(0, "gReflectivity");


	//Obtain handles for normal map shader
	mhFXTech1		= mFX1->GetTechniqueByName("NormalMapTech");
	mhWVPfx1		= mFX1->GetParameterByName(0, "gWVP");
	mhWorldInv		= mFX1->GetParameterByName(0, "gWorldInv");
	mhMtrlfx1		= mFX1->GetParameterByName(0, "gMtrl");
	mhLightfx1		= mFX1->GetParameterByName(0, "gLight");
	mhEyePosWfx1		= mFX1->GetParameterByName(0, "gEyePosW");
	mhTexfx1		= mFX1->GetParameterByName(0, "gTex");
	mhNormalMap		= mFX1->GetParameterByName(0, "gNormalMap");
}
```

In the above function, the environment mapping and normal mapping effects gets compiled and their relative framework variable handles get assigned to them. The same function is found in the water and skybox classes.

```
void Scene::Draw(D3DXMATRIX pMatrix, D3DXVECTOR3 CamPos)
{
	// Setup the rendering FX
	HR(mFX0->SetTechnique(mhFXTech0));

	mFX0->SetValue(mhLight, &mLight, sizeof(DirLight));
	mFX0->SetValue(mhReflectivity, &reflectivity1, sizeof(float));

	D3DXMATRIX worldInvTrans;
	D3DXMatrixInverse(&worldInvTrans, 0, &pMatrix);
	D3DXMatrixTranspose(&worldInvTrans, &worldInvTrans);
```

```
	mFX0->SetMatrix(mhWVP, &pMatrix);
	mFX0->SetMatrix(mhEyePos, &pMatrix);
	mFX0->SetMatrix(mhWorldInvTrans, &pMatrix);
	mFX0->SetMatrix(mhWorld, &pMatrix);
	mFX0->SetValue(mhWorld, &pMatrix, sizeof(D3DXVECTOR3));

	//Draw skysphere here
	gSky->draw(pMatrix, CamPos);

	//Draw the water here
	gWater->draw(pMatrix, CamPos);

	//Draw environment mapped skull mesh here
	gMesh->DrawEnvBlnd(&pMatrix, mFX0, mhMtrl,  mhEnvMap, mhTex);

	//Draw normal mapped scene geometry here
	mFX1->SetValue(mhLightfx1, &mLight, sizeof(DirLight));
	mFX1->SetMatrix(mhWorldInv, &mWorldInv);
	mFX1->SetMatrix(mhEyePosWfx1, &pMatrix);
	mFX1->SetMatrix(mhWVPfx1, &pMatrix);
	mFX1->SetTechnique(mhFXTech1);

	gLevel->DrawDiffNM(&pMatrix, mFX1, mhNormalMap, mhMtrlfx1, mhTexfx1);
}
```

The shaders can be used in the scene by setting up the matrix and lighting transformation parameters of the shader effects, and passing them as pointers to the relative drawing functions. In this case, the normal mapping effect handle is passed into the the level drawing function (once it has been set up), and the environment mapping effect handle is passed into the function used for drawing reflective skull in the 3D scene.

Finally, it is important to update the effect file handles along with the rest of the the scene, in case the global D3D device is reset or changes screen formats:

```
void Scene::SceneOnLostDevice()
{
	HR(mFX0->OnLostDevice());
	HR(mFX1->OnLostDevice());
	gSky->OnLostDevice();
	gWater->OnLostDevice();
}

void Scene::SceneOnResetDevice()
{
	HR(mFX0->OnResetDevice());
	HR(mFX1->OnResetDevice());
	gSky->OnLostDevice();
	gWater->OnResetDevice();
}
```

Now it's time to look at the actual code structure and syntax of the HLSL effect files. As mentioned before, HLSL is a lot like C, but perhaps a bit more simpler. The environment mapping shader is a good example shader to look at to see how HLSL shaders are written.

First, all of the global variables and structs are declared in the effect file. The global variables are used as references to the handles passed in by the application (see above):

```
uniform extern float4x4 gWorld;
uniform extern float4x4 gWorldInvTrans;
uniform extern float4x4 gWVP;
uniform extern Mtrl     gMtrl;
uniform extern DirLight gLight;
uniform extern float3   gEyePosW;
uniform extern texture  gTex;
uniform extern texture  gEnvMap;
uniform extern float    gReflectivity;
```

Additionally, the variable data structures that are used in the effect file are declared:

```
struct Mtrl
{
      float4 ambient;
      float4 diffuse;
      float4 spec;
      float  specPower;
};

struct DirLight
{
      float4 ambient;
      float4 diffuse;
      float4 spec;
      float3 dirW;
};

sampler TexS = sampler_state
{
      Texture = <gTex>;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
      MipFilter = LINEAR;
      AddressU  = WRAP;
      AddressV  = WRAP;
};

sampler EnvMapS = sampler_state
{
      Texture = <gEnvMap>;
      MinFilter = LINEAR;
      MagFilter = LINEAR;
      MipFilter = LINEAR;
      AddressU  = WRAP;
      AddressV  = WRAP;
};

struct OutputVS
{
    float4 posH    : POSITION0;
    float3 normalW : TEXCOORD0;
    float3 toEyeW  : TEXCOORD1;
    float2 tex0    : TEXCOORD2;
};
```

The above data structs are used to set the lighting, material and texture parameters that are used in the effect file. The most significant struct thought is the OutputVS struct. The OutputVS struct holds the output register values for the processed vertex data that gets passed to the pixel shader for rasterization and texturing operations.

The implemented techniques used in the vertex and pixel shaders will be discussed below along with their relative mathematical models. Below is the deceleration of the vertex processing operations stage of the effect file:

```
OutputVS EnvMapVS(float3 posL : POSITION0, float3 normalL : NORMAL0, float2
tex0: TEXCOORD0)
{
      OutputVS outVS = (OutputVS)0;
      return outVS;
}
```

Note that the return value of the vertex shader function is declared as the OutputVS struct. The input parameters represent the different values that can be passed into the corresponding registers in the programmable vertex processor on the GPU.

Next the pixel shader operations function is declared:

```
float4 EnvMapPS(float3 normalW : TEXCOORD0,  float3 toEyeW  : TEXCOORD1, float2
tex0    : TEXCOORD2) : COLOR
{
      float4 rgba = NULL;
      return rgba;
}
```

The default return value of the pixel shading stage is the pixels RGBA colour value. The pixel format is set up by the framework, but the pixel shader writes the rasterized and processed pixels to the screen buffer, from where they get converted to the declared screen buffer format before being displayed. The input values of the pixel shader are usually the sampled texture coordinates that the are specified in the vertex structure attributes of the geometric primitive that gets rasterized. The operations are not carried out on the texture coordinates themselves, but on the actual image data that gets mapped onto the geometric primitives surface. Various per-pixel operations such as alpha blending, diffuse mapping and lighting approximations are performed on the rasterized pixels of the geometric primitive before the processed pixel data is written to the frame buffer.
Finally, both the vertex and pixel shader function stages get compiled in what is called a technique. As mentioned earlier, different specifications of the vertex and pixel shaders can target different graphics hardware capabilities. A technique is the sub-routine of an HSLS shader file that compiles the specified vertex and pixel shader operation stages to a target graphic hardware capability specification. In this case, the target vertex and pixel shader specification is the vertex and pixel shader 2.0 specification, which is compatible with the majority of today's graphics hardware.
Additionally, an effect file technique can have a number of passes for each of the vertex and pixel data processing stages, where for each stage, a set of computations is performed on the vertex and pixel data. This allows the graphics programmer to implement complex rendering algorithms that may require various stages of vertex and pixel data processing, and where each stage has a different set of processing instructions for the vertex and pixel shader that is compiled for that specific pass.
Finally, various scene rendering parameters can also be set in each pass. In this case, the geometry culling mode is set to cull geometry whose vertex winding order is counter clock wise.

