7 Iluminación

En este capítulo se explicarán algunos modelos de iluminación, así como diferentes tipos de fuentes importantes en computación gráfica.

El ojo humano es más sensible a los cambios de brillo que a los de color, por lo que una imagen con efectos de iluminación transmite más información al espectador de forma más eficaz.[23]

La iluminación es uno de los elementos que le da más vida a un videojuego, y es que solo hay que recordar algunos títulos para distinguir la importancia que tuvieron. Por ejemplo, las aventuras de TomRaider, HalfLife, Gears of War, Final Fantasy o WarCraft. Son videojuegos que con el paso del tiempo, experiencia y tecnología aumentaron la sensación de realismo, exigiendo más por parte de los usuarios que de los mismos creadores.

Y aunque mucho tiene que ver el comportamiento de la luz en el mundo real, también es muy cierto que en este mundo de la computación gráfica no todo se maneja como en la naturaleza, y esto debido a que las máquinas en que se desarrollan y corren están limitadas. Y es que para pruebas basta el botón del error del punto flotante o errores de aproximación. En fin, en lo que se refiere a la parte de videojuegos y entretenimiento lo que se ve bien, está bien. Claro que en el caso de un CAD (Computer Aided Design) sería un desastre natural.

Los cálculos matemáticos para obtener los efectos de iluminación se explicarán de la mejor manera, para que se puedan implementar en un shader, ocupando Hight Level Shader Languaje (HLSL). Lo cual hará un poco difícil de explicar la codificación, pues este lenguaje de programación es diferente a C#. Además Visual Studio no proporciona las mismas herramientas para escribir este tipo de programas com sucede con sus lenguajes nativos. Es decir, no se tiene el apoyo del **IntelliSense**, o el realce de los tipos de datos y de métodos; etcétera. Por lo tanto, solo queda un editor de texto plano y llano.

La recomendación para este capítulo es descargar un IDE (Integrated Development Environment) para shaders. Pueden descargar el que más les agrade, sin embargo, los ejemplos que se verán a través de este escrito fueron desarrollados en FX Composer 2.5.

7.1 Shader

Shader: *procedimiento de sombreado e iluminación personalizado que permite al artista o programador especificar el renderizado de un vértice o un píxel.*[24]

*La palabra **shader** proviene directamente del software RenderMan de Pixar.*[25]

Estaría de sobra escribir algo de la historia del hardware programable, sin embargo, es necesario hacer una diferencia entre lo que anteriormente se utilizaba para crear un renderizado en tiempo real y el hardware programable.

La programación fija, llamada **fixed function**, está sujeta a las APIs de renderizado, la cual hacía las mismas funciones de transformación, desde que se tenían los datos de los vértices hasta la representación visual en la pantalla, a este proceso de transformación se le conoce como **pipiline**. Esto producía efectos pobres y no tenía le flexibilidad que necesitaban los artistas.

Con el paso del tiempo, y a causa de los buenos efectos obtenidos a partir del renderizado **off-line**, en las películas y anuncios de televisión. Los ingenieros de hardware comenzaron a ver la posibilidad de hacer más flexible el pipeline.

El **vertex shader** fue una de las mejoras que vinieron a eliminar el procesador de vértices de la tarjeta gráfica y los sustituía por un microprocesador programable, sin embargo, aún no se podía lograr efectos sorprendentes en tiempo real como los vistos en filmes.

[23] http://www.nvidia.es/object/IO_20020107_6675.html

[24] http://www.srg.es/files/apendice_tuberia_programable.pdf

[25] ídem

Luego vino la verdadera revolución en el renderizado en tiempo real con la llegada de los **fragment shaders**, pues ahora se podían implementar muchos más modelos de iluminación que anteriormente se limitaba a Flat y Gouraud. Algunos de los modelos de iluminación que ahora se implementan son Phong, Blinn, Cell, Mapping, Toon, etcétera.

Anteriormente se hacia la programación de shaders por medio del lenguaje ensamblador, sin embargo, y como era de suponerse, pasar a un lenguaje de alto nivel era la vía más apropiada, si es que se quería avanzar más rápido.

Por lo que ahora se tienen en el mercado, tres lenguajes importantes para shader: HLSL (High Level Shading Languaje) de Microsoft, GLSL (OpenGL Shading Languaje) de OpenGl, y Cg(C for graphics) creado por Nvidia. El primero corre sobre DirectX y XNA, el segundo lenguaje es para ocuparla con OpenGL; y por último Cg correrá tanto en OpenGL como en DirectX, lo cual lo convierte en uno de los lenguajes de programación de hardware más versátil, sin embargo, en este texto se usa HLSL, porque XNA con ayuda de su ContentManger hará más sencillo cargar el programa de sombreado.

Como este texto no trata con profundidad el lenguaje HLSL y el texto está dirigido para programadores, no para artistas, es recomendable que las matemáticas explicadas más adelante y el código se traten de entender lo mejor posible, y esto se tratará de lograr con ejemplos de iluminación básica, por lo que se espera que en un futuro el lector pueda entender un poco más, cualquier shader en HLSL, y porque no, en GLSL o en Cg.

En los siguientes subtemas se explicaran a groso modo el significado de cada modelo de iluminación, acompañado de algunos cálculos matemáticos y al final de cada uno de ellos, un ejemplo ocupando HLSL en FX Composer, para visualizar en tiempo real los cambios en las variables del shading.

Cabe aclarar, si es que no lo he mencionado anteriormente, no se profundizará en el lenguaje de shading, ni en el uso del IDE FX Composer y mucho menos, en la forma artística en que puede manejarse esta herramienta. Pues el fin de este tema es introducir al lector al hardware programable con ejemplos sencillos.

7.2 Iluminación ambiental

El modelo de iluminación ambiental es la intensidad en que se refleja la propiedad del material en todas direcciones, de una manera difusa, por lo que no se debe considerar como auto-iluminante, aunque así lo aparenta. Este modelo muestra un objeto de color monocromático, a menos que una de sus partes poligonales tenga otra propiedad, o sea, otro color. Y aunque no es de mucho interés manejar la iluminación ambiental por separado, es importante para complementar los siguientes modelos.

La ecuación de iluminación es la siguiente:

$$I = I_a k_a$$

Donde I es la intensidad resultante, I_a es la intensidad de la luz ambiental, y que puede considerarse constante para todos los objetos. El coeficiente de reflexión ambiental,k_a es una propiedad del material que refleja la cantidad de luz ambiental; su valores varían entre 0 y 1.

Tómese en cuenta que el coeficiente de reflexión ambiental, como en los demás modelos de iluminación, es una conveniencia empírica y no corresponde a ninguna propiedad física de los materiales reales.

7.2.1 HLSL. Modelo de iluminación ambiental

El ejemplo siguiente aplica el modelo de iluminación ambiental, ocupando HLSL para darle al lector desde el inicio una introducción a la sintaxis.

Pero antes hay que redefinir la ecuación de iluminación ambiental para el shader. Los términos de intensidad de luz ambiental, I_a y el coeficiente de reflexión ambiental, k_a quedaran como un único elemento, el color del material ambiental, esto es para fines prácticos lo que no significa que no se puedan añadir en el código. Por lo tanto la ecuación sería la siguiente:

$$I = colorMaterialAmbiental$$

Este primer ejemplo toma el color de entrada desde la aplicación XNA para colorear la geometría de un modelo 3D. Esta interacción entre la aplicación XNA y el shader necesita de variables **uniform**, que no son más que variables "globales" al estilo del lenguaje de C, pero su característica principal es que son de sólo lectura.

La forma de definir una variable en HLSL es la siguiente:

```
[Store_Class][Type_Modifier] Type Name
[Index][:Semantic][Annotations][=Initial_Value][:Packsoffset][:Register];
```

Store_Class: son modificadores opcionales que le sugieren al compilador el alcance y tiempo de vida de las variables; este modificador puede ser especificado en cualquier orden[26].

Type_Modifier: es opcional el modificador de tipo de variable.

Type: es cualquier tipo enlistado en la Tabla 7-1.

Tabla 7-1

Tipos intrínsecos	Descripción
Bufer	Bufer, que contiene uno o más escalares.
Scalar	Un componente escalar.
Vector, Matrix	Componente múltiple vector o matriz.
Sampler, Shader, Texture	Sampler, shader u objeto textura
Struct, Use Defined	Estructura personalizad o tipo definido[27]

Name[Index]: es una cadena ASCII que identifica únicamente a una variable shader.

Semantic: es una información opcional como parámetro, usado por el compilador para ligar las entradas y salidas de los shaders, o sea el **VertexShader** y **PixelShader**. Hay varias semánticas predefinidas para el vertex y pixel shaders, además de que no se pueden crear otras. El compilador ignorará la semántica a menos que se declaren como variables globales o como parámetro dentro de un shader.

Annotanios: es un metadato opcional, en la forma de un string, conectado a una variable global. Una anotación es usada por el framework del efecto e ignorada por HLSL.

Initial_Value: es el valor inicial y es opcional; el número de valores dependerá del número de componentes en **Type**. Cada una de las variables marcadas como extern deberán ser inicializadas con un valor literal; cada variable marcada como static deberá ser inicializada como contante.

Las variables globales no se deben marcar como **extern** o **static**, y si se inicializan no serán compiladas dentro del shader y podrán ser optimizadas.

Packoffset: es una palabra reservada para empaquetar manualmente una constante.

Register: es una palabra reservada para asignarle a una variable un registro en particular.

El listado siguiente muestra el programa que utiliza el modelo de iluminación ambiental, para la geometría.

[26] Para mayor información consulte la siguiente página: http://msdn.microsoft.com/en-us/library/bb509706(v=VS.85).aspx
[27] Para mayor información consulte la siguiente página: http://msdn.microsoft.com/en-us/library/bb509587(v=VS.85).aspx

Las líneas 1 – 5, del Código 7-1, es un comentario multi línea enmarcado entre **/*** y ***/**, al estilo de C; también pueden hacerse comentarios de una sola línea empleando **//**.

Enseguida del comentario se crean las variables globales, líneas 6 – 14, éstas se consideran variables **uniform**, así que no es necesario colocar el **Storage_Class uniform**.

Las variables **world**, **view** y **projection** son matrices de 4 renglones por 4 columnas de tipo **float**, véase que le precede dos puntos y el nombre de la variable, pero con la primera letra mayúscula, esto es una anotación que ayudará a visualizar el resultado de la compilación en FX Composer, y no es necesario que tenga el mismo nombre que la variable. Por lo tanto, las semánticas para las siguientes variables, **view** y **projection**, tienen el mismo fin que **World**. HLSL no tomará en cuenta estas semánticas, pero se han incluido en este programa, simplemente para poder visualizar el resultado final en el IDE.

Código 7-1

```
1.      /*
2.      Modelo de iluminación ambiental.
3.      xintalalai@live.com.mx
4.      Carlos Osnaya Medrano
5.      */
6.
7.      float4x4 world : World;
8.      float4x4 view : View;
9.      float4x4 projection : Projection;
10.
11.     float4 colorMaterialAmbiental
12.     <
13.         string UIName = "Color Ambiental";
14.         string UIWidget = "Color";
15.     > = {0.05F, 0.05F, 0.05F, 1.0F};
16.
17.     float4 mainVS(float3 posicion : POSITION) : POSITION
18.     {
19.         float4x4 wvp = mul(world, mul(view, projection));
20.         return mul(float4(posicion.xyz, 1.0F), wvp);
21.     }
22.
23.     float4 mainPS() : COLOR
24.     {
25.         return colorMaterialAmbiental;
26.     }
27.
28.     technique ModeloAmbiental
29.     {
30.         pass p0
31.         {
32.             VertexShader = compile vs_3_0 mainVS();
33.             PixelShader = compile ps_3_0 mainPS();
34.         }
35.     }
```

La variable **colorMaterialAmbiental** es un vector de cuatro componentes escalares de tipo **float**, y cada uno de ellos representa los componentes **RGBA** del color. Enseguida de la declaración de esta variable se puede ver la inclusión de un metadato encerrado entre **<** y **>**, y la inicialización de la variable; esto con el fin de manipular con la mayor sencillez el color en FX Composer. HLSL descartará este metadato y no se tomará en cuenta en la aplicación XNA, pero que repito, es para poder manejar las variables de una manera más sencilla en FX Composer.

El metadato **UIName**, de **colorMaterialAmbiental** le asigna un alias de tipo **string** al nombre, pero que será ignorado por HLSL, empero que se reflejará en la interfaz gráfica del usuario (GUI, por su soglas en inglés) de FX Composer. Se utiliza también, el **string UIWidget** para abrir una ventana llamada **Color Picker** en FX Composer, la cual ayuda a seleccionar un color, véase Ilustración 7-1.

La inicialización de la variable es como cualquier asignación a un arreglo de flotantes en C; entre llaves y separado por comas se asignan los valores correspondientes, no olvide colocar el punto y coma al final de la asignación.

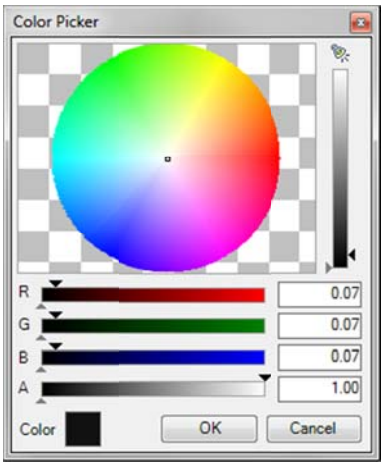

Ilustración 7-1 Color Picker

Las líneas 16 – 19 es la subrutina que servirá para hacer los cálculos correspondientes a cada vértice que sea leído por el shader, regularmente llamado **vertex shader**. Esta subrutina necesita datos de entrada, además de las variables globales, que se obtienen a partir de los parámetros de ésta, línea 16. Nótese que la declaración es parecida a las funciones del lenguaje de C, pero con la diferencia que ésta debe recibir por lo menos una semántica y arrojar otra.

Las semánticas son regularmente estructuras, cuyos componentes son semánticas predefinidas de HLSL. Por ejemplo:

```
struct VertexShaderEntrada
{
    float4 Posicion : POSITION;
    float2 CoordenadaTextura : TEXCOORD0;
};
struct VertexShaderSalida
{
    float4 Posicion : POSITION;
    float2 CoordenadaTextura : TEXCOORD0;
};
```

Algunas de las semánticas establecidas para las entradas son:

- POSITIONn
- TEXCOORDn
- NORMAL
- COLORn

Y para datos de salida:

- POSITION
- COLORn
- TEXCOORDn[28]

En donde **n** es número entero.

[28] Para mayor información acerca de las semánticas, consulte la siguiente dirección: http://msdn.microsoft.com/en-us/library/bb509647(v=VS.85).aspx

Tome en cuenta la integridad de los datos, pues a veces puede haber truncamiento de datos, provocando errores en tiempo de ejecución. También tome en cuenta que las semánticas deben ser las mismas en ambas estructuras de entrada como de salida.

En este ejemplo no se manejan las semánticas como estructuras de entrada y de salida, por lo que en los parámetros de la función **mainVS**, línea 16, se deben incluir después del nombre de la variable y después de los parámetros de la función, pues esto indica al compilador que el resultado arrojado por la subrutina será de tipo **POSITION**. Por lo menos el vertex shader debe arrojar la posición del vértice, para que el pixel shader lo tome como una semántica de interpoladores como se muestra en la subrutina mainPS, líneas 22 – 25, también llamada pixel shader.

Este vertex shacer toma como entrada la posición del vértice, y también puede agregarse otras, como las coordenadas de textura, la normal y la tangente del vértice. En este ejemplo solo se necesita el valor de la posición del vértice para implementar el modelo de iluminación ambiental.

El valor arrojado por el vertex shader es de tipo **float** y que gracias a la semántica **POSITION**, es posible que el **pixel shader** pueda interpolarlo para su uso.

Los interpoladores o variables **Varying** sirven para la comunicación entre los vertex shader y pixel shaders.

Estos interpoladores son para mantener la congruencia de la información, por ejemplo, en la Ilustración 7-2 se muestra una superficie formada por tres vértices v1, v2 y v3, y cada uno de ellos con sus respectivas normales **nv1**, **nv2** y **nv3**. Estos datos sólo son útiles cuando se está trabajando a nivel de vértice, y que se recomienda que a este nivel se haga la mayoría de los cálculos, pues el número de veces que se ejecuta es menor que si se trabaja a nivel píxel.

Para que el **píxel shader**, también llamado **fragment shader**, pueda hacerse de los datos correspondientes, debe interpolar la información recibida del vertex shader antes de comenzar cualquier cálculo. La normal de color verde, **nf**, es la normal correspondiente a ese píxel y es el resultado de la interpolación que se hace al momento de pasar los datos del vertex shader al píxel shader. Es por eso que se debe tener una buena integración de datos en el momento de declarar las variables, y el orden de las semánticas correspondientes a las de tipo **uniform**.

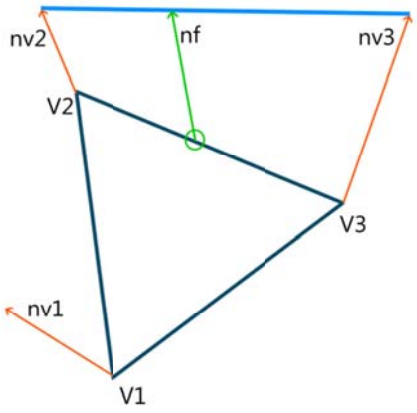

Ilustración 7-2 Interpolación de vertex shader a pixel shader

En el vertex shader, **mainVS**, se harán las transformaciones que regularmente hace el pipeline. Cambiar del espacio de coordenadas locales a las coordenadas de mundo, luego a las coordenadas de vista y por último se transforma a las coordenadas de proyección.

