Universidad Nacional Autónoma de México

Facultad de Ingeniería

Elaboración de un manual para programar en XNA Game Studio 3.1

Tesis

Que para obtener el título de:

Ingeniero en Computación

Presenta:

Carlos Osnaya Medrano

Ciudad Universitaria, México D.F., junio de 2011

1 Contenido

Objetivo

Debido a la falta de textos en español sobre programación orientada a gráficos 3D y al rezago que se tiene en los libros de tecnología, por su rápido cambio. Se elaborará material bibliográfico para la creación de programas en 3D, utilizando XNA Game Studio 3.1; como apoyo educativo para el estudiante de Computación Gráfica; así como motivar, enseñar, y generar el interés en la producción de programas interactivos, como pueden ser los videojuegos.

Parte I

1 Introducción

En el mercado de los videojuegos, México ha comenzado a crecer demasiado en estos últimos tres años que lo ha posicionado en el cuarto lugar internacionalmente y el primero en Latinoamérica, pero sólo como consumidor. En el año 2010 tuvo un valor de 12, 857 millones de pesos.

Este nicho comenzó a crecer debido al aumento de nuevos dispositivos multimedia y redes sociales, que han dado cabida al entretenimiento digital. Entre los diferentes dispositivos, las consolas de videojuegos han alcanzado a penetrar en los hogares mexicanos como la televisión, en especial la consola Xbox 360™ que ha ocupado el 61% de preferencia. Además, los teléfonos celulares inteligentes han comenzado a incrementarse entre la población, sin embargo, la tardía penetración del ancho de banda y el precio alto de dicho servicio a mermado el uso de la Internet en estos dispositivos, aun así, se han moldeado nuevos jugadores o *gamers*, en estos móviles.

El éxito de este mercado ha sido tal, que ha superado al cine, y la tendencia a mejorar la experiencia del usuario ha hecho que éste deje su asiento y se vuelva partícipe de lo que ve, como es el caso con Kinect™, cuyo éxito se denota en dejar a un lado el control para sustituirlo por todo el cuerpo de quien juega.

1.1 México y los videojuegos

Hablando específicamente de las consolas de videojuegos, por ser el de mayor consumo y preferencia entre los *gamers*, los videojuegos son producidos por empresas cuyos centros se encuentran en Estados Unidos de América, Japón, Canadá y Gran Bretaña; muchas de estas empresas tienen subsidiarias en otros países, en los cuales México no tiene hasta el momento alguna.

Latinoamérica ha emprendido la batalla en el mercado de los videojuegos, y no necesariamente se hace pensar que al ser México el primer consumidor en esta región es el pionero en la producción. Es Colombia y Argentina quienes han dado los primeros pasos con empresas como ©Immersion Games & Graphics y ©Sabarasa Inc. Y en comparación con los emporios de los videojuegos, existen en México anexas de estas empresas que producen videojuegos.

El Estado Mexicano ha vislumbrado la producción de software con el Programa para el Desarrollo de la Industria de Software (PROSOFT), el cual busca el desarrollo en este rubro otorgando apoyos a proyectos que sustenten las tecnologías de la información (TI). Además, la Secretaría de Economía ha lanzado la iniciativa Juego de Talento para fomentar el desarrollo de software interactivo, en especial los videojuegos, y su objetivo es descubrir las aptitudes que se encuentran en México para aterrizarlas en la creación de empresas que destinen su producto al mercado nacional e internacional.

1.2 La Interfaz de Programación de Aplicaciones XNA

XNA es conjunto de bibliotecas orientadas al desarrollo de aplicaciones interactivas, sobre la consola de videojuegos Xbox 360, equipos basados en Windows y teléfonos móviles. Las diferentes bibliotecas de XNA ofrecen operación sobre gráficos 2D, gráficos 3D, audio, video, entradas estándar, entradas no estándar y red.

XNA se integra al entorno de desarrollo de Visual Studio, ofreciendo todas las ventajas de una herramienta profesional y de misión crítica.

XNA está basado en el Framework de .NET, sin embargo, ofrece su propio Framework para poder operar entre las diferentes plataformas.

XNA es un bloque más en la estructura de .NET de Microsoft, por lo que el lenguaje de programación para trabajar es cualquiera que esté bajo las reglas del Common Language Runtime (CLR)[1], como lo es Visual Basic .NET, C# o C++ administrado. Sin embargo, la adopción de C# como lenguaje de programación

[1] Para mayor información acerca de .NET visite el sitio de Desarrollador 5 estrellas: http://www.mslatam.com/latam/msdn/comunidad/dce2005/

primario, ha sido por su sencillez y semejanza con C++, en la cual las Aplication Programming Interface (API) DirectX y OpenGL están diseñadas.

Por lo anterior XNA facilita la escritura de programas interactivos, con el menor número de líneas de codificación y la integración sencilla entre los diferentes dispositivos, lo que ha vuelto a esta API en versátil para los programadores principiantes.

2 Diseño del manual de XNA en español

2.1 Justificación

A la falta de bibliografía especializada en programación de gráficos 3D, y en idioma español. El manual de programación 3D surge como respuesta a las necesidades de volcar el conocimiento adquirido en el aula en una consola de videojuegos, PC y posteriormente en el teléfono celular. Todo eso, sin cambiar el lenguaje de programación y el entorno de desarrollo.

XNA es un conjunto de bibliotecas que permite desarrollar aplicaciones gráficas 2D y 3D, sobre las tres plataformas en que el mundo de los medios interactivos está pululando. Las herramientas son gratuitas, y la posibilidad de vender el producto final a un costo de suscripción justo, hizo atractivo la generación de un libro que trata de motivar, enseñar, y generar el interés en la producción de programas interactivos, como pueden ser los videojuegos.

2.2 Estructura

El manual está constituido por nueve capítulos, los primeros seis muestran el uso de las clases principales de dibujado en 3D en XNA, y no demuestran una gran complicación para el lector. Los últimos tres capítulos hacen uso de las clases explicadas en los seis primeros, y aumenta la complejidad del código, pues abarca conceptos matemáticos computación gráfica y un nuevo lenguaje de programación. Sin embargo, se ha tratado de explicar de la mejor manera para su comprensión.

En cada capítulo se muestra una serie de ejemplos con su código explicado línea a línea, al finalizar cada uno se muestran imágenes que ayudan a demostrar el resultado esperado. En ocasiones se deja al lector un ejercicio para que verifique los diferentes valores que pueden tomar algunos métodos, o se deja que complemente algunos ejemplos vistos con anterioridad.

En seguida se muestra un resumen de los capítulos del manual de programación en XNA.

Vertex buffer. En este capítulo se explican las clases **VertexBuffer** y **DynamicVertexBuffer**, clases que representan el búfer de vértices del dispositivo gráfico. Para mostrar las diferencias entre cada clase, se utiliza un arreglo de vértices para dibujar un triángulo.

Primitivas. Aquí se muestra cada una de las primitivas que XNA ofrece para dibujar. En cada una de ellas se utiliza un mismo arreglo de vértices para mostrar las diferencias entre cada una de ellas; a excepción de la última, en donde se le asigna al búfer de vértices otro arreglo.

Creando objeto 3D. En este capítulo se muestra cómo posicionar la cámara en el mundo tridimensional, a utilizar la entrada de datos por medio del teclado y el gamepad del Xbox 360, para trasladar, rotar y/o escalar un cubo. Este cubo es creado a partir de un arreglo de vértices y uno de índices.

Textura. Se presenta un plano, creado a partir de un arreglo de vértices, en donde texturiza una imagen. A la textura se le aplican diferentes filtros predefinidos en XNA, también se utilizan todos los modos de direccionamiento que presenta XNA.

Texto. XNA tiene dos formas de presentar texto en pantalla, una es a partir de un XML con todas las propiedades de la fuente y la otra es a partir de una imagen con todos los caracteres a mostrar.

Iluminación. Se deja a un lado XNA y se manejan los shaders, para explicar modelos de iluminación básicos con diferentes tipos de fuente de iluminación. Y se introduce al lector al lenguaje de programación sobre hardware, High Level Languaje (HLSL).

Cómo agregar un efecto en XNA. Se integran los efectos, vistos en el capítulo anterior, en una solución de XNA. Se hacen pequeños cambios a los shaders, pero solo como cuestión de ilustración; se le deja al lector tratar de entender los cambios y el crear la aplicación que totalice los efectos de iluminación en XNA.

Colisión. El último capítulo revisa las clases **BoundingSphere**, **BoundingBox** y **Ray**. Para cada una de ellas se presenta un caso de colisión entre ellas mismas, a excepción de la clase **Ray**, ya que no es una envolvente. Se crean las clases **Camara**, **Mira**, **Proyectil**, **ModeloEstatico** y **ModeloDinamico**; que complementan los tres ejemplos de colisión.

2.3 A quién va dirigido el manual de programación en XNA

El manual de programación nació con la idea de apoyar a los estudiantes que estén cursando, o hayan cursado, la materia de Computación gráfica, en la carrera de Ingeniería en computación, de la Facultad de Ingeniería en la Universidad Nacional Autónoma de México. Sin embargo, en la Internet se encontró mucho interés por parte de personas apasionadas con los gráficos, y sobre todo, con la nueva tecnología que representa XNA Game Studio.

El manual está enfocado en los gráficos 3D, por lo tanto, se recomienda que el público lector de esta obra tenga los siguientes conocimientos, solo para una mejor comprensión.

- Lenguaje de programación C#.
- Paradigma de programación orientado a objetos.
- Lenguaje de programación C.
- Geometría Analítica.
- Algebra Lineal.

Aunque la anterior lista representa un obstáculo, se ha tratado de explicar de la mejor manera los ejemplos, para aquellas que estén por involucrarse en esta materia.

2.4 Apoyos académicos y requisitos

El manual de programación estará acompañado por los programas fuentes, listos para ser ejecutados desde un inicio. También se incluirán videos demostrativos de algunos ejemplos que requieran más que una imagen.

Además se incluirá una copia de Visual Studio 2008 Express, XNA Game Studio 3.1 y FX Composer. Todo este software se puede descargar gratuitamente de las siguientes direcciones Web.

- http://www.microsoft.com/express/download/
- http://www.microsoft.com/downloads/details.aspx?FamilyID=80782277-d584-42d2-8024-893fcd9d3e82&displaylang=en
- http://developer.nvidia.com/object/fx_composer_home.html

2.4.1 Software

XNA Game Studio funciona sobre plataformas Windows, en la Tabla 2-1 se muestran las versiones de los sistemas operativos admitidos para la instalación de XNA. Se recomienda la actualización del sistema operativo

Tabla 2-1[2]

Sistema Operativo	Versiones admitidas
Windows XP	• Home Edition • Professional • Media Center Edition • Tablet PC Edition
Windows Vista	• Home Basic • Home Premium • Business • Enterprise • Ultimate
Window 7	• Home Basic • Home Premium • Professional • Enterprise • Ultimate

En el caso del Xbox 360, se debe contar con XNA Game Studio Connect. Este software se descarga desde el bazar del Xbox 360.

XNA Game Studio 3.1 necesita del entorno de desarrollo Visual Studio 2008, en cualquiera de sus versiones Express, Standard, Professional o Team System. Las especificaciones de software y hardware de cada una de ellas varían.

2.4.2 Hardware

El requisito adicional en hardware, sobre la PC, es una tarjeta gráfica que admita Shader Model 3.0 y DirectX 9.0c. Este requisito es indispensable para poder ejecutar los ejemplos mostrados en el capítulo de Iluminación. Se recomienda la actualización de los controladores de la tarjeta gráfica.

Para probar los ejemplos en el Xbox 360, se debe contar con un disco duro para su almacenamiento.

[2] Tabla de requisitos tomada de la siguiente dirección: http://msdn.microsoft.com/es-mx/library/bb203925.aspx

Parte II Manual

1 Vertex Buffer

1.1 VertexBuffer

Los modelos tridimensionales que se pueden ver en los videojuegos están conformados por un conjunto de planos triangulares en el espacio tridimensional. Cada plano está constituido por una terna de puntos y una triada de aristas. Los puntos, alejándonos del término matemático, almacenan información sobre su posición, color, textura, normal, tangente y binormal. Los puntos en el espacio, por lo menos deben contener un conjunto de tres coordenadas (x, y, z). Dado la posible información que representan estos puntos, se les ha denominado vértices.

El vertex buffer es una región de memoria, en la tarjeta gráfica, para almacenar vértices. Para establecer dichos datos en el búfer de vértices, XNA ofrece las clases **VertexBuffer** y **DynamicVertexBuffer**, esta última hereda de la primera y VertexBuffer no se recomienda para el uso sobre el Xbox360[3].

En el ejemplo siguiente, se muestra el uso del VertexBuffer para dibujar un triángulo. Comience por abrir Visual Studio y cree un nuevo proyecto; para ello, vaya al menú principal y seleccione **Archivo\Nuevo\Proyecto**. Enseguida se abrirá una ventana de diálogo para seleccionar el tipo de proyecto y plantilla instalados. En tipos de proyecto, seleccione **XNA Game Studio 3.1** y en la parte de plantillas tome **Windows Game (3.1)**, véase Ilustración 1-1.

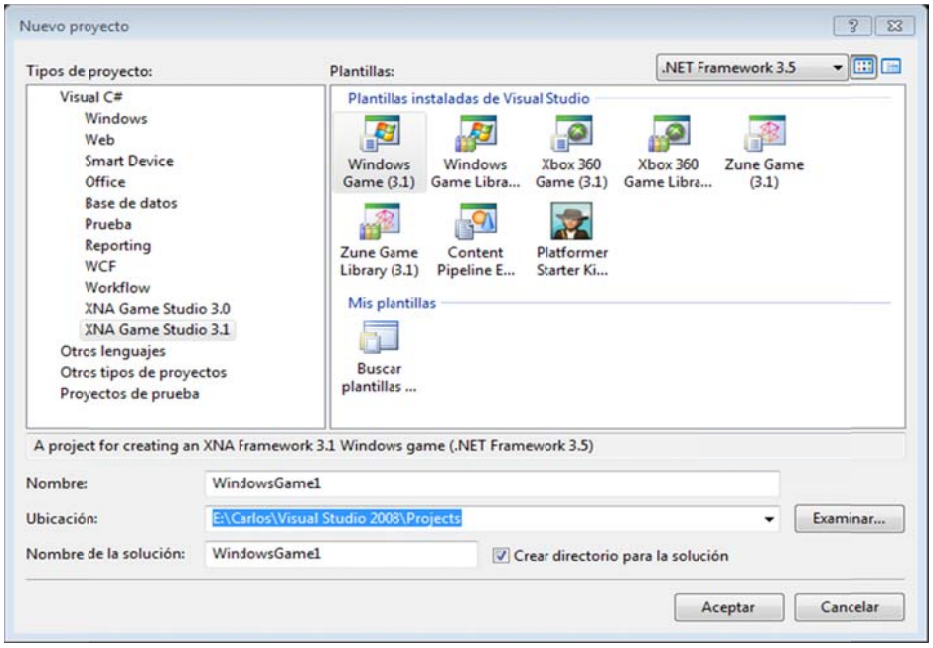

Ilustración 1-1 Ventana de diálogo: Nuevo proyecto

Al crear su nueva aplicación, podrá visualizar que automáticamente se crea todo lo necesario para comenzar, es más, puede oprimir F5 para ejecutarla y ver una ventana azul, véase Ilustración 1-2.

[3] Para mayor información acerca del vertex buffer consulte: http://msdn.microsoft.com/en-us/library/bb198836.aspx

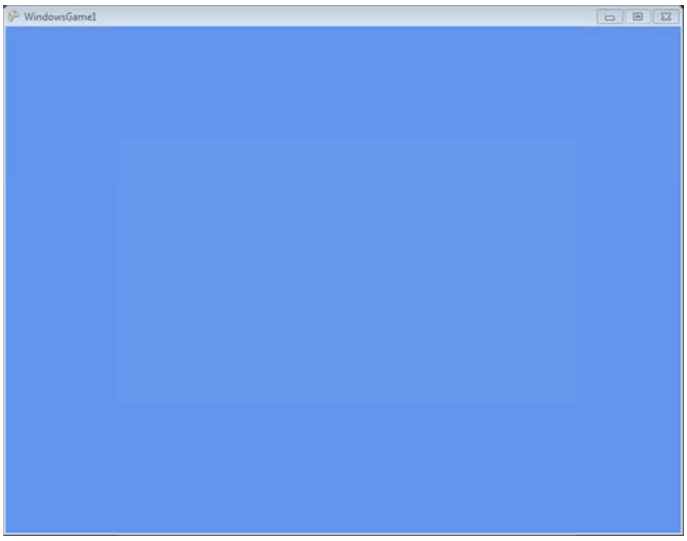

Ilustración 1-2 Programa predefinido

Claro está, que esto hace más rápido el desarrollo de la aplicación, sin embargo, hay que explicar unas cosas antes de continuar, y sólo serán las necesarias para ver el triángulo dibujado en la ventana.

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Estas son las directivas que se necesitan para crear el triángulo y que tiene que colocar al inicio del archivo Game1.cs creado por Visual Studio.

- **System** es el espacio de nombre que contiene las clases fundamentales que definen los datos, los eventos, las interfaces, etcétera.[4]
- **Microsoft.XNA.Framework** es el espacio de nombre que contiene las clases necesarias para crear juegos.
- **Microsoft.XNA.Framework.Graphics** es el espacio de nombre que contiene los métodos de la API de bajo nivel.

```
namespace TutorialX01
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
    }
}
```

Visual Studio crea el **namespace**, cuyo nombre corresponde al del proyecto; la clase **Game1**, que hereda de la clase **Game** y la variable de instancia **GraphicsDeviceManager** que maneja la configuración y administración del dispositivo gráfico, que es la tarjeta de video de la computadora. La clase **Game** provee de la inicialización básica del dispositivo gráfico, de la lógica del juego y el código del render.

Para el ejemplo se añaden las siguientes líneas, después de la declaración del dispositivo gráfico **GraphicsDeviceManager**.

```
VertexDeclaration vertexDeclaration;
```

[4] Para mayor información acerca de System visite la siguiente dirección: http://msdn.microsoft.com/es-es/library/system.aspx

```
BasicEffect effect;
VertexBuffer vertexBuffer;
```

VertexDeclaration es la clase que representa la declaración de un vértice, debido a que existen diferentes tipos de vértices definidos en XNA, estos pueden ser:

- **VertexPositionColor**. Esctructura personalizada que contiene posición y color.
- **VertexPositionColorTexture**. Estructura personalizada que contiene posición, color y textura.
- **VertexPositionNormalTexture**. Estructura personalizada que contiene posición, normal y textura.
- **VertexPositionTexture**. Estructura personalizada que contiene posición y textura.

En este ejemplo se utiliza **VertexPositionColor**, más adelante se mostraran los demás tipos de vértices. En DirectX se definía la estructura del tipo de vértice, sin embargo, XNA da estos cuatro tipos de estructuras como parte ésta, sin más que llamarlas como cualquier tipo de dato.

BasicEffect representa un shader versión 1.1, dando soporte para el color de los vértices, textura e iluminación. En XNA es necesario utilizar un tipo de shader para mostrar en pantalla el dibujo, a diferencia de DirectX u OpenGL.

VertexBuffer es la clase que representa el buffer de vértices para manipular los recursos.

Ahora se instancia los vértices que crearan el triángulo, por lo que se usa un arreglo de **VertexPositionColor**. El constructor sería el siguiente:

```
public VertexPositionColor (Vector3 position, Color color)
```

Tabla 1-1

Propiedad	Descripción
position	Es un Vector3, por lo que representa la posición del vértice en coordeanas x, y ,z.
color	Es el color del vértice representado por los colores R,G,B.

```
private readonly VertexPositionColor[] vertices =
{
    new VertexPositionColor(new Vector3(0.0F, 1.0F, 0.0F), Color.White),
    new VertexPositionColor(new Vector3(1.0F, -1.0F, 0.0F), Color. White),
    new VertexPositionColor(new Vector3(-1.0F, -1.0F, 0.0F), Color. White)
};
```

En el constructor, de la clase Game1, se inicializa el dispositivo gráfico y algunas propiedades de la ventana en donde se mostrará todo el mundo que se creé, en la Tabla 1-2 se muestran algunas de estas propiedades[5].

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    this.IsMouseVisible = true;
    this.Window.Title = "xintalalai Tutorial 00";
    this.Window.AllowUserResizing = true;
```

[5] Para mayor información de los miebros de la clase Game consulte la siguiente dirección: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game_members.aspx

```
}
```

Tabla 1-2

Propiedad	Descripción
IsMouseVisible	Muestra u oculta el puntero del mouse sobre la venta, por default está oculto.
Window.Title	Es el título de la ventana, por default el título es el nombre que se le dió a la solución en el momento de crearla.
Window.AllowUserResizing	Permite máximizar o redimensionar el tamaño de la venta.

En el método **Initialize** es donde se inicializa toda lógica o cualquier recurso no gráfico. Por default dentro del método se tiene la inicialización base de la clase **Game**, así que hay que agregar unas cuantas líneas para crear la declaración del vértice, el efecto así como el búfer de vértices.