```
technique EnvMapTech
{
    pass P0
    {
        vertexShader = compile vs_2_0 EnvMapVS();
        pixelShader  = compile ps_2_0 EnvMapPS();
        CullMode = CCW;
    }
}
```

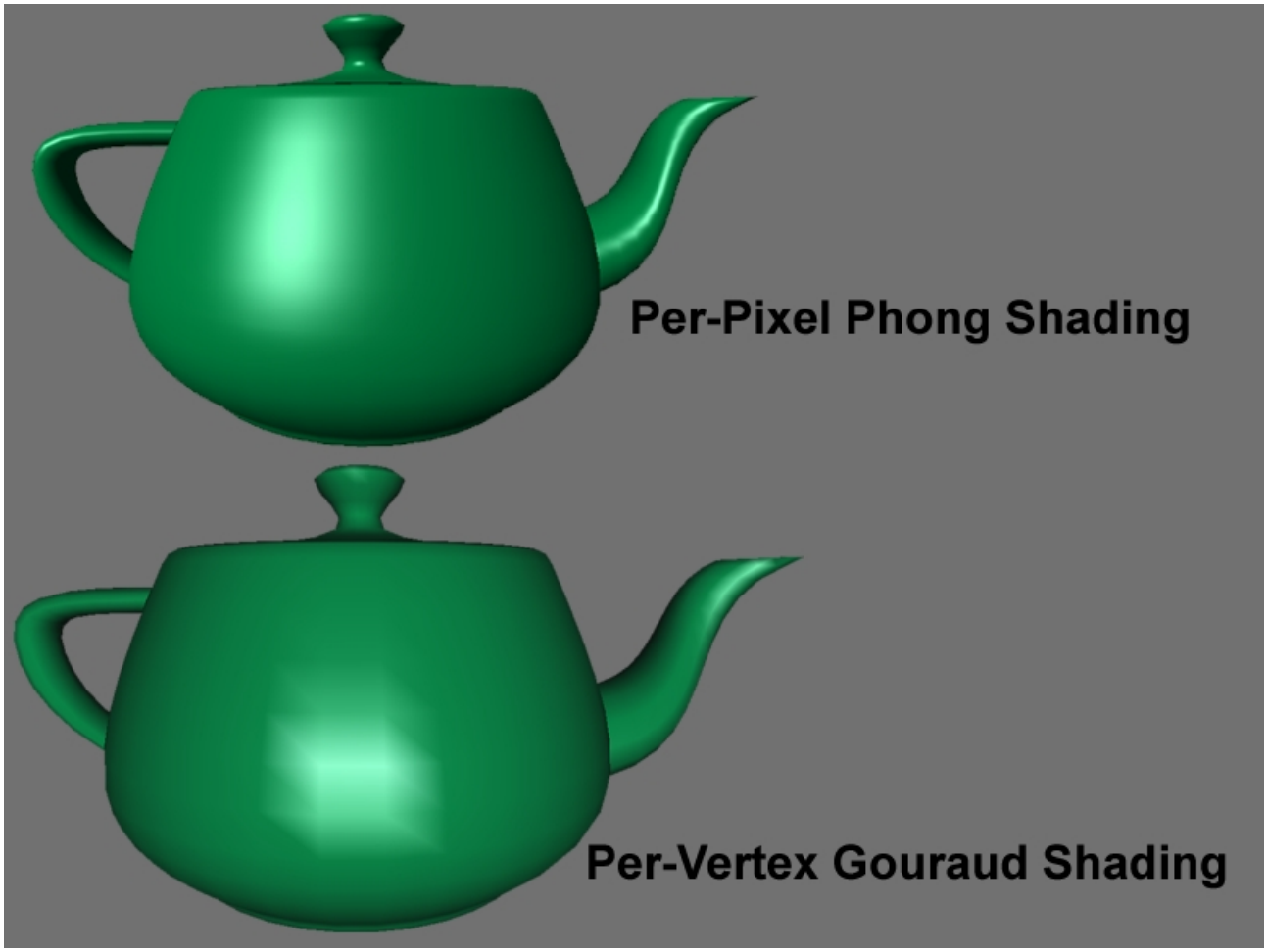
Per-Pixel Lighting

One of the most significant aspects of the coursework application is the implementation of per-pixel lighting. Per-pixel lighting computations are performed during the fragment rasterization stage by the programmable fragment processor, and these operations are programmed using the pixel shader.

Per-pixel lighting is a term used to describe interpolation of lighting colour values across the rasterized primitives pixel data. This means that instead of doing the lighting interpolation calculations during the vertex processing stage of the pipeline, the lighting interpolation is done during the fragment rasterization stage of the pipeline. This allows for the colour gradient values based on the vertex normal data to be interpolated across the rasterized geometric primitives pixel data, making the lighting much more refined. This in turn allows for smoother surface shading of the rendered mesh.

This report does not go into detail about the directional lighting model used in the coursework application. However, should the reader wish to read more about the mathematical theory behind directional lighting, it is recommend that the reader looks at another paper written by the author of this report, entitled “AG09202A PlayStation 2 3D Graphics Programming Coursework Report), available either from the author upon request or from Dr Henry S. Fortuna.

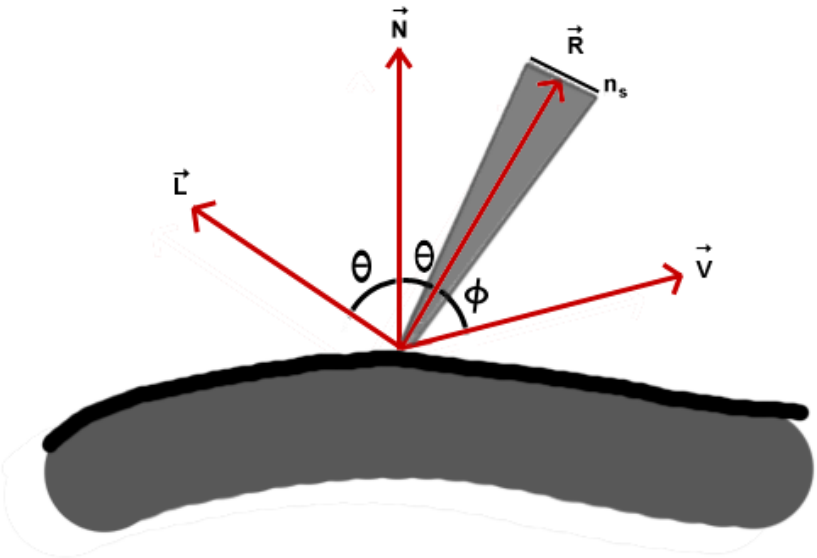
The below picture shows the difference in the low aesthetic quality of per-vertex Gouraud shading algorithm in comparison to the superior per-pixel Phong shading algorithm:



Phong Lighting

The Phong lighting model is a shading algorithm with two component models used for approximation of reflective surfaces and approximation of surface lighting. The Phong lighting model is used to interpolate ambient, diffuse and specular components of a given surface of a mesh, taking into account the lighting model that's implemented in the given scene (directional, spot or point, among others).

The Phong specular-reflection model is used to approximate illumination of shiny surfaces. Specular reflection of a surface can be illustrated as the following:



Specular reflection occurs when the incident of the light that is reflected from the surface is reflected in near or full amount. This reflection of light occurs around the specular reflection angle. The more shiny a surface is, the less light it absorbs, thus the more it reflects. Therefore, the specular reflection angle is larger on surfaces that absorb more light, and smaller on those that reflect more light then they absorb. The smaller the specular reflection angle, the higher the concentration of light around the reflection angle is.

The above diagram shows a surface that is being hit by a ray of light, represented as the reflection direction unit vector R. The unit vector N represents the surface normal vector, L represents the unit vector that is pointed towards the light source (reflection vector), and V is the unit vector that is the viewers eye position (or the world camera position). The angle of incidence of the light source is measured as the total size of theta, while the angle phi is the viewing angle from the reflection direction R to the viewers eye.

The above diagram of specular reflection can be modelled mathematically and presented as the Phong specular-reflection model:

$$I_{spec} = W(\Theta) I_l \cos^{n_s} \Phi$$

Cos^n_s (phi) is used to set the specular reflection of the given surface. The values of phi range from 0 to 90 degrees, which in turn makes the value of cos vary from 0 to 1. The n_s degree component of cos is a specular-reflection parameter. The value of n_s is used to set the “shininess” of the surface. Values above 100 set for n_s are used for highly reflective surfaces, while less reflective surfaces tend to have a value nearing 1.

