SHADERS

Durante los próximos días aprenderemos todo lo necesario para trabajar con iluminación dinámica y efectos avanzados mediante shaders en DirectX.

LightManager

Como siempre realizaremos un mánager que nos permitirá gestionar las diferentes luces que tendremos dentro de nuestro juego. El Mánager de luces contendrá tres tipos de luces diferentes:

- *Omni*, es un tipo de luz que ilumina en todas direcciones
- *Direccional*, un tipo de luz que ilumina en una única dirección
- *Spot*, un tipo de luz que ilumina en una dirección según un ángulo máximo

La clase la definimos de forma similar al siguiente código:

```
class CLightManager : public CTemplatedMapManager<CLight>
{
	public:
		CLightManager();
		~CLightManager();
		void Load(const std::string &FileName);
		void Render(CRenderManager *RenderManager);
};
```

Lights

Para controlar las luces crearemos una clase base de tipo CLight que contendrá una estructura como la siguiente:

```
class CLight : public CObject3D
{
	public:
		enum TLightType
		{
			OMNI=0,
			DIRECTIONAL,
			SPOT
		};
	protected:
		CColor		m_Color;
		TLightType	m_Type;
		std::string	m_Name;
		bool		m_RenderShadows;
		float		m_StartRangeAttenuation;
		float		m_EndRangeAttenuation;

		static TLightType GetLightTypeByName(const std::string &StrLightType);
	public:
		CLight();
		virtual ~CLight();

		void SetName(const std::string &Name);
		const std::string &GetName();
		void SetColor(const CColor &Color);
		const CColor & GetColor() const;
```

```
        void SetStartRangeAttenuation(const float StartRangeAttenuation);
        float GetStartRangeAttenuation() const;
        void SetEndRangeAttenuation(const float EndRangeAttenuation);
        float GetEndRangeAttenuation() const;
        bool RenderShadows() const;

        void SetType(const TLightType Type);
        TLightType GetType() const;

        virtual void Render(CRenderManager *RM);
};
```

Destacar los siguientes atributos/métodos:

- name: contendrá el nombre de la luz
- color: contendrá el color de la luz
- render shadows: nos dirá si esta luz proyectará sombras
- start range attentuation: nos dirá el rango de comienzo de la atenuación de la luz
- end range attentuation: nos dirá el fin del rango de la atenuación de la luz
- type: contendrá el tipo de luz OMNI, DIRECTIONAL, SPOT

Una vez creada la clase base pasaremos a crear una clase por cada una de los diferentes tipos de luces que tendremos en el juego:

```
class COmniLight : public CLight
{
public:
        COmniLight() : CLight() {}
};
```

La clase COmniLight a priori no introducirá ninguna propiedad nueva respecto a la clase Clight.

```
class CDirectionalLight : public CLight
{
protected:
        CPoint3D                                        m_Direction;
public:
        CDirectionalLight() : CLight() {}
        void SetDirection(const CPoint3D &Direction);
        CPoint3D GetDirection() const;
        virtual void Render(CRenderManager *RM);
};
```

La clase CDirectionalLight implementará la propiedad dirección de la luz respecto a la clase CLight.

```
class CSpotLight : public CDirectionalLight
{
        protected:
                float                                   m_Angle;
                float                                   m_FallOff;
        public:
                CSpotLight();
                void SetAngle(float Angle);
                float GetAngle() const;
                void SetFallOff(const float FallOff);
```

```
        float GetFallOff() const;
};
```

Por último la clase CSpotLight introduce las siguientes propiedades respecto a la clase CDirectionalLight de la que heredará:

- angle: dirá la propiedad del ángulo del foco de luz

Por último la clase CSpotLight introduce las siguientes propiedades respecto a la clase CDirectionalLight de la que heredará:

- angle: dirá la propiedad del ángulo del foco de luz
- falloff: dirá la propiedad del ángulo final del foco de luz

Luces en MAXScript

Para generar el fichero xml dónde estableceremos las luces de nuestro juego realizaremos un script de 3D Studio MAX.

Para realizar dicho script deberemos comprender los diferentes tipos de luces que utiliza el 3D Studio MAX y exportar sus propiedades.

Para exportar los tres diferentes tipos de luces utilizaremos las luces de tipo *Omni, Target Direct y Target Spot* que nos encontramos dentro del sub-menú de luces Standard.