```
protected override void Initialize()
{
    base.Initialize();
}
```

La declaración de **VertexDeclaration** necesita de dos parámetros; el primero es el dispositivo gráfico en el que se asocian los vértices, el segundo es un arreglo de elementos de vértices, que ya contiene la estructura de **VertexPositionColor**.

```
protected override void Initialize()
{
    vertexDeclaration = new VertexDeclaration(GraphicsDevice,
        VertexPositionColor.VertexElements);
    vertexBuffer = new VertexBuffer(GraphicsDevice, 3 * VertexPositionColor.SizeInBytes,
        BufferUsage.WriteOnly);

    effect = new BasicEffect(GraphicsDevice, null);
    effect.VertexColorEnabled = true;

    vertexBuffer.SetData<VertexPositionColor>(vertices, 0, vertices.Length);

    base.Initialize();
}
```

El constructor del **VertexBuffer** está sobrecargado, el que se tomó en cuenta es el que específica el tamaño y su uso.

```
VertexBuffer (GraphicsDevice, Int32, BufferUsage)
```

Tabla 1-3

Parámetros	Descripción
graphicsDevice	Es el dispositivo gráfico asociado con el vertex buffer.
sizeInBytes	Es el número de bytes alojados en el vertex buffer.

Usage	Es la opción que identifica el comportamiento del vertex buffer.

Para inicializar el efecto básico que proporciona XNA se utiliza:

```
public BasicEffect (GraphicsDevice device, EffectPool effectPool)
```

Tabla 1-4

Parámetros	Descripción
device	Es el dispositivo gráfico que creará el efecto.
effectPool	Especifica un conjunto de recursos para compartir entre los efectos.

BasicEffect.VertexColorEnabled es una propiedad que habilita el uso de vertices con colores.

```
public void SetData<T> (T[] data, int startIndex, int elementCount)
```

El método **SetData** de **VertexBuffer** adquiere los datos a copiar en el búfer de vértices, donde **T** es un tipo de dato en el búfer.

Tabla 1-5

Parámetros	Descripción
data	Es un arreglo que será copiado al vertex buffer.
starIndex	Indica el índice a partir del cual se copiarán los datos.
elementCount	Es el número de elementos máximos a copiar.

Siguiendo con el código generado automáticamente, por Visual Studio, tenemos los siguientes métodos:

```
protected override void LoadContent()
{
}

protected override void UnloadContent()
{
}

protected override void Update(GameTime gameTime)
{
base.Update(gameTime);
}
```

- **LoadContent** es llamado cuando los recursos gráficos necesitan ser cargados.
- **UnloadContent** es llamado cuando los recursos gráficos necesitan ser liberados.
- **Update** actualiza el estado de la sesión de multiplayer, como actualizar o supervisar el estado de los dispositivos de entrada, como el gamepad.

Como parte final del archivo **Game1.cs** se tiene el método que dibujará las geometrías, y otras cosas más en pantalla, que por el momento será un triángulo.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);
```

```
    Single apecto = GraphicsDevice.Viewport.AspectRatio;
    effect.World = Matrix.Identity;
    effect.View = Matrix.CreateLookAt(new Vector3(0, 0, 5),
        Vector3.Zero,
        Vector3.Up);
    effect.Projection = Matrix.CreatePerspectiveFieldOfView(1, apecto, 1, 10);

    GraphicsDevice.RenderState.FillMode = FillMode.WireFrame;
    GraphicsDevice.VertexDeclaration = vertexDeclaration;
    GraphicsDevice.Vertices[0].SetSource(vertexBuffer, 0, VertexPositionColor.SizeInBytes);

    effect.Begin();
        effect.CurrentTechnique.Passes[0].Begin();
        GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
        effect.CurrentTechnique.Passes[0].End();
    effect.End();

    base.Draw(gameTime);
}
```

La primera línea limpia el **viewport** especificando un color, en este caso negro. Se asignan valores a la matriz de mundo del efecto, en este caso con la matriz identidad, esta matriz sirve para hacer cabios de posición del modelo. Así también se le asigna un valor a la matriz de vista, que sirve para cambiar la posición y dirección de la cámara, por medio del método estático **CreateLookAt**. Por el momento no se explicaran a fondo estos términos, pues se verá en los siguientes ejemplos, por ahora sólo escríbalos.

Projection: es la matriz para cambiar la imagen 3D a 2D que será dibujada en la pantalla de la computadora; hay diferentes formas de proyecciones que se verán posteriormente.

Fillmode: es un numerador que especifica cómo se rellena el triángulo; existen tres modos de relleno de las geometrías, los cuales pueden ser punto, malla o sólido.

En el modo de relleno **Point** se dibujan los vértices de la geometría sin conectarlos entre sí, véase Ilustración 1-3.

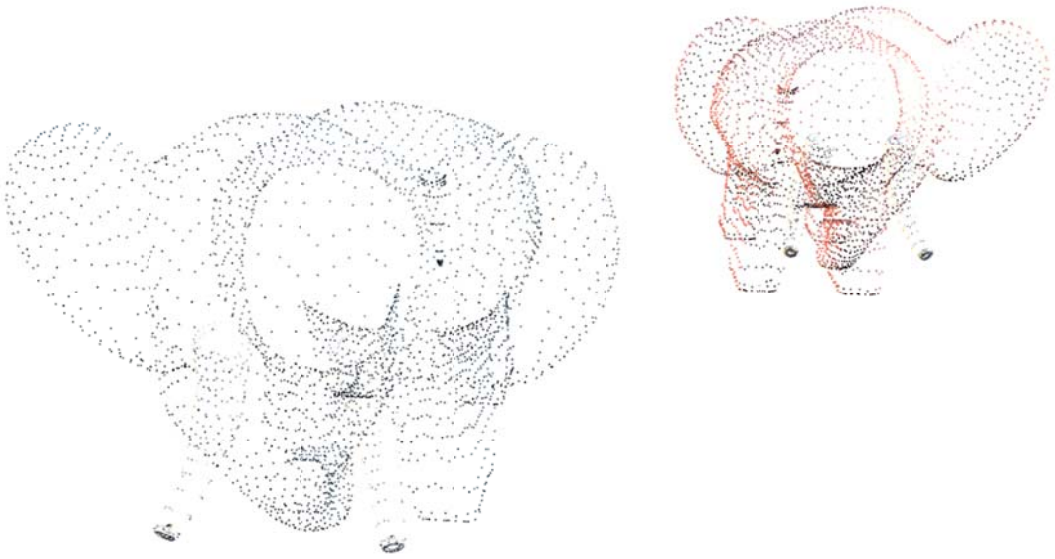

Ilustración 1-3 Fillmode.Point

En el modo de relleno **WireFrame**, sólo se dibujan los lados que conectan los vértices de la geometría, véase Ilustración 1-4.

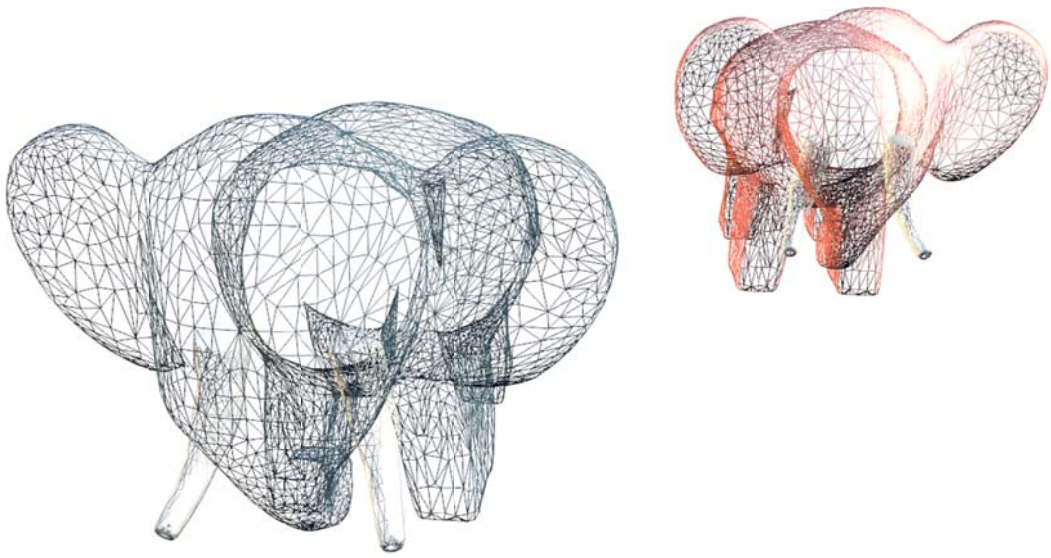

Ilustración 1-4 Fillmode.WireFrame

En el modo de relleno **Solid**, se dibujan los lados que conectan los vértices y los rellena, como en la Ilustración 1-5.

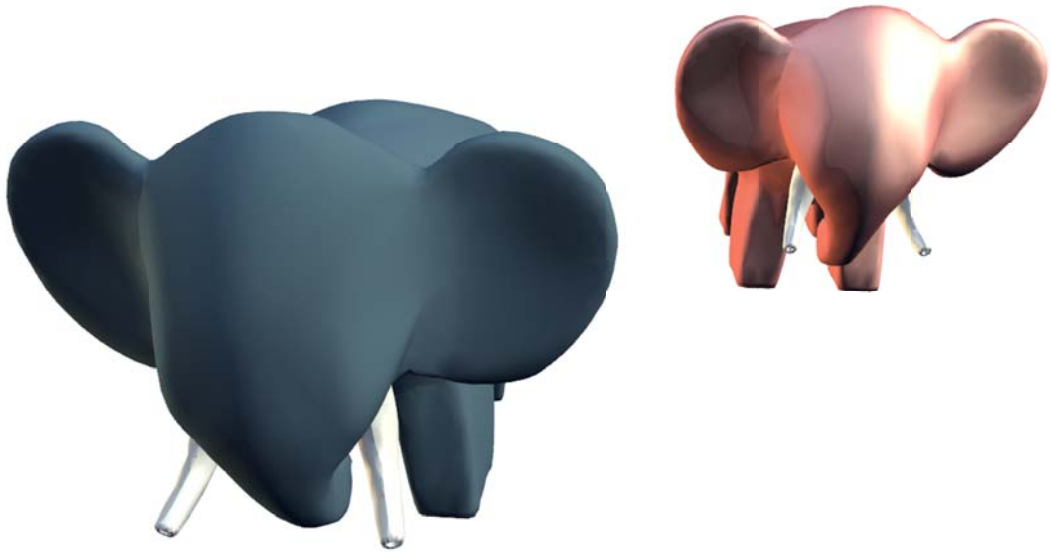

Ilustración 1-5 Fillmode.Solid

A **GraphicsDevice.VertexDeclaration** se le asigna la declaración del vértice al dispositivo gráfico. A **GraphicsDevice.Vertices[0].SetSource** se le asigna el contenido del búfer de vértices.

```
public void SetSource (VertexBuffer vb, int offsetInBytes,int
vertexStride)
```

Tabla 1-6

Parámetro	Descripción

vb	El verttex buffer de donde se tomaran los datos.
offsetInBytes	Es el byte a partir del cual serán copiados los datos.
vertexStride	Es el tamaño en bytes de los elementos en el vertex buffer.

Para dibujar la geometría se utilizan los métodos del efecto, así que se envuelve entre un **Begin**, método que comienza el efecto; y un **End**, método que finaliza el efecto, al método **GraphicsDevice.DrawPrimitives.**

También se debe envolver entre las técnicas que contiene el efecto y sus pasadas, en este caso solo tienen una y también comienza con un **Begin** y termina con un **End**.

```
public void DrawPrimitives(PrimitiveType primitiveType, int startVertex,
int primitiveCount)
```

DrawPrimitives dibuja una secuencia no indexada de la geometría, especificando el tipo de primitiva. En el siguiente capítulo se verán los distintos tipos de primitivas que ofrece XNA.

Tabla 1-7

Parámetro	Descripción
primitiveType	Describe el tipo de primitiva a dibujar.
startVertex	Indica el primer vértice a partir del cual comenzará a dibujar.
primitiveCount	Es el número de primitivas a dibujar.

Para concluir este capítulo, en el archivo **Program.cs**, que crea Visual Studio, se encuentra el método **Main**, el cual corresponde al punto de inicio del programa.

```
static void Main(string[] args)
{
    using (Game1 game = new Game1())
    {
        game.Run();
    }
}
```

El método **Run** de la clase **Game** es para inicializar el juego, para mantener en un bucle el dibujo y para comenzar el procesamiento de eventos de la aplicación.

Genere el proyecto en el menú **Generar** y corrija cualquier excepción de compilación que haya ocurrido. Si todo salió bien corra la aplicación con **F5** o vaya al menú **Depurar** para iniciar el programa, y verá un triángulo como el que se muestra en la Ilustración 1-6.

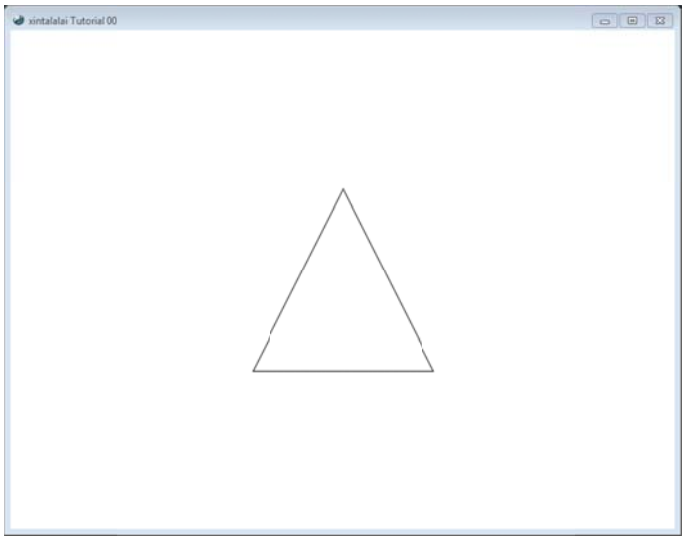

Ilustración 1-6 Triángulo

En el Código 1-1 se muestra el enlistado completo que debería tener **Game1.cs**.

Código 1-1

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Graphics;
4.
5.      namespace TutorialX01
6.      {
7.          public class Game1 : Microsoft.Xna.Framework.Game
8.          {
9.              GraphicsDeviceManager graphics;
10.             VertexDeclaration vertexDeclaration;
11.             BasicEffect effect;
12.             VertexBuffer vertexBuffer;
13.
14.             private readonly VertexPositionColor[] vertices =
15.             {
16.                 new VertexPositionColor(new Vector3(0.0F, 1.0F, 0.0F), Color.White),
17.                 new VertexPositionColor(new Vector3(1.0F, -1.0F, 0.0F), Color.White),
18.                 new VertexPositionColor(new Vector3(-1.0F, -1.0F, 0.0F), Color.White)
19.             };
20.
21.
22.             public Game1()
23.             {
24.                 graphics = new GraphicsDeviceManager(this);
25.                 this.IsMouseVisible = true;
26.                 this.Window.Title = "xintalalai Tutorial 00";
27.                 this.Window.AllowUserResizing = true;
28.                 Content.RootDirectory = "Content";
29.             }
30.
31.             protected override void Initialize()
32.             {
33.                 vertexDeclaration = new VertexDeclaration(GraphicsDevice,
34.                     VertexPositionColor.VertexElements);
35.                 vertexBuffer = new VertexBuffer(GraphicsDevice, 3 *
36.                     VertexPositionColor.SizeInBytes, BufferUsage.WriteOnly);
37.
38.                 effect = new BasicEffect(GraphicsDevice, null);
39.                 effect.VertexColorEnabled = true;
40.
41.                 vertexBuffer.SetData<VertexPositionColor>(vertices, 0, vertices.Length);
42.
```

```
43.                 base.Initialize();
44.             }
45.
46.             protected override void LoadContent()
47.             {
48.             }
49.
50.             protected override void UnloadContent()
51.             {
52.             }
53.
54.             protected override void Update(GameTime gameTime)
55.             {
56.                 base.Update(gameTime);
57.             }
58.
59.             protected override void Draw(GameTime gameTime)
60.             {
61.                 GraphicsDevice.Clear(Color.Black);
62.
63.                 Single apecto = GraphicsDevice.Viewport.AspectRatio;
64.                 effect.World = Matrix.Identity;
65.                 effect.View = Matrix.CreateLookAt(new Vector3(0, 0, 5),
66.                     Vector3.Zero,
67.                     Vector3.Up);
68.                 effect.Projection = Matrix.CreatePerspectiveFieldOfView(1,
69.                     apecto, 1, 10);
70.
71.                 GraphicsDevice.RenderState.FillMode = FillMode.WireFrame;
72.                 GraphicsDevice.VertexDeclaration = vertexDeclaration;
73.                 GraphicsDevice.Vertices[0].SetSource(vertexBuffer, 0,
74.                     VertexPositionColor.SizeInBytes);
75.
76.                 effect.Begin();
77.                 effect.CurrentTechnique.Passes[0].Begin();
78.                 GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleList, 0, 1);
79.                 effect.CurrentTechnique.Passes[0].End();
80.                 effect.End();
81.
82.                 base.Draw(gameTime);
83.             }
84.         }
85.     }
```

1.2 DynamicVertexBuffer

Al igual que la clase **VertexBuffer**, la clase **DynamicVertexBuffer** sirve para almacenar una lista de vértices, sin embargo, esta clase funciona para un arreglo dinámico de vértices, mientras que el otro es para un arreglo no dinámico. Así mismo lo recomiendan para el uso de aplicaciones sobre el Xbox360, en vez del **VetertexBuffer**; porque se puede sobre pasar el tamaño de la memoria de 10MB del EDRAM de la consola.

VertexBuffer.SetData no será necesario para escribir los datos en el búfer de vértices. Para lo anterior se cuentan con otros métodos de entrada de datos, pero que por ahora sólo se mostrara el adecuado para el triángulo.

```
public void DrawUserPrimitives<T> (PrimitiveType primitiveType, T[]
vertexData, int vertexOffset, int primitiveCount)
```

Tabla 1-8

Parámetro	Descripción
primitiveType	Describe el tipo de primitiva a dibujar
vertexData	Es el arreglo de vértices
vertexOffset	Es el índice del vértice a partir del cual se copiaran los datos al vertex buffer.
primitiveCount	Es el número de primitivas máximo que se dibujaran.

En la línea 65 del Código 1-2 se muestra el fácil uso de **DrawUserPrimitive**. El resto del código es el mismo que en el ejemplo anterior.

Código 1-2

```
1.      public class Game1 : Microsoft.Xna.Framework.Game
2.      {
3.          GraphicsDeviceManager graphics;
4.
5.          VertexDeclaration vertexDeclaration;
6.          BasicEffect effect;
7.          private readonly VertexPositionColor[] vertices =
8.          {
9.              new VertexPositionColor(new Vector3(0.0F, 1.0F, 0.0F), Color.Blue),
10.             new VertexPositionColor(new Vector3(1.0F, -1.0F, 0.0F), Color.Green),
11.             new VertexPositionColor(new Vector3(-1.0F, -1.0F, 0.0F), Color.Red)
12.         };
13.
14.         public Game1()
15.         {
16.             graphics = new GraphicsDeviceManager(this);
17.             Content.RootDirectory = "Content";
18.             this.IsMouseVisible = true;
19.             this.Window.Title = "xintalalai Tutorial 01";
20.             this.Window.AllowUserResizing = true;
21.         }
22.
23.         protected override void Initialize()
24.         {
25.             vertexDeclaration = new VertexDeclaration(GraphicsDevice,
26.                 VertexPositionColor.VertexElements);
27.
28.             effect = new BasicEffect(GraphicsDevice, null);
29.             effect.VertexColorEnabled = true;
30.
```

```
31.             base.Initialize();
32.         }
33.
34.         protected override void LoadContent()
35.         {
36.         }
37.
38.         protected override void UnloadContent()
39.         {
40.         }
41.
42.         protected override void Update(GameTime gameTime)
43.         {
44.             base.Update(gameTime);
45.         }
46.
47.         protected override void Draw(GameTime gameTime)
48.         {
49.             GraphicsDevice.Clear(Color.Black);
50.
51.             Single apecto = GraphicsDevice.Viewport.AspectRatio;
52.             effect.World = Matrix.Identity;
53.             effect.View = Matrix.CreateLookAt(new Vector3(0, 0, 5),
54.                 Vector3.Zero,
55.                 Vector3.Up);
56.             effect.Projection = Matrix.CreatePerspectiveFieldOfView(1, apecto, 1, 10);
57.
58.             GraphicsDevice.RenderState.FillMode = FillMode.WireFrame;
59.
60.             GraphicsDevice.VertexDeclaration = vertexDeclaration;
61.
62.             effect.Begin();
63.             effect.CurrentTechnique.Passes[0].Begin();
64.             GraphicsDevice.DrawUserPrimitives<VertexPositionColor>(
65.                 PrimitiveType.TriangleList, vertices, 0, 1);
66.             effect.CurrentTechnique.Passes[0].End();
67.             effect.End();
68.
69.             base.Draw(gameTime);
70.         }
71.     }
```

2 Primitivas

Las primitivas son elementos básicos para dibujar cualquier geometría en el espacio tridimensional. XNA ofrece un conjunto de éstas, y cada una se explicará a continuación.

Así como en el capítulo anterior, se seguirá usando el **VertexBuffer** y el **DynamicBuffer** para los ejemplos.

Abra Visual Studio y cree un nuevo proyecto de **XNA Game Studio 3.1**, seleccionando la plantilla **Windows Game (3.1)**. Escriba la declaración del búfer de vértices y las variables de instancia de los vértices, en el archivo que **Game1.cs**; recuerde que se está trabajando en los archivos que se crean por default.