Esta transformación se hace sobre cada uno de los vértices; para la transformación se ocupan las matrices de mundo, vista y proyección; estas matrices las proporciona la API de XNA y DirectX.

En la línea 18 se declara una matriz de 4 renglones por 4 columnas, que se inicializa con la multiplicación de las variables **world**, **view** y **projection**. La función **mul**[29] es una función intrínseca de HLSL; multiplica dos matrices A y B siempre y cuando el número de columnas de la matriz A sea igual al número de renglones de la matriz B. Así que está función trabaja de la misma manera que se haría una multiplicación de matrices en algebra lineal. El resultado es una matriz cuya dimensión es igual al número de renglones de la matriz A por el número de columnas de la matriz B.

Tómese en cuenta que la multiplicación de matrices no es conmutativa, así que hay que cuidar el orden en que se le dan las matrices a la función intrínseca **mul**.

Luego de haber obtenido la matriz **wvp**, se multiplica por la posición del vértice para hacer la transformación de las coordenadas locales a las de proyección, línea 19. Pero antes de hacer dicha operación se debe homogenizar el vector de posición del vértice, esto es para poder realizar la multiplicación. En este caso se crea un nuevo vector de cuatro elementos de tipo flotante, donde los primeros tres son iguales a las coordenadas **x**, **y** y **z** de la posición del vértice, y el último es uno. Como valor de retorno del vertex shader es la posición del vértice transformado, utilizando la palabra reservada **return**.

El pixel shader, **mainPS**, líneas 22 – 25, no tiene una semántica de entrada explícita, pues en este caso se toma como default la posición del vértice. Pero si tiene una explícita de salida que indica que regresará el color del píxel, este valor de retorno es el color que se toma a partir de la entrada de la aplicación XNA, y en este caso de FX Composer, línea 24. Aquí es importante aclarar que este color de retorno iluminara o coloreara de igual forma la geometría, esto representa el material ambiental del objeto y la ecuación del modelo ambiental.

Por último, se tiene la técnica con la cual se hará pasar la información de la geometría, líneas 27 – 34, para su transformación directa en el hardware programable. Se debe comenzar por escribir la palabra **technique** y enseguida el nombre, línea 27. El nombre de la técnica como el de las subrutinas puede contener caracteres alfanuméricos y guión bajo.

En el cuerpo de la la técnica, encerrada entre **{** y **},** se incluyen las pasadas por las que se harán las transformaciones de la información de la geometría, y estas deben comenzar con la palabra reservada **pass**, luego del nombre. Esto regularmente se utiliza para efectos más elaborados, por lo que no se incluye en la mayoría de los ejemplos de este texto, y que por el momento no se verá.

La declaración de las variables **VertexShader** y **PixelShader** se deben de hacer dentro de **pass**, líneas 31 y 32, cuya sintaxis debe cumplir con lo siguiente:

```
PixelShader = compile ShaderTarget ShaderFunction(…);
VertexShader = compile ShaderTarget ShaderFunction(…);
```

Tabla 7-2

Parámetros	Descripción
XXXShader	Una variable shader, que representa el shader compilado. Puede PixelShader o VertexShader.
ShaderTarget	Es el model shader contra el que se compila.
ShaderFunction(…)	Es una cadena ASCII que contiene el nombre de la función de entrada del shader; esta es la función que comienza la ejecución cuando el shader es invocado. El (…) representa los argumentos del shader.

[29] Para conocer todas las funciones intrínsecas que ofrece HLSL, consulte la siguiente página web: http://msdn.microsoft.com/en-us/library/ff471376(v=VS.85).aspx

Los **model shader**, dependerán del API con la cual se hará su ejecución, la GPU[30] y el sistema operativo. Sin embargo, XNA no tiene soporte para todas las versiones de model shader, por lo que trabajara al igual que Direct3D 9, cuyos modelos de shaders se limita a las versiones 1, 2 y 3. En la tabla siguiente se enlista los model shader que pueden ocuparse[31].

Tabla 7-3

Model Shader	Shader Profile
Shader Model 1	vs_1_1
Shader Model 2	ps_2_0, ps_2_x, vs_2_0,.vs_2_x
Shader Model 3	ps_3_0, vs_3_0

Las funciones **mainVS** y **mainPS** que se ocupan para la declaración de las variables shaders, no deben contener como argumentos las semánticas que ocupa la API como entradas, pero si debe especificarse aquellas que se declaren como **uniform** y que puedan servir como entradas extras para la activación de cierto cálculo. En este ejemplo las funciones de shaders contienen únicamente las semánticas que la API necesita como entrada, por lo que en las líneas 32 y 33 se deja vacío los argumentos.

7.2.2 Inciando FX Composer

FX Composer 2.5 es un IDE para el desarrollo de shaders orientado para artistas y programadores, éste último es en el que se enfocará este texto, pues permite escribir directamente en su editor de texto, visualizar en tiempo real el resultado del shader y una interfaz sencilla para manipular las variables uniform que pueda tener el shader. En la Ilustración 7-3 se muestra la interfaz gráfica de FX Composer 2.5.

[30] Antes de ejecutar el shader se debe saber el modelo de shader que puede ejecutar la GPU, por lo que es menester revisar las características del dispositivo gráfico.

[31] Para mayor información acerca de los model shader visite la página siguiente: http://msdn.microsoft.com/en-us/library/bb509626(v=VS.85).aspx

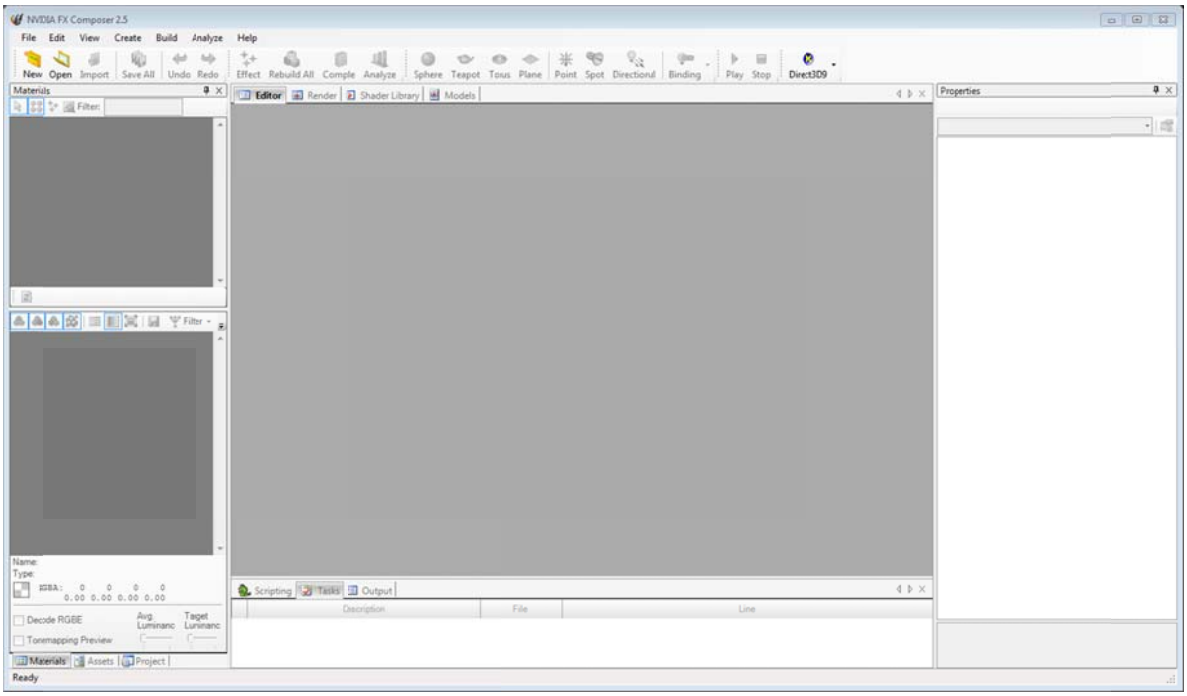

Ilustración 7-3 GUI de Fx Composer 2.5

Una vez iniciado FX Composer 2.5 cree un nuevo proyecto oprimiendo las teclas **Ctrl+Mayús+N**, o en **File\New Project** como se ve en la Ilustración 7-4, para abrir un cuadro de diálogo.

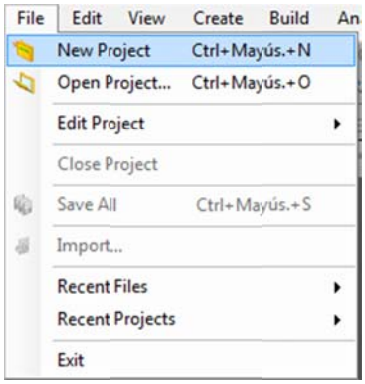

Ilustración 7-4 Nuevo proyecto

En el cuadro de diálogo, Ilustración 7-5, se tiene el **TextBox Name** y el **ListBox Location**, para darle nombre al proyecto y localidad en el dispositivo de almacenamiento. Es recomendable dejar habilitada la creación del directorio del proyecto, pues esto permite una mejor organización. Para cambiar de localidad el proyecto el botón abrirá otro cuadro de diálogo para buscar en dónde quisiera que se alojara.

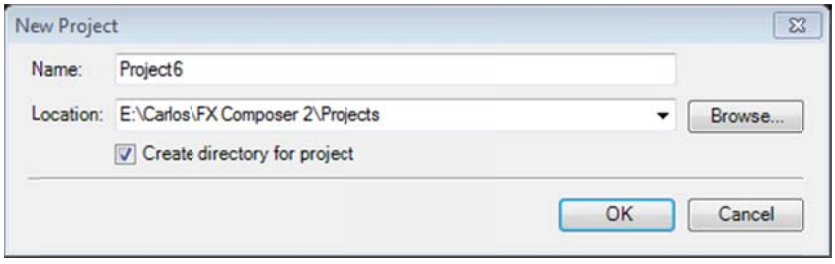

Ilustración 7-5 Ventana de diálogo

Una vez que se tiene el nombre del proyecto y la localidad, oprima el botón **OK**. En seguida se activarán iconos del **ToolBar,** y en el **TabPage Render** se activará el viewport que mostrará la geometría.

La inserción de un nuevo **Effect** consiste en crear un nuevo shader en los diferentes lenguajes de shaders disponibles en FX Composer 2.5. En la barra de tareas oprima en la etiqueta **Create\Add Effect…**, como se ve en la Ilustración 7-6.

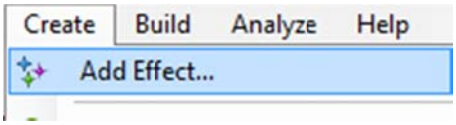

Ilustración 7-6 Añadir nuevo efecto

Enseguida se abre una ventana de diálogo **Effect Wizard**, Ilustración 7-7, para agregar un nuevo efecto al proyecto. En este caso sólo se trabajará con HLSL FX por lo que se habilita la casilla son dicho nombre en el cuadro de diálogo.

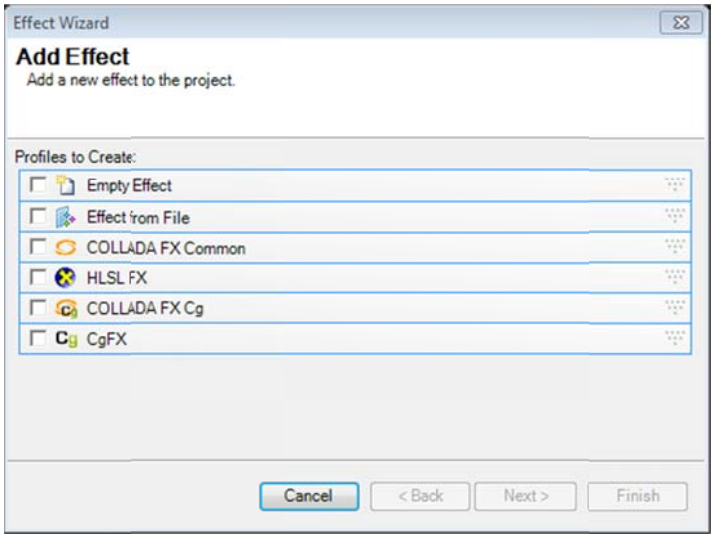

Ilustración 7-7 Ventana de diálogo para añadir nuevo efecto

Ya seleccionado, el perfil a crear, se oprime el botón **Next** para seguir con la selección de la plantilla **Empty** que ofrece FX Composer de una lista predeterminada que tiene, véase Ilustración 7-8.

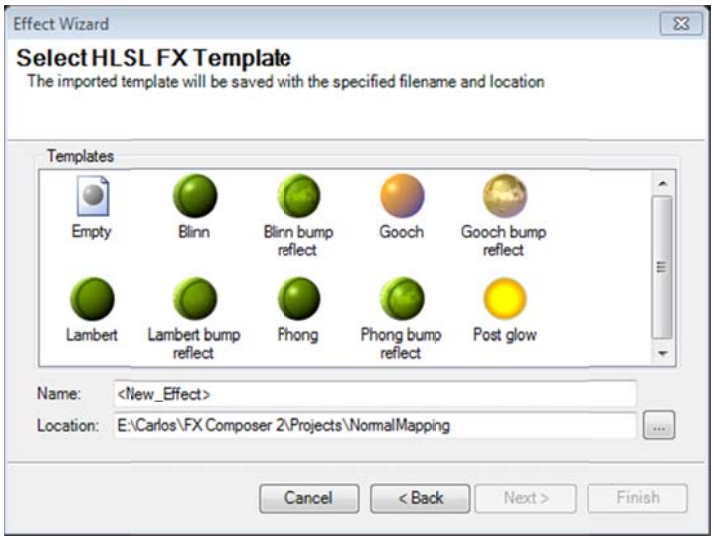

Ilustración 7-8 Plantillas para HLSL

En este punto se debe dar un nombre al archivo fx que se creará y su localización. Después de haber nombrado al archivo, oprima el botón **Next**.

Este último apartado es para darle nombre al efecto, se recomienda dejar el establecido por FX Composer, pues corresponderá al asignado en el archivo ***.fx**, de la misma manera que el nombre del material, también se recomienda dejar activada la casilla **Create a material for this effect**. Oprima el botón **Finish**, véase Ilustración 7-9.

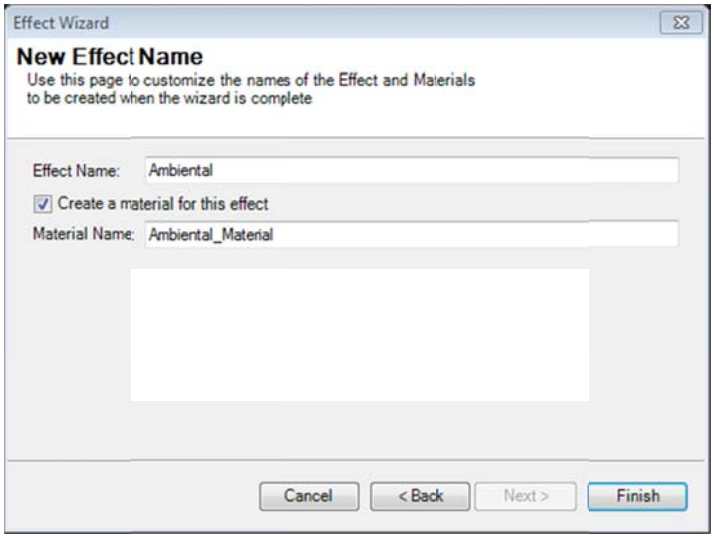

Ilustración 7-9 Ventana de diálogo para el nombre del efecto

En la parte de materiales de la GUI podrá ver que se ha creado un nuevo material llamado **Ambiental_Material**, el cual solo corresponde a una muestra del efecto en una esfera. Como se ha seleccionado la plantilla **Empty**, la esfera es más un círculo blanco. Haga doble clic sobre el material **Ambiental_Material** para poder editarlo, véase Ilustración 7-10.

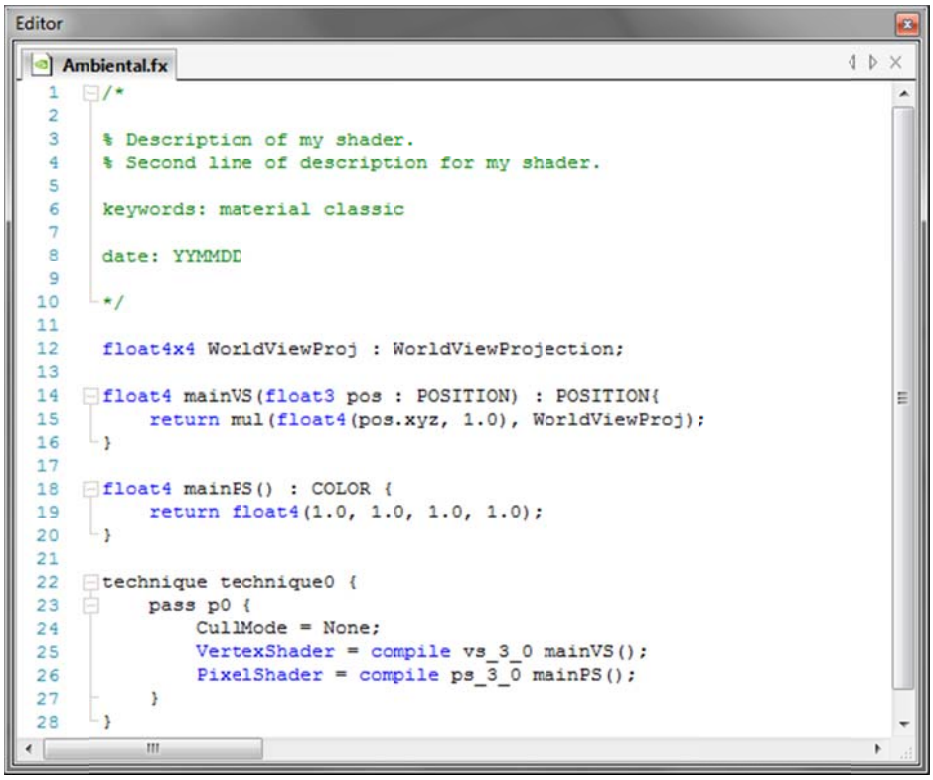

Ilustración 7-10 Editor de texto de FX Composer

El editor de texto ofrece el realce de palabras reservadas, el nombre de las variables de HLSL y los comentarios.