De cada una de las luces exportaremos la posición y el color. Utilizando el siguiente fragmento de código MAXScript.

```
-- para exportar la posición de la luz
$.pos

-- para exportar el color aplicado en la luz
$.rgb

-- para exportar la atenuación en la luz
$.farAttenStart
$.farAttenEnd
```

Para exportar la dirección de la luz tanto si es de tipo Target Direct o Target Spot utilizaremos el siguiente código.

```
-- para exportar la dirección de la luz
local dir=($.target.pos-$.pos)
```

Para exportar el ángulo de la luz en caso de ser Target Spot utilizaremos el siguiente código.

```
-- para saber el ángulo y del falloff en grados del spot light
$.hotspot
$.Falloff
```

Un fichero en formato xml de la exportación de las luces sería similar al siguiente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<lights>
        <light name="Light01" type="spot" pos="58.335 6.15266 85.1666" dir="-0.758065 -
                0.193092 -0.622939" color="0.223529 0.788235 1.0" angle="0.226893"
                fall_off="0.977384" att_start_range="28.0" att_end_range="40.0"/>
        <light name="Light02" type="directional" pos="-31.3958 35.6993 -74.0128" dir="0.555733 -
                0.350478 0.753874" color="1.0 1.0 1.0" render_shadows="true"
                att_start_range="8000.0" att_end_range="20000.0"/>
        <light name=" Light03" type="omni" pos="51.2215 0.0 38.0957" color="1.0 1.0 1.0"
                att_start_range="80.0" att_end_range="200.0"/>
        <light name="Light04" type="spot" pos="-20.0064 20.3447 29.1889" dir="-0.573021 -
                0.490473 0.656569" color="1.0 1.0 1.0" angle="0.436332" fall_off="1.0472"
                att_start_range="8.0" att_end_range="10.0"/>
</lights>
```

¿Qué es un shader?

Según la wikipedia:

Shader

"La tecnología **shaders** es cualquier unidad escrita en un lenguaje de sombreado que se puede compilar independientemente. Es una tecnología reciente y que ha experimentado una gran evolución destinada a proporcionar al programador una interacción con la GPU hasta ahora imposible. Los shaders son utilizados para realizar transformaciones y crear efectos especiales, como por ejemplo iluminación, fuego o niebla. Para su programación los shaders utilizan lenguajes específicos de alto nivel que permitan la independencia del hardware.

Historia

En el 2000 la serie 2 de tarjetas GeForce permitía a la GPU hacerse cargo de funciones de transformación e iluminación que hasta ahora debía hacerlas la CPU, sin embargo no fue hasta la GeForce 3 (2001) que se incluyó la posibilidad de programarlas con la primera versión del modelo de sombreado. Existen numerosas versiones, se debe tener en cuenta que cuanto más reciente es la versión más limita el número de tarjetas gráficas sobre las que el programa puede operar correctamente.

Lenguajes de sombreado

Para la escritura de esas instrucciones, los programadores hacen uso de lenguajes de programación diseñados específicamente para ello. Cada uno de estos lenguajes de programación necesita enlazarse mediante una API, entre otras DirectX u OpenGL. Existen otros lenguajes pero los siguientes son los más conocidos.

- HLSL es la implementación propiedad de Microsoft, la cual colaboró junto a Nvidia para crear un lenguaje de sombreado. Este lenguaje se debe utilizar junto a DirectX (la primera versión para la que se puede utilizar es DirectX 8.0). Anteriormente al DirectX 8 (DirectX 7, 6, 5...) se utilizaba otro método el cual era más complicado y complejo para ser utilizado. (Entre lo que era el lenguaje, creación de objetos, sonidos, partículas, entre otras).
- GLSL es el lenguaje desarrollado por el grupo Khronos. Está diseñado específicamente para su uso dentro del entorno de OpenGL. Sus diseñadores afirman que se ha hecho un gran esfuerzo para lograr altos niveles de paralelismo. Su diseño se basa en C y RenderMan como modelo de lenguaje de sombreado.

- CG lenguaje propiedad de la empresa Nvidia resultante de su colaboración con Microsoft para el desarrollo de un lenguaje de sombreado. Su principal ventaja es

que puede ser usado por las APIs OpenGL y DirectX. Otra ventaja de este lenguaje es el uso de perfiles. Estos lenguajes no son totalmente independientes del hardware por lo tanto es recomendable crear programas específicos para diferentes tarjetas gráficas. Los perfiles de CG se encargan de elegir para su ejecución el más adecuado de los programas disponibles para el hardware.

Tipos de procesadores shader

A continuación se presentan los diferentes tipos de procesadores shader que la GPU tiene, las cantidades de cada uno crecen con celeridad entre generaciones de gráficas.

Los shaders trabajan de la siguiente manera.

- El programador envía un conjunto de vértices que forman su escena gráfica a través de un lenguaje de propósito general.
- Todos los vértices pasan por el vertex shader donde pueden ser transformados y se determina su posición final.
- El siguiente paso es el geometry shader dónde se pueden eliminar o añadir vértices.
- Posteriormente los vértices son ensamblados formando primitivas que son rasterizadas, proceso en el cual las superficies se dividen en puntos que corresponden a píxeles de la pantalla.
- El Píxel/Fragment shader se encarga de modificar estos puntos. Por último se producen cierto tests, entre ellos el de profundidad que determina que punto es dibujado en pantalla.

Los tipos de shaders que nos encontramos son los siguientes:

- Vertex shader: Permite transformaciones sobre coordenadas, normal, color, textura, etc. de un vértice. No puede saberse el orden entre vértices ni pasarse información entre ellos (esto ocurre también en el resto de tipos).
- Geometry Shader: Es capaz de generar nuevas primitivas dinámicamente así como de modificar existentes. Un ejemplo claro es la decisión de utilizar o eliminar vértices en una malla polígonal según la posición del observador aplicando la técnica nivel de detalle.
- Píxel/Fragment shader: En primer lugar aclarar la difererencia entre fragmento y píxel. Desde la Khronos group se apuesta por diferenciar que fragmento es lo que se procesa puesto que existen múltiples relacionados con un mismo píxel de la pantalla. La terminología seguida por Microsoft y Nvidia es la de píxel que puede dar lugar a confusión ya que no se trabaja con lo de la pantalla sino con los de cada figura. En este procesador se pueden hacer diversas transformaciones como cambiar la profundidad o trabajar con texels así como calcular efectos de iluminación con gran precisión. Todo lo ejecutado debe determinar el color que debería aplicarse sobre el píxel en caso de ser usado. También es útil para modificar la profundidad."

EffectManager

Como siempre realizaremos un mánager de efectos dónde tendremos almacenados mediante la clase CTemplatedMapManager los diferentes efectos que cargaremos mediante un fichero .xml.

La declaración de la clase sería similar a la siguiente:

```
class CEffectManager : public CTemplatedMapManager<CEffectTechnique>
{
        private:
                typedef std::map<int,std::string>          TDefaultTechniqueEffectMap;
                TDefaultTechniqueEffectMap                 m_DefaultTechniqueEffectMap;
                CMatrix                    m_WorldMatrix, m_ProjectionMatrix, m_ViewMatrix,
                                           m_ViewProjectionMatrix;
                CMatrix                    m_LightViewMatrix, m_ShadowProjectionMatrix;
                CPoint3D                   m_CameraEye;
                CTemplatedMapManager<CEffect>      m_Effects;
                CEffectTechnique                   *m_StaticMeshTechnique;
                CEffectTechnique                   *m_AnimatedModelTechnique;
        public:
                CEffectManager();
                ~CEffectManager();
                const CMatrix & GetWorldMatrix() const;
                const CMatrix & GetProjectionMatrix() const;
                const CMatrix & GetViewMatrix() const;
                const CMatrix & GetViewProjectionMatrix();
                const CPoint3D & GetCameraEye();
                const CMatrix & GetLightViewMatrix() const;
                const CMatrix & GetShadowProjectionMatrix();

                void ActivateCamera(const CMatrix &ViewMatrix, const CMatrix &ProjectionMatrix,
                                const CPoint3D &CameraEye);
                void SetWorldMatrix(const CMatrix &Matrix);
                void SetProjectionMatrix(const CMatrix &Matrix);
                void SetViewMatrix(const CMatrix &Matrix);
                void SetViewProjectionMatrix(const CMatrix &ViewProjectionMatrix);
                void SetLightViewMatrix(const CMatrix &Matrix);
                void SetShadowProjectionMatrix(const CMatrix &Matrix);
                void SetCameraEye(const CPoint3D &CameraEye);

                void Load(const std::string &FileName);
                void Reload();

                std::string GetTechniqueEffectNameByVertexDefault(unsigned short VertexType);
                size_t GetMaxLights() const;

                CEffect * GetEffect(const std::string &Name);
                CEffectTechnique * GetEffectTechnique(const std::string &Name);