```
VertexDeclaration vertexDeclaration;
BasicEffect effect;
VertexBuffer vertexBuffer;

VertexPositionColor[] vertices ={
    new VertexPositionColor(new Vector3(-2.0F, 0.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(-1.0F, 2.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(0.0F, 0.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(1.0F, 1.5F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(1.5F, 0.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(2.2F, 1.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(2.5F, 0.0F, 2.0F), Color.White)};
VertexPositionColor[] abanico = {
    new VertexPositionColor(new Vector3(0.0F, 1.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(1.0F, 1.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(1.0F, 0.0F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(0.5F, -0.5F, 2.0F), Color.White),
    new VertexPositionColor(new Vector3(-0.5F, -1.0F, 2.0F), Color.White)};
```

En este caso se declararon dos arreglos de vértices, el arreglo **vertice** es para mostrar cinco de las seis primitivas de XNA, y el arreglo **abanico** es para mostrar la primitiva de abanico.

Dentro del constructor escriba las siguientes líneas para modificar algunas propiedades de la ventana en que se mostraran las geometrías, son las mismas propiedades que se explicaron en el capítulo anterior.

```
this.IsMouseVisible = true;
this.Window.AllowUserResizing = true;
this.Window.Title = "xintalalai Tutorial 02a";
```

En el método **Initialize** se crean las nuevas instancias del búfer de vértices y el efecto.

```
vertexDeclaration = new VertexDeclaration(GraphicsDevice,
VertexPositionColor.VertexElements);
vertexBuffer = new VertexBuffer(GraphicsDevice, 7 * VertexPositionColor.SizeInBytes,
BufferUsage.WriteOnly);
vertexBuffer.SetData<VertexPositionColor>(vertices, 0, vertices.Length);
effect = new BasicEffect(GraphicsDevice, null);
effect.VertexColorEnabled = true;
```

Lo único que cambia en esta declaración es el número de elementos que contendrá el búfer de vértices, así que se multiplica el número de elementos del arreglo por el tamaño en bytes de la estructura **VertexPositionColor**, es decir:

```
vertexBuffer = new VertexBuffer(GraphicsDevice, vertices.Length *
    VertexPositionColor.SizeInBytes, BufferUsage.WriteOnly);
```

En el llenado del búfer de vértices se utiliza el primer arreglo para mostrar las siguientes primitivas:

- **PointList**
- **LineList**
- **LineStrip**
- **TriangleList**
- **TriangleStrip**

2.1 PointList

La primitiva **PoinList** toma una lista de vértices y los muestra como puntos en el espacio, en el orden en que se encuentran en el búfer de vértices.

Código 2-1

```
1.      GraphicsDevice.Clear(Color.Black);
2.      Single aspecto = GraphicsDevice.Viewport.AspectRatio;
3.      effect.World = Matrix.Identity;
4.      effect.View = Matrix.CreateLookAt(new Vector3(0.0F, 0.0F, 6.0F),
5.          Vector3.Zero,
6.          Vector3.Up);
7.      effect.Projection = Matrix.CreatePerspectiveFieldOfView(1.0F,
8.          aspecto, 1.0F, 10.0F);
9.      GraphicsDevice.RenderState.FillMode = FillMode.WireFrame;
10.     GraphicsDevice.VertexDeclaration = vertexDeclaration;
11.     GraphicsDevice.Vertices[0].SetSource(vertexBuffer, 0,
12.         VertexPositionColor.SizeInBytes);
13.
14.     effect.Begin();
15.     effect.CurrentTechnique.Passes[0].Begin();
16.     GraphicsDevice.DrawPrimitives(PrimitiveType.PointList, 0, 7);
17.     effect.CurrentTechnique.Passes[0].End();
18.     effect.End();
```

El Código 2-1 se deberá escribir dentro del método **Draw**, note que la mayoría del código es el mismo que en el capítulo anterior, excepto la línea 16 en donde el tipo de primitiva será **PointList** y el número máximo de elementos que se dibujarán serán 7; por lo que podrá cambiarlo por la propiedad **Lenght** del arreglo **vertices**.

Inicie la aplicación y verá siete puntos como en la Ilustración 2-1.

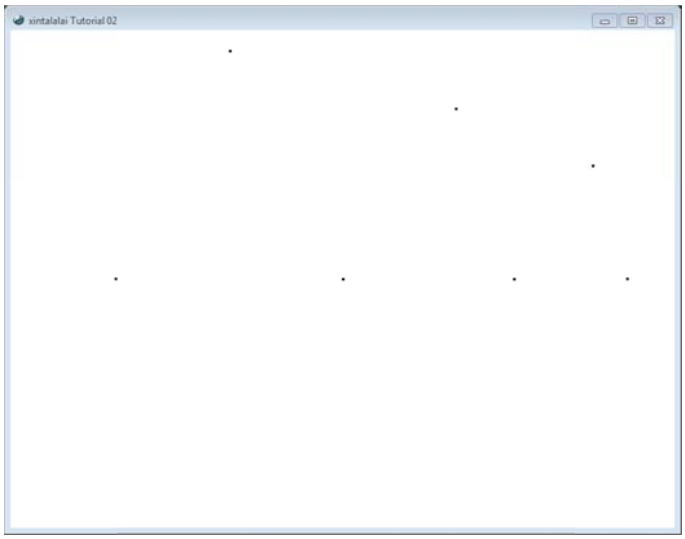

Ilustración 2-1 PointList

2.2 LineList

LineList es una primitiva que dibuja líneas rectas a partir de un par de vértices del búfer de vértices. En el ejemplo se tienen siete vértices con lo que se dibujarán tres líneas, el último no se ocupa.

Modifique el método **DrawPimitives** del Código 2-1, línea 16, cambiando el parámetro **PrimitiveType.PointList** por **PrimitiveType.LineList** y el número de elementos a tres.

```
GraphicsDevice.DrawPrimitives(PrimitiveType.LineList, 0, 3);
```

Oprima **F5** y verá algo similar a la Ilustración 2-2. Como se puede ver, las líneas van del **vertice** 0 al 1, del 2 al 3 y del 4 al 5.

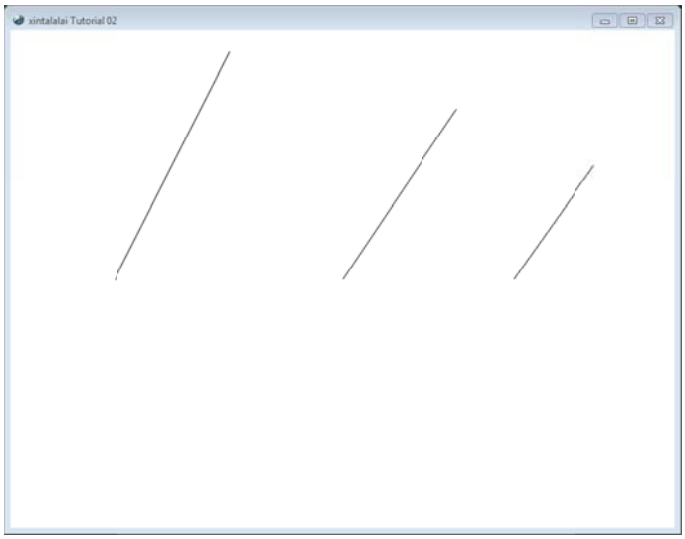

Ilustración 2-2 LineList

2.3 LineStrip

LinStrip es la primitiva que une dos puntos para crear una línea recta, tomando como punto de inicio el segundo punto de la línea anterior, excepto la primera línea que no le antecede otra.

El número de rectas que se pueden dibujar dado un número de vértices será igual a:

$$Número\ de\ LineStrip = Número\ de\ vértices - 1$$

$$Número\ de\ vértices \geq 2$$

Modifique el tipo de primitiva del Código 2-1, línea 16, por **PrimitiveType.LineStrip** y el número de elementos a dibujar por seis, o por el número de elementos del arreglo vértice menos uno.

```
GraphicsDevice.DrawPrimitives(PrimitiveType.LineStrip, 0, 6);
```

Oprima **F5** y vera una imagen similar a la Ilustración 2-3.

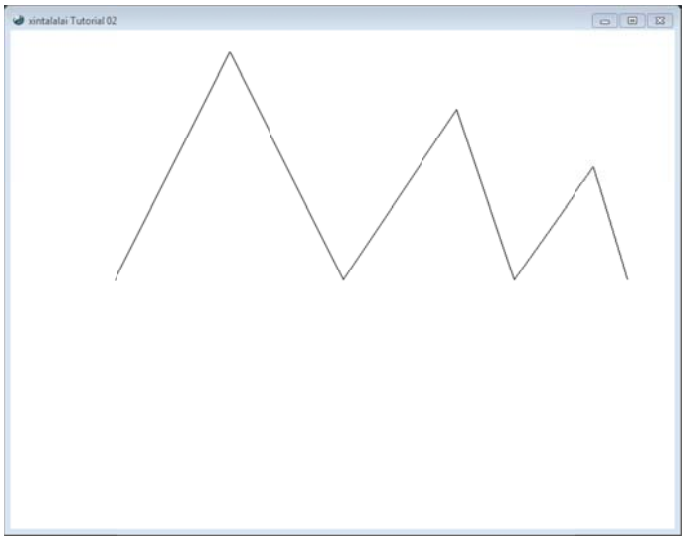

Ilustración 2-3LineStrip

2.4 Modos de Culling

El culling es un algoritmo que determina la visibildad de las caras, sirve para dibujar sólo aquella geometría que se alcanza a ver, por ejemplo, en un cubo a lo más que podemos ver son tres de sus caras. La manera de conocer qué caras se dibujarán es por medio de la normal de ésta. La normal es un vector perpendicular al plano y la dirección de éste dependerá del sentido en que se vayan dibujando las primitivas.

Por default XNA tiene el modo **CullCounterClockwiseFace** activado, es decir, oculta aquellas caras que por orden en el búfer de vértices se asemejan al sentido contrario a las manecillas del reloj, más adelante se verá un ejemplo. El modo **CullClockwiseFace**, oculta aquellas caras que en el búfer de vértices tengan el orden semejante al sentido de las manecillas del reloj. El modo **None** hace caso omiso del modo de culling, para dibujar todas las caras.

Escriba dentro del método Draw en el Código 2-1, línea 13, el modo de culling **None**.

```
GraphicsDevice.RenderState.CullMode = CullMode.None;
```

Una manera sencilla de conocer qué caras no son dibujadas, es por medio de la regla de la mano derecha para el caso de **CullCounterClockwiseFace** y la regla de la mano izquierda para el caso **CullClockwiseFace**.

2.5 TriangleList

TriangleList es la primitiva que toma triadas de vértices para dibujar un triángulo. La manera de saber qué número de elementos se pueden dibujar dado un número de vértices es la siguiente:

$$Número\ de\ TriangleList = \frac{Número\ de\ Vértices}{3}$$

$$Número\ de\ TriangleList\ \geq 1$$

$$Número\ de\ TriangleList \in \mathbb{N}$$

El número de elementos será igual a la parte entera del resultado de la división, además de que debe ser mayor a cero.

Modifique el método **Draw** en el Código 2-1, línea 16; el primer parámetro cámbielo por **Primitive.TriangleList** y el tercer parámetro por 2.

```
GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleList, 0, 2);
```

Oprima **F5** para iniciar la aplicación y verá una imagen similar a la Ilustración 2-4.

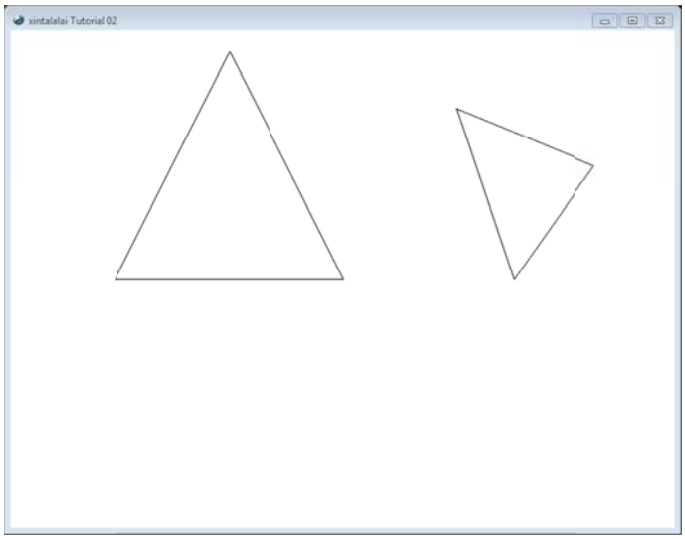

Ilustración 2-4 CullMode.None

Como puede ver, la primitiva toma los tres primeros vértices para crear el triángulo de la derecha, posteriormente toma los siguientes tres vértices para dibujar el triángulo de la izquierda, por lo que no se toma el último vértice.

Ahora modifique la línea 13 del Código 2-1 por **CullClockwiseFace**, para que oculte el triángulo cuya primitiva se dibuje en el sentido de la manecillas del reloj.

```
GraphicsDevice.RenderState.CullMode = CullMode.CullClockwiseFace;
```

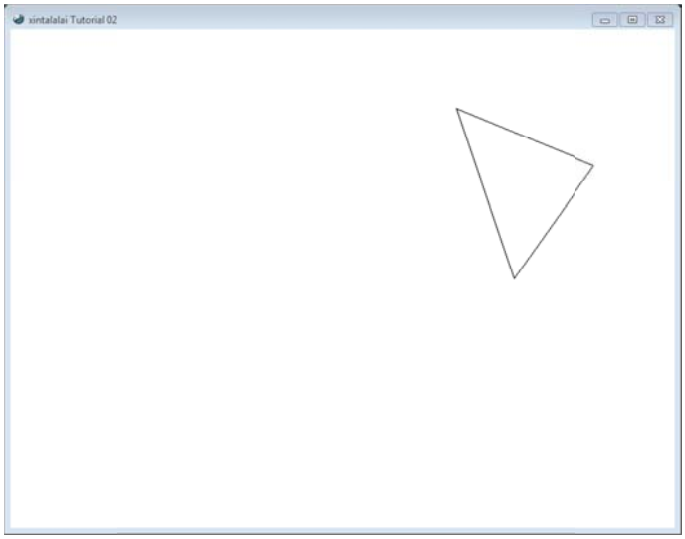

Ilustración 2-5 CullMode.CullClockwiseFace

Después cambie el modo del culling a **CullCounterClockwiseFace** para ocultar el triángulo cuya primitiva se dibuje en el sentido contrario a las manecillas del reloj.

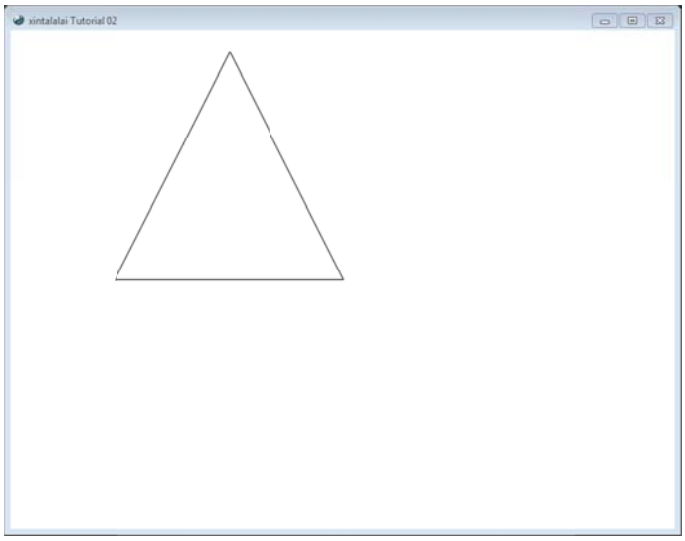

Ilustración 2-6 CullMode.CullCounterClockwiseFace

2.6 TriangleStrip

TriangleStrip es la primitiva que permite dibujar con eficiencia triángulos, pues toma dos de los últimos vértices del triángulo anterior más un nuevo vértice para crearlo, excepto la primera triada de **vertices** del búfer de vértices. Es decir, el primer triángulo toma los vértices 1, 2 y 3; el segundo toma los vértices 2, 3 y 4; el tercer triángulo toma los vértices 3, 4 y 5.hasta concluir con la lista contenida en el búfer de vértices.

Cambie el método **DrawPrimitives** en el Código 2-1, línea 16, en el primer y último parámetro.

```
GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 5);
```

Inicie la aplicación con la tecla **F5**, y verá algo similar a la Ilustración 2-7.

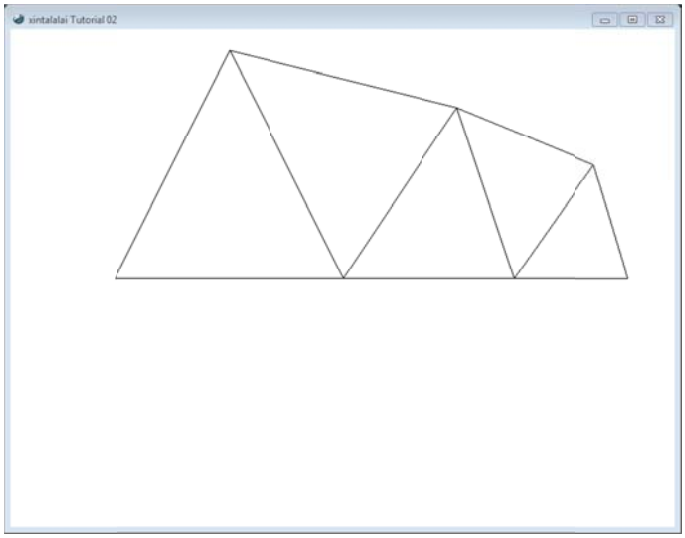

Ilustración 2-7 TriangleStrip

Para conocer la cantidad máxima de **TriangleStrip** que se pueden dibujar a partir de un número dado de vértices se resta el número de vértices menos dos.

$$Número\ de\ TriangleStrip = Número\ de\ vértices - 2$$

2.7 TriangleFan

Esta última primitiva utiliza el primer vértice del búfer de vértices, el último vértice del triángulo anterior y el vértice siguiente. Es decir, el primer triángulo está formado por los vértices 1, 2 y 3; el segundo triángulo por los vértices 1, 3 y 4; el tercer triángulo está constituido por los vértices 1, 4 y 5.

De nueva cuenta, cambie los parámetros uno y tres del método **DrawPimitives**, del Código 2-1, línea 16.