Borre todo el contenido del editor de texto y escriba el Código 7-1 de este texto. Compile la aplicación con la tecla **F6** y vera el cambio inmediato en el material **Ambiental_Material**. Si tiene un error en tiempo de compilación busque el error en las líneas transcritas en el editor de texto, pues este ejemplo ha sido probado perfectamente.

Para ver el resultado en el viewport que se encuentra en la **TabPage Render**, primero obtenga el foco oprimiendo en la pestaña. Luego seleccione uno de los modelos predefinidos de FX Composer en la barra de tareas, o importe un modelo en el menú **File\Import…**

En el viewport se visualiza una elefante con el material **DefaultMaterial**, ésta no se puede editar pero si se manipular sus propiedades.

Ahora hay que añadir el material **Ambiental_Material** al modelo, para esto arrastre el material de la sección **Materials** hacia el elefante, y en automático verá el cambio en el viewport, véase Ilustración 7-11.

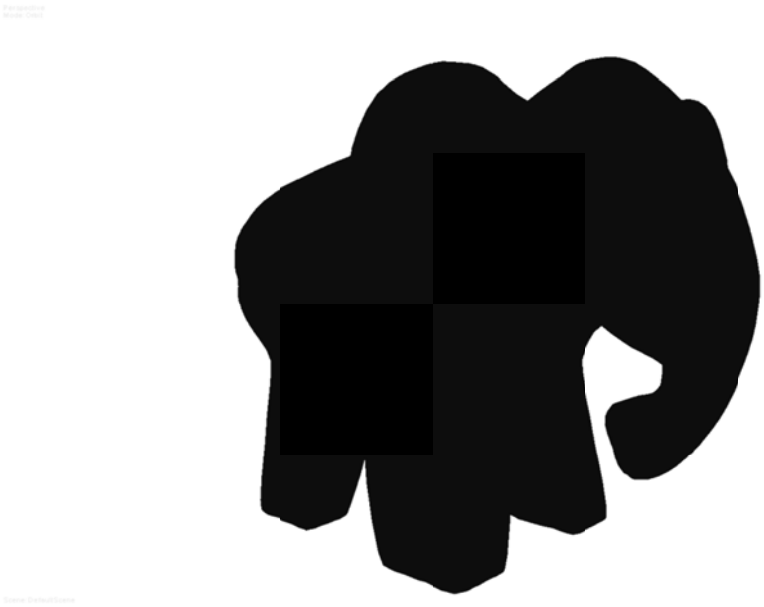

Ilustración 7-11 Elefante con iluminación ambiental

Para cambiar el **Color Ambiental**, seleccione de la parte **Properties** el parámetro con dicho nombre y se abrirá el **Color Picker** para seleccionar el color con el ratón.

Al seleccionar el color dentro del círculo y la intensidad en la barra, se verá el cambio inmediato en el viewport y en el material.

7.2.3 HLSL. Modelo de iluminación ambiental con textura

La textura es uno de los parámetros que permiten dar al renderizado una mejor vista de los que queremos que aparente tal modelo. Es por eso que en este ejemplo sólo se añadirá una textura al shader anterior, y que en los siguientes modelos de iluminación no sea necesario volver a explicar. También se cambiará la manera en que se escriben las semánticas de entrada y salida para los shaders vertex y pixel.

La declaración de la variable **texturaModelo** de tipo **Texture2D** en el Código 7-2, línea 16, es la variable que contendrá la textura provista por la API como entrada, la declaración de ésta es la misma que para cualquier variable **uniform**.

```
Type Name
```

Tabla 7-4

Parámetro	Descripción
Type	Es uno de los siguientes tipos: Texture1D, Texture1DArray, Texture2D, Texture2DArray, Texture3D, TextureCube.
Name	Cadena ASCII que identifica el nombre de la variable.

Después de haber declarado la variable es recomendable escribir de inmediato el tipo de sampleo de la textura. La siguiente sintaxis declara el estado del sampleo para Direct3D 9[32].

[32] Para mayor información acerca del tipo de muestreo de la textura en Direct3D 9 y Direct3D 10 visite la siguiente dirección web: http://msdn.microsoft.com/en-us/library/bb509644(VS.85).aspx

```
sampler Name = SamplerType{Texture = <texture_variable>; [state_name =
state_value;]…}
```

Tabla 7-5

Parámetro	Descripción
sampler	Palabra reservada que se requiere.
Name	Cadena ASCII única que identifica el nombre de la variable sampler.
SamplerType	[entrada] Tipo de muestreo que es uno de los siguientes: sampler, sampler1D, sampler 2D, sampler3D, samplerCUBE, sampler_state, SamplerState.
Texture = <texture_variable>	Una variable de textura. La palabra reservada Texture es requerida para establecer la regla de la mano izquierda o derecha. Los corchetes de ángulo son para establecer la regla de la mano izquierda y si se omiten es para establecer la regla de la mano derecha.
state_name = name_value	[entrada] Asignación de estados opcional. Todas las asignaciones de estado pueden aparecer fuera del bloque encerrado por { y }. Cada asignación será separada con un punto y coma. La siguiente lista muestra los posibles nombres de estado: AddressU AddressV AddressW BorderColor Filter MaxAnisotropy MaxLOD MinLOD MipLODBias Los valores de los estados serán igual a los ofrecidos por Direct3D o XNA[33]

Las líneas 21 – 29 muestran el tipo de sampleo que se utiliza para la textura, y si se recuerda el Textura no será necesario explicar lo estados y valores que puede tomar el tipo de sampleo.

La declaración de estructuras que le siguen al tipo de sampleo es opcional, pero es más sencillo de entender, de escribir y de mantener. Estas estructuras representan la semántica de entrada y salida para el vertex shader, líneas 48 – 56, y como semántica de entrada para el pixel shader, líneas 58 – 68.

[33] Para mayor información acerca de los valores que ofrece Direct3D consulte la siguiente dirección web: http://msdn.microsoft.com/en-us/library/bb173347(v=VS.85).aspx

Código 7-2

```
1.      /*
2.      Modelo de iluminación ambiental.
3.      xintalalai@live.com.mx
4.      Carlos Osnaya Medrano
5.      */
6.      float4x4 world : World;
7.      float4x4 view : View;
8.      float4x4 projection : Projection;
9.
10.     float4 colorMaterialAmbiental
11.     <
12.         string UIName = "Color Ambiental";
13.         string UIWidget = "Color";
14.     > = {0.05F, 0.05F, 0.05F, 1.0F};
15.
16.     texture2D texturaModelo
17.     <
18.         string UIName = "Textura";
19.     >;
20.
21.     sampler modeloTexturaMuestra = sampler_state
22.     {
23.         Texture = <texturaModelo>;
24.         MinFilter = Linear;
25.         MagFilter = Linear;
26.         MipFilter = Linear;
27.         AddressU = Wrap;
28.         AddressV = Wrap;
29.     };
30.
31.     struct VertexShaderEntrada
32.     {
33.         float4 Posicion : POSITION;
34.         float2 CoordenadaTextura : TEXCOORD0;
35.     };
36.
37.     struct VertexShaderSalida
38.     {
39.         float4 Posicion : POSITION;
40.         float2 CoordenadaTextura : TEXCOORD0;
41.     };
42.
43.     struct PixelShaderEntrada
44.     {
45.         float2 CoordenadaTextura : TEXCOORD0;
46.     };
47.
48.     VertexShaderSalida mainVS(VertexShaderEntrada entrada)
49.     {
50.         VertexShaderSalida salida;
51.         float4x4 wvp = mul(world, mul(view, projection));
52.         salida.Posicion = mul(entrada.Posicion, wvp);
53.         salida.CoordenadaTextura = entrada.CoordenadaTextura;
54.
55.         return salida;
56.     }// fin del VertexShader mainVS
57.
58.     float4 mainPS(PixelShaderEntrada entrada,
59.         uniform bool habilitarTextura) : COLOR
60.     {
61.         if(habilitarTextura)
62.         {
63.             float4 texturaColor = tex2D(modeloTexturaMuestra,
64.                 entrada.CoordenadaTextura);
65.             colorMaterialAmbiental *= texturaColor;
66.         }
67.         return colorMaterialAmbiental;
68.     }// fin del PixelShader mainPS
69.
```

```
70.     technique Texturizado
71.     {
72.         pass p0
73.         {
74.             VertexShader = compile vs_3_0 mainVS();
75.             PixelShader = compile ps_3_0 mainPS(true);
76.         }
77.     }// fin de la técnica Texturizado
78.
79.     technique Material
80.     {
81.         pass p0
82.         {
83.             VertexShader = compile vs_3_0 mainVS();
84.             PixelShader = compile ps_3_0 mainPS(false);
85.         }
86.     }// fin de la técnica Material
```

Para que las estructuras puedan servir como semánticas, es necesario que todos sus campos sean semánticos.

La estructura **VertexShaderEntrada** y **VertexShaderSalida** tienen los mismos campos, se ha dejado de esta manera para una mejor comprensión del código, por lo que no es necesario escribir ambos. Estas estructuras albergan el campo de tipo **float4** que representa la posición del vértice, y cuya semántica es **POSITION**. El segundo campo es de tipo **float2,** representa la coordenada de textura que contiene el vértice, por lo tanto su semántica es **TEXCOORD0**.

La estructura **PixelShaderEntrada**, líneas 43 – 46, contiene un sólo campo de tipo **float2** que representa la coordenada de textura del píxel; la semántica debe ser la misma que arroja como resultado el vertex shader, en este caso es **TEXCOORD0**.

La subrutina **mainVS** toma como entrada la estructura **VertexShaderEntrada** y da como resultado una estructura **VertexShaderSalida**, esta sintaxis es igual que en el lenguaje C y C#, por lo que no se profundizara. Hay que recordar que estas subrutinas que se compilan como shaders deben tener como entradas y salidas semánticas de HLSL.

En la línea 50 se declara la variable salida de tipo **VertexShaderSalida**, que fungirá como dato de salida del vertex shader. Enseguida se inicializan los campos de dicha variable, líneas 52 y 53. La coordenada de textura del vértice de entrada se le asigna sin ninguna modificación al campo **CoordenadaTextura** de la variable salida. Y por último, se regresa como dato de salida la variable **salida**.

El pixel shader **mainPS**, líneas 58 – 68, sigue ofreciendo como salida, el tipo de dato **float4** que representa el color final del píxel, gracias a la semántica **COLOR**. A diferencia al ejemplo anterior, éste toma como parámetro de entrada una estructura y una variable de tipo **uniform,** que no tiene semántica; lo anterior servirá para habilitar la textura, en el resultado final del píxel. Las variables que se den como argumentos en las subrutinas que se compilen como vertex o pixel shaders deben tener semántica, en dado caso de no hacerlo se debe incluir la palabra reservada **uniform,** antes del tipo de dato de la variable.

El cálculo necesario para tener como resultado final la combinación de cualquier modelo de iluminación, y la textura, es una multiplicación uno a uno, es decir elemento por elemento, entre el arreglo de cuatro elementos que representa el color obtenido a partir de las operaciones del modelo de iluminación, por el resultado de la función intrínseca **tex2D** de HLSL, línea 63 – 65.

`tex2D(s, t)`[34]

Tabla 7-6

Parámetro	Descripción

[34] Para mayor información acerca de las funciones intrínsecas para textura, revise la siguiente página web: http://msdn.microsoft.com/en-us/library/ff471376(v=VS.85).aspx

s	[entrada]Estado de muestreo de la textura.
t	[entrada]Coordenada de textura.

Con el condicional **if** y la variable **habilitarTextura** de tipo **bool**, línea 62, se procesan los datos correspondientes para texturizar el píxel, en caso contrario sólo se pinta con el color ambiental.

En este ejemplo se utilizan dos técnicas **Texturizado**, líneas 70 – 77, y **Material**, líneas 79 – 86. También se pudo haber utilizado una variable global para habilitar la textura, pero se ha dejado así para fines educativos. La compilación del pixel shader **mainPS**, líneas 75 y 84, pasan como parámetro de entrada el valor **true** o **false** directamente.

7.2.4 Añadiendo nuevo efecto en FX Composer

Retomando el proyecto ya hecho en el dubtema 7.2.2, añada un nuevo efecto tomando la plantilla **Empty**. Luego borre todo el contenido del nuevo efecto y escriba el listado anterior, **Ambiental con Textura**. Compile el efecto y quite cualquier error de compilación si así sucediese.

Cambie el material del modelo en el viewport, arrastrando el material **Ambiental con Textura** al modelo tridimensional, verá que la geometría es de un sólo color, esto es porque no tiene una textura como entrada.

Haga clic en el parámetro **Textura** en la parte **Properties** de FX Composer 2.5, Ilustración 7-12, seleccione la opción **Image** para abrir un cuadro de diálogo llamado **Textures**, Ilustración 7-13.

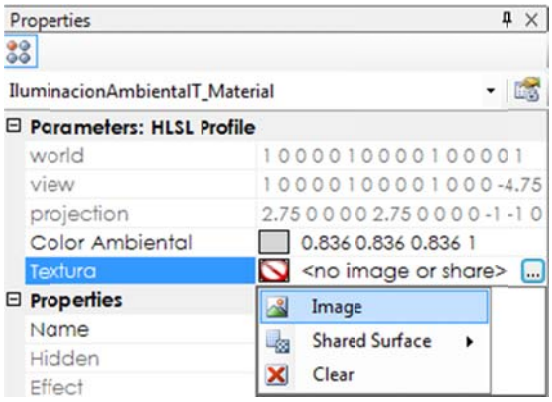

Ilustración 7-12 Campo Textura

Luego oprima el icono **Add Image**, aquel con forma de cruz o el símbolo más, en seguida se abrirá otra ventana de diálogo para seleccionar las imágenes tomadas de memoria secundaria. Sin embargo, no todas podrán agregarse al shader, pero sí estarán enlistadas en el cuadro de diálogo **Textures**, como lo muestra la Ilustración 7-13. Una vez seleccionada la imagen oprima el botón **OK** y de inmediato verá en el viewport del **TabPage Render** que la textura está envolviendo al modelo.

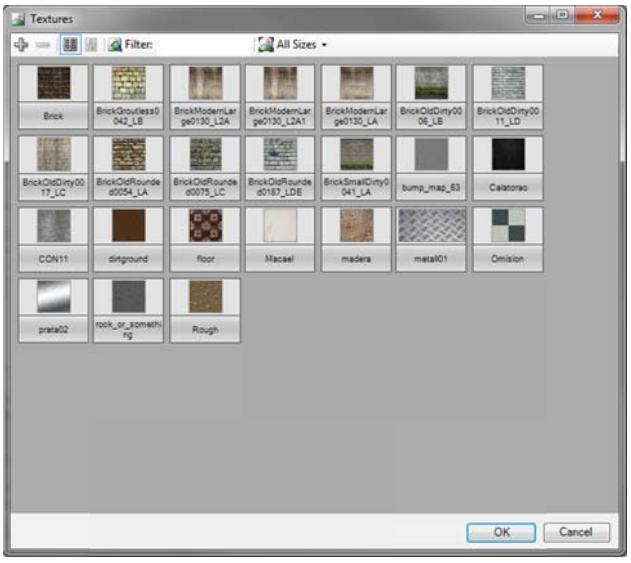

Ilustración 7-13 Texturas

FX Composer no ejecuta todas las técnicas que el archivo fx contiene, pero sí las compila. Así que para ver el efecto que no se está ejecutando comenté la técnica que esté más arriba del archivo.

Por último, cambie los valores al color ambiental para ver el resultado final sobre la textura.

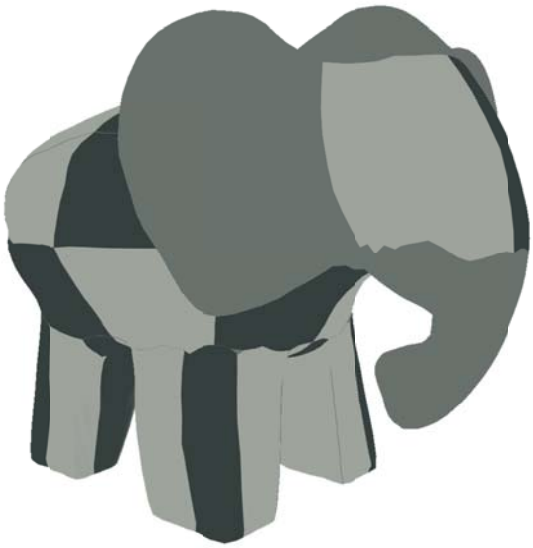

Ilustración 7-14 Ambiental con textura

7.3 Iluminación difusa

La iluminación ambiental no es de gran interés, pues los colores son planos y no da una sensación de volumen de los objetos. La iluminación difusa junto con la ambiental obtiene ese sentido de volumen, gracias a las diferentes intensidades en cada vértice o píxel, esta cantidad de luz reflejada depende del ángulo formado entre la normal del vértice o píxel y el haz de luz incidente que proviene de una fuente de iluminación, véase Ilustración 7-15.