The equation component W(theta) is the specular-reflection coefficient and is used to approximate the specular intensity of a surface. The value of W(theta) increases as the angle of the incidence increases. If the angle of incidence is 90 degrees, the value of W(90 degrees) becomes 1 and the surface perfectly reflects the incident light rays. I_l is the intensity of the light source.

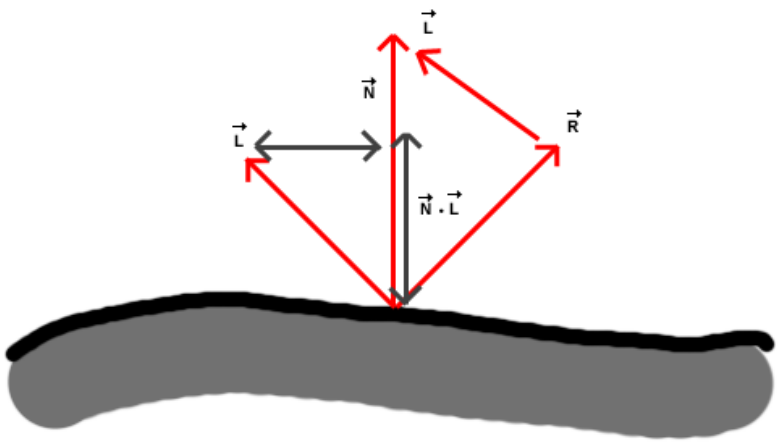
The Phong specular-reflection model can be optimized by replacing the the equation component W(theta) with a constant value of K_s (specular constant), which is in the range from 0 to 1. Then the angle between the viewer and the specular reflection can be calculated using vector arithmetic, by finding the dot product between the vectors V and R. This optimization then determines the intensity of the specular reflection a given point on the meshes surface:

$$I_{spec} = K_s I_l (\overline{V} . \overline{R})^{n_s}$$

The vector R (specular reflection vector), is obtained as:

$$\overline{R} = (2\overline{N} . \overline{L})\overline{N} - \overline{L}$$

The L reflection vector value is obtained by finding the dot product between L and the surface normal vector N. The reason why the dot product between N and L is multiplied by 2 is because it calculated for both projection to the specular reflection vector R and to the light reflection vector L. Below is a diagram to illustrate this:



The second component of the Phong lighting model is the method for interpolation of surface lighting. This algorithm is commonly referred to as Phong shading. Phong shading performs interpolation of lighting values given the averaged surface normal of a geometric primitive. Unlike Gouraud shading where the lighting intensity is interpolated in a bilinear manner per-vertex, the Phong shading algorithm performs linear interpolation of lighting gradients of the averaged pixel normals on a per-pixel level, during the rasterization stage of the graphics pipeline. This allows for smoother shading of surfaces and more realistic rendering of specular reflections as the rendering artefacts of uneven gradient lighting, as found with Gouraud shading, are virtually eliminated.

Lighting interpolation using Phong shading is done at each pixel rather then each vertex, making Phong shading more computationally expensive then Gouraud shading. The Phong lighting interpolation method performs linear interpolation of each averaged pixel normal of the lit primitive. This can be used in addition to the Phong reflection model, which in turn will compute the final pixel colour of the given primitive at the rasterization stage of the graphics pipeline. The linear interpolation of the lighting values from the normals of a given shaded surface can be approximated using Taylor-series expansion, as recommended in the book *Computer Graphics, Second Edition* (Hearn, Pauline Baker. 1994).

In terms of implementing the Phong lighting model in HLSL, the effect file is used to add diffuse Phong shading as well as Phong specular-reflection to a given mesh surface. Below are the main excerpts form the effect file, which show the vertex and pixel shader computation stages.
First the vertex shader is used to compute the averaged unit normal of each vertex of the surface. The vertex position of the mesh is then transformed to world coordinates using the world matrix, before being transformed by the homogeneous clipping coordinates using the world-view-projection matrix product.

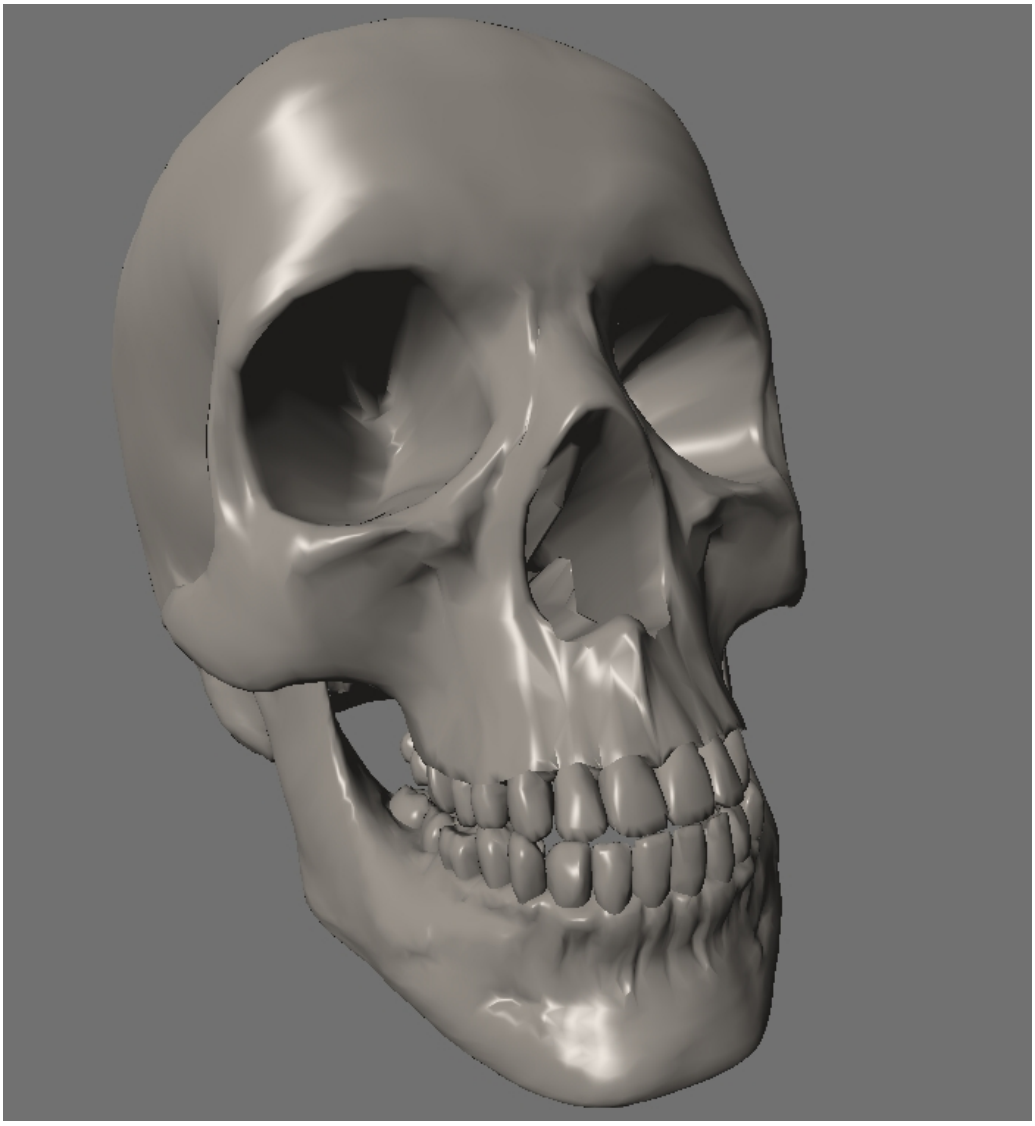
```
OutputVS PhongDirLtTexVS(float3 posL : POSITION0, float3 normalL : NORMAL0,
float2 tex0: TEXCOORD0)
{
      OutputVS outVS = (OutputVS)0;