```
GraphicsDevice.DrawPrimitives(PrimitiveType.TriangleFan, 0, 3);
```

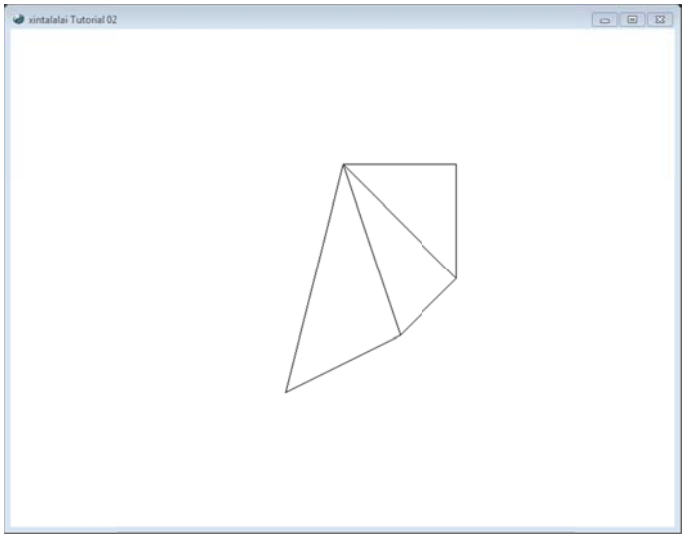

Ilustración 2-8 TriangleFan

El número de **TriangleFan** máximos que se pueden extraer de una lista de vértices, se calcula al restar dos, al número de vértices del búfer, con lo cual se obtiene la siguiente fórmula:

$$Número\ de\ TriangleFan = Número\ de\ vértices - 2$$

3 Creando objeto 3D

En este capítulo se dibujará un cubo a partir de un arreglo de vértices y un índice que indica que número de vértice debe conformar el triángulo.

Un cubo se conforma de seis caras, ocho vértices y doce triángulos, sin embargo, no se declararan treinta y seis vértices, pues habría redundancia en los datos haciéndolo ineficaz para este ejemplo. Así que el índice llevará el orden en que se deben dibujar los triángulos.

Comience por abrir Visual Studio y cree una nueva solución para XNA. Declare en el archivo **Game1.cs** como variables de instancia los vértices, el efecto, el vertexbuffer, el indexbuffer y un arreglo de enteros que servirá como índice.

Código 3-1

```
1.      VertexDeclaration vertexDeclaration;
2.      BasicEffect effect;
3.      VertexBuffer vertexBuffer;
4.      IndexBuffer indexBuffer;
5.      Color color = new Color(new Vector3(0.05859375F, 0.12890625F, 0.1484375F));
6.
7.      VertexPositionColor[] vertices ={
8.          new VertexPositionColor(new Vector3(-1.0F, -1.0F, 1.0F), Color.White),
9.          new VertexPositionColor(new Vector3(-1.0F, 1.0F, 1.0F), Color.White),
10.         new VertexPositionColor(new Vector3(1.0F, 1.0F, 1.0F), Color.White),
11.         new VertexPositionColor(new Vector3(1.0F, -1.0F, 1.0F), Color.White),
12.         new VertexPositionColor(new Vector3(-1.0F, -1.0F, -1.0F), Color.White),
13.         new VertexPositionColor(new Vector3(-1.0F, 1.0F, -1.0F), Color.White),
14.         new VertexPositionColor(new Vector3(1.0F, 1.0F, -1.0F), Color.White),
15.         new VertexPositionColor(new Vector3(1.0F, -1.0F, -1.0F), Color.White)
16.     };
17.
18.     Int32[] indices ={0, 1, 2,    // Frente
19.         0, 2, 3,
20.         4, 6, 5,   // Posterior
21.         4, 7, 6,
22.         4, 5, 1,   // Izquierda
23.         4, 1, 0,
24.         3, 2, 6,   // Derecha
25.         3, 6, 7,
26.         1, 5, 6,   // Tapa
27.         1, 6, 2,
28.         4, 0, 3,   // Base
29.         4, 3, 7};
```

Las línea 4 del Código 3-1 representa: el búfer de índices, que describe el orden de dibujo de los vértices en el búfer de vértices.

Se pueden generar nuevos colores con el constructor **Color** que tiene ocho sobrecargas, línea 5, en este caso se utilizó la sobrecarga que recibe como parámetro una estructura **Vector3**, cuyas coordenadas (x, y, z) representan el rojo, el verde y el azul respectivamente. El rango que puede contener cada coordenada (x, y, z) es de 0.0F a 1.0F.

En el método **Initialize** de la clase **Game1**, inicialice el búfer de vértices, el búfer de índices y el efecto.

Código 3-2

```
1.      protected override void Initialize()
2.      {
3.          vertexDeclaration = new VertexDeclaration(GraphicsDevice,
4.              VertexPositionColor.VertexElements);
5.          vertexBuffer = new VertexBuffer(GraphicsDevice, vertices.Length *
6.               VertexPositionColor.SizeInBytes, BufferUsage.WriteOnly);
7.          // inicialización del index buffer
8.          indexBuffer = new IndexBuffer(GraphicsDevice, typeof(Int32), indices.Length,
9.              BufferUsage.WriteOnly);
10.
```

```
11.         effect = new BasicEffect(GraphicsDevice, null);
12.         effect.VertexColorEnabled = true;
13.         vertexBuffer.SetData<VertexPositionColor>(vertices, 0, vertices.Length);
14.         indexBuffer.SetData<Int32>(indices, 0, indices.Length);
15.
16.         base.Initialize();
17.     }
```

El constructor de **IndexBuffer** que se ocupó tiene la siguiente forma:

```
public IndexBuffer(GraphicsDevice graphicsDevice, Type indexType, int
elementCount, BufferUsage usage)
```

En la Tabla 3-1 se muestra el significado de cada parámetro que toma el constructor.

Tabla 3-1

Parámetro	Descripción
graphicsDevice	Es el dispositivo gráfico asociado con el búfer de índice.
indexType	Es el tipo usado por los valores del índice.
elementCount	Es el número de valores en el búfer.
Usage	Es un conjunto de opciones que identifican el comportamiento de los recursos del búfer de índice.

En la línea 14 del Código 3-2, la asignación de los datos de un arreglo al búfer de índice se utilizó el siguiente método.

```
public void SetData<T>(T[] data, int startIndex, int elementCount)
```

Donde **T** es el tipo de dato de los elementos del arreglo a copiar en el búfer.

Tabla 3-2

Parámetro	Descripción
data	Es el arreglo de datos a copiar.
startIndex	Es el índice a partir del cual comenzará a copiar.
elementCount	Es el número de elementos a copiar.

Ya sólo falta mandar a dibujar el cubo, así que en el método **Draw** escriba las siguientes líneas para poder ver el cubo en modo **WireFrame**.

Código 3-3

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(color);
4.
5.          Single aspecto = GraphicsDevice.Viewport.AspectRatio;
6.          effect.World = Matrix.Identity;
7.          effect.View = Matrix.CreateLookAt(new Vector3(0.0F, 0.0F, -4.0F),
8.              Vector3.Zero,
9.              Vector3.Up);
```

```
10.         effect.Projection = Matrix.CreatePerspectiveFieldOfView(1, aspecto, 1, 1000);
11.
12.         GraphicsDevice.RenderState.FillMode = FillMode.WireFrame;
13.         GraphicsDevice.VertexDeclaration = vertexDeclaration;
14.         GraphicsDevice.Vertices[0].SetSource(vertexBuffer, 0,
15.             VertexPositionColor.SizeInBytes);
16.         GraphicsDevice.Indices = indexBuffer;
17.
18.         effect.Begin();
19.         effect.CurrentTechnique.Passes[0].Begin();
20.         GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0,
21.             0, indices.Length, 0, 12);
22.         effect.CurrentTechnique.Passes[0].End();
23.         effect.End();
24.         base.Draw(gameTime);
25.     }
```

GraphicsDevice.Indices es una propiedad del dispositivo gráfico para establecer el búfer de índices al dispositivo gráfico.

El método de dibujo, ahora toma un arreglo de enteros que indica el orden en que serán dibujados los triángulos.

```
public void DrawIndexedPrimitives(PrimitiveType primitiveType, int
baseVertex, int minVertexIndex, int numVertices, int startIndex, int
primitiveCount)
```

En la Tabla 3-3 se muestra el significado de cada uno de los parámetros.

Tabla 3-3

Parámetro	Descripción
primitiveType	Es el tipo de primitiva a dibujar.
baseVertex	Es el índice a partir del cual se tomará en cuenta.
minVertexIndex	Es el índice del vértice usado durante la llamada.
numVertices	Es el número de vértices usados durante la llamada.
startIndex	Índice a partir del cual se dibujará.
primitiveCount	Es el número de primitivas a dibujar.

Inicie el programa con **F5** y podrá ver algo parecido a la Ilustración 3-1; si tuvo problemas de compilación resuélvalas y vuelva a intentarlo.

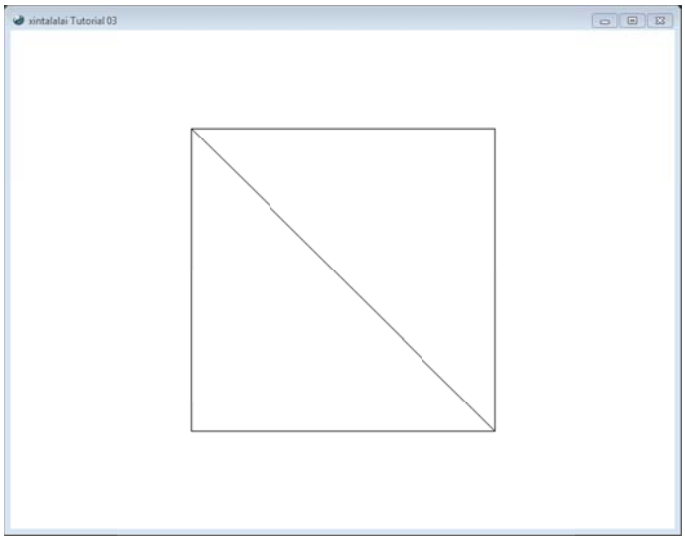

Ilustración 3-1 Cubo

Aunque a primera vista lo que aparece en el viewport no se parezca a un cubo, no es así, esto se debe a la posición de la cámara que se encentra frente a un lado del cubo. Así que ahora modificaremos la posición de la cámara, sólo para darnos una idea de que en realidad se ha dibujado una figura tridimensional.

3.1 Posicionando cámara

Para posicionar la cámara haremos uso del método **CreateLookAt** que tiene tres vectores cómo parámetros que indican: la posición de la cámara, el punto de observación y un vector unitario que indica dónde es hacia arriba, véase Ilustración 3-2.

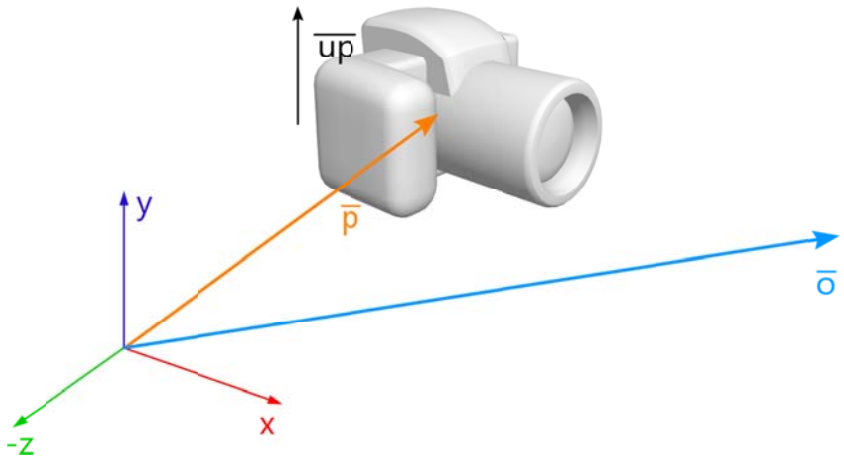

Ilustración 3-2 Parámetros del método CreateLookAt

En este ejemplo sólo se cambiará la posición de la cámara sin cambiar los otros dos vectores. Así que comience por declarar los vectores de la cámara como variables de instancia de la clase **Game1**.

```
Vector3 posicion = new Vector3(0.0F, 0.0F, -4.0F);
Vector3 vista = Vector3.Zero;
Vector3 arriba = Vector3.Up;
```

Ahora asígnelas en los parámetros de **CreateLookAt**, líneas 7 – 9, Código 3-3.

```
effect.View = Matrix.CreateLookAt(posicion, vista, arriba);
```

Para cambiar la posición de la cámara se utilizará el teclado, y sabiendo que XNA se creó para que la creación de videojuegos fuera sencilla, pues sí, también se utilizará el control 360 para cambiar la cámara.

3.2 Teclado

Comencemos con el teclado pues es el dispositivo de entrada de datos más antiguo. Pero primero definamos con qué teclas se harán los cambios.

Tecla	Movimiento
Flecha Izquierda	Disminuye la coordenada X
Flecha Derecha	Incrementa la coordenada X
Flecha Arriba	Incrementa la coordenada Y
Flecha Abajo	Disminuye coordenada Y
Avanza página	Incrementa coordenada Z
Retrocede página	Disminuye coordenada Z

Ahora hay que escribir las siguientes líneas de código dentro del método **Update** de la clase **Game1**.

```
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Left))
    posicion.X -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Right))
    posicion.X += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Up))
    posicion.Y += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Down))
    posicion.Y -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.PageUp))
    posicion.Z += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.PageDown))
    posicion.Z -= 0.1F;
```

La clase **Keyboard** del namespace **Microsoft.Xna.Framework.Input**, es la encargada de verificar qué tecla se oprimió o dejó de oprimirse. El método **static GetState** tiene dos sobrecargas, la primera no obtiene ningún parámetro, y la segunda toma el jugador asociado al control.

IsKeyDown regresa un valor booleano si se ha presionado la tecla que se pasó como parámetro; el parámetro debe ser un elemento de la numeración **Keys**.

3.3 Control 360

Antes de continuar con el manejo del control 360, es importante que se verifique que este dispositivo se encuentre funcionando correctamente; por default Windows Vista y Windows 7 tiene el controlador instalado, sin embargo, en versiones anteriores se tiene que descargar el controlador del control, en la siguiente liga pueden descargarlo.

http://www.microsoft.com/hardware/download/download.aspx?category=Gaming

En la Ilustración 3-3 se muestra el control del Xbox 360 con los nombres de los componentes de la clase **GamePad**.

Ilustración 3-3 Control 360

Ahora sí, coloque las siguientes líneas de código dentro del método **Update** de la clase **Game1**, y por debajo de las líneas que manejan el teclado.

```
if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X > 0)
    posicion.X += 0.1F * GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X;
if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X < 0)
    posicion.X -= 0.1F * -GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X;
if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y > 0)
    posicion.Y += 0.1F * GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y;
if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y < 0)
    posicion.Y -= 0.1F * -GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y;
if (GamePad.GetState(PlayerIndex.One).Triggers.Right > 0)
    posicion.Z += 0.1F * GamePad.GetState(PlayerIndex.One).Triggers.Right;
if (GamePad.GetState(PlayerIndex.One).Triggers.Left > 0)
    posicion.Z -= 0.1F * GamePad.GetState(PlayerIndex.One).Triggers.Left;
```

Al igual que con el teclado, la clase **GamePad** tiene métodos estáticos que muestran el estado de cada uno de los componentes del control 360.

GetState tiene dos sobrecargas, en la primera recibe el número del jugador asociado al control; en la segunda sobrecarga recibe el número del jugador (**PlayerIndex)** y el valor de la numeración que especifica el uso de la zona muerta (**GamePadDeadZone**).

Los valores que puede tomar **GamePadDeadZone** es: **Circular**, **IndependentAxes** y **None**. El valor de **Circular** combina los posiciones **x** y **y** de cada uno de los sticks y es comparado con la zona muerta; esto es una mejor práctica, que utilizar los ejes independientes. **IndependentAxes** compara a cada eje con la zona muerta independientemente, y es el valor que se da por default en el método **GetState** con un parámetro. **None** regresa los valores sin procesar de cada stick como un arreglo; esto es cuando intenta implementar su propia zona muerta.

Los sticks regresan un **Vector2** con valores de punto flotante entre -1.0F y 1.0F. Los triggers regresan un valor de punto flotante entre 0.0F y 1.0F. Éstos últimos se utilizaran para cambiar el valor de la posición de la cámara.

Ya puede oprimir **F5** para ejecutar la aplicación y verificar que puede mover la cámara de posición con el teclado y el control del Xbox 360; en caso de tener un error de compilación corríjalo y vuélvalo a intentar.

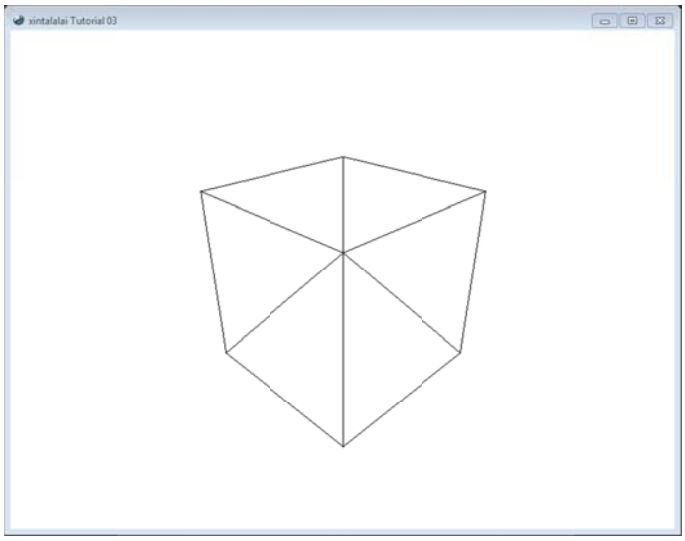

Ilustración 3-4 Cambio de posición

3.4 Traslación

Para comenzar con los tipos de transformaciones que puede realizarle a un modelo gráfico, agregue las siguientes variables de instancia a la clase **Game1**.

```
Vector3 traslacion = Vector3.Zero;
Single escala = 1.0F;
Single rotacionX = 0.0F;
Single rotacionY = 0.0F;
Single rotacionZ = 0.0F;
```

La traslación consiste en mover los vértices del modelo a una nueva posición y esto se logra por medio de la siguiente declaración:

```
effect.World = Matrix.CreateTranslation(traslacion);
```

Recuerde que la matriz **World** sirve para realizar las transformaciones sobre la geometría, así que la asignación se le hace a la propiedad del efecto, en el método **Draw** de la clase **Game1**.

El método static **CreateTranslation** crea una matriz de translación, la cual tiene cuatro sobrecargas, pero todas comparten la idea de aceptar coordenadas **x**, **y** y **z**, de una manera u otra, en este caso fue un **Vector3D**. Sus coordenadas se tomaran como la translación respecto al eje que representan. Es decir, si el vector tiene como coordenadas (1.0, -2.0, 0.0) los vértices tendrán que trasladarse una unidad sobre el eje **x**, menos dos unidades sobre el eje **y**, y cero unidades sobre el eje **z**.

Al igual que para mover la posición de la cámara, la siguiente tabla muestra qué teclas deberán oprimirse para trasladar el modelo 3D.

Tabla 3-4

Tecla	Movimiento
D	Disminuye la coordenada X
A	Incrementa la coordenada X

W	Incrementa la coordenada Y
S	Disminuye la coordenada Y
Z	Incrementa la coordenada Z
X	Disminuye la coordenada Z

Agregue las siguientes líneas de código para trasladar el cubo, recuerde que deben escribirse dentro del método **Update** de la clase **Game1**.

```
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.D))
    traslacion.X -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.A))
    traslacion.X += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.W))
    traslacion.Y += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.S))
    traslacion.Y -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Z))
    traslacion.Z += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.X))
    traslacion.Z -= 0.1F;
```

Inicie la solución oprimiendo **F5**, si hay algún error que compilación resuélvalo y vuelva a intentar.

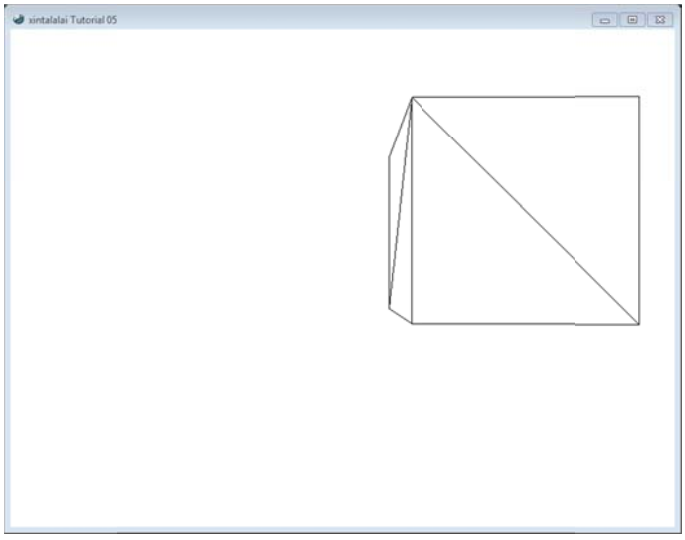

Ilustración 3-5 Traslación

3.5 Escala

Para cambiar el tamaño de los modelos se crea una matriz de escala, la cual tiene seis sobrecargas para aceptar valores de tipo flotante o estructuras de tipo **Vector3**. El valor que se le asigne al método estático **CreateScale**, se usará para cambiar el tamaño del modelo. Algunas de las sobrecargas son:

```
Matrix.CreateScale (Single)
Matrix.CreateScale (Single, Single, Single)
Matrix.CreateScale (Vector3)
```

La primera sobrecarga escala las coordenadas de los vértices por el valor del parámetro. En la segunda sobrecarga cada parámetro representa la escala sobre el eje **x**, **y** y **z**, respectivamente. Y por último, el parámetro que toma un **Vector3** toma los valores de sus coordenadas para cambiar el tamaño del modelo, muy parecido a la sobrecarga anterior.