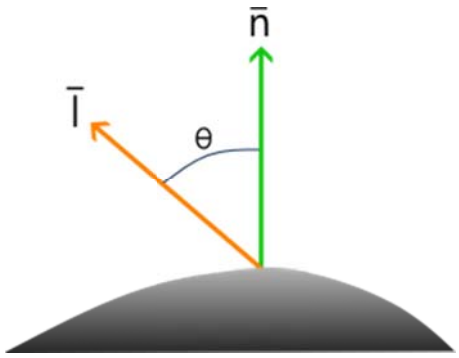

Ilustración 7-15 Ángulo formado entre el haz de luz y la normal de la superficie

La reflexión difusa o también llamada **reflexión Lambertiana**, no toma en cuenta la posición del observador, por lo tanto es independiente de éste y, sólo es proporcional al coseno del ángulo de incidencia de luz y la normal.

La ecuación de iluminación difusa es:

$$I = I_f k_d \cos\theta$$

Donde I_f es la intensidad de la fuente luminosa; k_d es el coeficiente de reflexión difusa del material que varía entre cero y uno. Regularmente el rango del ángulo se encuentra entre 0° a 90°, esto es con el fin de hacer autoocluyente la reflexión difusa, véase Ilustración 7-16.

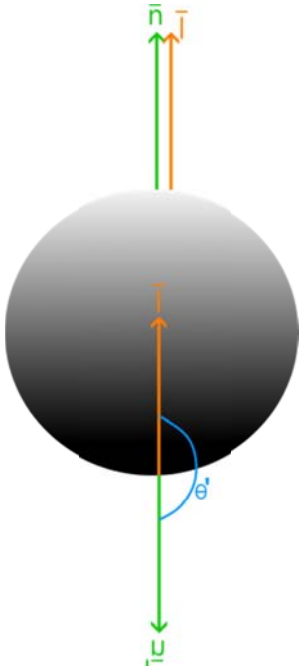

Ilustración 7-16 Autoocluyente

Para conocer el ángulo entre dos vectores, se utiliza la siguiente expresión:

$$\theta = ang\ \cos\frac{\bar{a}\cdot\bar{b}}{|\bar{a}||\bar{b}|}$$

Donde $\bar{a}$ y $\bar{b}$ son dos vectores cualesquiera.

Despejamos el coseno del ángulo y obtenemos la siguiente expresión:

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

Ahora se toman los valores de los vectores $\bar{n}$ y $\bar{l}$, donde el vector del haz de luz es unitario y el coseno el ángulo debe ser positivo, para ser autoocluyente la expresión quedaría como:

$$\cos\theta = max(\bar{n}\cdot\bar{l}, 0)$$

Donde max selecciona el valor máximo entre el producto punto y cero. La ecuación de iluminación difusa quedaría escrita como:

$$I = I_f k_d max(\bar{n}\cdot\bar{l}, 0)$$

Para obtener una iluminación más realista se agrega el modelo de iluminación ambiental, al modelo difuso, esto se logra sumando ambos modelos de iluminación.

$$I = I_a k_a + I_f k_d max(\bar{n}\cdot\bar{l}, 0)$$

El vector del haz de luz $\bar{l}$ dependerá del tipo de fuente de iluminación. En este texto se manejaran tres tipos básicos de fuentes, mismas que ofrece DirectX en su pipeline, **Direccional**, **Puntual** y **Spot**. Pero antes se reescribe la ecuación anterior para adaptarlo al shader de los siguientes ejemplos.

La intensidad de iluminación y coeficiente de reflexión ambiental se sustituye por el color ambiental del material; la intensidad de iluminación y coeficiente de reflexión difusa se sustituyen por el color difuso del material.

$$I = colorMaterialAmbiental + colorMaterialDifuso\left(max(\bar{n}\cdot\bar{l}, 0.0)\right)$$

7.3.1 Fuente de iluminación direccional

La fuente de iluminación direccional es semejante a tener una fuente en el infinito con una cantidad constante de energía, por lo que no importa la distancia del objeto a la fuente.

También se puede considerar que el vector del haz de iluminación direccional es paralelo para todos los vértices o pixeles del modelo tridimensional. En la Ilustración 7-17 se muestra que el vector de iluminación es paralelo para todos los vértices o píxeles del cubo.

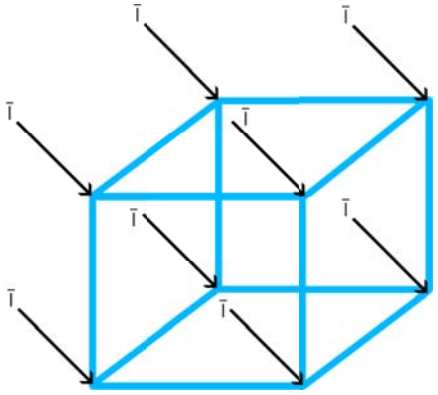

Ilustración 7-17 Iluminación direccional

Por lo tanto, la única información que se requiere de este tipo de fuente es su dirección; el vector unitario es el adecuado para este tipo de luz. También se puede considerar el color de la fuente, pero en este ejemplo no se tomará en cuenta.

Retomando la ecuación de iluminación, hasta ahora descrita, se reescribirá para el shader de manera que sea práctico usar.

En los siguientes ejemplos los cálculos necesarios para obtener el color final sobre el material y textura del modelo tridimensional se hará por píxel, esto no significa que no se puedan hacer por vértice, es más, las operaciones que se hagan por píxel son muy similares a los que se harían por vértice. Así que si el lector quiere probar estos ejemplos sobre cada vértice, se espera no le sea difícil de lograrlo. También se ha tomado la decisión de dejar las operaciones por píxel, pues el resultado final es mejor, visualmente, pero requiere más poder de GPU en comparación si se hiciese por vértice.

En el siguiente enlistado similar al visto con anterioridad, se explicarán las cosas nuevas que se han agregado para el modelo de iluminación difusa, reutilizando el ejemplo del modelo de iluminación ambiental con textura.

Código 7-3

```
1.      /*
2.      Email: xintalalai@live.com.mx
3.      Autor: Carlos Osnaya Medrano
4.
5.      Modelo de iluminación: Especular o Difusa.
6.      Tipo de fuente: Direccional.
7.      Técnica empleada: PixelShader.
8.      Material clásico.
9.      */
10.
11.     float4x4 world : World;
12.     float4x4 view : View;
13.     float4x4 projection : Projection;
14.
15.     float3 direccionLuz
16.     <
17.         string UIName = "Dirección de Luz";
18.     >;
19.
20.     float4 colorMaterialAmbiental
21.     <
22.         string UIName = "Material ambiental";
23.         string UIWidget = "Color";
24.     > = {0.05F, 0.05F, 0.05F, 1.0F};
25.
26.     float4 colorMaterialDifuso
27.     <
28.         string UIName = "Material difuso";
29.         string UIWidget = "Color";
30.     > = {0.24F, 0.34F, 0.39F, 1.0F};
31.
32.     texture2D texturaModelo
33.     <
34.         string UIName = "Textura";
35.     >;
36.     sampler modeloTexturaMuestra = sampler_state
37.     {
38.         Texture = <texturaModelo>;
39.         MinFilter = Linear;
40.         MagFilter = Linear;
41.         MipFilter = Linear;
42.         AddressU = Wrap;
43.         AddressV = Wrap;
44.     };
45.
46.     struct VertexShaderEntrada
47.     {
48.         float4 Posicion : POSITION;
49.         float3 Normal : NORMAL;
50.         float2 CoordenadaTextura : TEXCOORD0;
51.     };
52.
53.     struct VertexShaderSalida
54.     {
55.         float4 Posicion : POSITION;
```

```
56.         float2 CoordenadaTextura : TEXCOORD0;
57.         float3 WorldNormal : TEXCOORD1;
58.         float3 WorldPosition : TEXCOORD2;
59.     };
60.
61.     struct PixelShaderEntrada
62.     {
63.         float2 CoordenadaTextura : TEXCOORD0;
64.         float3 WorldNormal : TEXCOORD1;
65.         float3 WorldPosition : TEXCOORD2;
66.     };
67.
68.     VertexShaderSalida MainVS(VertexShaderEntrada entrada)
69.     {
70.         VertexShaderSalida salida;
71.         float4x4 wvp = mul(world, mul(view, projection));
72.         salida.Posicion = mul(entrada.Posicion, wvp);
73.         salida.CoordenadaTextura = entrada.CoordenadaTextura;
74.         salida.WorldNormal = mul(entrada.Normal, world);
75.         salida.WorldPosition = mul(entrada.Posicion, world);
76.
77.         return salida;
78.     }// fin del vertex shader MainVS
79.
80.     float4 MainPS(PixelShaderEntrada entrada,
81.         uniform bool habilitarTextura) : COLOR
82.     {
83.         // modelo de iluminación difusa
84.         // iluminación difusa
85.         direccionLuz = normalize(direccionLuz);
86.         float reflexionDifusa = max(dot(-direccionLuz, entrada.WorldNormal), 0.0F);
87.         float4 difuso = colorMaterialDifuso * reflexionDifusa;
88.         float4 color;
89.         // color final
90.         color = colorMaterialAmbiental + difuso;
91.
92.         // modelo de iluminación especular
93.
94.
95.
96.         // texturizado
97.         if(habilitarTextura)
98.         {
99.             float4 colorTextura = tex2D(modeloTexturaMuestra,
100.                 entrada.CoordenadaTextura);
101.            color *= colorTextura;
102.        }
103.
104.        // color final
105.        color.a = 1.0F;
106.        return color;
107.    }// fin pixel shader MainPS
108.
109.    technique Texturizado
110.    {
111.        pass P0
112.        {
113.            VertexShader = compile vs_3_0 MainVS();
114.            PixelShader = compile ps_3_0 MainPS(true);
115.        }
116.    }
117.
118.    technique Material
119.    {
120.        pass P0
121.        {
122.            VertexShader = compile vs_3_0 MainVS();
123.            PixelShader = compile ps_3_0 MainPS(false);
124.        }
125.    }
```

En la declaración de variables globales se han agregado la dirección de la luz, líneas 15 – 18, Código 7-3, y el color del material difuso, líneas 26 – 30. En la estructura de entrada para la subrutina vertex shader, se ha agregado el campo **Normal** de tipo **float3** y semántica **NORMAL**, línea 49, que recibirá la normal asociado al vértice. Para la estructura **VertexShaderSalida** se ha añadido el campo **WorldNormal** de tipo **float3** con semántica **TEXCOORD1**, línea 57; y el campo **WorldPosition** de tipo **float3** con semántica **TEXCOORD2**, línea 58. Estos dos últimos campos son para hacerle pasar al pixelshader de manera coherente los datos de la normal y posición utilizando la semántica **TEXCOORD**. En la estructura **PixelShaderEntrada** se agregan los mismos campos **WorldNormal** y **WorldPosition** que los que contiene **VertexShaderSalida**, véase que tienen la misma semántica.

Dentro de la subrutina **MainVS**, líneas 68 – 78, se inicializan los campos de la variable inmediata salida. Los nuevos campos de la estructura **VertexShaderSalida** deben contener la normal y posición del vértice en el espacio de coordenadas de mundo, para eso hay que multiplicar la normal y posición del vértice por la matriz de mundo **world**, líneas 74 y 75.

La dirección de la luz debe ser normalizada, es decir, debe convertirse el vector a un vector unitario, pues lo único que interesa de este tipo de fuente es la dirección de ésta. La función intrínseca **normalize** regresa el vector de cualquier tamaño normalizado. En la línea 85 se le asigna a la misma variable **direccionLuz**, el resultado de la normalización. Además con esto se asegura que el vector dado por la API sea unitario, también se pudo haber normalizado desde ésta y evitarle el cálculo a la GPU.

La intensidad difusa, como se había explicado con anterioridad, depende del coseno del ángulo formado entre la normal del vértice o píxel y el vector de dirección de la fuente de iluminación. Por lo que el valor máximo de la intensidad difusa será cuando la dirección de luz sea paralela a la normal y el menor será cuando sean ortogonales. Para hacer autoocluyente, como se indica en la ecuación del modelo de iluminación difusa, se utiliza la función intrinseca **max**, línea 86, que devuelve el valor máximo entre los dos parámetros que recibe. El primer parámetro es el resultado de la función intrínseca **dot** que obtiene el producto punto entre dos vectores, y el primero de **dot** es **direcciónluz** multiplicado por menos uno; el cambio de dirección de **direccionLuz**, es para darle coherencia entre la entrada del dato por parte del usuario y el cálculo para la intensidad, pues lo que se pregunta es hacia a dónde apunta la luz, no de dónde proviene la luz. El segundo parámetro de la función **max** es cero, pues los valores que debe normalmente tomar la intensidad difusa es entre cero y uno. La asignación del resultado de la función **max** a la variable **reflexionDifusa** de tipo **float**, línea 86, es para una mejor comprensión del ejemplo, por lo que no es necesario declararla.

Luego de haber obtenido la intensidad de iluminación difusa, se debe de multiplicar por el color del material difuso para luego sumarlo al modelo de iluminación ambiental. Primero se declara la variable **difuso** y se inicializa con la multiplicación entre el escalar **intesidadDifusa** por el vector **colorMaterialDifuso**, línea 87. La variable **color**, línea 88, servirá para almacenar el color final, a partir de la suma de los modelos de iluminación.

El comentario de la línea 92 indica que a partir de ahí se harán los operaciones para el modelo de iluminación especular.

Hasta este punto es todo lo nuevo en este código, todo el resto es el mismo que en el ejemplo anterior. Cree un proyecto nuevo en FX Composer para probar el shader, si no conoce cómo crear uno, lea el subtema 7.2.2, si encuentra un error de compilación, corrija y pruebe de nuevo.

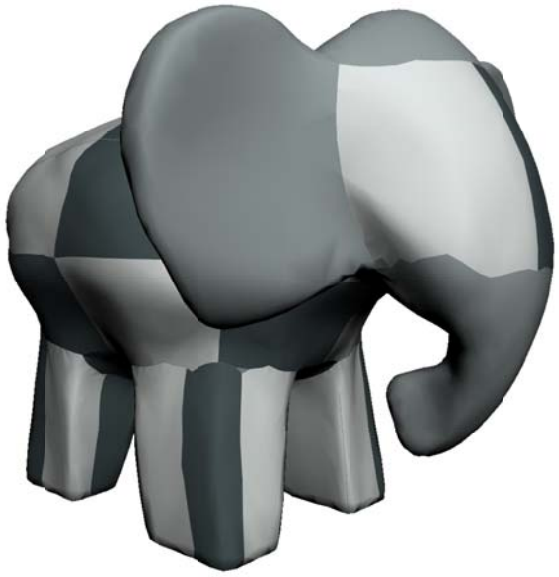

Ilustración 7-18 Difuso con textura

Como se puede ver en la Ilustración 7-18 el modelo tiene un aspecto mucho mejor que en el visto en la Ilustración 7-14. Pruebe con la técnica **Material**, comentando la técnica **Texturizado**, líneas 108 – 115, para poder apreciar mejor la sensación de volumen del objeto, como se muestra en la Ilustración 7-19.

Ilustración 7-19 Difuso sin textura

7.3.2 Fuente de iluminación puntual

La fuente de iluminación puntual es similar a tener un foco incandescente en un cuarto totalmente oscuro. Esto hace que la dirección de la fuente de iluminación al vértice varíe dependiendo de la posición en la cual se encuentre la fuente y/o el vértice. En la Ilustración 7-20 se muestra el vector de posición de la fuente $\bar{p}_f$ y los vectores $\bar{\imath}_x$de cada vértice o píxel.

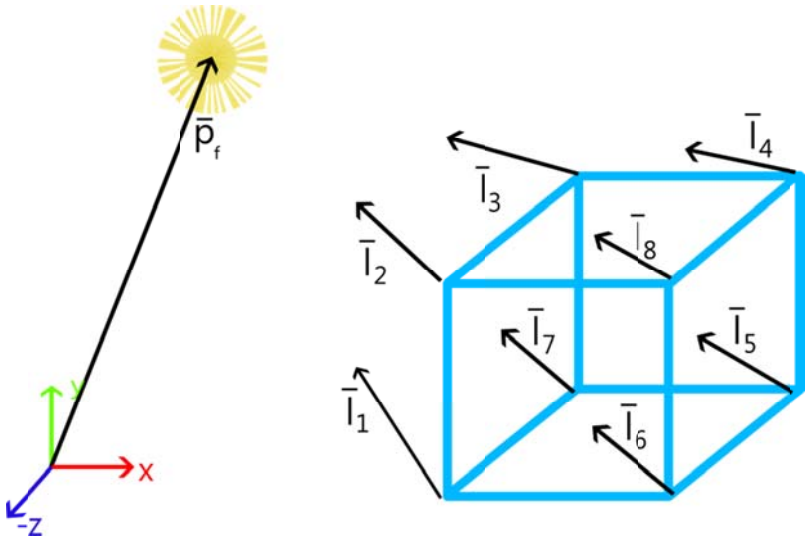

Ilustración 7-20 Fuente puntual

Otra característica importante de este tipo de fuente, es la **atenuación**. La atenuación como en el mundo real, dependerá de la distancia entre el foco y el objeto; entre más cerca se encuentre de la fuente mayor intensidad de luz reflejada se tiene; entre más alejado del foco menor intensidad de luz se podrá reflejar. Una tentativa para obtener esta atenuación es la inversa del cuadrado de la distancia.

$$f_{at} = \frac{1}{d^2{}_l}$$

Sin embargo, como lo indican los pioneros en graficación por computadora, esto no siempre funciona bien. Así que una forma útil, permitiendo una mayor gama de efectos es:

$$f_{at} = min\left(\frac{1}{c_1 + c_2 d_l + c_3 d^2{}_l}, 1\right)$$

Donde c_1, c_2 y c_3 son constante definidas por el usuario. La primera constante evita que el denominador sea demasiado pequeño cuando la luz esté cerca. La función **min** limita a un valor a uno, para asegurar que siempre se atenúe.