                CEffectTechnique * GetStaticMeshTechnique() const;
                void SetStaticMeshTechnique(CEffectTechnique *StaticMeshTechnique);
                CEffectTechnique * GetAnimatedModelTechnique() const;
                void SetAnimatedModelTechnique(CEffectTechnique *AnimatedModelTechnique);

                void CleanUp();
};
```

Destacaremos los siguientes atributos de la clase:

- *m_WorldMatrix*, estableceremos la matriz de world en el effectmanager y la almacenaremos en esta variable.

- *m_ProjectionMatrix*, estableceremos la matriz de projection en el effectmanager y la almacenaremos en esta variable.
- *m_ViewMatrix*, estableceremos la matriz de view en el effectmanager y la almacenaremos en esta variable.
- *m_ViewProjectionMatrix*, estableceremos la matriz de viewprojection en el effectmanager y la almacenaremos en esta variable.
- *m_LightViewMatrix, matriz de view creado a partir de las propiedades de luz que utilizaremos para generar el efecto de shadowmap.*
- *m_ShadowProjectionMatrix,matriz de proyección creado a partir de las propiedades de la luz que utilizaremos para generar el efecto de shadowmap.*
- *m_CameraEye*, estableceremos la posición del ojoe de la cámara en el effectmanager y la almacenaremos en esta variable.
- *m_DefaultEffectTechniqueMap*, en este mapa guardaremos las techniques por defecto para un determinado tipo de vértice.
- m_Effects, mediante este mapa guardaremos los diferentes archivos de efectos que carguemos.
- m_StaticMeshTechnique, en este parámetro guardaremos la technique que queramos utilizar al renderizar las mallas estáticas, en caso de que contenga un valor de NULL se utilizará la technique por defecto de cada malla.
- m_AnimatedModelTechnique, en este parámetro guardaremos la technique que queramos utilizar al renderizar los modelos animadas, en caso de que contenga un valor de NULL se utilizará la technique por defecto de cada modelo.

Como siempre cargaremos los effects a partir de un fichero xml similar al siguiente.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<effects>
        <default_technique vertex_type="19" technique="NormalTextureVertexTechnique"/>

        <effect name="DefaultEffect" file="data/effects/ DefaultEffect.fx"/>
        <effect name="Cal3DEffect" file="data/effects/cal3d.fx"/>

        <technique name=" NormalTextureVertexTechnique " effect=" DefaultEffect "
                use_world_matrix="true" use_world_view_projection_matrix="true" use_lights="true"
                num_of_lights="4" use_camera_position="true" use_projection_matrix="true"
                use_world_view_matrix="true" use_view_to_light_projection_matrix="true"/>
</effects>
```

Effect

Como siempre crearemos una clase para encapsular la clase LPD3DXEFFECT de DirectX, para ello crearemos una clase similar a la siguiente.

```
class CEffect
{
	private:
		std::string			m_FileName;
		LPD3DXEFFECT		m_Effect;
		BOOL		m_LightsEnabled[MAX_LIGHTS_BY_SHADER];
		int		m_LightsType[MAX_LIGHTS_BY_SHADER];
		float		m_LightsAngle[MAX_LIGHTS_BY_SHADER];
		float		m_LightsFallOff[MAX_LIGHTS_BY_SHADER];
		float		m_LightsStartRangeAttenuation[MAX_LIGHTS_BY_SHADER];
		float		m_LightsEndRangeAttenuation[MAX_LIGHTS_BY_SHADER];
		CPoint3D	m_LightsPosition[MAX_LIGHTS_BY_SHADER];
		CPoint3D	m_LightsDirection[MAX_LIGHTS_BY_SHADER];
		CPoint3D	m_LightsColor[MAX_LIGHTS_BY_SHADER];

		D3DXHANDLE m_WorldMatrixParameter, m_ViewMatrixParameter,
			m_ProjectionMatrixParameter;
		D3DXHANDLE m_WorldViewMatrixParameter,
			m_ViewProjectionMatrixParameter,
			m_WorldViewProjectionMatrixParameter;
		D3DXHANDLE m_ViewToLightProjectionMatrixParameter;
		D3DXHANDLE m_LightEnabledParameter, m_LightsTypeParameter,
			m_LightsPositionParameter, m_LightsDirectionParameter,
			m_LightsAngleParameter, m_LightsColorParameter;
		D3DXHANDLE m_LightsFallOffParameter,
			m_LightsStartRangeAttenuationParameter,
			m_LightsEndRangeAttenuationParameter;
		D3DXHANDLE m_CameraPositionParameter;
		D3DXHANDLE m_BonesParameter;
		D3DXHANDLE m_TimeParameter;

		void SetNullParameters();
		void GetParameterBySemantic(const std::string &SemanticName, D3DXHANDLE
			&l_Handle);
		bool LoadEffect();
		void Unload();
	public:
		CEffect();
		~CEffect();

		bool SetLights(size_t NumOfLights);
		bool Load(const std::string &FileName);
		bool Reload();
		//DirectX Methods Interface
		LPD3DXEFFECT GetD3DEffect() const;
		D3DXHANDLE GetTechniqueByName(const std::string &TechniqueName);
};
```

Destacaremos las siguientes propiedades/métodos de la clase CEffect.