En este caso se usó un solo valor para cambiar en **x**, **y** y **z** el tamaño del cubo. Las siguientes teclas son para cambiar el tamaño de la geometría.

Tecla	Acción
E	Incrementa el tamaño.
R	Disminuye el tamaño

Escriba el código siguiente en el método **Update** de la clase **Game1**, después del que traslada la geometría.

```
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.E))
    escala += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.R))
{
    escala -= 0.1F;
    if (escala < 0.1F)
        escala = 0.1F;
}
```

Se coloca un condicional **if** en la disminución de la escala, pues si llega a ser cero el modelo desaparecería, y si llegará a ser menor que cero se llegaría a cambiar el culling del modelo, lo que haría sería mostrarnos las paredes internas del cubo.

Ahora hay que generar la matriz y multiplicarla por los valores anteriores, en el método **Draw** de la clase **Game1**.

```
effect.World = Matrix.Identity * Matrix.CreateTranslation(traslacion) *
Matrix.CreateScale(escala);
```

Oprima **F5** y pruebe los cambios que puede hacer con el teclado; si hay un problema de compilación resuélvalo y trate de nuevo.

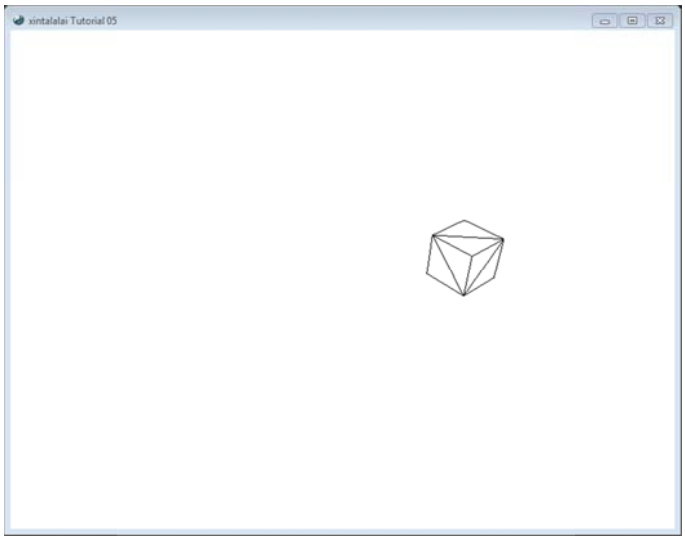

Ilustración 3-6 Escala

3.6 Rotación

Como en cualquier otro juego en tercera persona, en el que se observa al personaje; el modelo 3D se traslada sobre un mundo y también rota para poder cambiar de dirección mientras corren por sus vidas u otra cosa que les pida cambiar de rumbo. La rotación es la última transformación que se hará al cubito con el que se ha estado trabajando, y que éstas, las transformaciones, no son exclusivas de los modelos 3D, también puede aplicarse a la cámara.

La rotación se hace alrededor de un eje, y para eso se crea una matriz por medio de los métodos **Matrix.CreateRotationX**, **Matrix.CreateRotationY** y **Matrix.CreateRotationZ.** Esta terna de métodos recibe un parámetro en punto flotante que representa el ángulo en radianes que será rotado alrededor de algún eje coordenado.

En la Tabla 3-5 se muestra las teclas con las que manejará el cambio de orientación del cubo.

Tabla 3-5

Tecla	Acción
J	Disminuye el ángulo de rotación en Y.
L	Incrementa el ángulo de rotación en Y.
I	Incrementa el ángulo de rotación en X.
K	Disminuye el ángulo de rotación en X.
U	Incrementa el ángulo de rotación en Z.
O	Disminuye el ángulo de rotación en Z.

Escriba las líneas código dentro del método Update y después de las líneas que manipulan el tamaño del cubo.

```
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.J))
```

```
    rotacionY -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.L))
    rotacionY += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.I))
    rotacionX += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.K))
    rotacionX -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.U))
    rotacionZ += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.O))
    rotacionZ -= 0.1F;
```

Ahora hay que multiplicar la matriz generada por la matriz de mundo del efecto; en este punto hay que tener cuidado al colocar el ordene de las matrices, y que queda fuera del alcance de este documento, pues la multiplicación de las matrices no es conmutativa, así que en este ejemplo se coloca primero las matrices de rotación y después la de traslación.

```
effect.World = Matrix.Identity * Matrix.CreateScale(escala) *
    Matrix.CreateRotationX(rotacionX) * Matrix.CreateRotationY(rotacionY) *
    Matrix.CreateRotationZ(rotacionZ) *
    Matrix.CreateTranslation(traslacion);
```

Así que la siguiente ilustración se muestra lo que pasa cuando se coloca primero las matrices de rotación y luego la de traslación.

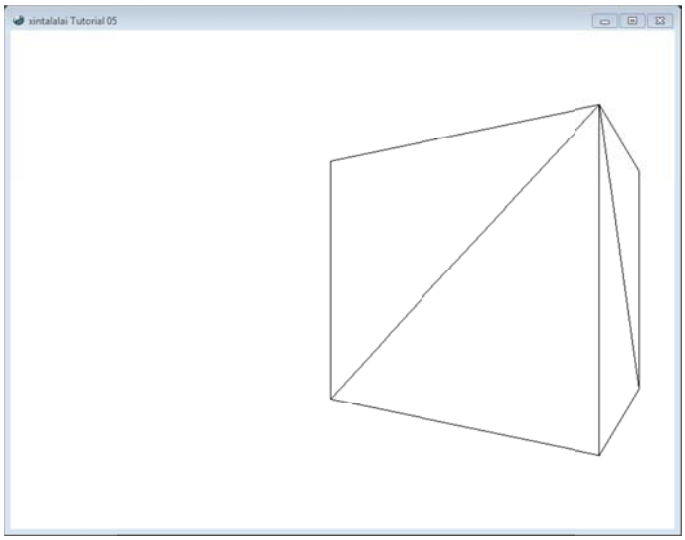

Ilustración 3-7 Rotar y trasladar

Se oprimió una de las teclas que rota el cubo y luego se oprimió una que traslada al cubo; es como no si hubiera movido los ejes coordenados, es decir, se rota alrededor del eje **y** y luego se traslada sobre el eje **x**, conservando la misma distancia con el viewport.

Cambiando el orden de las matrices, primero la matriz de traslación y luego las de rotación, se aprecia un cambio importante en la ilustración siguiente.

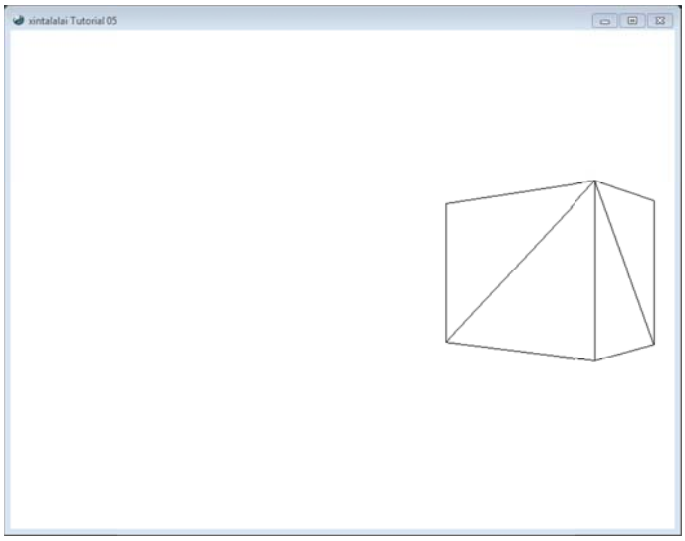

Ilustración 3-8 Rotar y trasladar

Se oprimieron las mismas teclas que en el ejemplo anterior para apreciar el cambio; en este caso es como si se hubiera cambiado la orientación de los ejes coordenados, es decir, primero se rota el cubo alrededor del eje **y** (es como se hubieran girado también los ejes coordenados), y luego se trasladó el cubo sobre el eje **x**, lo que lo aleja del viewport.

4 Textura

Las texturas sirven como papel tapiz que se “pegan” sobre los modelos 3D para darle un aspecto más real, sin embargo, existen técnicas más avanzadas para generar la sensación de realismo y que se basan en el shader pero que no reemplazan a las texturas, las mejoran.

En este capítulo se mostrará un ejemplo sencillo, un plano formado por dos triángulos y texturizado.

Las texturas son simplemente imágenes digitalizadas que deben tener ciertas características para que la tarjeta gráfica pueda soportarla. Anteriormente el tamaño de las imágenes no pasaba de los 512x512 pixeles, empero la evolución de las GPUs (Graphics Processing Unit) dió cabida para experimentar con mejores imágenes. Así que hasta ahora las imágenes deberán medir con potencias de 2, es decir, imágenes que tengan las siguientes medidas serán aceptadas: 512x512, 512x1024, 256x512, 2048x1024, 1024zx1024, etcétera.

Los formatos de imágenes que podrán agregar en la solución de un proyecto de XNA se muestran en la Tabla 4-1.[6]

Tabla 4-1

Formato	Significado
Bmp	Microsoft Windows bitmap.
Dds	DirectDrawSurface.
Dib	Microsoft Windows bitmap.
Hdr	High dynamic-range.
Jpg	Joint Photographic Experts Group (JPEG) compressed
Pfm	Portable float map.
Png	Portable Network Graphics.
Ppm	Portable pixmap.
Tga	Truevision Targa image.

Cada formato de imagen tiene sus características particulares que las hacen importantes para cada fin; mientras unos son prácticamente una matriz de datos, lo que los hacen mucho más grandes en bits y más completos; otros son menos grandes en bits pero con menor información y además agregan sus códigos para ser decodificados; otros tantos agregan funciones como el canal alfa para crear sprites. Así que en el momento de seleccionar el formato de las imágenes hay que tener en cuenta la calidad, el tamaño en bits y el tiempo que tarda en cargarse. Este texto no abarca este tipo de cuestiones, pues está fuera de su alcance.

4.1 El téxel

De la misma manera que las imágenes tienen al pixel como unidad de medida, para altura y anchura, las texturas tienen al téxel como su unidad, y en muchas de las veces es igual a uno. Además, pueden ser leídos

[6] Para mayor información sobre los formatos de imágenes visite la siguiente página web: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.imagefileformat.aspx

o escritos desde una GPU. Específicamente, un téxel puede ser cualquiera de los formatos de textura disponible representados en la numeración **SurfaceFormat**.[7]

Para conocer qué parte de la textura le corresponde a un vértice es necesario asignarle una coordenada de textura; las coordenadas de texturas se llaman **u** y **v**. El origen de estas coordenadas se sitúa en la esquina superior izquierda de la imagen; la coordenada **u** aumenta hacia la derecha y la coordenada **v** aumenta hacia abajo. La Ilustración 4-1 muestra en cada esquina las coordenadas que le corresponderían, en el caso que la misma figura fuera la unidad.

La esquina superior izquierda el valor de las coordenadas son las mismas, cero; en la esquina superior derecha el valor de la coordenada **u** es uno; en la esquina inferior izquierda la coordenada **v** es uno y la coordenada **u** es cero; y por último, en la esquina inferior derecha las coordenadas tienen el mismo valor, uno.

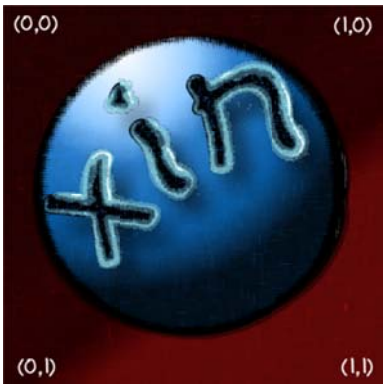

Ilustración 4-1 Textura

4.2 Filtros

Las geometrías no siempre coincidirán con el tamaño de las texturas, es decir, la distancia entre dos vértices puede medir más o medir menos, comparada con una unidad de textura, por lo que puede haber un mayor o menor número de pixeles asociados al téxel. Por lo tanto existen dos modos de sampleo o de muestreo, el de magnificación y el de minificación, que permiten seleccionar el color del pixel final. Los filtros se encargan de calcular ese color del pixel, y XNA ofrece varios de éstos, que están enumerados en **TextureFilter**[8], véase la Tabla 4-2. Tratar de explicar cómo funciona cada filtro está fuera del alcance de este texto; si quiere conocer más, acerca de los tipos de filtros, busque en libros de procesamiento digital de imágenes.

Tabla 4-2

Miembro	Descripción
Anisotropic	*Filtro de textura anisotrópico que se utiliza como filtro de ampliación o reducción de la textura. Este tipo de filtro compensa la distorsión producida por la diferencia de ángulo entre el polígono de la textura y el plano de la pantalla.*[9]
GaussianQuad	Es el filtro Gaussiano con máscara de 4 x 4 para la magnificación y la

[7] Para mayor información visite la siguiente página web: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.surfaceformat.aspx

[8] Para mayor información visite la siguiente página web: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.texturefilter.aspx

[9] Texto tomado de: http://msdn.microsoft.com/es-es/library/microsoft.windowsmobile.directx.direct3d.texturefilter(VS.85).aspx

	minificación.
Linear	*Filtro de interpolación bilineal utilizado como un filtro de ampliación o reducción de la textura. Se utiliza un promedio ponderado de un área 2x2 de téxels (elementos de textura de un píxel) alrededor del píxel deseado. El filtro de la textura utilizado entre los niveles de mipmap es una interpolación de mipmaps trilineal en la que la impresora de trama realiza la interpolación lineal del color del píxel, utilizando los téxels de las dos texturas de mipmap más cercanas.*
None	*Los mipmaps están deshabilitados. En su lugar, la impresora de trama utiliza el filtro de ampliación.*
Point	*Filtro de punto utilizado como un filtro de ampliación o reducción de la textura. Se utiliza el téxel con las coordenadas más próximas al valor de píxel deseado. El filtro de textura utilizado entre los niveles de mipmap se basa en el punto más cercano; es decir, la impresora de trama utiliza el color del téxel de la textura de mipmap más cercana.*[10]
PyramidalQuad	Usa un filtro paso banda con máscara de 4x4 para la magnificación y minificación de la textura.

4.3 Mipmaps

Los mipmaps son una secuencia de texturas que parten de una original, cada textura es la mitad de su tamaño de su antecesora, excepto la primera. Esto ayuda a mejorar la imagen final, quitando ese parpadeo cuando hay cambios bruscos, u otros problemas visuales.

4.4 Plano con textura

Ahora que ya se ha explicado un poco sobre las texturas en el mundo de la computación gráfica, es momento de revisar un ejemplo que maneje una textura sobre una geometría simple. Dos triángulos adyacentes formaran un cuadrado con una textura.

Comience por abrir Visual Studio 2008 y cree un nuevo proyecto para XNA, seleccione la plantilla **Windows Game (3.1)**.

Tomando parte del código del ejemplo anterior, solo se explicaran las nuevas líneas de código.

Ahora agregue las siguientes variables de instancia, dentro de la definición de la clase pero fuera de todo método, en la clase **Game1** (a menos que le haya cambiado el nombre)que hereda de la clase Game.

Código 4-1

```
1.      VertexDeclaration vertexDeclaration;
2.      VertexPositionNormalTexture[] vertices ={
3.          new VertexPositionNormalTexture(new Vector3(-1.0F, -1.0F, -1.0F),
4.              new Vector3(0.0F,0.0F,-1.0F), new Vector2(0.0F,1.0F)),
5.          new VertexPositionNormalTexture(new Vector3(-1.0F,1.0F,-1.0F),
6.              new Vector3(0.0F,0.0F,-1.0F), new Vector2(0.0F,0.0F)),
7.          new VertexPositionNormalTexture(new Vector3(1.0F,1.0F,-1.0F),
8.              new Vector3(0.0F,0.0F,-1.0F), new Vector2(1.0F,0.0F)),
9.          new VertexPositionNormalTexture(new Vector3(1.0F,-1.0F,-1.0F),
10.             new Vector3(0.0F,0.0F,-1.0F), new Vector2(1.0F,1.0F))
11.     };
12.     Int32[] indices = { 0, 1, 2, 0, 2, 3 };
```

[10] Texto tomado de: http://msdn.microsoft.com/es-es/library/microsoft.windowsmobile.directx.direct3d.texturefilter(VS.85).aspx

```
13.     Texture2D textura;
14.     BasicEffect efecto;
15.     Vector3 posicion = new Vector3(0.0F, 0.0F, 2.0F);
16.     Vector3 vista = Vector3.Zero;
17.     Vector3 traslacion = Vector3.Zero;
18.     Single escala = 1.0F;
19.     Single rotacionX = 0.0F;
20.     Single rotacionY = 0.0F;
21.     Single rotacionZ = 0.0F;
```

El tipo de vértice en esta ocasión es **VertexPositionNormalTexture**, línea 2, debe contener dos **Vector3** que contendrán las coordenadas espaciales y la normal respectivamente; y un **Vector2** para las coordenadas de textura.

El objeto **textura**, línea 13, representa una malla 2D de téxeles y cada téxel es direccionado por un vector con coordenadas **u** y **v**.

En el método **Initialize** de la clase **Game1**, agregue las siguientes líneas de código para inicializar el efecto y el dispositivo gráfico.

```
vertexDeclaration = new VertexDeclaration(GraphicsDevice,
VertexPositionNormalTexture.VertexElements);
efecto = new BasicEffect(GraphicsDevice, null);
efecto.TextureEnabled = true;   // se habilita la textura del efecto básico

// modos de sampleo
GraphicsDevice.SamplerStates[0].MagFilter = TextureFilter.Linear;
GraphicsDevice.SamplerStates[0].MinFilter = TextureFilter.Linear;
GraphicsDevice.SamplerStates[0].MipFilter = TextureFilter.Linear;
```

En este caso necesitamos habilitar la propiedad **Textura** del **EffectBasic** que nos proporciona XNA; como puede ver no se ha habilitado el color de los vértices, pues no tiene sentido al carecer éstos de dicha información, pero haga la prueba para que vea que es lo que pasa.

Se inicializa el modo de sampleo del dispositivo gráfico con **SamplerStates**, que recupera una colección de objetos **SamplerState** del **GraphicsDevice**. Y se le asignan a cada una de las propiedades **MagFilter**, **Minfilter** y **MipFilter**; el filtro de tipo **Linear**.

Antes de continuar con el código es necesario añadir la imagen en el proyecto, así que para agregar un nuevo elemento haga clic secundario sobre **Content**, en el **Explorador de Soluciones** de Visual Studio y seleccione **Agregar**, véase Ilustración 4-2.

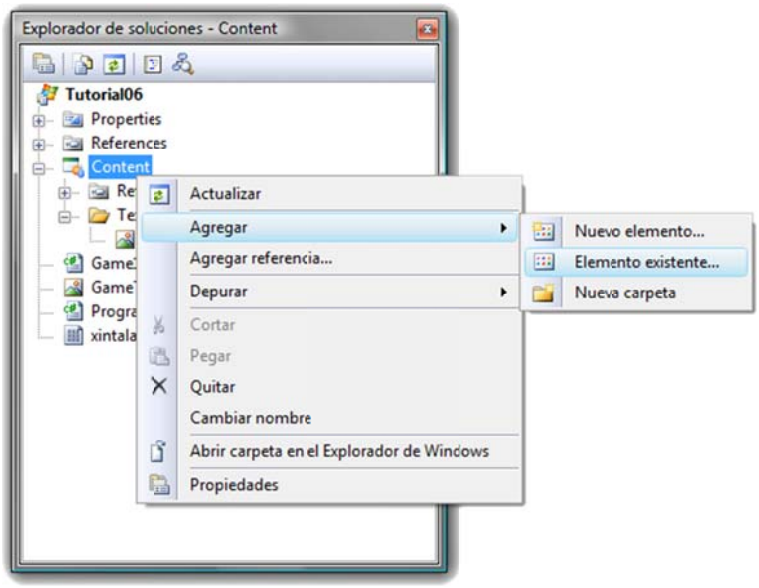

Ilustración 4-2 Agregar\ Elemento

En este caso se agregó una nueva carpeta, llamada **Texturas**, para almacenar las imágenes, y es que es mejor tener las cosas ordenadas. Luego haga lo mismo con la carpeta creada en **Content**, o sea la nueva carpeta creada con el nombre **Texturas**, y agregue **Elemento existente**, véase Ilustración 4-2.

Siguiendo con el código, dentro del método **LoadContent** escriba las siguientes líneas para cargar la textura y generar los **MipMaps**.

```
// Lectura de la textura
textura = Content.Load<Texture2D>(@"Texturas\Xin");
textura.GenerateMipMaps(TextureFilter.Linear);  // creación de los MipMaps
```

Hay dos maneras de cargar una textura desde un archivo, una es por medio el método genérico y estático **ContentManager.Load**[11]. Éste carga un recurso para ser procesado por el **Content Pipeline**. Y el otro es por medio del método **Texture2D.TextureFromFile**.