Una manera de conocer qué valores deben tener las constantes de atenuación es graficar la ecuación anterior, como se muestre en la Ilustración 7-21.

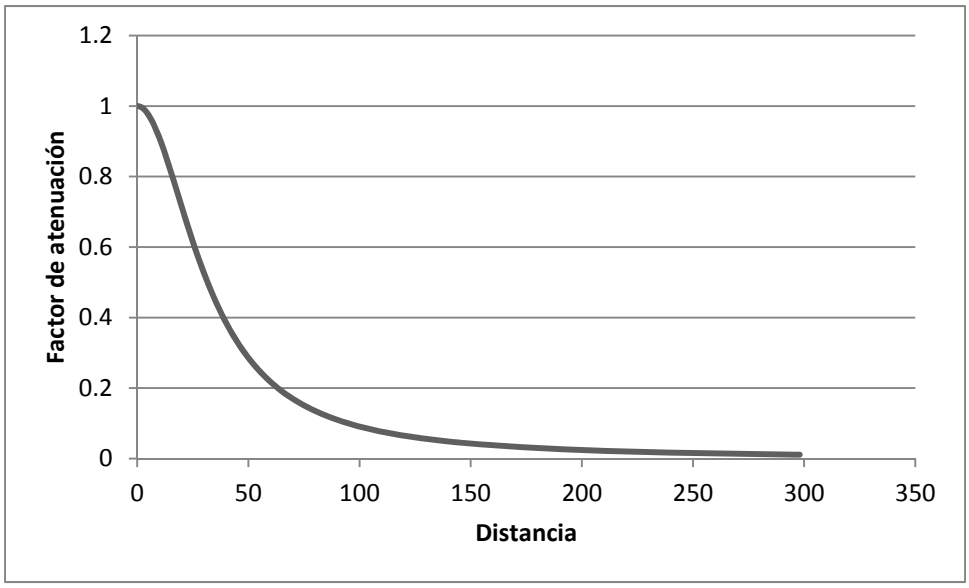

Ilustración 7-21 Factor de atenuación de iluminación. c1= 1, c2 = 0, c3 = 0.001

Regularmente los valores del factor de atenuación deben estar entre cero y uno, este valor máximo puede manipularse a partir de la primera constante, las demás constantes es para disminuir el factor de atenuación conforme aumenta la distancia. Tome en cuenta que estos valores, por lo menos para las constantes c_2 y c_3 deben ser menores que uno y mayores a cero.

Reescribiendo la ecuación de iluminación con el factor de atenuación f_{at} para el modelo de iluminación difuso.

$$I = I_a k_a + f_{at} I_f k_d max(\bar{n} \cdot \bar{l}, 0)$$

Luego se sustituye el miembro derecho del factor de atenuación y tenemos la nueva ecuación de iluminación para la fuente tipo puntual y spot.

$$I = I_a k_a + min\left(\frac{1}{c_1 + c_2 d_l + c_3 {d^2}_l}, 1\right) I_f k_d max(\bar{n} \cdot \bar{l}, 0)$$

Muchas de las líneas del código siguiente son iguales a las vistas en la fuente de iluminación direccional, por lo tanto sólo describirán aquellas que sean relevantes.

En la declaración de variables globales se tiene la posición de la fuente de iluminación como **posicionLuz**, una variable de tipo **float3**, líneas 15 – 18, Código 7-4. Las constantes de atenuación c_1, c_2 y c_3 y rango son variables de tipo **float**, líneas 32 y 33. La variable **rango** sirve para evaluar la atenuación a la distancia establecida por **rango**, es decir, aquel modelo que se encuentre dentro de la distancia establecida por **rango**, será afectada por la atenuación, en dado caso que estuviera fuera de esa distancia la atenuación sería cero, por lo que sólo la iluminación ambiental afectaría el color final del material y textura.

La variable **atenuado** de tipo **bool**, líneas 46 – 49, sirve para activar los cálculos de atenuación dentro de un condicional **if**.

Las estructuras de entrada y salida para los shaders vertex y pixel, siguen siendo las mismas que en el ejemplo anterior. En el pixel shader, líneas 87 – 132, el único cambio importante es el cálculo de la dirección de la luz a partir de la posición de dicha fuente, y los cálculo de atenuación.

El primer paso es calcular la dirección de la luz para cada píxel, esto se logra restando el vector de posición del píxel (o vértice) menos el vector de posición de la luz. Luego se normaliza dicho resultado y se le asigna a una nueva variable local, **direccionLuz** de tipo **float3**, línea 91.

Véase que el cálculo para la intensidad difusa, línea 92, no se multiplica la dirección de luz por menos uno, esto es porque se busca de dónde proviene el haz y no hacia donde apunta.

La variable **color** de tipo **float4** se utiliza para almacenar el color final que resulte de las operaciones para el color difuso, especular y atenuación. Cada vez que se termine con el cálculo de cualquier modelo de iluminación se le asignará el resultado a **color** como la suma entre el contenido de **color** y el resultado de la operación del modelo, excepto con el primero, línea 95.

Para habilitar la atenuación se hace uso del condicional **if**, líneas 102 – 120, y la variable uniform **atenuado**. Lo primero que hay que obtener es la distancia entre la fuente de iluminación y el píxel, para luego compararlo con el rango ofrecido por el usuario. La función intrínseca **distance** obtiene la distancia entre dos vectores, siempre y cuando sean de la misma dimensión, es decir, que tengan el mismo número de elementos. También se pudo haber obtenido el módulo de la resta entre posición de la fuente y el píxel, pero se ha dejado de esta manera para que el lector puedo comprender mejor el código. El resultado arrojado por **distance** se guarda en la variable local **distancia**, línea 105. En seguida se declara la variable de tipo **float**, **atenuación**, línea 107, que almacenará el factor de atenuación.

Ahora se compara la distancia entre la fuente al píxel y el rango establecido por el usuario, en caso que la distancia sea menor o igual al rango, se calcula el factor de atenuación, línea 114 y 115; en caso contrario el factor de atenuación es cero. Luego se multiplica el factor por el modelo de iluminación difuso, línea 119.

Después del condicional que habilita la atenuación, se suma el color del material ambiental al color final que se ha obtenido y luego se le asigna a la variable **color**, línea 122.

Código 7-4

```
1.      /*
2.      Email: xintalalai@live.com.mx
3.      Autor: Carlos Osnaya Medrano
4.
5.      Modelo de iluminación: Especular o Difusa.
6.      Tipo de fuente: Puntual.
7.      Técnica empleada: PixelShader.
8.      Material clásico.
9.      */
10.
11.     float4x4 world : World;
12.     float4x4 view : View;
13.     float4x4 projection : Projection;
14.
15.     float3 posicionLuz
16.     <
17.         string UIName = "Posición de Luz";
18.     >;
19.
20.     float4 colorMaterialAmbiental
21.     <
22.         string UIName = "Color ambiental";
23.         string UIWidget = "Color";
24.     > = {0.05F, 0.05F, 0.05F, 1.0F};
25.
26.     float4 colorMaterialDifuso
27.     <
28.         string UIName = "Color difuso";
29.         string UIWidget = "Color";
30.     > = {0.24F, 0.34F, 0.39F, 1.0F};
31.
32.     float c1, c2, c3; // constante de atenuación
33.     float rango;   // rango a evaluar la atenuación
34.
35.     texture2D texturaModelo;
36.     sampler modeloTexturaMuestra = sampler_state
37.     {
38.         Texture = <texturaModelo>;
39.         MinFilter = Linear;
40.         MagFilter = Linear;
```

```
41.         MipFilter = Linear;
42.         AddressU = Wrap;
43.         AddressV = Wrap;
44.     };
45.
46.     bool atenuado
47.     <
48.         string UIName = "Atenuado";
49.     > = false;
50.
51.     struct VertexShaderEntrada
52.     {
53.         float4 Posicion : POSITION;
54.         float3 Normal : NORMAL;
55.         float2 CoordenadaTextura : TEXCOORD0;
56.     };
57.
58.     struct VertexShaderSalida
59.     {
60.         float4 Posicion : POSITION;
61.         float2 CoordenadaTextura : TEXCOORD0;
62.         float3 WorldNormal : TEXCOORD1;
63.         float3 WorldPosition : TEXCOORD2;
64.     };
65.
66.     struct PixelShaderEntrada
67.     {
68.         float2 CoordenadaTextura : TEXCOORD0;
69.         float3 WorldNormal : TEXCOORD1;
70.         float3 WorldPosition : TEXCOORD2;
71.     };
72.
73.
74.     VertexShaderSalida MainVS(VertexShaderEntrada entrada)
75.     {
76.         VertexShaderSalida salida;
77.         float4x4 wvp = mul(world, mul(view, projection));
78.         salida.Posicion = mul(entrada.Posicion, wvp);
79.         salida.CoordenadaTextura = entrada.CoordenadaTextura;
80.
81.         salida.WorldNormal = mul(entrada.Normal, world);
82.         salida.WorldPosition = mul(entrada.Posicion, world);
83.
84.         return salida;
85.     }// fin del vertex shader MainVS
86.
87.     float4 MainPS(PixelShaderEntrada entrada, uniform bool texturizado) : COLOR
88.     {
89.         float4 color;
90.         // modelo de iluminación difusa
91.         float3 direccionLuz = normalize(posicionLuz - entrada.WorldPosition);
92.         float reflexionDifusa = max(dot(direccionLuz, entrada.WorldNormal), 0.0F);
93.         float4 difuso = reflexionDifusa * colorMaterialDifuso;
94.         // color final
95.         color = difuso;
96.
97.         // modelo de iluminación especular
98.
99.
100.
101.        // atenuación
102.        if(atenuado)
103.        {
104.            // atenuación
105.            float distancia = distance(entrada.WorldPosition,
106.            posicionLuz); //  distancia de la fuente al vértice
107.            float atenuacion;
108.            // Si la distancia entre la fuente y el vértice se
109.            // encuentra entre el rango
110.            // éste se verá afectado por la atenuación, en caso
111.            // contrario la atenuación
```

```
            // será de cero.
            if(distancia <= rango)
                atenuacion = min(1 / (c1 + (c2 * distancia) +
                (c3 * pow(distancia, 2.0F))), 1.0F);
            else
                atenuacion = 0;
            // color final
            color *= atenuacion;
        }// fin del if
        // color final
        color += colorMaterialAmbiental;

        // texturizado
        if(texturizado)
        {
            float4 texturaColor = tex2D(modeloTexturaMuestra,
                entrada.CoordenadaTextura);
            color *= texturaColor;
        }

        // color final
        color.a = 1.0F;
        return color;
    }// fin pixel shader MainPs

    technique Texturizado
    {
        pass P0
        {
            VertexShader = compile vs_3_0 MainVS();
            PixelShader = compile ps_3_0 MainPS(true);
        }
    }

    technique Material
    {
        pass P0
        {
            VertexShader = compile vs_3_0 MainVS();
            PixelShader = compile ps_3_0 MainPS(false);
        }
    }
```

Cree un nuevo proyecto en FX Composer 2.5 para probar el Código 7-4 y ver algo similar a la Ilustración 7-22; si no conoce cómo generar un proyecto en FX Composer 2.5 lea Inciando FX Composer.

Ilustración 7-22 Fuente puntual

Para saber que realmente se ha generado un tipo de fuente diferente, cambie de posición del modelo. Esto se logra seleccionando el modelo en el viewport luego oprima la letra **W** y aparecerá un **gizmo** con tres colores, azul, verde y rojo. Al acercar el cursor del ratón a alguna de la flechas del **gizmo** verá que estas cambian a un color blanco y el puntero cambia de figura. Cuando esto pasa haga clic y arrastre el puntero del mouse para mover el modelo en la dirección. Al cambiar de posición el modelo, los vectores de dirección de iluminación cambian y así también el color en cada píxel.

Juegue con los valores de entrada del shader en la parte **Parameters** de FX Composer, sobre todo con las constantes, recomiendo tener cerca la gráfica del factor de atenuación para tener una mejor referencia de lo que se hace.

7.3.3 Fuente de iluminación spot

La fuente de iluminación spot es similar a tener una lámpara sorda en un cuarto oscuro, este tipo de fuente ilumina una región en forma cónica, que gradualmente se atenúa dependiendo de la distancia del objeto a la posición de la fuente y del ángulo de abertura de la lámpara.

Las características de una fuente tipo spot están representadas en la Ilustración 7-23, donde la posición de la fuente está representado por el vector de posición $\bar{p}_f$, el objetivo de la luz o lugar hacia dónde apunta la fuente, es el vector de posición $\bar{t}_f$.

Además de incluir la atenuación de la distancia entre la fuente y el objeto, en la fuente spot existe la atenuación respecto al ángulo de apertura interno θ y externo ϕ. Esto indica que el objeto que se encuentre dentro del ángulo interno no sufrirá ninguna atenuación por parte del cono. Si el objeto se encuentra entre el ángulo interno y el externo será afectado por la atenuación del cono. Y si el objeto está fuera del ángulo externo el único modelo de iluminación que puede afectarlo es el ambiental.

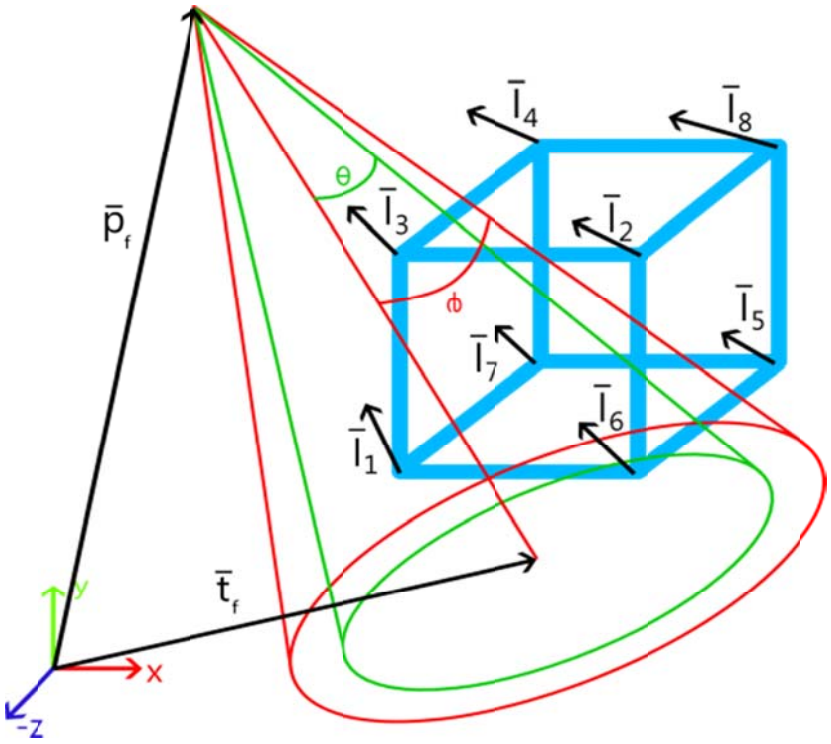

Ilustración 7-23 Fuente de iluminación spot

La manera de saber qué vértice o píxel debe ser afectado por la fuente, es por medio del ángulo que existe entre la normal y el vector de dirección de la fuente hacia el vértice o píxel. En la Ilustración 7-24 éste ángulo es α, el cual es menor cuando el vértice o píxel se encuentran dentro del cono de iluminación y, mayor cuando esté fuera.

Para comparar ángulos entre vectores se echara mano de la función trigonométrica coseno y la relación que existe entre ésta y el producto punto.