- *Variables D3DXHANDLE*, estas variables contendrán los handles de las variables que encontraremos dentro del efecto cargado.

- *SetLight*, establece las propiedades de las luces según el parámetro del número de luces.

- *GetD3DEffect*, devuelve el handle de DirectX del efecto.

- *GetTechniqueByName*, devuelve el handle de DirectX de la technique.

Technique

En este caso crearemos una clase para encapsular la información que utilizaremos para cada una de las techniques de nuestro effect. Su código será similar al siguiente.

```
class CEffectTechnique
{
	private:
		bool		m_UseCameraPosition;
		bool		m_UseInverseProjMatrix;
		bool		m_UseInverseViewMatrix;
		bool		m_UseInverseWorldMatrix;
		bool		m_UseLights;
		int		m_NumOfLights;
		bool		m_UseLightAmbientColor;
		bool		m_UseProjMatrix;
		bool		m_UseViewMatrix;
		bool		m_UseWorldMatrix;
		bool		m_UseWorldViewMatrix;
		bool		m_UseWorldViewProjectionMatrix;
		bool		m_UseViewProjectionMatrix;
		bool		m_UseViewToLightProjectionMatrix;
		bool		m_UseTime;
		CEffect		*m_Effect;
		D3DXHANDLE	m_D3DTechnique;
		std::string	m_TechniqueName;
	public:
		CEffectTechnique ();
		~ CEffectTechnique ();

		inline CEffect * GetEffect() const {return m_Effect;}
		bool BeginRender();
		bool Refresh();
		//DirectX Methods Interface
		D3DXHANDLE GetD3DTechnique();
};
```

Destacaremos las siguientes propiedades de la clase CEffect.

- *Variables booleanas*, estas variables booleanas determinarán si utilizamos o no un determinado parámetro en el shader.

- *BeginRender*, preparará las variables del effect según las variables booleanas.

- *Refresh*, recogerá la technique del efecto según el nombre de esta.

HLSL

Para la implementación de los shaders vamos a utilizar el lenguaje HLSL que como hemos dicho previamente es un lenguaje creado por Microsoft para la plataforma DirectX.

Su sintaxis es bastante similar a la de C con nuevos tipos y funciones determinados para la programación matemática.

```
// Para declarar una variable
bool g_VarName : LIGHTSENABLED = false;
// En este caso hemos declarado una variable de tipo booleana con nombre g_VarName con valor falso y le hemos definido una semántica que nos va a permitir después ser buscada en el effect a través de esta semántica

//Para declarer una variable de tipo Matriz 4x4 y con semántica WORLD
float4x4 g_WorldMatrix : WORLD;

//Para declarer una variable de tipo Vector 3 float con semántica CAMERAPOSITION
float3 g_CameraPosition : CAMERAPOSITION;


// Declarar una etapa de textura en la etapa s0 con los filtros activados de tipo lineal
sampler DiffuseTextureSampler : register( s0 ) = sampler_state
{
   MipFilter = LINEAR;
   MinFilter = LINEAR;
   MagFilter = LINEAR;
};

// Para declarar una variable
bool g_VarName : LIGHTSENABLED = false;

//Para declarer una estructura de vértice que utilizaremos como entrada para el Vertex Shader
struct TNORMAL_TEXTURED_VERTEX_VS {
        float3 Position   : POSITION;
        float3 Normal     : NORMAL;
        float4 UV         : TEXCOORD0;
};

//Para declarar una estructura de vértice que utilizaremos como salida para el Vertex Shader y entrada para el Píxel Shader
struct TNORMAL_TEXTURED_VERTEX_PS {
        float4 Hposition          : POSITION;
        float2 UV                 : TEXCOORD0;
        float3 WorldNormal        : TEXCOORD1;
        float3 WorldPosition      : TEXCOORD2;
};

//Implementación de un vertex shader simple
TNORMAL_TEXTURED_VERTEX_PS
NormalTexturedVertexVS(TNORMAL_TEXTURED_VERTEX_VS IN)
{
        TNORMAL_TEXTURED_VERTEX_PS OUT = (TNORMAL_TEXTURED_VERTEX_PS)0;
        OUT.WorldNormal = mul(IN.Normal,(float3x3)g_WorldMatrix);
        OUT.UV = IN.UV.xy;
        OUT.HPosition = mul(float4(IN.Position,1.0),g_WorldViewProjMatrix);
        OUT.WorldPosition=mul(float4(IN.Position,1.0),g_WorldMatrix).xyz;

        return OUT;
}
```

```
//Implementación de un pixel shader simple
float4 NormalTexturedVertexPS(TNORMAL_TEXTURED_VERTEX_PS IN) : COLOR
{
	// Extrae el color del pixel de la textura según las coordenadas de textura IN.UV
	return tex2D(DiffuseTextureSampler,IN.UV);
}

//Por último para implementar una technique para poder ser utilizada desde nuestro código  C
technique NormalTextureVertexTechnique
{
	//Pasada que vamos a implementar
	pass p0
	{
		//Activamos el Zbuffer, el Zwrite y la función de Z's que queremos utilizar
		ZEnable = true;
		ZWriteEnable = true;
		ZFunc = LessEqual;
		//Deshabilitamos el alphablend
		AlphaBlendEnable = false;
		//Tipo de culling que queremos utilizar
		CullMode = CCW;
		//Compilamos el Vertex Shader  y el Pixel Shader con su versión correspondiente
		VertexShader = compile vs_2_0 NormalTexturedVertexVS();
		PixelShader = compile ps_2_0 NormalTexturedVertexPS();
	}
}
```