Lo recomendable usar es el **Content.Load**, el cual recibe un **String** del **Asset Name**. El **Asset Name** es el nombre que se usará como referencia en tiempo de ejecución, por lo que no es necesario colocar la extensión del archivo.

Para saber el **Asset Name**, seleccione el archivo en el **Explorador de soluciones** de Visual Studio, y verá en la ventana **Propiedades** el **String** del **Asset Name**, véase Ilustración 4-3.

[11] Para mayor información revise la siguiente página web: http://msdn.microsoft.com/en-us/library/bb197848.aspx

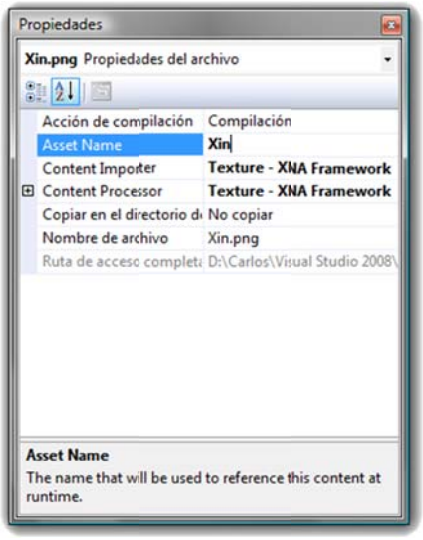

Ilustración 4-3 Asset Name

El **string** que debe recibir el método **LoadContent**, debe incluir los nombres de las carpetas anidadas en directorio asociado en el **ContentManager**, en donde se deposita el archivo creado por un **Digital Content Creation** (DCC)[12]. El **ContentManager** es una clase que carga el contenido del **Content Pipeline** en tiempo de ejecución[13].

Enseguida se generan los mipmaps de la textura con la constante **TextureFilter.Linear** con el que se filtra cada nivel del mipmap, por medio del método **Texture.GenerateMipMaps**.

```
public void GenerateMipMaps(TextureFilter filterType)
```

En el método **UnloadContent** se añade la siguiente línea para liberar el recurso ocupado, que en este caso es la textura.

```
protected override void UnloadContent()
{
    textura.Dispose();
}
```

Para poder apreciar que en realidad se ha colocado la textura sobre una geometría se toman las mismas líneas de transformaciones del ejemplo anterior y se escriben dentro del método **Update**.

```
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Left))
    posicion.X -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Right))
    posicion.X += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Up))
    posicion.Y += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Down))
    posicion.Y -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.PageUp))
    posicion.Z += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.PageDown))
```

[12] La DCC es una herramienta de creación de contenido digital.

[13] Para mayor información revise la siguientes páginas web: http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.content.contentmanager.aspx
http://msdn.microsoft.com/es-es/library/bb447756.aspx

```
    posicion.Z -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.D))

    traslacion.X -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.A))
    traslacion.X += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.W))
    traslacion.Y += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.S))
    traslacion.Y -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Z))
    traslacion.Z += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.X))
    traslacion.Z -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.E))
    escala += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.R))
{
    escala -= 0.1F;
    if (escala < 0.1F)
        escala = 0.1F;
}
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.J))
    rotacionY -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.L))
    rotacionY += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.I))
    rotacionX += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.K))
    rotacionX -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.U))
    rotacionZ += 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.O))
    rotacionZ -= 0.1F;
if (Keyboard.GetState(PlayerIndex.One).IsKeyDown(Keys.Escape))
    this.Exit();
```

La asignación de la textura a la propiedad **Texture** de la instancia **efecto**, línea 13, Código 4-2, es la textura que será aplicada hacia la geometría. El tipo de dato es un **Texture2D**, como lo indica la sintaxis siguiente.

```
public Texture2D Texture { get; set; }
```

El método **EnableDefaultLighting**, línea 14, habilita la iluminación por omisión de este efecto. La propiedad **PreferPerPixelLighting**, línea 15, obtiene o establece que la iluminación por píxel puede ser soportada por el dispositivo gráfico, o sea la GPU. Así que si no tiene una GPU con soporte mínimo para Pixel Shader 2.0 no escriba esta línea de código.

Código 4-2

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(color);
4.
5.          Single aspecto = GraphicsDevice.Viewport.AspectRatio;
6.          efecto.World = Matrix.Identity * Matrix.CreateScale(escala) *
7.              Matrix.CreateRotationX(rotacionX) *
8.              Matrix.CreateRotationY(rotacionY) * Matrix.CreateRotationZ(rotacionZ) *
9.              Matrix.CreateTranslation(traslacion);
10.         efecto.View = Matrix.CreateLookAt(posicion, vista, arriba);
11.         efecto.Projection = Matrix.CreatePerspectiveFieldOfView(1, aspecto, 1, 1000);
12.
13.         efecto.Texture = textura;
14.         efecto.EnableDefaultLighting(); // Iluminación por default del efecto básico
15.         efecto.PreferPerPixelLighting = true;   // iluminación por pixel
16.
17.         GraphicsDevice.RenderState.FillMode = FillMode.Solid;
```

```
18.         GraphicsDevice.VertexDeclaration = vertexDeclaration;
19.         GraphicsDevice.RenderState.CullMode = CullMode.None;
20.
21.         // comienza el trazado de la geometría
22.         efecto.Begin();
23.         efecto.CurrentTechnique.Passes[0].Begin();
24.         GraphicsDevice.DrawUserIndexedPrimitives<VertexPositionNormalTexture>(
25.             PrimitiveType.TriangleList,
26.             vertices, 0, vertices.Length, indices, 0, 2);
27.         efecto.CurrentTechnique.Passes[0].End();
28.         efecto.End();
29.
30.         base.Draw(gameTime);
31.     }
```

Ahora inicie la depuración oprimiendo **F5** para poder ver el plano con la textura pegada a ésta. Si todo salió bien, podrá ver una imagen similar a la Ilustración 4-4, si tiene algún error de compilación corrija y vuelva a intentar.

Ilustración 4-4 Plano con textura

Sobre todo mueva el modelo con el teclado para que pueda apreciar los cambios, como se muestra en la Ilustración 4-5.

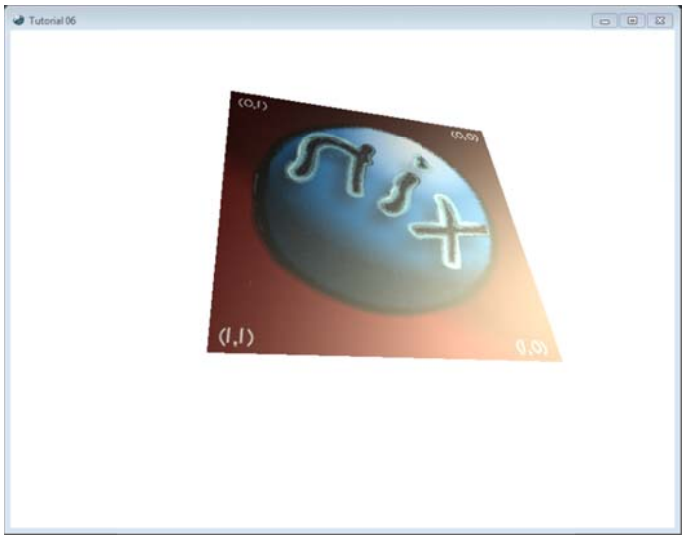

Ilustración 4-5 Transformaciones sobre el plano texturizado

4.5 Modos de direccionamiento

Hasta ahora las coordenadas de textura asignadas tienen valores de cero o uno, para cubrir toda la geometría con la imagen, pero ¿qué pasaría si estos valores fueran mayores a uno? Bueno la manera en que la textura ahora envolverá a la geometría dependerá del tipo de direccionamiento que se le asigne a las coordenadas **u** y **v**.

Para poder ver los modos de direccionamiento se cambiaran las coordenadas de textura por las siguientes.

```
VertexPositionNormalTexture[] vertices ={
    new VertexPositionNormalTexture(new Vector3(-1.0F, -1.0F, -1.0F),
         new Vector3(0.0F,0.0F,-1.0F),
         new Vector2(0.0F,5.0F)),
    new VertexPositionNormalTexture(new Vector3(-1.0F,1.0F,-1.0F),
        new Vector3(0.0F,0.0F,-1.0F),
        new Vector2(0.0F,0.0F)),
    new VertexPositionNormalTexture(new Vector3(1.0F,1.0F,-1.0F),
        new Vector3(0.0F,0.0F,-1.0F),
        new Vector2(5.0F,0.0F)),
    new VertexPositionNormalTexture(new Vector3(1.0F,-1.0F,-1.0F),
        new Vector3(0.0F,0.0F,-1.0F),
        new Vector2(5.0F,5.0F))
};
```

El modo de direccionamiento **Wrap** será el primero en revisar, éste sirve para repetir la textura sobre la geometría de manera que cubra toda.

En el método **Initialize** agregue las siguientes líneas de código, después de la asignación del filtro lineal.

```
GraphicsDevice.SamplerStates[0].AddressU = TextureAddressMode.Wrap;
GraphicsDevice.SamplerStates[0].AddressV = TextureAddressMode.Wrap;
```

En este caso se deja el mismo modo de direccionamiento para las coordenadas **u** y **v**, pero no significa que así sea.

AddressU y **AddressV** son propiedades que obtienen o establecen el direccionamiento sobre las coordenadas **u** y **v**, respectivamente. **TextureAddressMode** es una numeración que define el modo de direccionamiento de la textura.

Ejecute el programa y corrija cualquier error de compilación que sucediera, si tiene éxito logrará algo similar a la Ilustración 4-6.

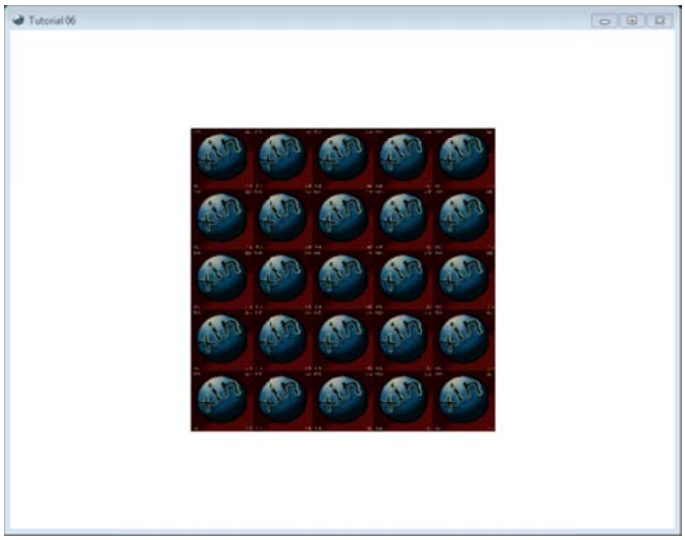

Ilustración 4-6 Wrap

El siguiente modo de direccionamiento corresponde a **Clamp**, las coordenadas de textura fuera del rango [0.0, 1.0] se definen con el color de la última columna y renglón de la textura para rellenar. Cambie la numeración de **TextureAddressMode.Wrap** a **TextureAddressMode.Clamp**.

```
GraphicsDevice.SamplerStates[0].AddressU = TextureAddressMode.Clamp;
GraphicsDevice.SamplerStates[0].AddressV = TextureAddressMode.Clamp;
```

Inicie el programa y corrija cualquier error de compilación que suceda, si tiene éxito podrá ver algo similar a la Ilustración 4-7.

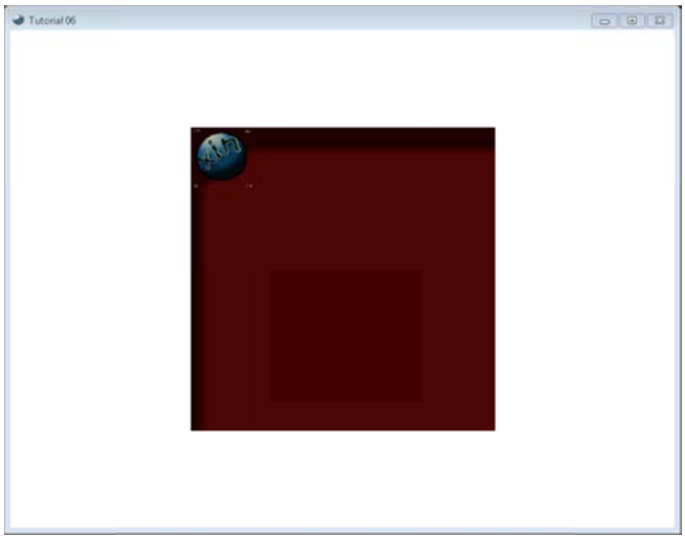

Ilustración 4-7 Clamp

El direccionamiento **Mirror** es similar al **Wrap**, sin embargo, en el momento de cubrir la geometría la siguiente imagen estará invertida, como si se estuviera viendo en un espejo; esto sucede hasta que se termine de cubrir la geometría.

Cambie la numeración **TextureAddressMode.Clamp** por **TextureAddressMode.Mirror** en las propiedades **AddresU** y **AddresV**.

```
GraphicsDevice.SamplerStates[0].AddressU = TextureAddressMode.Mirror;
GraphicsDevice.SamplerStates[0].AddressV = TextureAddressMode.Mirror;
```

Oprima **F5** para comenzar con la depuración y así correr el programa, si ocurre cualquier error de compilación o de tiempo de ejecución soluciónelo y vuelva a intentar. Si lo ha conseguido verá una imagen como la que se muestra en la Ilustración 4-8.

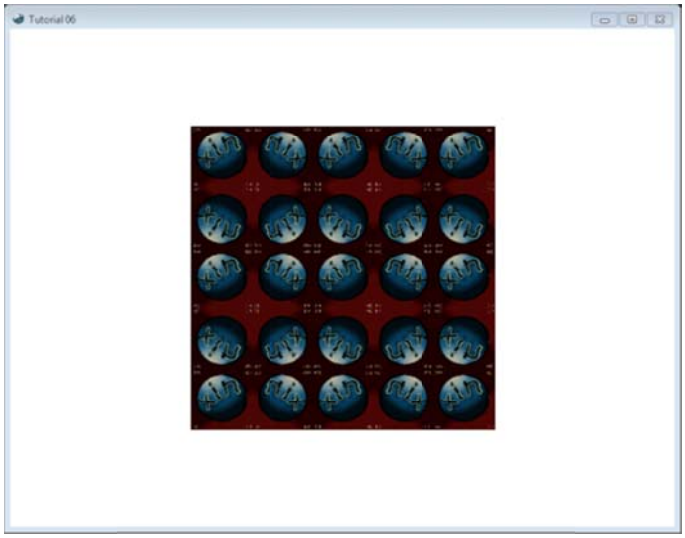

Ilustración 4-8 Mirror

Modo de direccionamiento **Border**, aquí se coloca una vez la textura sobre la geometría y el resto, si es que tiene, será rellenado por un color.

Cambie la numeración del modo de direccionamiento de las coordenadas **u** y **v** por **Border**, y agregue el color al borde. **BorderColor** es una propiedad que obtiene o establece el color del borde.

```
GraphicsDevice.SamplerStates[0].AddressU = TextureAddressMode.Border;
GraphicsDevice.SamplerStates[0].AddressV = TextureAddressMode.Border;
GraphicsDevice.SamplerStates[0].BorderColor = Color.LightBlue;
```

Inicie con la depuración del programa con **F5**, y corrija cualquier error para poder ver algo similar a la Ilustración 4-9.

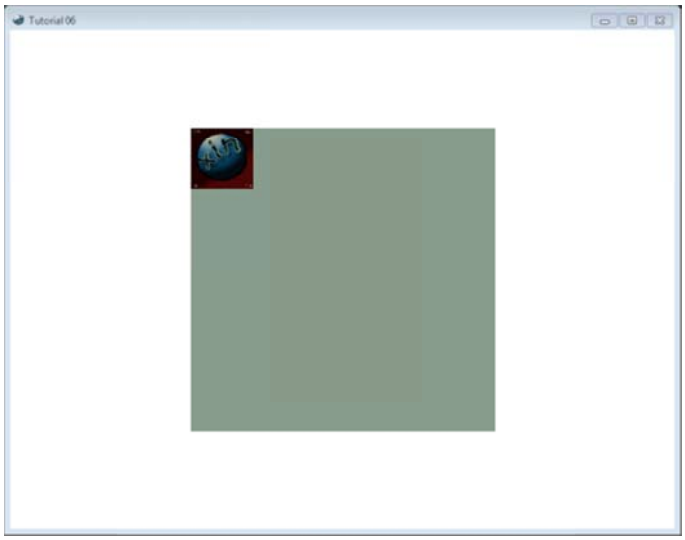

Ilustración 4-9 Border

Regularmente los modos de direccionamiento son iguales para **u** y **v**, sin embargo, pueden crearse combinaciones entre las distintas propiedades.

5 Texto

En esta parte del tutorial no se verá nada del mundo 3D, que más tiene en interés al autor, y es que hasta este momento XNA 3.1 no tiene una librería del texto en 3D, como si lo tiene DirectX u OpenGL, pero que de todas formas se tomará en cuenta en este escrito.

El tipo de fuente que se utiliza en XNA es prácticamente toda aquella que se tenga instalada en el Sistema Operativo o que se pueda agregar como una textura de una fuente. Las primeras se agregarán al proyecto como una descripción del Sprit Font, así que debe tener instalado la fuente. La segunda toma una imagen con los caracteres de la fuente, por lo que no necesita la fuente instalada en la computadora.

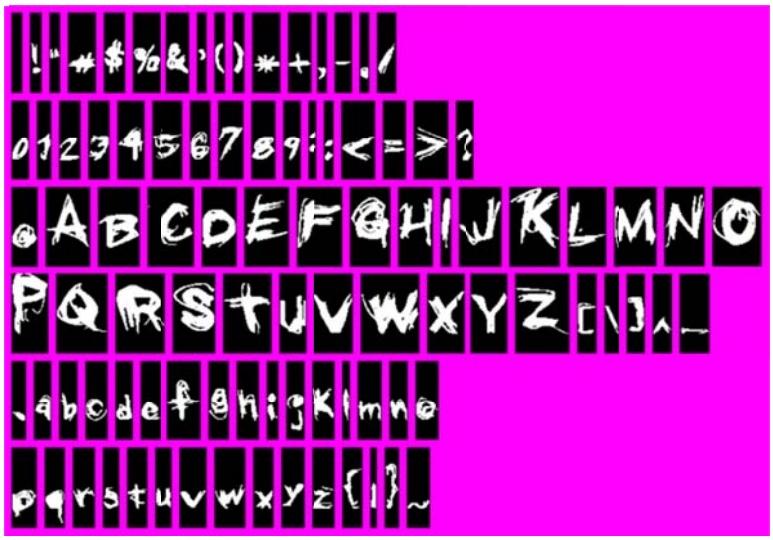

Ilustración 5-1 Sprite Font Texture

Para comenzar este ejemplo, necesitará descargar las imágenes de las fuentes que se agregaron en la misma carpeta en que descargo este tutorial. Pero si quieren crear las suyas pueden descargar la herramienta **Bitmap Font Maker de XNA CREATORS CLUB ONLINE**[14]. Estas texturas serán procesadas por el Content Pipeline de XNA como **Sprite Font Texture**, que es uno de los procesos estándar que proporciona XNA.

Lo que hace el proceso **Sprite Font Texture** es tomar una entrada, que en este caso es de tipo **Texture2DContent** que representa una textura regular de dos dimensiones; y la transforma en una fuente dando como salida a un **SpriteFontContent**. Esto lo hace al cambiar los pixeles de **Color.Magenta** a **Color.TransparentBlack**, y claro estas texturas tienen un canal alfa para hacer transparentes las regiones oscuras.

5.1 SpriteFont

En este ejemplo sólo se mostrará un conjunto de enunciados que muestran el uso de texto en XNA. Así que comenzaremos por crear un proyecto nuevo de XNA Game Studio 3.1, utilizando la plantilla **Windows Game (3.1)**.

Ahora hay que agregar el **SpritFont**, esto se hace en el explorador de soluciones del proyecto. Como ya se había visto anteriormente, coloque el cursor sobre el icono de **Content** y oprima el botón derecho del ratón para agregar un nuevo elemento al proyecto, como se muestra en la Ilustración 5-2.

[14] http://creators.xna.com/es-ES/utilities/bitmapfontmaker

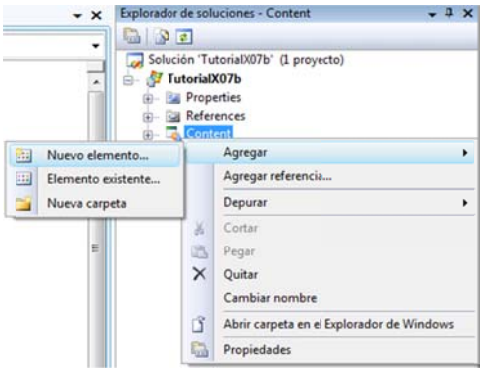

Ilustración 5-2 Nuevo elemento…

Inmediatamente se abre una ventana diálogo, **Agregar nuevo elemento – Content**, como se muestra en la Ilustración 5-3. Seleccione la plantilla **Sprite Font**, es la que muestra una letra A, cambie el nombre por el de la fuente que quiere colocar en la aplicación y dé clic en el botón **Agregar**; no es necesario que el nombre del **SpriteFont** sea el mismo que el de la fuente, es sólo por orden. Recuerde que para poder utilizar el **SpritFont** debe tener instalada la fuente en el Sistema Operativo.

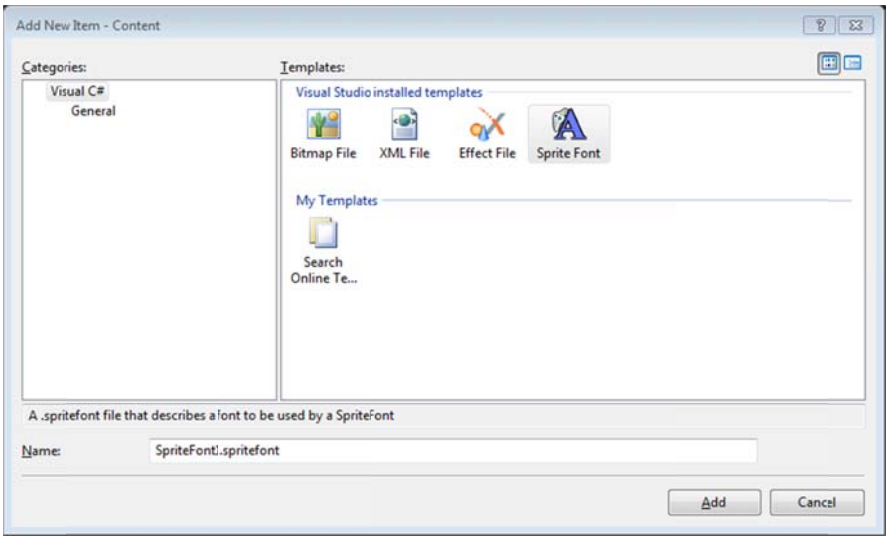

Ilustración 5-3 Agregar un Sprite Font

Visual Studio abrirá de inmediato el archivo ***.spritefont** que agregamos al proyecto anteriormente. Como podrá ver es un archivo XML que contiene la descripción de una fuente y que será leído por el Content Pipeline de XNA; no es necesario que aprenda sobre XML para poder agregar texto en XNA, sin embargo, sería una buena idea hacerlo porque usted mismo podría crear sus propia definición de documentos o esquema para instanciar los modelos gráficos dentro de un mundo 3D, dándoles su poción, orientación y nombre a cada uno de ellos. Además que .NET tiene su librería para manipular XML.

En fin, el XML contiene diez etiquetas que corresponden a las propiedades que comúnmente tienen las fuentes, las cuales se presenta en la Tabla 5-1[15].

Tabla 5-1

Etiqueta	Tipo de dato	Descripción

[15] Para más información revise la siguiente dirección: http://msdn.microsoft.com/en-us/library/bb447759.aspx

<FontName>	String	Es el nombre de la fuente que se tiene instalada en el Sistema Operativo, y que prácticamente toma cualquiera que esté, excepto las de tipo *.fon.
<Size>	Single	Es el tamaño en punto flotante de la fuente.
<Spacing>	Single	Es el número de píxeles a añadir entre cada carácter cuando la cadena se dibuja.
<UseKerning>	Boolean	Especifica si será usado el ajuste de espacio cuando se dibuje la fuente. Su valor por default es true.
<Style>	"Regular," "Bold," "Italic," o "Bold, Italic"	Es el estilo de la fuente a ser importada.
<DefaultCharacter>	Char	Es el carácter Unicode que substituirá al carácter que no se encuentra en la fuente importada. La especificación de esta etiqueta es opcional.
<CharacterRegions>	Uno o más etiquetas <CharacterRegion>	Uno o más rangos numéricos indicando que subconjunto de caracteres Unicode será importado.
<CharacterRegion>	Una etiqueta <Start> y una <End>	Es el inicio y el final de una región de caracteres Unicode.
<Start>	Char	Es el primer carácter Unicode a incluir en un <CharacterRegion>
<End>	Char	Es el último carácter Unicode a incluir en un <CharcaterRegion>

En la línea 4 del Código 5-1 el nombre de la fuente se ha cambiado por **Kids**, en la línea 6 se ha puesto el valor de 20 para que se pueda apreciar la cadena en la ilustración que enseguida aparecerá; el valor del espaciado será de 2, línea 8, pues por default es 0 y a veces por el tipo de fuente queda muy amontonado como le ocurre a **Kids**. También se ha des comentado la etiqueta **DefaultCharacter**, por si hubiera alguna letra de la cadena que no pueda ser encontrada en el intervalo de caracteres de código ASCII imprimible que se le ha asignado, líneas 18 y 19.

Código 5-1