Las condiciones para conocer qué parte del objeto deben ser iluminados, atenuados o ignorados es la siguiente[35]:

$$AtenuaciónAngular = \begin{cases} 0.0 & \cos\alpha \leq \cos\varphi \\ \left(\dfrac{\cos\alpha - \cos\varphi}{\cos\theta - \cos\varphi}\right)^{falloff} & \cos\varphi < \cos\alpha < \cos\theta \\ 1.0 & \cos\alpha \geq \cos\theta \end{cases}$$

[35] http://wiki.gamedev.net/index.php/D3DBook:%28Lighting%29_Direct_Light_Sources

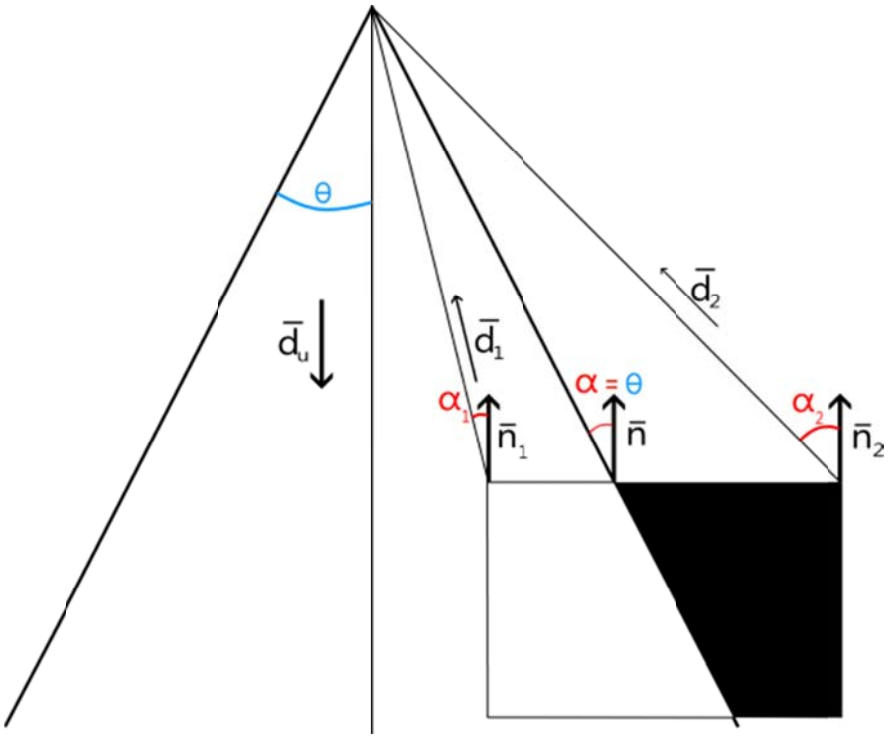

Ilustración 7-24 Fuente spot

Y la ecuación del modelo de iluminación quedaría expresada como:

$$I = I_a k_a + min\left(\frac{1}{c_1 + c_2 d_l + c_3 d^2{}_l}, 1\right) I_f k_d max(\bar{n} \cdot \bar{l}, 0) AtenuaciónAngular$$

Tómese en cuenta que la función coseno tiene su valor máximo cuando el ángulo es cero y menor cuando es $\frac{\pi}{2}$ radianes, esto por considerar que también es una fuente autoocluyente. Así que la anterior condición acerca de la atenuación es correcta.

El exponente **falloff** es un valor tipo flotante que regularmente debe tomar valores positivos, en la Ilustración 7-25 se tiene una gráfica de la atenuación angular, un nombre que de ahora en adelante le he propuesto para diferenciarla de la atenuación de distancia.

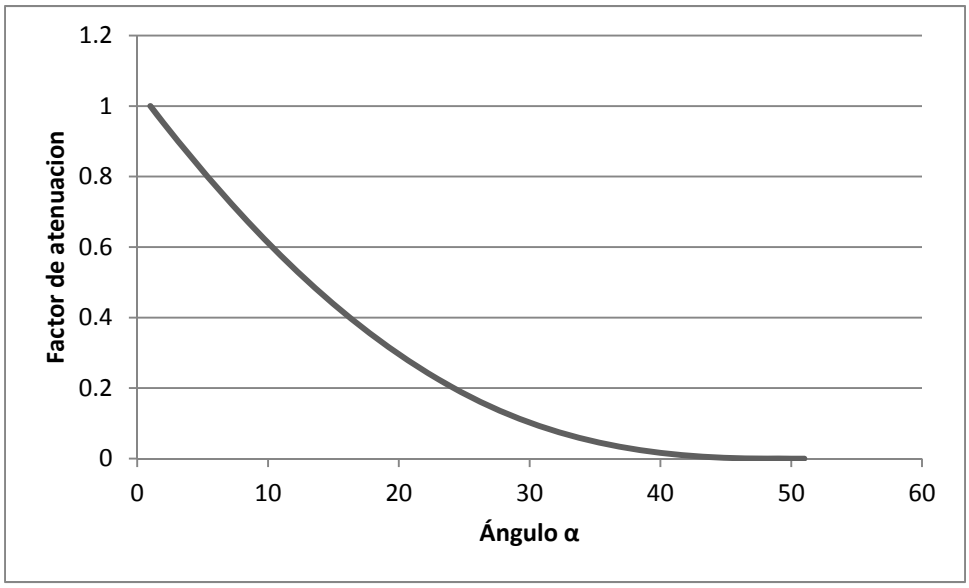

Ilustración 7-25 Factor de atenuación angular. falloff = 3, θ = 10°, ϕ = 15°

El Código 7-5 tiene muchas cosas ya vistas en el primer ejemplo, Fuente de iluminación direccional, se ha suprimido la variable uniform **direccionLuz**, y se ha modificado la subrutina **MainPS**.

Las variables globales nuevas son los ángulos **interno** y **externo**, líneas 15 y 28 respectivamente, estas son de tipo **float**, y es importante que los valores que se den a estas variables sea en radianes, la API que es la encargada de ofrecer dichas entradas extras de información, ofrecerá los datos íntegros para el shader.

La posición y el blanco de la luz spot, líneas 19 y 33, también son variables uniform que complementan los datos de entrada para el shader; ambos son de tipo **float3**.

El exponente de atenuación angular **falloff**, línea 24, las constantes de atenuación de distancia, línea 49, y el rango línea 50, son variables de tipo flotante que corresponden a las ecuaciones de atenuación, todas ellas son las nuevas entradas de información por parte del usuario.

En la subrutina **MainPS**, líneas 103 – 162, se hacen las operaciones correspondientes para cada píxel. Lo primero que se hace es declarar la variable **color**, línea 105, para almacenar el resultado obtenido de cada uno de los modelos de iluminación.

El vector unitario $\bar{d}_u$, indica la dirección de la fuente spot, por lo tanto es la resultante de la resta entre el vector de posición de luz y el vector de posición del blanco de ésta. En la línea 106 se asigna dicha operación a la variable **direccionSpot**.

La dirección de la luz respecto al píxel es el vector unitario de la resta del vector de posición de luz menos la posición del píxel, línea 108, que se almacena en la variable **direccionLuz**. Esta variable sirve para calcular la intensidad de la iluminación difusa, líneas 109 – 112.

Los cosenos de los ángulos internos y externos bien pudieron haberse calculado por parte de la API. Pero aquí se calculan con la función intrínseca **cos**, a cada ángulo, líneas 118 y 119, y se almacenan en nuevas variables tipo **float**, este exceso de declaraciones de variables sólo es para fines didácticos, por lo que la optimización del código se deja al lector como ejercicio. El coseno del ángulo α es el producto punto entre **direccionSpot** y **direccionLuz**, recordando hacer autoocluyente se selecciona el valor máximo entre cero y el producto escalar, línea 117.

El condicional **if**, líneas 122 – 126, hacen la segunda evaluación del condicional de atenuación ángular, es decir, se pregunta si el ángulo α se encuentra entre el ángulo interno y el externo, en dado caso que así sea se ejecuta la operación de atenuación angular y se multiplica el resultado por la variable color para luego volverla a asigna a la misma, líneas 124 y 125.

Si el ángulo α no se encuentra entre θ y ϕ, se pregunta si es mayor que ϕ, en caso que así sea el factor de atenuación es cero y se le multiplica la variable color para luego volver a asignarle la resultante al mismo color, línea 129.

Código 7-5

```
1.      /*
2.      Email: xintalalai@live.com.mx
3.      Autor: Carlos Osnaya Medrano
4.
5.      Modelo de iluminación: Especular o Difusa.
6.      Tipo de fuente: Spot.
7.      Técnica empleada: PixelShader.
8.      Material clásico.
9.      */
10.
11.     float4x4 world: World;
12.     float4x4 view : View;
13.     float4x4 projection : Projection;
14.
15.     float anguloInterno
16.     <
17.         string UIName = "Ángulo interno";
18.     >;// ángulo interno en radianes
19.     float3 blancoLuz
20.     <
21.         string UIName = "Blanco de la luz";
22.     >;// hacia dónde ilumina la luz
23.
24.     float falloff
25.     <
26.         string UIName = "Fallof";
27.     >;// exponente de atenuación
28.     float anguloExterno
29.     <
30.         string UIName = "Ángulo externo";
31.     >;// ángulo externo, que se toma de referencia para atenuar la luz
32.
33.     float3 posicionLuz
34.     < string UIName = "Posición de la fuente";
35.     >;
36.
37.     float4 colorMaterialAmbiental        // color de la luz ambiental
38.     <
39.         string UIName = "Luz Ambiental";
40.         string UIWidget = "Color";
41.     >  = {0.07F, 0.07F, 0.07F, 1.0F};
42.
43.     float4 colorMaterialDifuso    // color de la luz Difusa
44.     <
45.         string UIName = "Luz difusa";
46.         string UIWidget = "Color";
47.     > = {0.24F ,0.34F, 0.39F, 1.0F};
48.
49.     float c1, c2, c3;     // constantes de atenuación
50.     float rango;          // rango a evaluar la atenuación
51.
52.     texture2D texturaModelo;
53.     sampler modeloTexturaMuestra = sampler_state
54.     {
55.         Texture = <texturaModelo>;
56.         MinFilter = Linear;
57.         MagFilter = Linear;
58.         MipFilter = Linear;
59.         AddressU = Wrap;
60.         AddressV = Wrap;
61.     };
62.
63.     bool atenuado
64.     <
```

```
65.         string UIName = "Atenuado";
66.     > = false;
67.
68.     struct VertexShaderEntrada
69.     {
70.         float4 Posicion : POSITION;
71.         float3 Normal : NORMAL;
72.         float2 CoordenadaTextura : TEXCOORD0;
73.     };
74.
75.     struct VertexShaderSalida
76.     {
77.         float4 Posicion : POSITION;
78.         float2 CoordenadaTextura : TEXCOORD0;
79.         float3 WorldNormal : TEXCOORD1;
80.         float3 WorldPosition : TEXCOORD2;
81.     };
82.
83.     struct PixelShaderEntrada
84.     {
85.         float2 CoordenadaTextura : TEXCOORD0;
86.         float3 WorldNormal : TEXCOORD1;
87.         float3 WorldPosition : TEXCOORD2;
88.     };
89.
90.     VertexShaderSalida MainVS(VertexShaderEntrada entrada)
91.     {
92.         VertexShaderSalida salida;
93.         float4x4 mvp = mul(world, mul(view, projection));
94.         salida.Posicion = mul(entrada.Posicion, mvp);
95.         salida.CoordenadaTextura = entrada.CoordenadaTextura;
96.         salida.WorldNormal = mul(entrada.Normal, world);
97.         salida.WorldPosition = mul(entrada.Posicion, world);
98.
99.         return salida;
100.    }// fin del Vertex Shader MainVS
101.
102.    // Pixel Shader
103.    float4 MainPS(PixelShaderEntrada entrada, uniform bool texturizado) : COLOR
104.    {
105.        float4 color;
106.        float3 direccionSpot = normalize(posicionLuz - blancoLuz);
107.
108.        float3 direccionLuz = normalize(posicionLuz - entrada.WorldPosition);
109.        float reflexionDifusa = max((dot(direccionLuz, entrada.WorldNormal)), 0.0F);
110.        float4 difusa = colorMaterialDifuso * reflexionDifusa;
111.        // color final
112.       color = difusa;
113.        // modelo de iluminación especular
114.
115.
116.        // datos para la atenuación
117.        float cosAlfa = max(dot(direccionSpot, direccionLuz), 0.0F);
118.        float cosTeta = cos(anguloInterno);
119.        float cosFi = cos(anguloExterno);
120.
121.        // el ángulo alfa es mayor al ángulo interno y menor al ángulo externo
122.        if(cosAlfa < cosTeta && cosAlfa > cosFi)
123.        {
124.            float facAteAng = pow((cosAlfa - cosFi) / (cosTeta - cosFi), falloff);
125.            color *= facAteAng;
126.        }
127.        else if(cosAlfa <= cosFi) // el ángulo alfa es mayor al ángulo externo
128.        {
129.            color *= 0.0F;
130.        }// fin del else
131.
132.        // atenuación respecto a la distancia del foco
133.        if(atenuado)
134.        {
135.            // atenuación difusa
```

```
136.            float distancia = distance(entrada.WorldPosition,
137.            posicionLuz); //  distancia de la fuente al vértice
138.            float facAteDist;
139.            // Si la distancia entre la fuente y el vértice se encuentra entre el rango
140.            // éste se verá afectado por la atenuación, en caso contrario la
141.            // atenuación será de cero.
142.            if(distancia <= rango)
143.                facAteDist = min(1 / (c1 + (c2 * distancia) +
144.                    (c3 * pow(distancia, 2.0F))), 1.0F);
145.            else
146.                facAteDist = 0;
147.            // color final
148.            color *= facAteDist;
149.        }// fin del if
150.
151.        // color final
152.        color += colorMaterialAmbiental;
153.
154.        if(texturizado)
155.        {
156.            float4 colorTextura = tex2D(modeloTexturaMuestra,
157.                entrada.CoordenadaTextura);
158.            color *= colorTextura;
159.        }
160.        color.a = 1.0F;
161.        return color;
162.    }// fin del Pixel Shader MainPS
163.
164.    technique Texturizado
165.    {
166.        pass P0
167.        {
168.            VertexShader = compile vs_3_0 MainVS();
169.            PixelShader = compile ps_3_0 MainPS(true);
170.        }
171.    }
172.
173.    technique Material
174.    {
175.        pass P0
176.        {
177.            VertexShader = compile vs_3_0 MainVS();
178.            PixelShader = compile ps_3_0 MainPS(false);
179.        }
180.    }
```

Estas son las nuevas parte en el píxel shader **MainPS**, en comparación con los anteriores ejemplos de fuentes de iluminación.

Cabe aclarar que algunas restricciones a los datos de entrada para el shader, se pueden hacer desde la API, dejando lo más importante en el shader.

Cree un nuevo proyecto en FX Composer 2.5, para saber cómo, lea Inciando FX Composer de este texto. Transcriba el Código 7-5 y corrija cualquier error de compilación si así sucediese. Pruebe con los siguientes valores en los parámetros del shader:

- Ángulo interno: 0.01
- Blanco de la luz: 0 0 0
- Falloff: 0.8
- Ángulo externo: 0.1
- Posición de la fuente: 0 10 0
- Luz ambiental: (al gusto)
- Luz difusa: (al gusto)
- c1 : 1
- c2: 0
- c3: 0.001

- Rango: 20
- texturaModelo: (al gusto)
- Atenuado: true

Pruebe con el plano ofrecido por FX Composer en la barra de tareas, para poder observar que se crea un círculo en donde se ilumina por el spot y después solo es por la luz ambiental. Si con la textura no alcanza a distinguir muy bien está definición solo cambie el valor de los colores de los materiales ambiental o difuso, de preferencia tome un color más oscuro para el ambiental. Otra manera es ejecutar la técnica Material del shader, así que comente la técnica **Texturizado** y podrá ver algo similar a la Ilustración 7-26.

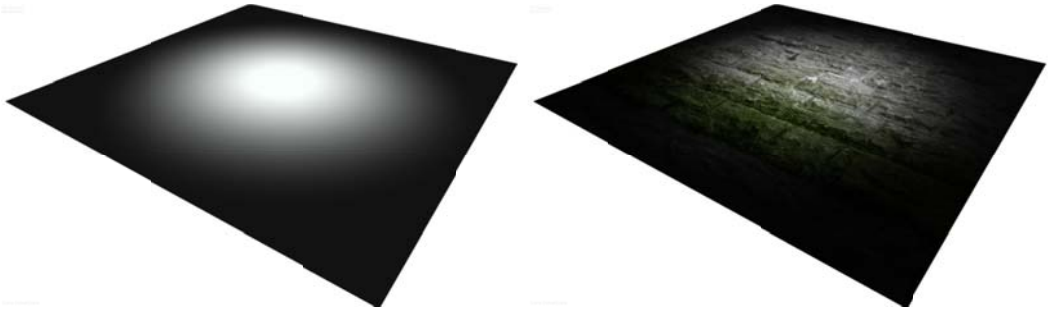

Ilustración 7-26 Fuente Spot

Pruebe con otros modelos y muévalos de lugar para ver que en realidad se trata de una fuente spot.

7.4 Iluminación especular

La iluminación especular se puede apreciar en toda superficie brillante, como los plásticos o metales. La máxima iluminación se encuentra en un punto llamado **highlight** generado por la reflexión especular, este punto de iluminación regularmente es del color de la fuente.

Otra característica de la iluminación especular es que el **highlight** se mueve cuando el observador cambia de posición, esto es como si estuviera siguiendo al observador.

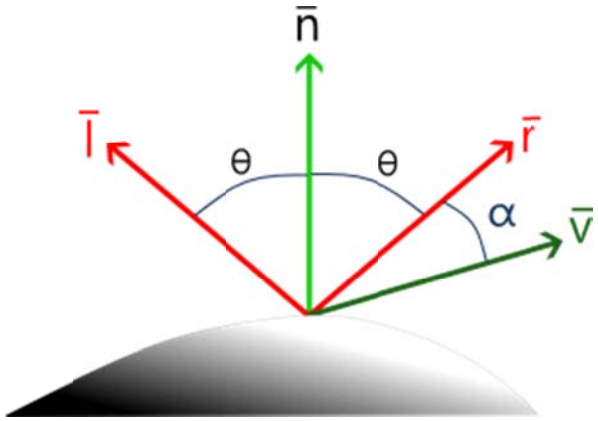

Ilustración 7-27 Reflexión especular

Además, la luz sólo se refleja en la dirección $\bar{r}$ que es la inversa del haz de luz $\bar{l}$ respecto a la normal $\bar{n}$. La intensidad de la reflexión expecular depende del ángulo α entre el vector de reflexión $\bar{r}$ y el vector de dirección del observador $\bar{v}$. Esto indica que el valor máximo de reflectancia se tiene cuando α es cero y decrece cuando aumenta. La caída es dada aproximadamente por:

$$\cos^n \alpha$$

Donde **n** es el exponente de reflexión especular del material. El rango de valores que toma **n** regularmente es de uno a varios cientos, dependiendo el material.