Vertex Declaration

Igual que hicimos en el VertexType definiendo un método para decirle al DirectX el tipo de vértice que vamos a utilizar, crearemos un método para cada uno de los diferentes tipos de vértices que tengamos, que nos devolverá un vertex declaration que será la forma en como le diremos al shader el tipo de vértice que utilizaremos.

Para implementar dicho método crearemos una variable estática de tipo LPDIRECT3DVERTEXDECLARATION9 que será utilizada por todos los vértices de ese mismo tipo.

Por ejemplo:

```
// Para un vértice de tipo normal + coordenada de textura + geometría
struct TNORMAL_TEXTURED_VERTEX
{
	float                                   x, y, z;
	float                                   nx, ny, nz;
	float                                   tu, tv;
	static LPDIRECT3DVERTEXDECLARATION9                    s_VertexDeclaration;
	static LPDIRECT3DVERTEXDECLARATION9 & GetVertexDeclaration();
	static void ReleaseVertexDeclaration()
	{
		CHECKED_RELEASE(s_VertexDeclaration);
	}
};

LPDIRECT3DVERTEXDECLARATION9 & GetVertexDeclaration()
{
```

```
        if(s_VertexDeclaration==NULL)
        {
                D3DVERTEXELEMENT9 l_VertexDeclaration[] =
                {
                        { 0,  0  ,  D3DDECLTYPE_FLOAT3,  D3DDECLMETHOD_DEFAULT,
                          D3DDECLUSAGE_POSITION, 0 },
                        { 0,  12,   D3DDECLTYPE_FLOAT3,  D3DDECLMETHOD_DEFAULT,
                          D3DDECLUSAGE_NORMAL, 0 },
                        { 0,  24,   D3DDECLTYPE_FLOAT2,  D3DDECLMETHOD_DEFAULT,
                          D3DDECLUSAGE_TEXCOORD, 0 },
                        { 0,  32,   D3DDECLTYPE_FLOAT2,  D3DDECLMETHOD_DEFAULT,
                          D3DDECLUSAGE_TEXCOORD, 1 },
                        D3DDECL_END()
                };
                CRenderManager.GetDevice()->CreateVertexDeclaration(l_VertexDeclaration,
                        &s_VertexDeclaration);
        }
        return s_VertexDeclaration;

}
```

Uso de un CEffect

Para la carga de un efecto utilizaremos el método de DirectX D3DXCreateEffectFromFile con un código similar al siguiente.

```
LPD3DXBUFFER l_ErrorBuffer=NULL;
HRESULT  l_HR=D3DXCreateEffectFromFile(PSRender.GetDevice(),  l_FileName.c_str(),  NULL,
      NULL, D3DXSHADER_USE_LEGACY_D3DX9_31_DLL, NULL, &m_Effect, &l_ErrorBuffer);
if(l_ErrorBuffer)
{
        Info("Error creating effect '%s':\n%s", l_FileName.c_str(), l_ErrorBuffer-
              >GetBufferPointer());
        CHECKED_RELEASE(l_ErrorBuffer);
        return false;
}
```

Para encontrar una variable HLSL dentro del effect utilizaremos el método de DirectX GetParameterBySemantic, el cual nos devolverá un D3DXHANDLE con la dirección de memoria dónde se encuentra nuestra variable en el shader.