      outVS.normalW = mul(float4(normalL, 0.0f), gWorldInvTrans).xyz;

      float3 posW  = mul(float4(posL, 1.0f), gWorld).xyz;

      outVS.toEyeW = gEyePosW - posW;

      outVS.posH = mul(float4(posL, 1.0f), gWVP);

      outVS.tex0 = tex0;

    return outVS;
}
```

The pixel shader is used to perform the actual lighting calculations. In the pixel shader, the L N R and V vectors are computed and are used to compute the final RGBA colour value computed by the pixel shader, using the specular, ambient and diffuse lighting colour terms computed from values obtained from vector arithmetic operations on the LNRV vectors.

```
float4 PhongDirLtTexPS(float3 normalW : TEXCOORD0, float3 toEyeW  : TEXCOORD1,
float2 tex0 : TEXCOORD2) : COLOR
{
      normalW = normalize(normalW);
      toEyeW  = normalize(toEyeW);

      float3 lightVecW = -gLight.dirW;
      float3 r = reflect(-lightVecW, normalW);

      float t  = pow(max(dot(r, toEyeW), 0.0f), gMtrl.specPower);
      float s = max(dot(lightVecW, normalW), 0.0f);

      float3 spec = t*(gMtrl.spec*gLight.spec).rgb;
      float3 diffuse = s*(gMtrl.diffuse*gLight.diffuse).rgb;
      float3 ambient = gMtrl.ambient*gLight.ambient;
```

```
        float4 texColor = tex2D(TexS, tex0);

        float3 color = (ambient + diffuse)* texColor.rgb + spec;

    return float4(color, gMtrl.diffuse.a*texColor.a);
}
```

Below is a picture of the final result of implementing the Phong lighting model in HLSL:



Texturing Techniques

This part of the report will talk about the various texturing techniques used in the development of the DirectX framework application, in order to achieve various real-time rendering effects.

Diffuse Texturing

The term diffuse texturing is used to describe the process of copying image data from an image file and mapping the pixel colour values of the image onto a the pixels of the rasterized geometry. The location to where the pixels of the image are copied onto the face of the given rasterized geometric primitive is defined by the UV coordinates that are stored as a vertex data attribute passed in during the vertex processing stage of the graphics pipeline.
The final colour of the rasterized pixel is interpolated along with the colour value of the copied image data pixel as well as the surface lighting colour values, during the fragment processing stage in the graphics pipeline. This in turn allows for the display of textured primitives.
In essence, a texture can be though of as an 2D array map stored in memory, where each value in the map corresponds to the colour value of each of the pixels in the image file that is loaded into the application (either during run time or during a loading function).

During the rasterization stage, the texture pixel data is blended with the rasterized pixel data of the geometric primitive, where the texture pixels (texels) are mapped from the texture in memory to one of the temporary registers on the fragment processor where the copied image pixel data is blended with the rasterized primitive pixel data. The mapping of the image pixel data onto the geometric primitive is done in screen space. In most cases, the size of the image is not the same as the size of the geometric primitive in screen space. If the image is smaller then the coordinated mapping area of the primitives surface, the texture data sampled from the image is stretched across the surface of the given primitive.
Depending on the size of the texture in comparison to the size geometric primitive in screen space (which changes per frame), if the texture is smaller, it is stretched and if it is larger, it is shrinked down (this is correctly referred to as magnification and minification of texels, where texels are used to represent a textured pixel unit of the texture map). Constantly updating the texels based on the users view of the scene and converting them to screen space, per frame during the run time of the application, causes rendering artefacts to appear.

These rendering artefacts can be smoothed out with the use of filtering. Direct3D offers a few built in filtering options. The filtering method that was used in the coursework application is the linear filtering method. This filter works by averaging the colour value of 2x2 texels around a rasterized pixel in screen space. Anisotropic filtering can also be used, but it is more computationally expensive.
In addition to the use of filtering, mip-mapping can also be used. The mip-mapping is the process of taking the default texture size, and making many more copies of it (up to 10, referred to as mipmap levels), with each copy being half the size of the previous. These texture copies are then stored in texture memory along with the original size texture. Then as the screen space pixels are updated per frame, a mipmap of the nearest size is selected and mapped onto the given rasterized geometric primitive.

Additionally, a texture may go over the specified texture coordinates of a given textured primitive, in which case a corresponding address mode is used to determine what happens to these overdrawn texture coordinates. In most cases, the texture will simply repeat itself from the coordinate where it is overdrawn (the UV texture coordinate range for both U and V texture coordinate screen space vectors is in the range from 0 to 1), or it can be clamped or mirrored. The default mode use by Direct3D is the texture tiling mode.

The size of the texture loaded in memory depends on the image file format. A typical image file used for 3D graphics running on modern hardware may be 24-bit, with 8 bits for each of the RGB channels, or it may be 32bit, with 8 bits for each of the RGB channels plus an additional 8bits for the alpha channel (used for alpha blending, aka texture transparency). Direct3D stores the textures in memory as surfaces, and there are many different formats for textures available. The generic file format for DirectX surfaces is the DDS file format, which is used to represent different textures formats in memory.
Since most image files that are converted to texture data can be very large, thus take up a lot of memory, textures can be compressed. This allows the texture to maintain it's aesthetic quality (to a degree), while being smaller in size.
Textures can also be animated over the given geometry surfaces by changing and updating the texture UV coordinates per frame.

Finally, it should be noted that diffuse texturing, by default, requires only one texture, but multiple textures can be rendered over a single geometric surface texture map. This is called multi-texturing, and makes use of texel blending during the fragment processing stage in the graphics pipeline. Since the diffuse texture is blended with the interpolated surface lighting colour, the mapped surface texels do not take into account the change of the interpolated surface lighting colour. This can be solved with the implementation of normal mapping, discussed below.

Normal Mapping

Below is a picture showing the combination of a normal map with a diffuse map, which are then applied to a given mesh surface:

Normal mapping is a technique used to simulate changes in high frequency surface gradients. This refers to things like bumps, grooves, dents, scratches, etc, commonly found on everyday surfaces. The human eye is able to detect these subtle details in real life due to perception of light changes affecting the appearance of these details. In order to simulate these high frequency details, normal mapping is used. The reason why normal mapping is used is because it would be otherwise computationally expensive to model these surface details using actual geometric primitives. Normal mapping approximates these details using texturing techniques.

A normal map is a texture file that holds XYZ directional values in place of it's RGB channels. These direction values present the normal vectors of each pixel of the given rasterized surface. This in turns allows for lighting values to be calculated for each pixel based on the normal vectors provided by the normal map. Therefore instead of performing lighting calculations of the geometry surface at each vertex normal, the lighting interpolation is done at each pixel normal. The Phong lighting model is ideal to use in such a situation, as it performs per-pixel lighting interpolation.
It is beyond the scope of this report to go into details about the techniques used for the generation of normal maps, so instead the next few paragraphs will describe how to use normal map textures that have already been generated.

Since the normal maps contains vector normal values stored in screen space, and and the scene lighting is relative to world space, using normal maps requires making the normal maps pixel normals to be visually responsive to the scene lighting. This can be accomplished by transforming the scene lighting data to the screen space coordinates of the polygon onto which the normal map is interpolated over.

Using a 3x3 matrix matrix, the lighting values affecting each pixel normal can be computed using the pixel normals stores in the normal map, rather then based on the surfaces constant per-pixel lighting interpolation.
In order to calculate this, three different vectors are needed in order to transform the world lighting information to the screen space pixel data for interpolation and rasterization. The three different vector data components required are the tangent, binormal and normal vectors. The tangent and the binormal vectors are used as tangent values of the given surface, and the normal vector is the normal of that surface. The tangent and the binormal surface values are calculated from the XYZ coordinate values stored for each pixel in the normal map. This in turn allows the world lighting data in the scene to be used when interpolating the surface colour value of a given normal mapped surface.

Now, the since the normal map is relative to screen space coordinates, the world lighting coordinates need to be transformed to screen space coordinates in order for the final surface colour to be successfully interpolated. This is done by defining the screen space coordinates of the normal map using a 3x3 matrix:

$$M_{\Delta} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}$$

The above matrix transforms the normal map coordinates from screen space to world space. So to transform the lighting coordinates from world space to screen space, the transpose of the above matrix is used:

$$M_{\Delta}^{-1} = M_{\Delta}^{T} = \begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

In the coursework application, this is implemented as the following function (in mesh.cpp, as part of the XMesh class, though the same technique is used in the Water class):

```
void XMesh::LoadandInitNMMap(char * meshFile, char * meshTex, char * nmTex)
{
	ID3DXMesh* tempMesh = 0;

	LoadXFile(meshFile, &tempMesh, mMtrl, mTex);
	D3DXCreateTextureFromFile(gd3dDevice, nmTex, &mNMTex);
	D3DXCreateTextureFromFile(gd3dDevice, meshTex, &mLevelTex);

	D3DVERTEXELEMENT9 elems[MAX_FVF_DECL_SIZE];
	UINT numElems = 0;
	HR(NMapVertex::Decl->GetDeclaration(elems, &numElems));

	// Clone the mesh to the NMapVertex format
	ID3DXMesh* clonedTempMesh = 0;
	HR(tempMesh->CloneMesh(D3DXMESH_MANAGED, elems, gd3dDevice,
	&clonedTempMesh));

	// Now use D3DXComputeTangentFrameEx to build the TNB-basis for each
	vertexin the mesh
	HR(D3DXComputeTangentFrameEx(
	  clonedTempMesh,
	  D3DDECLUSAGE_TEXCOORD, 0,
	D3DDECLUSAGE_BINORMAL, 0,
	  D3DDECLUSAGE_TANGENT, 0,
	D3DDECLUSAGE_NORMAL, 0,
	0,
	0,
	0.01f, 0.25f, 0.01f,
	&mLevel,
	0));
	// Done with temps.
	Release(tempMesh);
	Release(clonedTempMesh);
}
```

The highlighted function **D3DXComputeTangentFrameEx()** is used to compute the TBN values of a given surface.

The HLSL effect file is used perform the normal map based lighting calculations. The vertex shader is used to transform the lighting information from world space to screen space.