Este término manejado por Warnock, suponía que la reflexión estaba en la misma posición que el punto del observador, por lo que no tenía un resultado apropiado, y no fue sino hasta que Phong Bui – Tuong considero al observador y las luces en diferentes posiciones.

La característica que añadió Phong es que la luz reflejada especularmente depende del ángulo de incidencia θ. La fracción de luz reflejada suele asignarse como la constante k_s, coeficiente de reflexión especular del material, que regularmente varía entre 0 y 1. Así el modelo de iluminación queda completo con la reflexión especular:

$$I = I_a k_a + I_f k_d max(\bar{n} \cdot \bar{l}, 0) + \ k_s \cos^n \alpha$$

El cálculo del vector de reflexión $\bar{r}$, puede obtenerse a partir de la Ilustración 7-28, como la suma del vector auxiliar $\bar{a}$ más el vector $\lambda\bar{n}_u$, conocido como componente vectorial de $\bar{l}$ sobre $\bar{n}$, donde λ es un escalar y $\bar{n}_u$ es el vector unitario de $\bar{n}$.

$$\bar{r} = \bar{a} + \lambda\bar{n}_u \qquad (1)$$

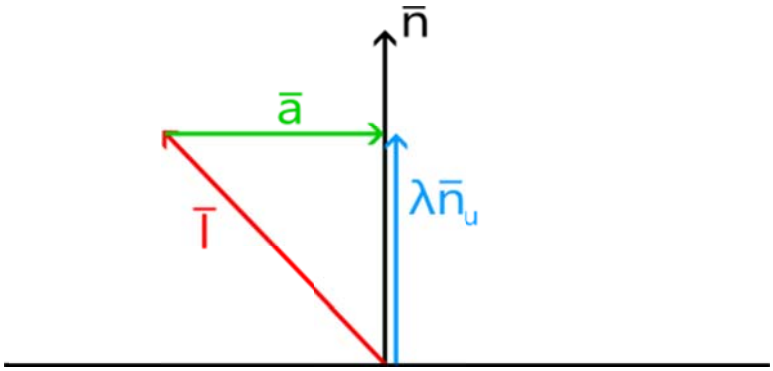

Ilustración 7-28 Vector auxiliar

Dado que el vector $\bar{a}$ es el resultado de la resta entre $\lambda\bar{n}_u$ menos el vector de dirección del haz de luz $\bar{l}$.

$$\bar{a} = \lambda\bar{n}_u - \bar{l} \qquad (2)$$

El escalar λ es el componente escalar de $\bar{l}$ sobre $\bar{n}$, que está dado por la siguiente expresión:

$$\lambda = \frac{\bar{l}\cdot\bar{n}}{|\bar{n}|} \qquad (3)$$

Sustituyendo la ecuación (3) en (2) se tiene:

$$\bar{a} = \left(\frac{\bar{l}\cdot\bar{n}}{|\bar{n}|}\right)\frac{\bar{n}}{|\bar{n}|} - \bar{l} \qquad (4)$$

Una vez sustituyendo las ecuaciones (3) y (4) en (1), el vector de reflexión $\bar{r}$ estará dado por la siguiente expresión:

$$\bar{r} = \left(\frac{\bar{l}\cdot\bar{n}}{|\bar{n}|}\right)\frac{\bar{n}}{|\bar{n}|} - \bar{l} + \left(\frac{\bar{l}\cdot\bar{n}}{|\bar{n}|}\right)\frac{\bar{n}}{|\bar{n}|}$$

$$\bar{r} = 2\left(\frac{\bar{l}\cdot\bar{n}}{|\bar{n}|}\right)\frac{\bar{n}}{|\bar{n}|} - \bar{l}$$

Como el módulo de la normal es igual a uno, el vector $\bar{r}$ estará dado por:

$$\bar{r} = 2(\bar{l}\cdot\bar{n})\bar{n} - \bar{l}$$

Una vez más reescribiendo la ecuación del modelo de iluminación se tiene en términos de la normal y el haz de luz.

$$I = I_a k_a + I_f k_d max(\bar{n}\cdot\bar{l}, 0.0) + \ k_s \max\left(\left(2(\bar{l}\cdot\bar{n})\bar{n} - \bar{l}\right)\cdot\bar{v}, 0.0\right)^n$$

La manera de seleccionar el valor adecuado para el exponente de reflexión especular y el coeficiente de reflexión especular es graficar $k_s \cos^n \alpha$, como se muestra en la Ilustración 7-29. El exponente hace que la reflexión sea suave o puntual; el coeficiente especular sólo aumenta o disminuye la intensidad de la reflexión.

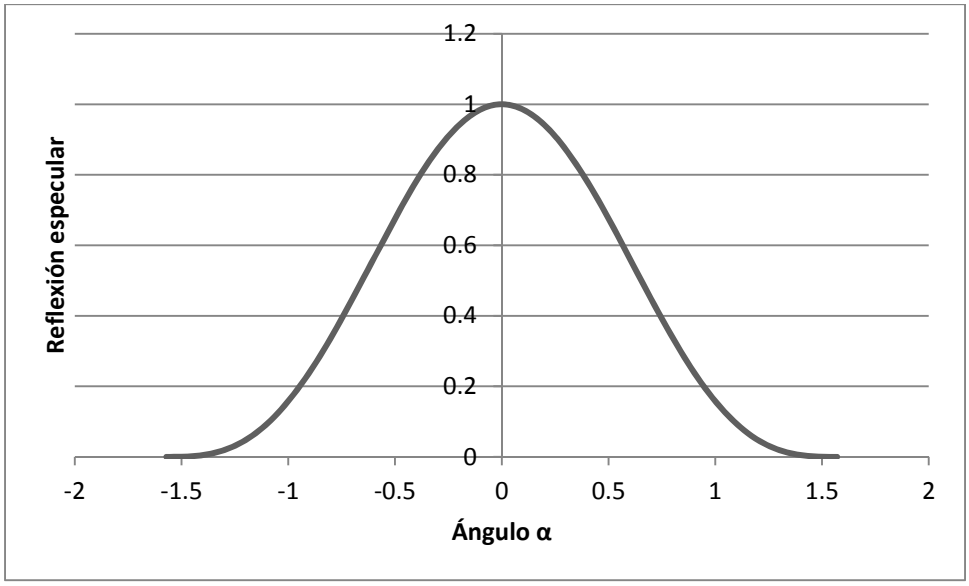

Ilustración 7-29 Reflexión especular n = 3

Como hasta ahora, se reescribirá la ecuación del modelo de iluminación para la implementación del shader. La intensidad de iluminación y el coeficiente de reflexión ambiental se sustituyen por el color ambiental del material; la intensidad de iluminación de la fuente y el coeficiente de reflexión difusa del material se sustituye por el color difuso del material; y se añade el color especular del material, dando como resultado la siguiente expresión:

$$I = colorMaterialAmbiental + colorMaterialDifusa\left(max(\bar{n}\cdot\bar{l}, 0.0)\right) + k_s colorMaterialEspecular\left(max\left(\left(2(\bar{l}\cdot\bar{n})\bar{n}-\bar{l}\right)\cdot\bar{v}, 0.0\right)^n\right)$$

Como la reflexión especular es el último modelo de iluminación que se suma, se reutilizaran los ejemplos de las fuentes de iluminación de la reflexión difusa anteriores.

7.4.1 Reflexión especular. Actualizando códigos de fuentes de iluminación

En los enlistados de los ejemplos Fuente de Iluminación Direccional, Código 7-3, Puntal, Código 7-4, y Spot, Código 7-5, vistos anteriormente se debe agregar las siguientes variables globales para la reflexión especular.

Código 7-6

```
1.      float ks
2.      <
3.          string UIWidget = "slider";
4.          float UIMin = 0.0F;
5.          float UIMax = 10.0F;
6.          float UIStep = 0.05F;
7.          string UIName = "Coeficiente de reflexión especular";
8.      > = {0.05F};
9.
10.     float n
11.     <
12.         string UIWidget = "slider";
13.         float UIMin = 0.0F;
```

```
14.         float UIMax = 128.0F;
15.         float UIStep = 1.0F;
16.         string UIName = "Exponente de reflexión especular";
17.     > = 5.0F;
18.     float4 colorMaterialEspecular
19.     <
20.         string UIName = "Material especular";
21.         string UIWidget = "Color";
22.     > = {1.0F, 1.0F, 1.0F, 1.0F};
23.     float3 posicionCamara
24.     <
25.         string UIName = "Posición cámara";
26.     >;
27.     bool modIluEsp
28.     <
29.         string UIName = "Modelo Iluminación Especular";
30.     > = false;
```

En la declaración del coeficiente y exponente de reflexión especular, líneas 1 – 8 y 10 – 17 respectivamente, se hace uso de una nueva anotación. Ésta sirve para hacer aparecer un control **TrackBar** horizontal, o **Slider**, en la GUI de FX Composer, véase ilustración 7 – 31.

Ilustración 7-30 Slider de FX Composer

La declaración de este control dentro de la anotación es de tipo **string** con nombre **UIWidget** e inicializado como **"slider"**. Estas anotaciones deben escribirse como están definidas en el Standard Annotations and Semantics (SAS) [36]. Las siguientes anotaciones después de la declaración del control son los parámetros de éste, como es el valor mínimo, máximo y el paso al cual cambiará el slider. Aunque no es necesario escribir dichas anotaciones a las variables globales, ha sido menester hacerlo en FX Composer para controlar los valores de una manera más sencilla.

La variable **modIluEsp** de tipo **bool**, línea 27, es para activar la reflexión especular en el condicional **if**, por default estará inhabilitada para disminuir el cálculo por píxel. Algunos recomiendan habilitar e inhabilitar este modelo de iluminación cuándo sea necesario.

En cada uno de los ejemplos Fuentes de Iluminación se dejó un comentario que dice **// modelo de iluminación especular**, enseguida debe escribir el siguiente código para las operaciones de reflexión especular.

Código 7-7

```
1.      if(modIluEsp)
2.      {
3.          // iluminación especular
4.          float3 reflexion = normalize(2 * entrada.WorldNormal *
5.              max(dot(direccionLuz, entrada.WorldNormal), 0.0F) - direccionLuz);
6.          float3 direccionCamara = normalize(posicionCamara - entrada.WorldPosition.xyz);
7.          float intensidadEspecular = pow(max(dot(reflexion, direccionCamara), 0.0F), n);
8.          float4 especular = ks * intensidadEspecular * colorMaterialEspecular;
9.
10.         color += especular * reflexionDifusa;
11.     }// fin del if
```

En la línea 4, del Código 7-7, se calcula el vector de reflexión especular, sólo hay que recordar que para la fuente direccional se debe multiplicar la dirección de la luz por menos uno.

```
float3 reflexion = normalize(2 * entrada.WorldNormal *
```

[36] Para mayor información acerca de las anotaciones usando SAS consulte la siguiente dirección electrónica: http://developer.nvidia.com/object/using_sas.html

```
max(dot(-direccionLuz, entrada.WorldNormal), 0.0F) + direccionLuz);
```

La dirección del observador, representado en las ecuaciones como el vector $\bar{v}$, es el vector unitario de la resta entre el vector de posición de la cámara menos el vector de posición del píxel, línea 6.

Por último, se hace uso de la reflexión especular para obtener el modelo de iluminación completo, líneas 7 – 10. La multiplicación de la reflexión difusa por la reflexión especular, línea 8, limita el brillo a la parte iluminada por la fuente en cuestión, si se omite dicha operación el objeto seguiría brillando en las partes no iluminadas.

Vuelva a compilar los shaders en FX Composer y juegue con los valores de la reflexión especular para obtener un material parecido al mostrado en la ilustración 7 – 32.

Ilustración 7-31 Reflexión especular

7.5 Múltiples fuentes de iluminación

Tomando como ejemplo el mundo real, la luz proviene de diferentes fuentes de iluminación como el sol, el foco, la lámpara de escritorio, la luz del monitor, etcétera. La suma de todas esas fuentes provoca diferentes tipos de resultados sobre los objetos que tienen su propio color. Estos colores propios son colores de material

- Ambiental
- Difuso
- Especular

Hasta el momento el color de la fuente no afectaba el modelo de iluminación, y se limitaba a tomar el color del material para multiplicarlo por la reflexión ambiental, difusa o especular, se puede decir que el color de la fuente siempre fue blanco y reflejaba el color real del material. También se consideró una sola fuente para iluminar el modelo tridimensional, lo que en la práctica resultaría no común.

En el siguiente ejemplo se aplican los tres tipos de fuentes y de cada una de ellas existe más de una. Es decir, existirán **n** luces direccionales, **n** puntual y **n** tipo spot. El número de luces de cada tipo será el mismo para todas.

El modelo de iluminación ambiental se aplicará una vez a la geometría, por lo que no se tomará en cuenta para el cálculo de cada luz.

7.5.1 Multi-pass rending

Multi pass rendering es el proceso de renderizar diferentes atributos de nuestra escena por separado.[37]

A veces los efectos más complejos necesitan de varias pasadas para obtener el resultado deseado. Al referirse como pasada, indica que se dibuja más de una vez, y cada una de ellas contribuye a generar un efecto más elaborado y enriquecido.

En este ejemplo que sigue, se crean dos PixelShader. El primero obtiene la iluminación ambiental y el segundo hace los cálculos necesarios para sumar todas las fuentes de iluminación al render anterior.

Ilustración 7-32 Suma de imágenes

En la Ilustración 7-32 se logra ver que la técnica de multi – pass rendering es la suma de cada uno de los renderizados, o también se puede ver como una sobre posición de cada uno de ellos.

En la segunda pasada se habilita el **Alpha blending** para combinar el modelo de iluminación ambiental con la suma de las fuentes de iluminación. Recuérdese que para el blending se debe cambiar el modo de renderizado de la tarjeta gráfica, y esto se logra desde XNA o si lo prefieren desde el shader. En este caso se le dejará a XNA hacer los cambios necesarios para cambiar las propiedades del dibujado.

7.5.2 Ejemplo Multiluces

En este último ejemplo de shaders, se reutiliza parte del código de los anterios programas en HLSL, y se crea por fin estructuras que representan las fuentes de iluminación direccional, puntual y de spot. Estas estructuras permitirán generar varias fuentes de iluminación, por lo que lo hace un shader de iluminación básico lo más parecido a lo que el fixed pipeline de DirectX u OpenGL ofrecen.

La estructura de la fuente direccional **LuzDireccional**, líneas 87 -92, Código 7-8, tiene como campos la dirección de la fuente, el color y la opción de encender o apagarla.

La estructutra de la fuente puntual **LuzPuntual**, líneas 94 – 102, tiene los campos de posición de la fuente, el color, el rango de alcance, las constantes de atenuación, la opción de habilitar o inhabilitar la atenuación y la posibilidad de encender o apagar la luz.

La fuente de tipo spot representada por la estructura **LuzSpot**, líneas 104 – 116, tiene como campos la posición de la fuente, el blanco de hacia dónde apunta, el color, el ángulo interno y externo de la fuente; el exponente de atenuación de abertura de la fuente, el rango de alcance, las constantes de atenuación, la opción de habilitar o inhabilitar la atenuación por distancia de la fuente, y la posibilidad de encender o apagar la luz.

Nótese que no se ocupó el tipo de dato booleano en los campos **Encender** y **Atenuar** en las estructuras de las fuentes. Esto debido a que el compilador de shaders de Microsoft, limita a este tipo de dato a un determinado número; lo que varía entre FX Composer y XNA; por lo tanto se han dejado como tipos enteros.

El límite de operaciones por fuente se límita por la variable **numLuces**, línea 118, que sirve como variable de escape en los ciclos **for**.

Los arreglos de estructuras de las fuentes, son representadas por las variables:

[37] Traducción hecha a partir de http://www.3drender.com/light/compositing/index.html

- **direccionales[]**, línea 259
- **puntuales[]**, línea 260
- **spots[]**, línea 261

La reservación de memoria para cada una debe hacerse en tiempo de compilación, y como se había mencionado anteriormente dependerá de la versión del compilador, por lo que el número máximo permitido podría variar.

El listado presenta un vertex shader, **MainVS** ,líneas 121 – 132, que será suficiente para el multi – pass rendering. Es aquí en donde se harán las transformaciones que hace el fixed pipeline, cosa que en ejemplos anteriores ya se explicó. Véase que no existe una estructura como parámetro en el vertex shader, por lo que cada variable de entrada desde XNA, se especifica su semántica.

La caraterística particular de un multi – pass rendering, es que se tienen diferentes pixel shaders, para cada pasada del shader, y rara vez diferentes vertex shaders. En este ejemplo se tienen dos pixel shaders, **MainPSAmbiental**, líneas 135 – 144, y **MainPS**, líneas 265 - 296. El primero representa el modelo de iluminación ambiental y el segundo los modelos de iluminación difuso o especular con la suma de cada una de las fuentes.