```
1.      <?xml version="1.0" encoding="utf-8"?>
2.      <XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
3.        <Asset Type="Graphics:FontDescription">
```

```
4.          <FontName>Kids</FontName>
5.
6.          <Size>20</Size>
7.
8.          <Spacing>2</Spacing>
9.
10.         <UseKerning>true</UseKerning>
11.
12.         <Style>Regular</Style>
13.
14.         <DefaultCharacter>*</DefaultCharacter>
15.
16.         <CharacterRegions>
17.           <CharacterRegion>
18.             <Start>&#32;</Start>
19.             <End>&#126;</End>
20.           </CharacterRegion>
21.         </CharacterRegions>
22.       </Asset>
23.     </XnaContent>
```

Dejando a un lado el XML, es momento de regresar a C#, y comenzaremos declarando una instancia de **SpritFont** en las variables de instancia de la clase **Game1** que generó Visual Studio en el archivo **Game1.cs**.

Código 5-2

```
1.      public class Game1 : Microsoft.Xna.Framework.Game
2.      {
3.          GraphicsDeviceManager graphics;
4.          SpriteBatch spriteBatch;
5.          SpriteFont kids;
6.
7.          public Game1()
8.          {
9.              graphics = new GraphicsDeviceManager(this);
10.             Content.RootDirectory = "Content";
11.             this.Window.Title = "TutorialX07";
12.             this.Window.AllowUserResizing = true;
13.             this.IsMouseVisible = true;
14.         }
15.
16.         protected override void Initialize()
17.         {
18.             base.Initialize();
19.         }
20.
21.         protected override void LoadContent()
22.         {
23.             spriteBatch = new SpriteBatch(GraphicsDevice);
24.
25.             kids = Content.Load<SpriteFont>("Kids");
26.         }
27.
28.         protected override void UnloadContent()
29.         {
30.         }
31.
32.         protected override void Update(GameTime gameTime)
33.         {
34.             if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
35.                this.Exit();
36.
37.             base.Update(gameTime);
38.         }
39.
40.         protected override void Draw(GameTime gameTime)
41.         {
42.             GraphicsDevice.Clear(Color.White);
43.
44.             spriteBatch.Begin();
45.             spriteBatch.DrawString(kids, "¡Hola mundo!", new Vector2(0.0F, 0.0F),
```

```
46.                 Color.Black);
47.             spriteBatch.End();
48.
49.             base.Draw(gameTime);
50.         }
51.     }
```

En el método **LoadContet**, líneas 21 – 26 Código 5-2, se carga el ***.spritfont** que se añadió en el **Content**. El tipo de dato que se usa es de tipo **SpritFont** en el método genérico **Content.Load**. Hay que recordar que sólo se debe tomar el **Asset Name** con el que **ContentManaged** lo identifica, no se necesita la extensión del archivo.

Como último paso, hay que mandar a dibujar el **SpriteFont** en el método **Draw**, líneas 40 - 50. Para comenzar hay que preparar el dispositivo gráfico para dibujar los sprites con **SpriteBatch.Begin**. Enseguida se dibuja la cadena con el método **SpritBatch.DrawString** que tiene seis sobrecargas, en este caso se seleccionó el primero que nos da el **IntelliSense**.

```
public void DrawString(SpriteFont spriteFont, string text, Vector2
position, Color color)
```

Cuyos parámetros se muestran en la Tabla 5-2.

Tabla 5-2

Parámetro	Descripción
spriteFont	El sprit de la fuente.
text	La cadena a dibujar.
position	La localidad, en coordenadas de la pantalla, donde será dibujado el texto.
color	Es el color para el texto.

Inicie el programa con la tecla **F5**, para poder ver algo similar a la Ilustración 5-4, si surge cualquier error de compilación encuentre el problema para resolverlo y trate de nuevo.

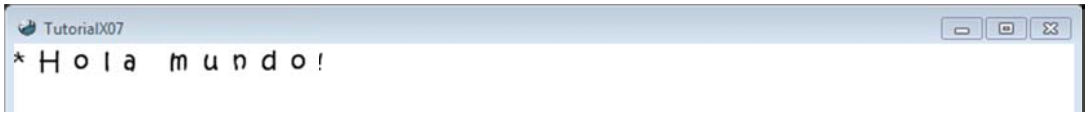

Ilustración 5-4 Carácter desconocido

Como se puede ver, cuando se agregan caracteres con acentos, diéresis o la letra “ñ”, no podrán mostrarse, a cambio se dibuja el signo que se le dio a la etiqueta **DefaultCharacter** del **Spritfont**; esto es porque dichos caracteres no se en encuentran dentro del intervalo que se le dio al XML[16]. Para resolver esto, hay que agregar otro intervalo de caracteres ASCII o aumentar el número en que termina el intervalo.

```
<CharacterRegions>
    <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#255;</End>
    </CharacterRegion>
```

[16] Véase la siguiente dirección para revisar en que intervalo del código ASCII se encuentran ciertos caracteres http://msdn.microsoft.com/en-us/library/4z4t9ed1(VS.71).aspx

En este caso se aumentó el intervalo, para mostrar los signos de puntuación que se ocupan en el español. Vuelva a correr el programa y verá que el mensaje escrito será el que quería mostrar.

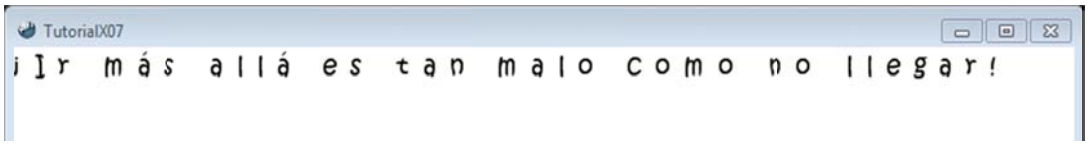

Ilustración 5-5 Aumenta el rango de caracteres reconocibles

Las demás sobrecargas del método **SpriteBatch.DrawString** para dibujar texto, se dejan como ejercicio para el lector.

5.2 Sprite Font Texture

En caso que el tipo de fuente no se encuentre instalado en el Sistema Opertativo, o se quiera enriquecer más el estilo de la fuente se podría exportar al programa una imagen con las letras necesarias que se ocupan en el proyecto, para ello el **Sprite Font Texture** es una buena solución.

Comience por agregar un nuevo elemento al **Content**; esta vez será un archivo de imagen como se muestra en la Ilustración 5-6. A estas alturas ya sabrá como agregar nuevos elementos en el **Content**, así que omitiremos esos pasos. La imagen debe tener ciertas propiedades para que el **Content Pipeline** pueda procesarla, y ésta son que tenga un color magenta y este habilitado el canal Alfa.

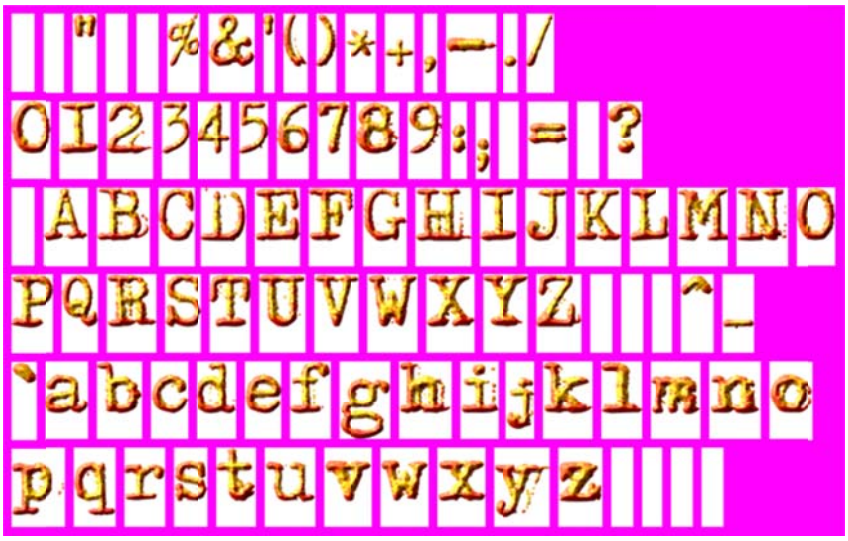

Ilustración 5-6 Adler

Sin embargo, hay que hacer unas modificaciones en el **Content Processor**, que es el que se encargara en procesar la imagen de forma adecuada para crear un **Sprite Font Texture**. Así que diriga el icono de la imagen que acaba de agregar en el **Content**, lo anterior se realiza en el explorador de soluciones, posteriormente haga clic para poder ver las propiedades del archivo. Por default, el valor que tiene la propiedad **Content Processor** de la lista **XNA Framework Content Pipeline**, es **Texture – XNA Framework**, véase Ilustración 5-7.

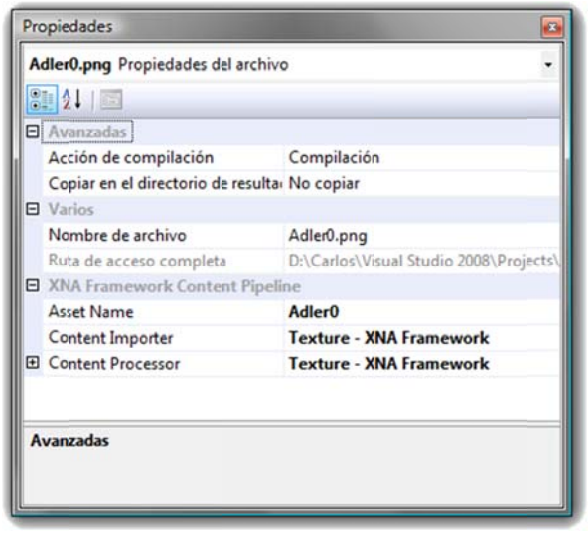

Ilustración 5-7 Propiedades de una imagen

Cambie la manera en que el **Content Pipeline** procesará la imagen por **Sprite Font Texture – XNA Framework**, como se ve en la Ilustración 5-8.

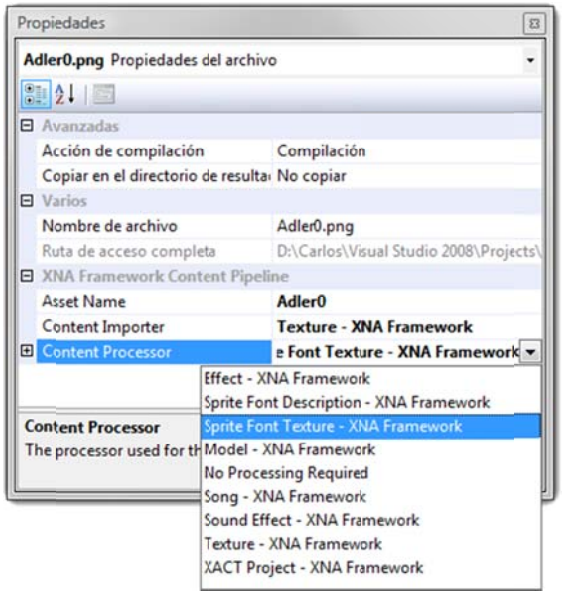

Ilustración 5-8 Cambiando el procesador de contenido

Cabe aclarar que la textura que ha tomado el **Content Pipeline** debe tener todos los caracteres necesarios para poder dibujar la cadena deseada, porque si no es así, también abra que dejar un carácter por default en caso que no exista.

Ahora hay que agregar el **Sprite Font Texture** al código, comience por declarar una variable de instancia en la clase **Game1**, línea5 Código 5-3.

En el método **LoadContent**, líneas 22 -29, al igual que en el ejemplo anterior, se carga el **Sprit Font Texture** con el nombre del **Asset**. Y es aquí donde se modifican las propiedades del **SpriteFont**, el más importante es el **SpriteFont.DefaultCharacter**, línea 26, pues no falta el símbolo que no se pueda representar y dé un error en tiempo de ejecución, hay que asegurarse también que el carácter que se tome por default esté en el **Sprite Font Texture**, porque si así no fuera se caería en el mismo problema y esto también lo es en el XML del ejemplo anterior.

Código 5-3