```
OutputVS NormalMapVS(float3 posL      : POSITION0,
                     float3 tangentL  : TANGENT0,
                     float3 binormalL : BINORMAL0,
                     float3 normalL   : NORMAL0,
                     float2 tex0      : TEXCOORD0)
{
    // Zero out our output.
      OutputVS outVS = (OutputVS)0;

      // Build TBN-basis.
      float3x3 TBN;
      TBN[0] = tangentL;
      TBN[1] = binormalL;
      TBN[2] = normalL;

      // Matrix transforms from object space to tangent space.
      float3x3 toTangentSpace = transpose(TBN);

      // Transform eye position to local space.
      float3 eyePosL = mul(float4(gEyePosW, 1.0f), gWorldInv);

      // Transform to-eye vector to tangent space.
      float3 toEyeL = eyePosL - posL;
      outVS.toEyeT = mul(toEyeL, toTangentSpace);

      // Transform light direction to tangent space.
      float3 lightDirL = mul(float4(gLight.dirW, 0.0f), gWorldInv).xyz;
      outVS.lightDirT  = mul(lightDirL, toTangentSpace);

      // Transform to homogeneous clip space.
      outVS.posH = mul(float4(posL, 1.0f), gWVP);

      // Pass on texture coordinates to be interpolated in rasterization.
      outVS.tex0 = tex0;

      // Done--return the output.
    return outVS;
}
```

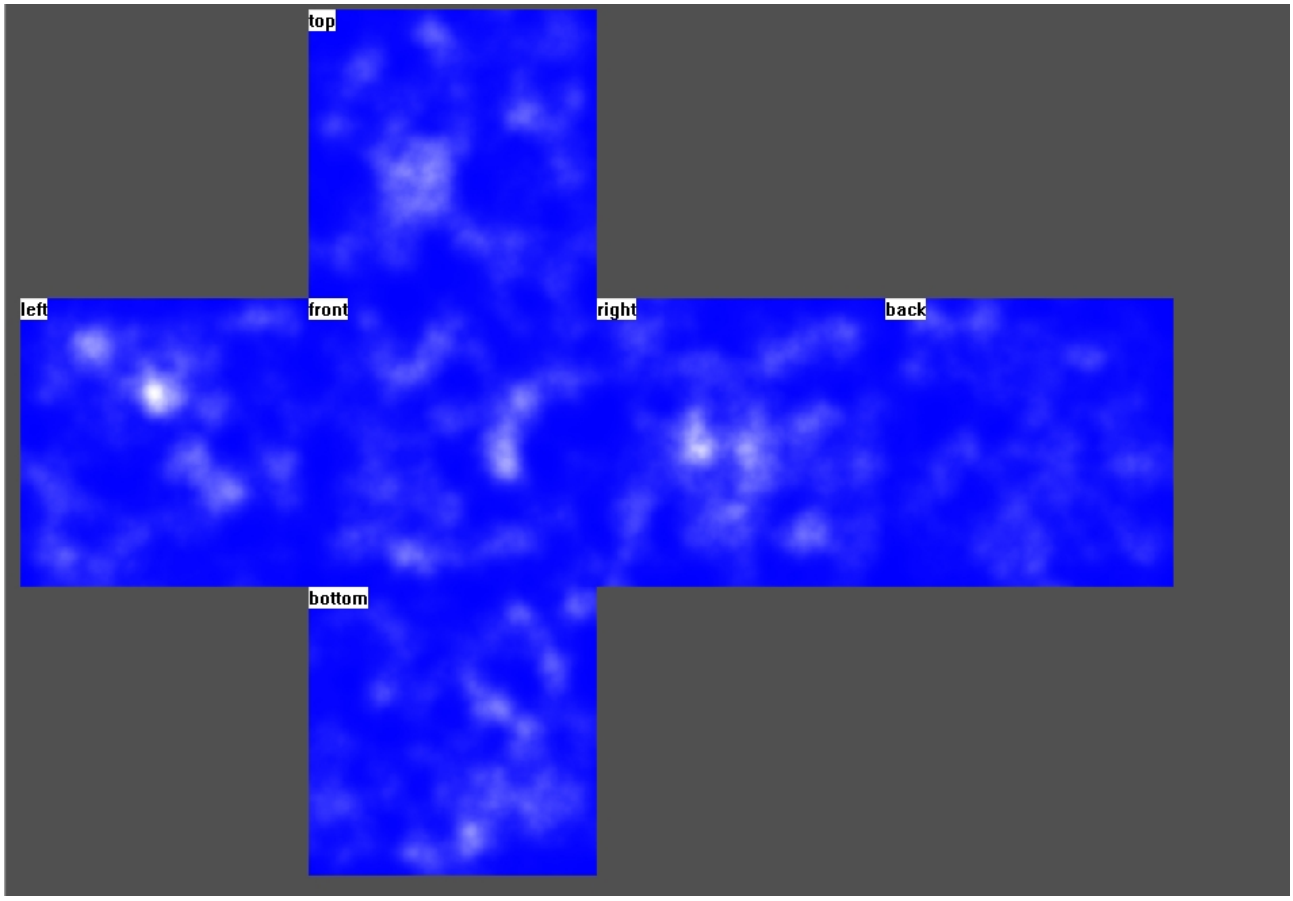
Then the pixel shader stage is used to perform the lighting interpolation during the rasterization stage of the surface fragment data. The only difference is that instead of using the constant uniform value pixel normals of the surface to interpolate the lighting, the normal values of each pixel are sampled from the provided normal map. Finally, since the normal map values of normals are in the range of 0 to 1, in order to use them for lighting interpolation, the normal values of the surface must be converted to the range of -1 to 1 (since a negative value normal is used to represent the back of the surface). The pixel shader does all of these computations in the following fragment operation stage function:

```
float4 NormalMapPS(float3 toEyeT    : TEXCOORD0,
                   float3 lightDirT : TEXCOORD1,
                   float2 tex0      : TEXCOORD2) : COLOR
{
      // Interpolated normals can become unnormal--so normalize.
      toEyeT    = normalize(toEyeT);
      lightDirT = normalize(lightDirT);

      // Light vector is opposite the direction of the light.
      float3 lightVecT = -lightDirT;

      // Sample normal map.
      float3 normalT = tex2D(NormalMapS, tex0);

      // Expand from [0, 1] compressed interval to true [-1, 1] interval.
      normalT = 2.0f*normalT - 1.0f;

      // Make it a unit vector.
      normalT = normalize(normalT);

      // Compute the reflection vector.
      float3 r = reflect(-lightVecT, normalT);

      // Determine how much (if any) specular light makes it into the eye.
      float t  = pow(max(dot(r, toEyeT), 0.0f), gMtrl.specPower);

      // Determine the diffuse light intensity that strikes the vertex.
      float s = max(dot(lightVecT, normalT), 0.0f);

      // If the diffuse light intensity is low, kill the specular lighting term.
      // It doesn't look right to add specular light when the surface receives
      // little diffuse light.
      if(s <= 0.0f)
           t = 0.0f;

      // Compute the ambient, diffuse and specular terms separatly.
      float3 spec = t*(gMtrl.spec*gLight.spec).rgb;
      float3 diffuse = s*(gMtrl.diffuse*gLight.diffuse).rgb;
      float3 ambient = gMtrl.ambient*gLight.ambient;

      // Get the texture color.
      float4 texColor = tex2D(TexS, tex0);

      // Combine the color from lighting with the texture color.
      float3 color = (ambient + diffuse)*texColor.rgb + spec;