En el pixel shader **MainPSAmbiental**, se multiplica el color del material ambiental por la textura si se ha habilitado la texturización, en caso contrario se pasa solo el color.

En el pixel shader **MainPS** se multiplica el color difuso por la textura, si ésta se ha habilitado. Luego se pasa por un bucle **for** para calcular cada una de las fuentes que estén encendidas. Esto se logra con tres condicionales **if**, cada uno verifica si el campo **Encender** es diferente de cero, en cada una de las distintas fuentes. Dependiendo el tipo de fuente se realizan las operaciones correspondientes con ayuda de de tres funciones: **FuenteDireccional**, **FuentePuntual** y **FuenteSpot**. Cada una de ellas devuelve un color que será sumado en una variable local en el **MainPS** y ese será el color final.

La función **FuenteDireccional**, líneas 192 – 199, toma como parámetros la estructura **LuzDireccional**, la posición del vértice, la normal y el color del material, éste último es el color de la suma del color difuso multiplicado por el de textura. Como todas las funciones hacen uso de los cálculos del modelo de reflexión Phong, se ha escrito una función que devuelve el color de dicho modelo. La función, con el mismo nombre del modelo, líneas 176 – 189, toma como entradas la dirección de la luz, la posición del vértice, la normal, el color de luz y el color del material. Las operaciones son las necesarias para obtener la reflexión difusa y la reflexión especular, siempre y cuando esté habilitado. Tomando el principio de divide y venceras, se ha construido una función llamada **ReflexionEspecular**, para hacer legible el programa. La función devuelve el color especular y toma como parámetros la dirección de luz, la posición del vértice, la normal y el color de luz.

Las operaciones que realizan las funciones **Phong** y **ReflexiónEspecular**, ya se explicaron en los ejemplos anteriores, por lo que si el lector tiene dudas por favor de regrese a consultarlos los apartados: Iluminación difusa e Iluminación especular.

La función **FuentePuntual**, líneas 201 – 217, toma como parámetros la estructura **LuzPuntual**, la posición del vértice, la normal y el color del material. La única entrada que cambia cuando se llama a la función Phong, es la dirección de luz, véase línea 205. Además esta fuente de iluminación puede atenuarse dependiendo la distancia que se encuentre el vértice de la fuente y del campo que habilita dicha operación, líneas 210 – 214. Como la fuente spot también puede ser atenuada de esta forma, se ha escrito una función que devuelve un escalar que indica la atenuación de la luz. En la líneas 147 – 161, la función **AtenuarFuente** realiza los cálculos necesarios, y ya vistos en ejemplos anteriores, que permiten disminuir la intensidad de luminancia de la fuente. Las entradas de esta función son la posición del vértice, la posición de la luz, el rango en que se permite la atenución y las constantes de atenuación.

La función **FuenteSpot**, líneas 219 – 258, toma prácticamente el código del shader descrito en el tema Fuente de iluminación spot, con la diferencia que separá los cálculos necesarios para limitar la iluminación, y los cálculos del modelo iluminación Phong.

Las técnicas, líneas 298 -325, **Texturizado** y **Material**, ambas tiene dos pasadas llamadas **Ambiental** y **MultiLuces**. La primera llama el vertex shader **MainVS** y el pixel shader **MainPSAmbiental**. Esta primera pasada sólo colorea el modelo tridemensional con el color ambiental, y dependiendo de la técnica habilita o inhabilita la textura.

En la segunda pasada vuelve a llamar al vertex shader **MainVS**, pues es necesario realizar el renderizado una vez más; y como pixel shader se llama **MainPS** que permite la suma de varias fuentes de luz en el mismo píxel.

Código 7-8

```
1.      /*
2.      Email: xintalalai@live.com.mx
3.      Autor: Carlos Osnaya Medrano
4.      Multiples luces.
5.      Modelo de iluminación: Especular o Difusa.
6.      Tipo de fuente: Direccional, Puntual y Spot.
7.      Técnica empleada: PixelShader.
8.      Material clásico.
9.      */
10.
11.     float4x4 world: World;
12.     float4x4 view : View;
13.     float4x4 projection : Projection;
14.
15.     float3 posicionCamara
16.     <
17.         string UIName = "Posición cámara";
18.     >;
19.
20.     float ks
21.     <
22.         string UIWidget = "slider";
23.         float UIMin = 0.0F;
24.         float UIMax = 10.0F;
25.         float UIStep = 0.05F;
26.         string UIName = "Coeficiente de reflexión especular";
27.     > = {0.05F};
28.
29.     float n
30.     <
31.         string UIWidget = "slider";
32.         float UIMin = 0.0F;
33.         float UIMax = 128.0F;
34.         float UIStep = 1.0F;
35.         string UIName = "Exponente de reflexión especular";
36.     > = 5.0F;
37.
38.     float4 colorMaterialAmbiental
39.     <
40.         string UIName = "Material ambiental";
41.         string UIWidget = "Color";
42.     >  = {0.07F, 0.07F, 0.07F, 1.0F};
43.
44.     float4 colorMaterialDifuso
45.     <
46.         string UIName = "Color Material";
47.         string UIWidget = "Color";
48.     > = {0.24F ,0.34F, 0.39F, 1.0F};
49.
50.     float4 colorMaterialEspecular
51.     <
52.         string UIName = "Material especular";
53.         string UIWidget = "Color";
54.     > = {1.0F, 1.0F, 1.0F, 1.0F};
55.
56.     bool habiEspec
57.     <
58.         string UIName = "Modelo Iluminación Especular";
```

```
59.     > = false;
60.
61.     texture2D texturaModelo;
62.     sampler modeloTexturaMuestra = sampler_state
63.     {
64.         Texture = <texturaModelo>;
65.         MinFilter = Linear;
66.         MagFilter = Linear;
67.         MipFilter = Linear;
68.         AddressU = Wrap;
69.         AddressV = Wrap;
70.     };
71.
72.     struct VertexShaderSalida
73.     {
74.         float4 Posicion : POSITION;
75.         float2 CoordenadaTextura : TEXCOORD0;
76.         float3 WorldNormal : TEXCOORD1;
77.         float3 WorldPosition : TEXCOORD2;
78.     };
79.
80.     struct PixelShaderEntrada
81.     {
82.         float2 CoordenadaTextura : TEXCOORD0;
83.         float3 WorldNormal : TEXCOORD1;
84.         float3 WorldPosition : TEXCOORD2;
85.     };
86.
87.     struct LuzDireccional
88.     {
89.         float3 Direccion;
90.         float4 Color;
91.         int Encender;
92.     };
93.
94.     struct LuzPuntual
95.     {
96.         float3 Posicion;
97.         float4 Color;
98.         float Rango;
99.         float C1, C2, C3;
100.        int Atenuar;
101.        int Encender;
102.    };
103.
104.    struct LuzSpot
105.    {
106.        float3 Posicion;
107.        float3 Blanco;
108.        float4 Color;
109.        float AnguloInterno;
110.        float AnguloExterno;
111.        float Falloff;
112.        float Rango;
113.        float C1, C2, C3;
114.        int Atenuar;
115.        int Encender;
116.    };
117.
118.    int numLuces = 4;
119.
120.    // Vertex Shader
121.    VertexShaderSalida MainVS(float3 posicion : POSITION,
122.    float3 normal : NORMAL, float2 coordenadaTextura : TEXCOORD0)
123.    {
124.        VertexShaderSalida salida;
125.        float4x4 mvp = mul(world, mul(view, projection));
126.        salida.Posicion = mul(float4(posicion, 1.0F), mvp);
127.        salida.WorldNormal = mul(normal, world);
128.        salida.WorldPosition = mul(float4(posicion, 1.0F), world);
129.        salida.CoordenadaTextura = coordenadaTextura;
```

```
130.
131.        return salida;
132.    }// fin del vertexshader MainVS
133.
134.    // Pixel Shader que obtiene la iluminación ambiental, con o sin textura
135.    float4 MainPSAmbiental(PixelShaderEntrada entrada,
136.        uniform bool habiTextura) : COLOR
137.    {
138.        float4 color = colorMaterialAmbiental;
139.        if(habiTextura)
140.        {
141.            color *=  tex2D(modeloTexturaMuestra, entrada.CoordenadaTextura);
142.        }
143.        return color;
144.    }// fin del pixelshader MainPSAmbiental
145.
146.    // Función que calcula la atenuación de la distancia de la fuente al objeto
147.    float AtenuarFuente(float3 worldPosicion, float3 posicionLuz, float rango,
148.        float c1, float c2, float c3)
149.    {
150.        float distancia = distance(worldPosicion, posicionLuz);
151.        float atenuacionD = 0;
152.        // si la distancia está dentro del rango se hace el cálculo de la
153.        // atenuación.
154.        if(distancia <= rango)
155.        {
156.            atenuacionD = min(1 / (c1 + (c2 * distancia) +
157.            (c3 * pow(distancia, 2.0F))), 1.0F);
158.        }// fin del if
159.
160.        return atenuacionD;
161.    }// fin de la función AtenuarFuente
162.
163.    // Función que calcula la reflexión especular.
164.    float4 ReflexionEspecular(float3 direccionLuz, float3 worldPosicion,
165.        float3 worldNormal, float4 colorLuz)
166.    {
167.        float3 reflexion = normalize(2 * worldNormal *
168.        max(dot(direccionLuz, worldNormal), 0.0F) - direccionLuz);
169.
170.        float3 direccionCamara = normalize(posicionCamara - worldPosicion);
171.        float reflexionEspecular = pow(max(dot(reflexion, direccionCamara), 0.0F), n);
172.
173.        return (ks * reflexionEspecular) * (colorLuz * colorMaterialEspecular);
174.    }// fin de la función ReflexionEspecular
175.
176.    float4 Phong(float3 direccionLuz, float3 worldPosicion, float3 worldNormal,
177.     float4 colorLuz, float4 colorMater)
178.    {
179.        float reflexionDifusa = max(dot(direccionLuz, worldNormal), 0.0F);
180.        float4 phong = reflexionDifusa * (colorLuz * colorMater);
181.        // activación de la reflexión especular
182.        if(habiEspec)
183.        {
184.            phong += ReflexionEspecular(direccionLuz, worldPosicion,
185.                worldNormal, colorLuz) * colorMater * reflexionDifusa;
186.        }// fin del if
187.
188.        return phong;
189.    }// fin de la función Phong
190.
191.    // Función que calcula la luz tipo direccional
192.    float4 FuenteDireccional(LuzDireccional luz, float3 worldPosicion,
193.        float3 worldNormal, float4 colorMater)
194.    {
195.        return Phong(-normalize(luz.Direccion), // dirección de luz
196.            worldPosicion,
197.            worldNormal, luz.Color,
198.            colorMater);
199.    }// fin de la función FuenteDireccional
200.
```

```
201.    float4 FuentePuntual(LuzPuntual luz, float3 worldPosicion, float3 worldNormal,
202.        float4 colorMater)
203.    {
204.        float4 color = Phong(
205.            normalize(luz.Posicion - worldPosicion),// dirección de luz
206.            worldPosicion,
207.            worldNormal, luz.Color,
208.            colorMater);
209.        // atenuar luz
210.        if(luz.Atenuar != 0)
211.        {
212.            color *= AtenuarFuente(worldPosicion, luz.Posicion, luz.Rango,
213.                luz.C1, luz.C2, luz.C3);
214.        }// fin del if
215.
216.        return color;
217.    }// fin de la función FuentePuntual
218.
219.    float4 FuenteSpot(LuzSpot luz, float3 worldPosicion, float3 worldNormal,
220.        float4 colorMater)
221.    {
222.        float4 color = 0;
223.        float3 direccionLuz = normalize(luz.Posicion - worldPosicion);
224.        float3 direccionSpot = normalize(luz.Posicion - luz.Blanco);
225.        // calculando cosenos de alfa, teta y fi
226.        float cosAlfa = max(dot(direccionSpot, direccionLuz), 0.0F);
227.        float cosTeta = cos(luz.AnguloInterno);
228.        float cosFi = cos(luz.AnguloExterno);
229.
230.        // atenuación angular entre 0 y 1
231.        if(cosAlfa < cosTeta && cosAlfa > cosFi)
232.        {
233.            float atenuacionAngular = pow((cosAlfa - cosFi) / (cosTeta - cosFi),
234.                luz.Falloff);
235.            color = atenuacionAngular * Phong(direccionLuz, worldPosicion,
236.                worldNormal, luz.Color,
237.                colorMater);
238.            // atenuar fuente
239.            if(luz.Atenuar != 0)
240.            {
241.                color *= AtenuarFuente(worldPosicion, luz.Posicion, luz.Rango,
242.                    luz.C1, luz.C2, luz.C3);
243.            }// fin del if
244.        }// fin del if
245.        else if(cosAlfa >= cosFi) // atenuación angular igual a uno
246.        {
247.            color = Phong(direccionLuz, worldPosicion, worldNormal,
248.                luz.Color, colorMater);
249.            // atenuar fuente
250.            if(luz.Atenuar)
251.            {
252.                color *= AtenuarFuente(worldPosicion, luz.Posicion, luz.Rango,
253.                    luz.C1, luz.C2, luz.C3);
254.            }
255.        }// fin del if
256.
257.        return color;
258.    }// fin de la función FuenteSpot
259.
260.    LuzDireccional direccionales[2];
261.    LuzPuntual puntuales[2];
262.    LuzSpot spots[2];
263.
264.
265.    float4 MainPS(PixelShaderEntrada entrada, uniform bool texturizado) : COLOR
266.    {
267.        float4 color = 0;
268.        float4 colorDifuso = colorMaterialDifuso;
269.
270.        if(texturizado)
271.        {
```

```
272.            colorDifuso *= tex2D(modeloTexturaMuestra, entrada.CoordenadaTextura);
273.        }// fin del if
274.
275.        for(int i = 0; i < numLuces; i++)
276.        {
277.            if(direccionales[i].Encender != 0)
278.            {
279.                color +=  FuenteDireccional(direccionales[i], entrada.WorldPosition,
280.                    entrada.WorldNormal, colorDifuso);
281.            }
282.            if(puntuales[i].Encender != 0)
283.            {
284.                color += FuentePuntual(puntuales[i], entrada.WorldPosition,
285.                    entrada.WorldNormal, colorDifuso);
286.            }
287.            if(spots[i].Encender != 0)
288.            {
289.                color += FuenteSpot(spots[i], entrada.WorldPosition,
290.                    entrada.WorldNormal, colorDifuso);
291.            }
292.
293.        }// fin del for
294.        color.a = 1.0F;
295.        return color;
296.    }// fin del pixelShader mainPS
297.
298.    technique Texturizado
299.    {
300.        pass Ambiental
301.        {
302.            VertexShader = compile vs_3_0 MainVS();
303.            PixelShader = compile ps_3_0 MainPSAmbiental(true);
304.        }
305.        pass MultiLuces
306.        {
307.            VertexShader = compile vs_3_0 MainVS();
308.            PixelShader = compile ps_3_0 MainPS(true);
309.        }
310.    }// fin de la técnica Texturizado
311.
312.    technique Material
313.    {
314.        pass Ambiental
315.        {
316.            VertexShader = compile vs_3_0 MainVS();
317.            PixelShader = compile ps_3_0 MainPSAmbiental(false);
318.        }
319.
320.        pass MultiLuces
321.        {
322.            VertexShader = compile vs_3_0 MainVS();
323.            PixelShader = compile ps_3_0 MainPS(false);
324.        }
325.    }// fin de la técnica Texturizado
```

Creé un nuevo proyecto en FX Composer y transcriba el enlistado anterior para crear un efecto en HLSL. Compile y añada el nuevo material a un modelo que FX Composer ofrece, de preferencia la tetera o esfera. En la parte de Parameters: **HLSL Profile**, se muestran todas las propiedades de entrada del shader. Los campos de las estructuras de las fuentes de iluminación no pueden tener anotaciones que habiliten la paleta de colores o los sliders, por lo tanto hay que escribir el valor numérico con el teclado.

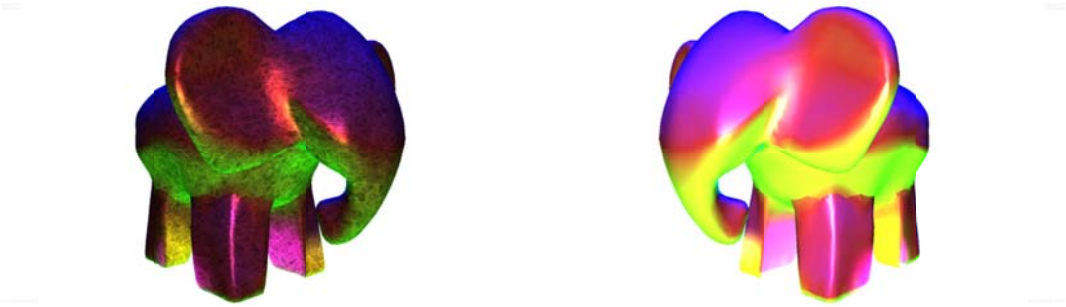

Ilustración 7-33 Multi pass rendering

Desafortunadamente FX Composer no puede hacer correr múltiples pasadas, o por lo menos en algunas tarjetas gráficas, así que toma la última que se encuentra en la técnica. En la Ilustración 7-33 se muestra el resultado sin aplicar la pasada **Ambiental**, con textura y sin textura.

En el siguiente capítulo se explicará cómo agregar los shaders creados en este apartado en XNA. Y se deja al lector estudiar acerca de este lenguaje, pues está fuera del alcance de este texto.