El código sería similar al siguiente.

```
l_Handle=m_Effect->GetParameterBySemantic(NULL,SemanticName.c_str());
if(l_Handle==NULL)
        Info("Parameter by semantic '%s' wasn't found on effect '%s'", SemanticName.c_str(),
              m_FileName.c_str());
```

Para modificar los valores de un effect en hlsl podemos utilizar los métodos de DirectX según el tipo de la variable que queramos modificar.

Los métodos a utilizar serían los siguientes.

```
HRESULT SetBool(D3DXHANDLE hParameter, BOOL b);
HRESULT SetBoolArray(D3DXHANDLE hParameter, CONST BOOL* pB, UINT Count);
HRESULT SetInt(D3DXHANDLE hParameter, INT n);
HRESULT SetIntArray(D3DXHANDLE hParameter, CONST INT* pn, UINT Count);
HRESULT SetFloat(D3DXHANDLE hParameter, FLOAT f);
HRESULT SetFloatArray(D3DXHANDLE hParameter, CONST FLOAT* pf, UINT Count);
HRESULT SetMatrix(D3DXHANDLE hParameter, CONST D3DXMATRIX* pMatrix);
```

```
HRESULT SetMatrixArray(D3DXHANDLE hParameter, CONST D3DXMATRIX* pMatrix, UINT
        Count);
```

Para encontrar una technique en HLSL dentro del effect utilizaremos el método de DirectX GetTechniqueByName, el cual nos devolverá un D3DXHANDLE con la dirección de memoria dónde se encuentra nuestra technique en el shader.

El código sería similar al siguiente.

```
D3DXHANDLE l_EffectTechnique=m_Effect->GetTechniqueByName(
        TechniqueName.c_str());
```

Por último para renderizar una malla deberemos utilizar un código similar al siguiente.

```
bool Render(CEffectTechnique *EffectTechnique) const
{
        LPDIRECT3DDEVICE9 l_Device=RenderManager.GetDevice();
        UINT l_NumPasses;
        LPD3DXEFFECT l_Effect=EffectTechnique->GetEffect()->GetD3DEffect();
        l_Effect->SetTechnique(EffectTechnique->GetD3DTechnique());
        if(SUCCEEDED(Effect->Begin(&l_NumPasses,0)))
        {
                l_Device->SetVertexDeclaration(T::GetVertexDeclaration());
                l_Device->SetStreamSource(0,m_VB,0,sizeof(T));
                l_Device->SetIndices(m_IB);
                for (UINT b=0;b<l_NumPasses;++b)
                {
                        l_Effect->BeginPass(b);
                        l_Device->DrawIndexedPrimitive(  D3DPT_TRIANGLELIST,  0,  0,
                            (UINT)m_VertexCount, 0, (UINT) m_IndexCount/3);
                        l_Effect->EndPass();
                }
                l_Effect->End();
        }
        return true;
}
```

Para concluir resumimos los pasos que debemos realizar para utilizar un shader implementado en hlsl.

- Cargar el fichero del effect
- Crear el vertex type del effect y su vertex declaration
- Establecer los valores de las variables que utilizamos dentro del effect
- Renderizar utilizando la technique que queramos utilizar del effect

Matemática básica

Para calcular la iluminación que implementaremos durante esta semana, recordaremos la siguiente fórmula matemática necesaria para el desarrollo de estas funcionalidades.

Ángulo entre dos vectores

El ángulo determinado por las direcciones de dos vectores $\mathbf{a}$ y $\mathbf{b}$ viene dado por:

$$\cos\theta = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}|\,|\mathbf{b}|}$$

En el caso de que los vectores estén normalizados, sus módulos |a| |b| valdrán 1 por consecuencia la fórmula anterior quedará de la siguiente manera.

$\cos\theta = a \cdot b$

En consecuencia en caso de que dos vectores sean iguales el ángulo será 0 y el cos de ese ángulo valdrá 1.

Iluminación

Para implementar la iluminación aplicada sobre un píxel debemos comprender varios conceptos.

Ambient

La luz ambiente nos establecerá una luz constante sobre el escenario dando una iluminación base. Para calcular este valor según la ayuda del DirectX.

Ambient Lighting = C_a*[G_a + Atti*Spoti*sum(Lai)]

Parameter	Default value	Type	Description
C_a	(0,0,0,0)	D3DCOLORVALUE	Material ambient color
G_a	(0,0,0,0)	D3DCOLORVALUE	Global ambient color
$Atten_i$	(0,0,0,0)	D3DCOLORVALUE	Light attenuation of the ith light.
$Spot_i$	(0,0,0,0)	D3DVECTOR	Spotlight factor of the ith light.
sum	N/A	N/A	Sum of the ambient light
L_{ai}	(0,0,0,0)	D3DVECTOR	Light ambient color of the *i*th light

Diffuse

La luz de difuso viene dada por el ángulo que forman la dirección de la luz con la normal del píxel. Para calcular este valor según la ayuda del DirectX.