      // Output the color and the alpha.
    return float4(color, gMtrl.diffuse.a*texColor.a);
}
```

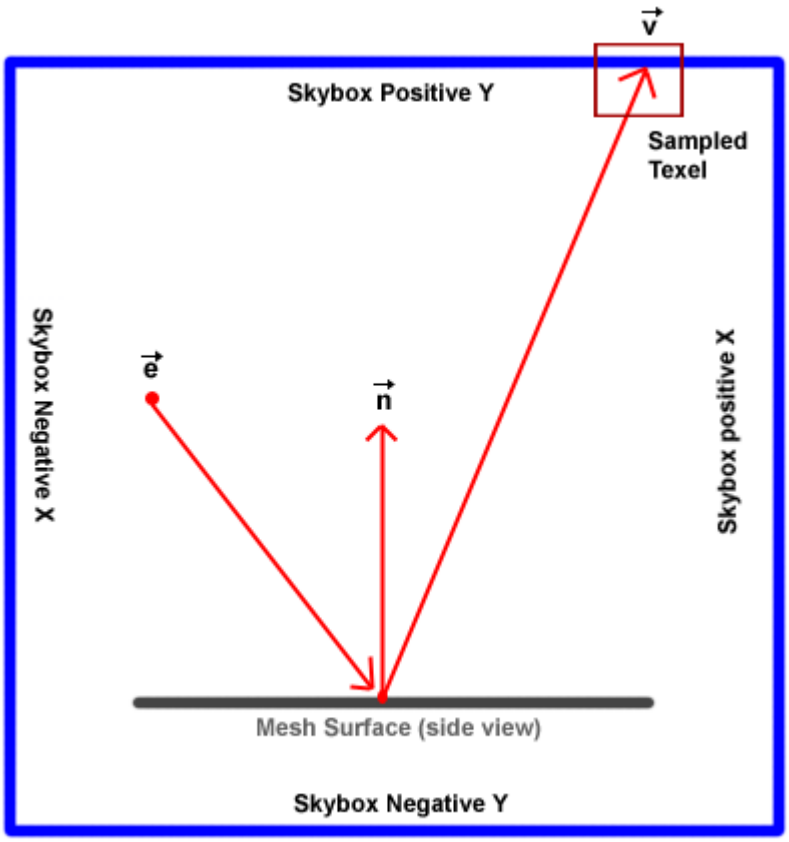
Environment Mapping

One of the other main features of the application coursework is the implementation of environment mapping. Environment mapping is the process of using texture data stored in a cube map as sampled texel data that is projected onto a given mesh surface. The projected texel data is interpolated in a way that imitates the given surface as reflecting the cube map. Since the cube map is used as the background image of the scene, the rendered surface will look as it it reflecting the surrounding scene.

A cube map is a texture format for storing 6 different textures (+-x, +-y, +-z) that are used as the background skybox or skysphere in the rendered scene. Below is an unfolded example of a simple cube map:



The approximated reflection model used with cube mapping can be illustrated with the following diagram:



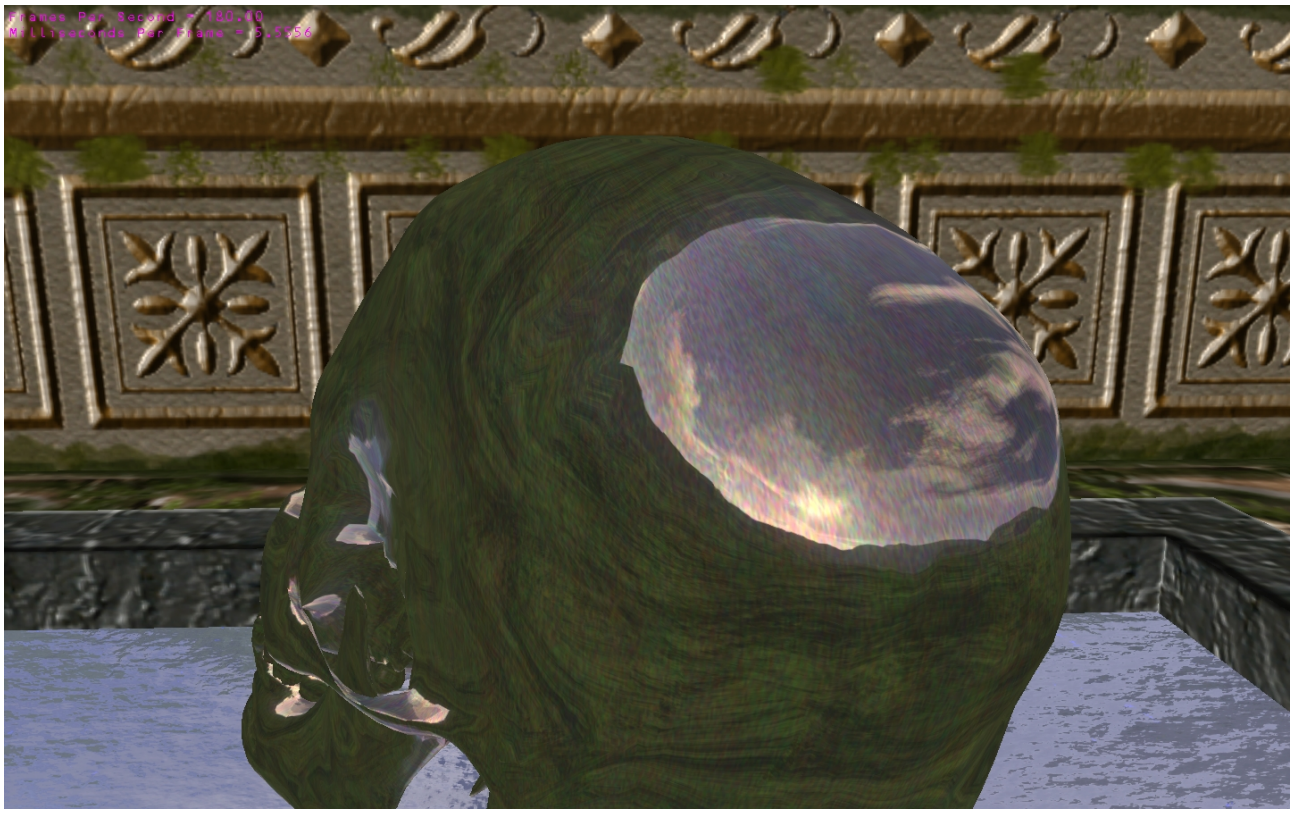
In the above diagram, the vector e is used to represent the users camera (or in theory the users eye), while the vector n is the surface unit normal vector. The texel from where the normal originates (based on the pixel normal value) is given the texture value of the sampled texel that the vector v intersects with. The sampled texel is then reflected onto the surface by taking it's texel value and copying it onto the original surface texel. The new surface texel value is then mirrored to vector e, which is the angle from where the scene camera is viewing the environment mapped surface. In the above example a skybox is used for simplicity, by likewise a primitive like a cylinder or a sphere can be used to represent the background imagery in the scene. In the coursework application, a sphere is used.

The environment mapping HLSL effect was slightly modified to include blending with an additional diffuse map. This in turn allows for the environment mapped surface to be blended with a diffuse texture. The blending of the surface is computed in the following equation:

$$\overline{F} = \alpha \overline{R} + (1 - \alpha) \overline{C}$$

The vector F represents the final RGB colour component. Alpha is used to denote the reflectivity amount applied to the reflected colour value of vector R, which is the sampled environment map texel. This is then added to the surface pixel colour (colour component vector C) that is multiplied with a blending value specified in the range of 0 to 1 (the value being calculated by subtracting alpha from 1). The colour component vector C holds the RGB values of the given surface texel. In case of the coursework application, the surface pixel colour is based on the ambient, diffuse and specular surface colour components of the given mesh (all three of which are interpolated using the Phong lighting model).

Below is a screenshot showing the diffuse texture pixel colours being blended with the environment map texels:



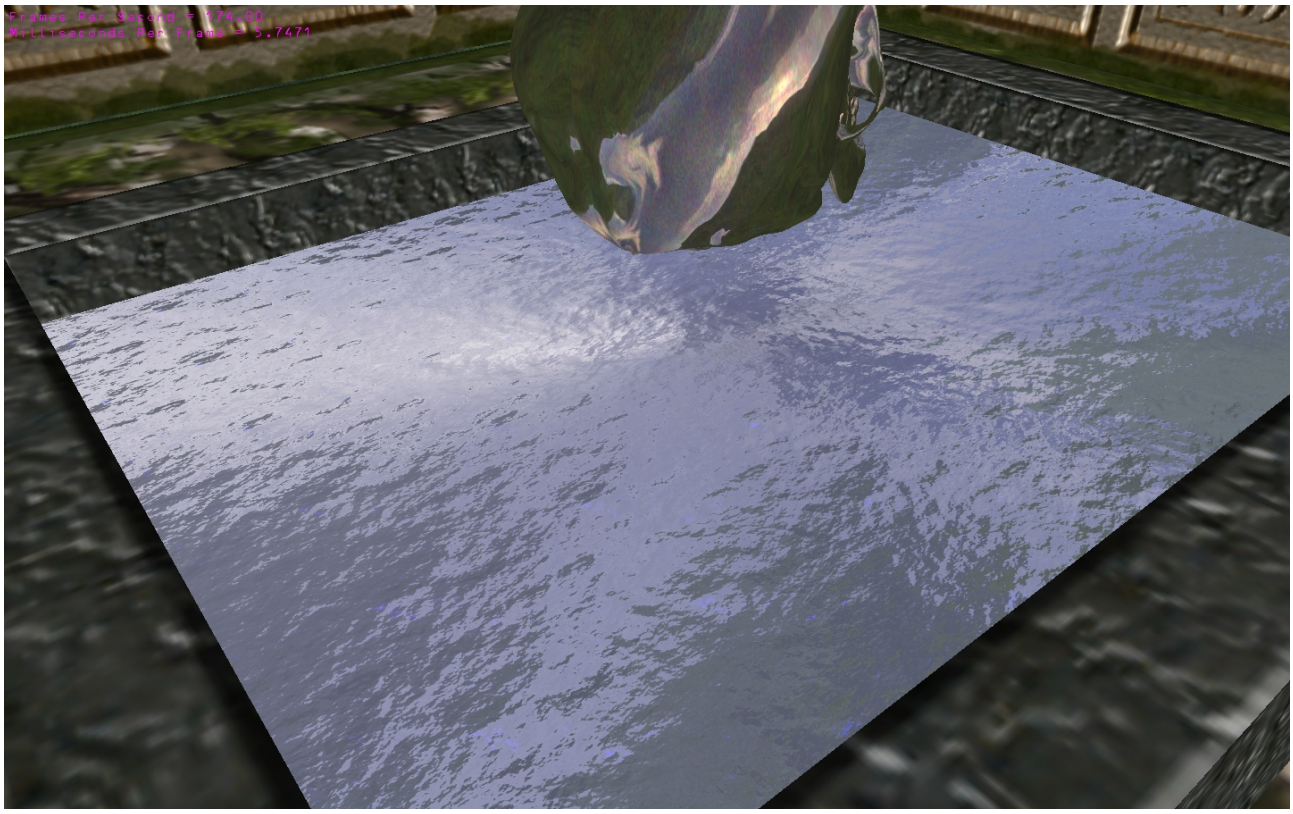
This is applied in the pixel shader operations stage in the following function (found in EnvMap.c):

```
float4 EnvMapPS(float3 normalW : TEXCOORD0,
                float3 toEyeW  : TEXCOORD1,
                float2 tex0    : TEXCOORD2) : COLOR
{
      // Interpolated normals can become unnormal--so normalize.
      normalW = normalize(normalW);
      toEyeW  = normalize(toEyeW);

      // Light vector is opposite the direction of the light.
      float3 lightVecW = gLight.dirW;

      // Determine the diffuse light intensity that strikes the vertex.
      float s = max(dot(lightVecW, normalW), 0.8f); //fixed by vlad: Increased
            diffuse lighting from 0.0f to 08.f, so it's not murky any more.

      // Get the texture color.
      float4 texColor = tex2D(TexS, tex0);

      // Get the reflected color.
      float3 envMapTex = -reflect(toEyeW, normalW); //fixed by vlad: Inversed
                  the refelction vector, now the y-axis reflections are correct
      float3 blendTex = tex2D(TexS, tex0).rgb * 1.0f; //fixed by vlad: texture
                                                        blending
      float3 reflectedColor = texCUBE(EnvMapS, envMapTex) * blendTex; //fixed by
                                                  vlad: Added blending with texture

      // Weighted average between the reflected color, and usual
      // diffuse/ambient material color modulated with the texture color.
      float3 ambientMtrl = gReflectivity*reflectedColor + (1.0f-
                           gReflectivity)*(texColor);
```

```
        float3 diffuseMtrl = gReflectivity*reflectedColor + (1.0f-
                                gReflectivity)*(texColor);

        // Compute the ambient, diffuse and specular terms separately.

        float3 diffuse = s*(diffuseMtrl);
        float3 ambient = ambientMtrl;


        float3 final = ambient*(0.5*texColor.rgb+0.5*reflectedColor) +
        diffuse*(0.5*texColor.rgb+0.5*reflectedColor);// + spec; //vlad: use spec
        if you want, but I prefer the final surface colour without it.

        // Output the color and the alpha.
    return float4( final  ,  texColor.a);
}
```

Water Effect Rendering

The final important effect to discuss in the coursework is the water effect. The water effect is in essence a combination of the normal mapping shader and the environment mapping shader, with animated normal map coordinates. Below is a screenshot showing the water rendering effect used in the coursework application:



The water shader works by taking two normal maps, and blending them over the given surface. These normal maps are used to represent the high-frequency waves of the rendered water surface. The interpolated normal mapped surface colour is then blended with the interpolated environment map surface colour. Finally, the U or V normal map texture coordinates are offset per frame in order to give the illusion of the normal map waves moving across the water surface.

The HLSL water effect implements the above methods the following way:

```
OutputVS WaterVS(float3 posL : POSITION0, float2 tex0 : TEXCOORD0)
{
    // Zero out our output.
      OutputVS outVS = (OutputVS)0;

      // Build TBN-basis.  For flat water grid in the xz-plane in
      // object space, the TBN-basis has a very simple form.
      float3x3 TBN;
      TBN[0] = float3(1.0f, 0.0f, 0.0f); // Tangent goes along object space x-
      axis.
      TBN[1] = float3(0.0f, 0.0f, -1.0f);// Binormal goes along object space -z-
      axis
      TBN[2] = float3(0.0f, 1.0f, 0.0f); // Normal goes along object space y-
      axis

      // Matrix transforms from object space to tangent space.
      float3x3 toTangentSpace = transpose(TBN);

      // Transform eye position to local space.
      float3 eyePosL = mul(float4(gEyePosW, 1.0f), gWorldInv).xyz;

      // Transform to-eye vector to tangent space.
      float3 toEyeL = eyePosL - posL;
      outVS.toEyeT = mul(toEyeL, toTangentSpace);

      // Transform light direction to tangent space.
      float3 lightDirL = mul(float4(gLight.dirW, 0.0f), gWorldInv).xyz;
      outVS.lightDirT  = mul(lightDirL, toTangentSpace);

      // Get to-eye in world space also for environment map look-up.
      outVS.toEyeW = gEyePosW - mul(float4(posL, 1.0f), gWorld);

      // Water plane lies in xz-plane, so geometric normal is just (0, 1, 0).
      outVS.normalW = mul(float4(0.0f, 1.0f, 0.0f, 0.0f), gWorld);

      // Transform to homogeneous clip space.
      outVS.posH = mul(float4(posL, 1.0f), gWVP);

      // Scroll texture coordinates.
      outVS.tex0 = tex0 + gWaveMapOffset0;
      outVS.tex1 = tex0 + gWaveMapOffset1;

      // Done--return the output.
    return outVS;
}


float4 WaterPS(float3 toEyeT    : TEXCOORD0,
               float3 lightDirT : TEXCOORD1,
               float2 tex0      : TEXCOORD2,
               float2 tex1      : TEXCOORD3,
               float3 toEyeW    : TEXCOORD4,
               float3 normalW   : TEXCOORD5) : COLOR
{
      // Interpolated normals can become unnormal--so normalize.
      // Note that toEyeW and normalW do not need to be normalized
      // because they are just used for a reflection and environment
      // map look-up and only direction matters.
      toEyeT    = normalize(toEyeT);
      lightDirT = normalize(lightDirT);

      // Light vector is opposite the direction of the light.
```

```
    float3 lightVecT = -lightDirT;

    // Sample normal map.
    float3 normalT0 = tex2D(WaveMapS0, tex0);
    float3 normalT1 = tex2D(WaveMapS1, tex1);

    // Expand from [0, 1] compressed interval to true [-1, 1] interval.
    normalT0 = 2.0f*normalT0 - 1.0f;
    normalT1 = 2.0f*normalT1 - 1.0f;

    // Average the two vectors.
    float3 normalT = normalize(0.5f*(normalT0 + normalT1));
    // Compute the reflection vector.
    float3 r = reflect(-lightVecT, normalT);

    // Determine how much (if any) specular light makes it into the eye.
    float t  = pow(max(dot(r, toEyeT), 0.0f), gMtrl.specPower);
    // Determine the diffuse light intensity that strikes the vertex.
    float s = max(dot(lightVecT, normalT), 0.5f);

    // If the diffuse light intensity is low, kill the specular lighting term.
    // It doesn't look right to add specular light when the surface receives
    // little diffuse light.
    if(s <= 0.0f) {t = 0.0f;}

    // Get the reflected color.  Also add some scale of normalT to
    // the environment map look-up vector.  This perturbs the look-up
    // vector in a different way each frame (since normalT changes),
    // which gives a ripple effect.
    float3 envMapTex = reflect(-toEyeW, normalW) + (normalT*2.0f);

    //Fixed by vlad: added transperancy to water texture
    float3 reflectedColor = texCUBE(EnvMapS, envMapTex);

    // Weighted average between the reflected color and usual diffuse
    material.
    float3 ambientMtrl = (0.5f*reflectedColor + 0.5f*gMtrl.ambient);
    float3 diffuseMtrl = (0.5f*reflectedColor + 0.5f*gMtrl.diffuse);