Diffuse Lighting = sum[C_d*L_d*(N·L_{dir})*Atten*Spot]

Parameter	Default	Type	Description

	value		
sum	N/A	N/A	Summation of each light's diffuse component.
C_d	(0,0,0,0)	D3DCOLORVALUE	Diffuse color.
L_d	(0,0,0,0)	D3DCOLORVALUE	Light diffuse color.
N	N/A	D3DVECTOR	Vertex normal
L_{dir}	N/A	D3DVECTOR	Direction vector from object vertex to the light.
Atten	N/A	FLOAT	Light attenuation.
Spot	N/A	FLOAT	Spotlight factor.

Specular

La componente specular de una luz remarcará el reflejo de esta sobre el material que le afecta. Para calcular este valor según la ayuda del DirectX.

Specular Lighting = C_s * sum[L_s * $(N \bullet H)^P$ * *Atten* * *Spot*]

Parameter	Default value	Type	Description
C_s	(0,0,0,0)	D3DCOLORVALUE	Specular color.
sum	N/A	N/A	Summation of each light's specular component.
N	N/A	D3DVECTOR	Vertex normal.
H	N/A	D3DVECTOR	Half way vector. See the section on the halfway vector.
P	0.0	FLOAT	Specular reflection power. Range is 0 to +infinity
L_s	(0,0,0,0)	D3DCOLORVALUE	Light specular color.
Atten	N/A	FLOAT	Light attenuation value.
Spot	N/A	FLOAT	Spotlight factor.

Para calcular el Half way Vector, de nuevo utilizamos la ayuda de DirectX.

H = norm(norm(C_p - V_p) + L_{dir})

Parameter	Default value	Type	Description
C_p	N/A	D3DVECTOR	Camera position.
V_p	N/A	D3DVECTOR	Vertex position.
L_{dir}	N/A	D3DVECTOR	Direction vector from vertex position to the light position.

Exportar normales en MAXScript

Para poder realizar shaders avanzados vamos a necesitar calcular las normales de los vértices según las normales de sus caras y los grupos de suavizado a las que pertenece.

Para ello podemos utilizar el siguiente código de MAXScript.

```
global UABVtxsNormals=#()

function UABClearNormalsArray =
(
	while UABVtxsNormals.count>0 do
	(
		deleteItem UABVtxsNormals 1
	)
)

function IsSmoothingGroupEnabled IdGroup Value =
(
	local ValueMask=2^(IdGroup-1)
	return (bit.and Value ValueMask)==ValueMask
)

function UABGetVertexNormal obj IdVertex SmoothValue FaceNormal =
(
	local HasNormal=false
	local Normal=point3 0 0 0
	for b=1 to 32 do
	(
		if((IsSmoothingGroupEnabled b SmoothValue)==true) then
		(
			Normal=Normal+UABVtxsNormals[IdVertex][b]
			HasNormal=true
		)
	)
	if HasNormal==false then
	(
		Normal=FaceNormal
	)
	return Normal
)

function UABCalcVertexsNormals obj =
(
	UABClearNormalsArray()
	local NumVtxs=getNumVerts obj

	for b=1 to NumVtxs do
	(
		UABVtxsNormals[b]=#()
		for t=1 to 32 do
		(
			UABVtxsNormals[b][t]=point3 0 0 0
		)
	)

	local NumFaces=getNumFaces obj
	local InvTransform=inverse obj.transform
	for IdFace=1 to NumFaces do
	(
		local IdxsFace=getFace obj IdFace
		local Vtx1=(getVert obj IdxsFace.x)*InvTransform
```

```
                local Vtx2=(getVert obj IdxsFace.y)*InvTransform
                local Vtx3=(getVert obj IdxsFace.z)*InvTransform

                local FaceNormal=getFaceNormal obj IdFace
                local SmoothValue=getFaceSmoothGroup obj IdFace

                for b=1 to 32 do
                (
                        if((IsSmoothingGroupEnabled b SmoothValue)==true) then
                        (
                        UABVtxsNormals[IdxsFace.x][b]=UABVtxsNormals[IdxsFace.x][b] +
                                FaceNormal
                        UABVtxsNormals[IdxsFace.y][b]=UABVtxsNormals[IdxsFace.y][b] +
                                FaceNormal
                        UABVtxsNormals[IdxsFace.z][b]=UABVtxsNormals[IdxsFace.z][b] +
                                FaceNormal
                        )
                )
        )
)
```