    // Compute the ambient, diffuse and specular terms separatly.
    float3 spec = t*(gMtrl.spec*gLight.spec).rgb;
    float3 diffuse = s*(diffuseMtrl*gLight.diffuse.rgb);
    float3 ambient = ambientMtrl*gLight.ambient;

    float3 final = (ambient + diffuse + spec);

    // Output the color and the alpha.
  return float4(final, gMtrl.diffuse.a);
}
```

Working with .X Mesh Files

The DirectX D3DX library comes with it's own set of mesh loading, processing and saving functions. The .X file format is a file format used for storing 3D data that can be displayed in a given scene. This 3D data contains information such as mesh data, material data and animation data. This allows for complex art assets to be integrated into a DirectX application that implement the D3DX mesh interface.

In the DirectX coursework application, the skull and the level models are both .X file format models. The mesh loading class that was implemented for use in the coursework application as part of the developed framework, is called the XMesh class, and is defined as:

```
class XMesh
{
public:

	//Class constructor
	XMesh(void);

	//Class destructor
	~XMesh();

	//.x file loading routine
	void LoadXFile(const std::string& filename,
                        ID3DXMesh** meshOut,
                        std::vector<Mtrl>& mtrls,
                        std::vector<IDirect3DTexture9*>& texs);

	void GenTexCoords(float texScale);

	//Load and initialize mesh
	void LoadandInitDiffMap(char * meshFile, char * meshTex,float texScale_);
	void LoadandInitEnvMap(char * meshFile, char * meshTex, char * envTex,
                       float texScale_);
	void LoadandInitNMMap(char * meshFile, char * meshTex, char * nmTex);

	//Draw mesh with diffuse texture only
	void DrawDiffuse(D3DXMATRIX* pMatrix, ID3DXEffect* pFX, D3DXHANDLE phMtrl,
                 D3DXHANDLE phTex);

	//Draw mesh with diffuse + normal map
	void DrawDiffNM(D3DXMATRIX* pMatrix, ID3DXEffect* pFX, D3DXHANDLE
                phNormalMap, D3DXHANDLE phMtrl, D3DXHANDLE phTex);

	//Draw mesh with environment map blending (diffuse + environment map)
	void DrawEnvBlnd(D3DXMATRIX* pMatrix, ID3DXEffect* pFX, D3DXHANDLE phMtrl,
                 D3DXHANDLE phTex, D3DXHANDLE phTex1);

	//Add mesh animation function (arbitrary rotations around the positive x-
axis)
	void AnimRotX(float fTime);

private:
	//private mesh class data members
	ID3DXMesh* mMesh;
	ID3DXMesh* mLevel;
	std::vector<Mtrl> mMtrl;
	std::vector<IDirect3DTexture9*> mTex;
	IDirect3DTexture9* mDefTex;
	IDirect3DTexture9* mLevelTex;
	IDirect3DTexture9* mNMTex;
	IDirect3DCubeTexture9* mEnvMap;
};
```

The XMesh class contains different methods for drawing meshes with different surface properties. These methods include:

`void DrawDiffuse()` : Draws a mesh with a diffuse texture only.
`void DrawDiffNM()` : Draws a mesh with a diffuse and a normal map.
`void DrawEnvBlnd()` : Draws a mesh with environment mapping and diffuse texture blending.

`Addtionaly, the` void GenTexCoords(float texScale) can be used automatically generate texure coordinates for meshes that do not have UV coordinates assigned to them.

The class also includes a method for simple animation of the mesh, which performs arbitrary rotations in the x-axis (and is updated once per frame):

```
void AnimRotX(float fTime)
```

As mentioned before, all of the geometry rendered by Direct3D must have specific vertex attributes that are streamed into the graphics pipeline. In case of the .X mesh file, these attributes are stored in the file and are compared to the vertex attributes in the ID3DXMesh interface buffers. The ID3DXMesh interface contains buffers that hold vertex and index attributes in order to stream the indexed vertex data through the graphics pipeline. The .X file mode format uses triangles as the base geometric primitive. The triangle geometric primitives are stored in a triangle list.

The main mesh loading function in the XMesh class is defined as:

```
void LoadXFile(const std::string& filename,
            ID3DXMesh** meshOut,
            std::vector<Mtrl>& mtrls,
            std::vector<IDirect3DTexture9*>& texs);
```

The LoadXFile() function takes in the mesh file name, the output mesh double pointer, the mesh material vector and the mesh texture vector. The meshOut double pointer variable is used for assigning the loaded mesh to a mesh instance in the XMesh class.
The reason why this method was implemented is because once the mesh is loaded, it can be optimized using ID3DXMesh mesh optimization functions. These optimization functions can be used to remove bow-ties (where two triangles share the same vertex, as this can cause problems when that vertex is modified) in the mesh, as well as to re-assign the mesh hierarchy in order to make the rendering of the mesh as fast as possible.

An important feature to mention about the .X model file format is the concept of mesh hierarchy (referred to as mesh subsets). A typical model is made up of one or more subsets (or commonly called “groups” in various 3D modelling packages). Each of these subsets is essentially an instance of the mesh with it's own copy of the mesh attributes. These attributes include things like the texture used for the mesh, the number of materials, smoothing groups and bone assignments (if animation is used).
Furthermore, each triangle in the mesh subset is given a unique ID, which is accessed by the ID3DX mesh interface and stored the attribute buffer. The size of the attribute buffer is equal to the size of the number of triangles in the mesh, and each element is stored as a DWORD variable (double word, meaning 64bits).
The vertex and index buffers are used to store the indexed vertices of each of the triangles in the mesh.

Finally, in order to draw the mesh, the following function is used:

```
myMesh->DrawSubset(subset_number);
```

The DrawSubset() function draws the specified mesh subset in mesh. If the mesh only has one subset, default value being 0, then the entire mesh is drawn.

Critical Analysis

The development of the DirectX coursework application proved be a very rewarding experience, as originally anticipated. One of the most important aspects of the development of the coursework application was the exposure the programmable graphics pipeline. This allowed for the development of the advanced effects featured in the coursework.

However, there were certain problems that were faced and solved to varying degrees of success, during the development of the DirectX framework. The first problem encountered in the development of the framework stemmed from the fact that this was the first time DirectX was used for3D graphics programming in this course. Unlike OpenGL, which has a state machine based architecture, DirectX is COM based and therefore heavily relies of object orientated programming principals and techniques.

There were certain issues in learning how to use the DirectX API properly, specifically Direct3D, and at the same time developing a useful framework. One of these issues was the lack of a solid framework design.
The framework that was developed thus suffers from certain design flaws, such as the fact that there are several memory leaks present in the application, which due to time constraints, weren't fixed. Also, the framework is based heavily on the examples provided by the author Frank D. Luna, from his book "*Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach*".

The framework does what it needs to do, and it is functional, but is not ideal and could have been done better. Many of the code examples presented in the book were modified and incorporated into the framework. All of these code examples were developed in such a way as to make them easy to understand, but not necessarily fast. Therefore, in the near future, faster and more sophisticated graphics programming techniques will be investigated, and a new and better framework will be developed.

Finally, due to time constraints and getting to grips with DirectX, the implementation of key-frame based skinned mesh animation was dropped. There were early attempts to incorporate this feature into the framework, but these attempts did not produce the desired results. Commitment to other projects prevented further investigation into properly implementing key-frame based skinned mesh animation. Though, in the near future, this feature will definitely be investigated and implemented.

Conclusion

In conclusion, having now obtained experience in both fixed function and programmable pipeline programming (via OpenGL and Direct3D), it is evident that Direct3D is a valid and sophisticated graphics programming API. The developed coursework application is able to demonstrate advanced graphics rendering techniques found in the majority of today's video games and interactive applications, made possible with the use of the Direct3D.

With further advancements in the Direct3D API every few months, it is clear that Direct3D programming experience is a useful credential to have for anyone who is interested in pursuing 3D graphics programming professionally.

References

Luna, D. F. 2006. *Introduction to 3D Game Programming With DirectX 9.0c: A Shader Approach*. Wordware Publishing, Inc.

Van Verth, M. J. Bishop, M. L. 2008. *Essential Mathematics for Games & Interactive Applications: A Programmer's Guide, Second Edition.* Morgan Kaufmann.

Akenine-Moller, T. Haines, E. Hoffman, N. 2008. *Real-Time Rendering - Third Edition*. A K Peters, Ltd.

Hearn, D. Baker , P. M. 1994. *Computer Graphics – Second Edition*. Prentice Hall.

Fernando, R. Kilgard, J. M. 2003. *The Cg Tutorial: The Definitive Guide to Programmable Real-time Graphics.* Addison Wesley.

Microsoft. 2010. *DirectX API Documentation.* Available from: http://msdn.microsoft.com/en-us/directx/default [Accessed 08 January 2011].