```
1.      public class Game1 : Microsoft.Xna.Framework.Game
2.      {
3.          GraphicsDeviceManager graphics;
4.          SpriteBatch spriteBatch;
5.          SpriteFont adler;
6.
7.          public Game1()
8.          {
9.              graphics = new GraphicsDeviceManager(this);
10.             Content.RootDirectory = "Content";
11.             this.Window.Title = "TutorialX07";
12.             this.Window.AllowUserResizing = true;
13.             this.IsMouseVisible = true;
14.
15.         }
16.
17.         protected override void Initialize()
18.         {
19.             base.Initialize();
20.         }
21.
22.         protected override void LoadContent()
23.         {
24.             spriteBatch = new SpriteBatch(GraphicsDevice);
25.             adler = Content.Load<SpriteFont>("Adler0");
26.             adler.DefaultCharacter = '*';
27.             adler.LineSpacing = 0;
28.             adler.Spacing = 0;
29.         }
30.
31.         protected override void UnloadContent()
32.         {
33.         }
34.
35.         protected override void Update(GameTime gameTime)
36.         {
37.             if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
38.                 this.Exit();
39.             base.Update(gameTime);
40.         }
41.
42.         protected override void Draw(GameTime gameTime)
43.         {
44.             GraphicsDevice.Clear(Color.Black);
45.
46.             spriteBatch.Begin();
47.             spriteBatch.DrawString(adler, DateTime.Now.TimeOfDay.ToString(),
48.             new Vector2(GraphicsDevice.Viewport.Width / 4.5F,
49.                 GraphicsDevice.Viewport.Height / 2),
50.                 Color.White);
51.             spriteBatch.End();
52.             base.Draw(gameTime);
53.         }
54.     }
```

En el método **Draw**, líneas 42 - 53, como en el ejemplo anterior de dibuja la cadena con el mismo método sobrecargado **SpriteBatch.Drawing**, línea 47. Sin embargo, esta vez se mostrará el reloj del sistema como la cadena; para ello se obtiene la propiedad **DateTime.Now.TimeOfDay** en forma de **String**. Y la posición de la cadena será tomando en cuenta el ancho y alto del Viewport. Para que no se vea alterado el color de la cadena, pues se pretende que el **Sprite Font Texture** tiene los colores y efectos que no nos puede dar el XML, se deja en **Color.White**.

Por fin, esto es todo con respecto a texto en XNA, o por lo menos lo más básico para colocar información valiosa en el Viewport, pues a veces no sabemos qué valores toma los vértices de nuestros modelos 3D o de colisión en tiempo real.

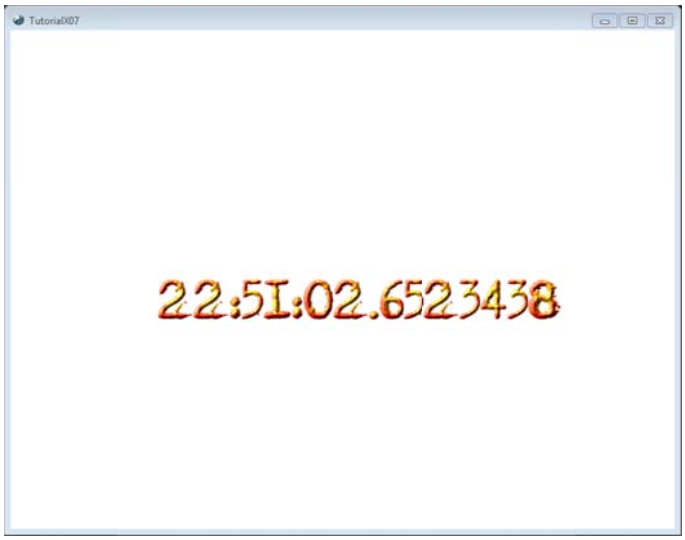

Ilustración 5-9 Reloj

Inicie el programa con la tecla **F5** y corrija cualquier error de compilación si así lo necesitará; verá un lindo reloj más o menos como la Ilustración 5-9.

6 Cómo cargar modelos 3D desde un archivo X y FBX

Máquinas que crean máquinas, software que crea software, así es cómo se hacen las cosas, pero todo tuvo un comienzo, es por eso que se explicó lo que era el búfer de vértices y el búfer de índices. Sin embargo, para crear aplicaciones más complejas es necesario apoyarse de herramientas ya construidas para no volver a reinventar la rueda. Por eso, en esta parte, se hará mención a la carga de modelos 3D, que al fin de cuentas fueron hechos en herramientas de modelación, como pueden ser 3D MAX®, XSI®, MAYA®, AutoCAD®, etc.

6.1 Formatos de archivos

Los modelos 3D se guardan en memoria secundaria, por lo que se tiene un archivo con toda la información necesaria para representar en XNA lo que se tenía en el modelador.

Estos archivos tienen su propia estructura para almacenar los vértices, las coordenadas de textura, las normales, los índices de los vértices, los índices de las coordenadas de textura, los materiales, etcétera. Algunos de los formatos de estos archivos son ***.3ds**, ***.x**, ***.obj**, ***.fbx**, ***.md5**, ***.dwg**, ***.max**, ***.exp**, etcétera. Sin embargo, no se entrará en detalle qué información guarda cada uno de ellos, pues estaría lejos del alcance de este escrito, eso y porque algunos están protegidos.

XNA no puede abrir todos los diferentes archivos que generan los modeladores 3D, esto le correspondería al programador. Para nuestra fortuna hay dos tipos de archivos que puede abrirse y cargarse, sin complicarnos la vida, estos son los ***.x** y los ***.fbx**.

El formato ***.x** fue introducido en el **DirectX 2.0** y que ahora puede estar codificado en forma binaria o en texto plano[17].

El formato ***.fbx** es una de las tecnologías de Autodesk, cuyo objetivo es la portabilidad de los datos 3D a través de las diferentes soluciones que existen en el mercado, también puede presentarse en forma binaria o en texto plano[18].

La forma de codificar los archivos, ya sea binaria o en texto, dependerá en qué es lo que le conviene mejor para la aplicación, o para el desarrollador. Cuando el texto es plano, es fácil revisar, por si llegará a haber algún problema a la hora cargar el modelo 3D en la solución, puesto que es texto que nosotros como humanos podemos interpretar de forma rápida, otra de las características importantes es que el archivo ocupa menos memoria en comparación con el binario. Por otra parte, los archivos que se guardan en forma binaria son muy fáciles de interpretar por la computadora, en comparación con el texto plano.

6.2 La clase Model

Para manipular de una manera más sencilla la información obtenida de un modelo 3D, mucho más elaborado, XNA ofrece una clase llamada **Model** y la ayuda de su **ContentManaged** para cárgalo. Las propiedades en las que se conserva la información de los modelos, de la clase **Model**, son **Meshes** y **Bones**.

Meshes: es una colección de objetos **ModelMesh** que componen el modelo, y cada **ModelMesh** puede moverse independientemente de otro y ser compuesto por múltiples materiales identificados como objetos **ModelMeshPart**.

Bones: es una colección de objetos **ModelBone** que describen cómo cada uno de los mesh en la colección **Meshes** de este modelo se relaciona con su mesh padre.

Pero ¿qué es el mesh?, el mesh como tal es donde se guarda la información de la geometría, por lo que tiene un vertex buffer y un index buffer, además de otros datos.

La clase **ModelMeshPart** es un lote de información de la geometría a enviar al dispositivo gráfico durante el rendering. Cada **ModelMeshpart** es una subdivisión de un objeto **ModelMesh**, así que la clase

[17] Para mayor información revise la siguiente liga http://msdn.microsoft.com/en-us/library/bb174837(VS.85).aspx

[18] Para mayor información revise la siguiente liga http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=6837478

ModelMesh está formada por varios objetos **ModelMeshpart**, y que típicamente se basan en la información del material. Por ejemplo, si en una escena se tiene una esfera y un cubo, y cada uno de ellos tiene diferentes materiales, éstos se representaran como objetos **ModelMeshpart** y la escena se representará como un **ModelMesh**. En la Ilustración 6-1, se muestra un **ModelMesh** que contiene tres **ModelMeshpart**.

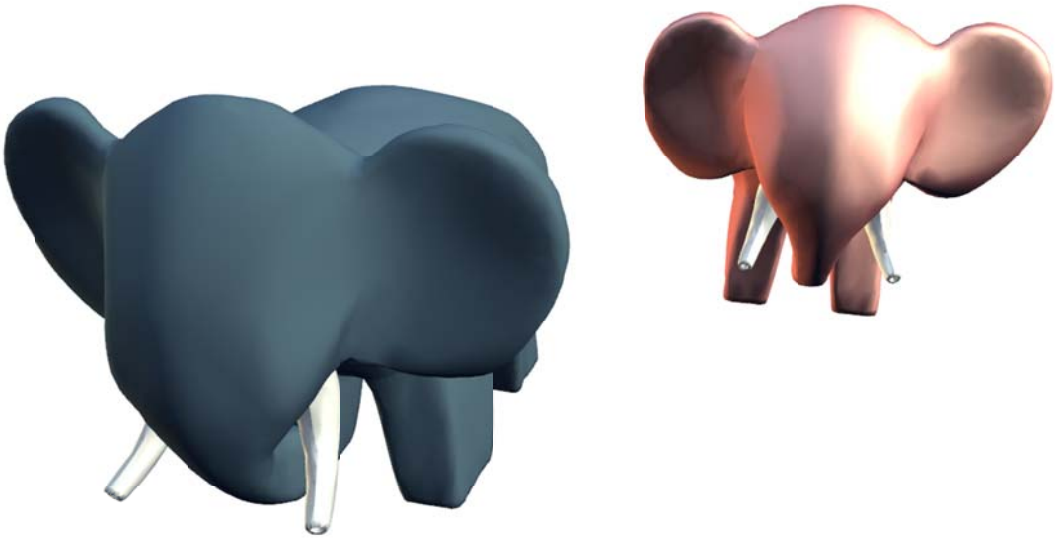

Ilustración 6-1 Tres ModelMeshpart diferentes

En fin, creo que ha sido momento de tener el ejemplo de cómo cargar un modelo 3D en una solución de XNA. El primer ejemplo será muy simple para comprender un poco más el cómo aprovechar las propiedades de la clase **Model**. (Cómo leer el búfer de vértices de un **ModelMeshPart** para crear nuestra propia envolvente de colisión)[19] Así que no se desesperen si quieren una solución rápida para sus necesidades, es mejor ir por las piedritas.

[19] Esto debido a la pregunta que surgía en los foros, ¿cómo leo el vertex buffer de un mesh?

6.3 Ejemplo 01

El siguiente ejemplo muestra cómo cargar un modelo 3D, un elefante, así que si quieren probar éste ejemplo con un archivo que contenga más de un modelo 3D, solo se verá uno. Tampoco será la mejor manera de hacerlo, pero creo menester para explicarlo. Pero no se preocupen, en los siguientes ejemplos de este capítulo se hará mejor la codificación.

Abra Visual Studio para crear una nueva solución de XNA, llegado hasta este punto no creo que sea necesario explicar cómo crear una nueva solución, así que enseguida añada un nuevo archivo ***.x** o ***.fbx**. XNA tiene predefinido tomar estos dos tipos de archivos para procesarlos. Para este ejemplo se utilizó la figura de un elefante, **Elefante.fbx**, que pueden descargar del mismo sitio en donde descargaron este texto. Es de esperarse que el **ContentManaged** sea el encargado de tomar estos tipos de archivos, así que en el momento de agregar el archivo, no hay que olvidar que es en el **Content** en donde hay que agregar el archivo, a menos que cambie la carpeta raíz del **Content**.

En el archivo **Game1.cs** escriba las siguientes líneas de código fuera del constructor pero dentro de la clase **Game1**:

```
Model modelo;
BasicEffect basicEffect;
```

La primera línea es un objeto de la clase **Model**, que no creo sea necesario explicar para qué sirve, el segundo objeto es el efecto básico que nos ofrece XNA, para mandar a dibujarlo.

En el método **LoadContent** de la clase **Game1** hay que leer el contenido del archivo **Elefante**, y se hará con ayuda del método genérico **Load del Content**.

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    modelo = Content.Load<Model>("Elefante");
}
```

Hay que recordar que el nombre que se le pasa como parámetro al método debe ser el que está en el **Asset name**.

Hasta el momento todo esto ha sido conocido, eso espero, así que hay que pasar al método **Draw**. Y es en esta parte que a muchos les parecerá mal, pero esa no es la intención, es nada más para que pueda explicarse mejor el ejemplo, así que la optimización se deja como ejercicio para el lector.

Código 6-1

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(Color.Black);
4.
5.          Matrix[] transformaciones = new Matrix[modelo.Bones.Count];
6.          modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
7.          // preparando el efecto
8.          basicEffect = (BasicEffect)modelo.Meshes[0].Effects[0];
9.          basicEffect.World = transformaciones[modelo.Meshes[0].ParentBone.Index];
10.         basicEffect.Projection = Matrix.CreatePerspectiveFieldOfView(1.0F,
11.             GraphicsDevice.Viewport.AspectRatio,
12.             1.0F, 1000.0F);
13.         basicEffect.View = Matrix.CreateLookAt(new Vector3(150.0F, 25.0F, 250.0F),
14.             new Vector3(-50.0F, 0.0F, 0.0F), Vector3.Up);
15.         basicEffect.EnableDefaultLighting();
16.
17.         ModelMesh modelMesh = modelo.Meshes[0];
18.         ModelMeshPart meshPart = modelMesh.MeshParts[0];
19.
```

```
20.         // preparando el dispositivo
21.         GraphicsDevice.Vertices[0].SetSource(
22.             modelMesh.VertexBuffer, meshPart.StreamOffset, meshPart.VertexStride);
23.         GraphicsDevice.VertexDeclaration = meshPart.VertexDeclaration;
24.         GraphicsDevice.Indices = modelMesh.IndexBuffer;
25.
26.         basicEffect.Begin(SaveStateMode.None);
27.         basicEffect.CurrentTechnique.Passes[0].Begin();
28.         GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
29.             meshPart.BaseVertex, 0, meshPart.NumVertices, meshPart.StartIndex,
30.             meshPart.PrimitiveCount);
31.         basicEffect.CurrentTechnique.Passes[0].End();
32.         basicEffect.End();
33.
34.         base.Draw(gameTime);
35.     }// fin del método Draw
```

En la línea 5, Código 6-1, se crea un arreglo de matrices con dimensión igual al número de objetos **ModelBone** que contenga el modelo. Esto servirá para dejar el modelo 3D, que teníamos en el modelador, en la misma posición en la que estaba. En la línea 6 se utiliza el método **CopyAbsoluteBoneTransformsTo** para copiar las transformaciones de cada uno de los huesos en la matriz que sea acaba de crear.

Ahora hay que inicializar el efecto y sus propiedades para poder visualizar el modelo 3D; en la línea 8 se hace una conversión explicita del tipo de dato, esto es para inicializar el efecto básico que nos proporciona XNA. Sin embargo, no inicializa todas las propiedades que quisiéramos, como son las luces y las matrices. Véase que de antemano tomamos el primer **ModelMesh** y su primer **ModelEffect** de éste; esto es porque sabemos que el archivo contiene un sólo modelo, así que sólo tiene un **ModelMesh** y un sólo **ModelEffect**. No es necesario que sepamos cuántos meshes contiene el archivo; porque al fin de cuentas es un arreglo, y si conocen el uso de arreglos en C#, verán que el propio arreglo conoce cuántos elementos tiene, y es porque es un objeto.

Así que enseguida le damos valores a las matrices **World**, **Projection** y **View**, líneas 9, 10 y 13 respectivamente. En la matriz **World** le asignamos la matriz de transformaciones, pero utilizando como indíce el del **ParentBone**. El **ParentBone** es una propiedad del **ModelMesh** que obtiene el **ModelBone** padre de ese mesh. El **ModelBone** de un mesh contiene una matriz de transformación que indica cómo es colocado con referencia al mesh padre del modelo. Trate de colocar el valor de cero y verá que no es lo igual, a menos que el modelo estuviera en el centro de referencia del sistema coordenado y orientado a los ejes.

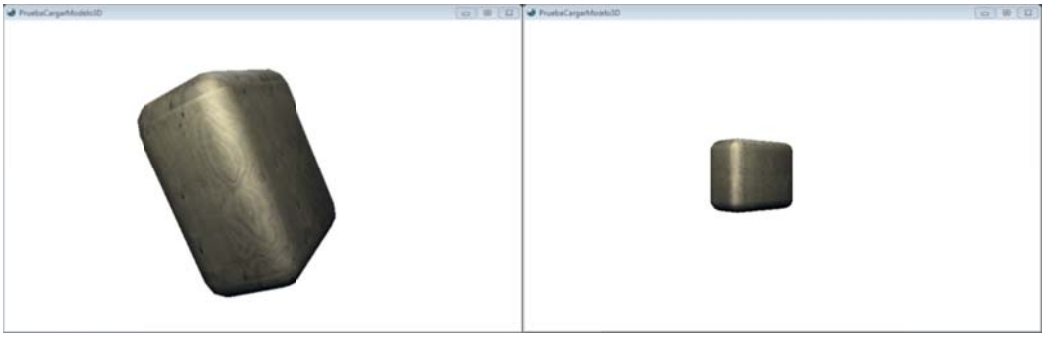

Ilustración 6-2 Posición Correcta e incorrecta

En la Ilustración 6-2 se muestra el resultado de haber respetado el índice proporcionado por **ParentBone**, y cuando no se toma en cuenta.

Para poder ver las características de los materiales del modelo, se habilita la luz que por default tiene el efecto, véase línea 15.

De antemano se conoce el número de meshes que contiene el archivo, así que se declara un **ModelMesh**, línea 17. Y también se conoce cuántos **MeshPart** tiene el **ModelMesh**, por lo que se declara en

la línea 18. Esto en sí, es malo hacerlo en esta parte del código, pero se hizo para poder explicarlo de la mejor manera.

Ahora hay que preparar el dispositivo gráfico con los datos a dibujar, el primero es el contenido del búfer de vértices, con el método **SetSource**, líneas 21 – 22. Lo interesante aquí es de dónde se obtiene la información, del **ModelMesh** y del **ModelMeshPart**; ambos pertenecen a la clase **Model**. Si recuerda el primer parámetro del método **SetSource**, es el mismo búfer de vértices, el cual es parte del **ModelMesh**. Después necesita el byte a partir del cual serán copiados los datos, lo cual es proporcionado por el **ModelMeshPart**. Y por último esté necesita el tamaño en bytes de los elementos del búfer de vértices, el cual también es proporcionado por el **ModelMeshPart**.

Después de lo anterior, ahora se necesita declarar el tipo de vértices que ocupará el dispositivo, para eso regresamos a usar el método **VertexDeclaration** y la propiedad con el mismo nombre del **ModelMeshPart**, línea 23.

Ya como último preparativo del dispositivo, se asigna el búfer de indices del **ModelMesh** al **Indexbuffer** de GraphicsDevice, línea 24.

En este ejemplo se conoce el número técnicas de rendereo, así que el índice en los arreglos **Passes** será cero, líneas 27 y 31.

Otra vez, se recurre al método **DrawIndexPrimitives** del dispositivo gráfico para mandar a dibujar la geometría, y obtenemos los datos de las propiedades **BaseVertex**, **NumVertices**, **StartIndex** y **PrimitiveCount** de la **ModelMeshPart**. No hace falta decir para qué son estas propiedades, así que si hay alguna duda revísese el capítulo 001.

Eso es todo para poder ver el elefante en el viewport, Ilustración 6-3; si hay algún error de compilación corrija y vuelva a intentarlo otra vez.

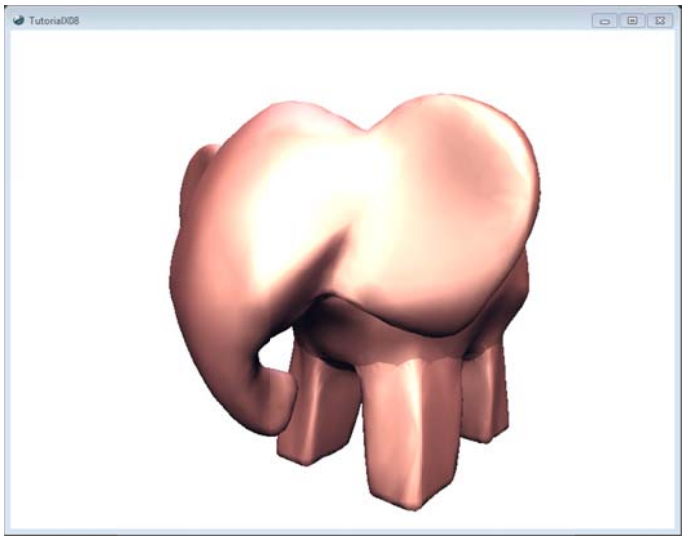

Ilustración 6-3 Un mesh

Como un agregado cambie la línea de la propiedad World del efecto por el siguiente enlistado.

```
tiempo = (Single)gameTime.TotalGameTime.TotalSeconds;
basicEffect.World = transformaciones[modelo.Meshes[0].ParentBone.Index] *
Matrix.CreateRotationX(tiempo) * Matrix.CreateRotationY(tiempo) *
Matrix.CreateRotationZ(tiempo);
```

En donde tiempo es una variable de instancia de la clase **Game1** de tipo **float**, y que se inicializa dentro del método **Draw** de la clase **Game1**, y antes de la asignación de datos a la propiedad **World**. A éste se le

asigna el valor total en segundos, del tiempo transcurrido desde el inicio de la aplicación. Y como se encuentra dentro del método **Draw** se actualizará cada vez que se mande a rasterizar.

En la asignación de valores a la propiedad **World** del efecto, se obtienen primero las matrices de rotación alrededor de los ejes coordenados **x**, **y** y **z**. Estas matrices se obtienen con ayuda de los métodos estáticos **CreateRotationX**, **CreateRotationY** y **CreateRotationZ** de la clase **Matrix**. La variable **tiempo**, de tipo **float,** representa el ángulo expresado en radianes.

Inicie una vez más la aplicación y verá rotar el modelo extraído de un archivo ***.fbx** o ***.x**.

6.4 Ejemplo 02

En un archivo se pueden encontrar varios y diferentes modelos, cada uno con sus respectivos materiales, texturas y/o efectos. Así que este ejemplo muestra cómo dibujar dichos modelos contenidos en un archivo.

Cree un nuevo proyecto para XNA Game Studio 3.1 y seleccione la plantilla **Windows Game (3.1)**. En **Content**, en **Explorador de soluciones**, agregue dos nuevas carpetas con los nombres **Modelos** y **Texturas** respectivamente.

Uno de los problemas al cargar archivos que contenían texturas, ya sea en formato ***.x** o ***.fbx**, es el nombre relativo del archivo de la textura. Este nombre relativo se crea cuando se exporta el archivo desde el modelador 3D. Este nombre contiene la ruta de donde se leía la textura o simplemente el nombre de la textura. Así que en el momento en que se trata de generar la solución, oprimiendo **F6**, aparece el error de **Missing asset**, esto es porque de forma automática XNA hace referencia al nombre del archivo de la textura que viene dentro de las propiedades del archivo ***.x** o ***.fbx**, y al no encontrar dicho archivo no permite continuar con la ejecución, aún si no se hace mención en los archivos ***.cs** de la solución de XNA.

Una solución es dejar el archivo ***.x** o ***.fbx** en el mismo lugar en donde se encuentran las texturas, pero esto dejaría todo revuelto, así que otra de las soluciones es modificar el nombre del archivo de textura que se hace mención en el arhivo ***.x** o ***.fbx**. En los achivos ***.x** lo encuentran con el metadato **TextureFilename** y en ***.fbx** como **RelativeFilename**. Pero esto también sería algo mal hecho, así que otra de las soluciones es crear carpetas que contengan por separado a las texturas, los shaders, los modelos, etcétera, en la solución que se está creando. Y a partir de ellas es toman los recursos necesarios para modelar y no estar modificando “a mano” dichos nombres de archivos de textura.

Los nombres de las carpetas son necesarios para este ejemplo, pues el archivo que contiene la geometría, hace referencia a los archivos de textura:

"..\..\Content\Texturas\Piso.bmp"

"..\..\Content\Texturas\Tablero.bmp"

"..\..\Content\Texturas\Checker.bmp"

Ahora agregue el archivo, en este caso se utilizó el archivo con nombre TeterasF03.FBX, en la carpeta **Modelos**, y los archivos de textura, en este caso fueron: Piso.bmp, Tablero.bmp y Checker.bmp, en la carpeta **Texturas**, y verá algo similar a la Ilustración 6-4.

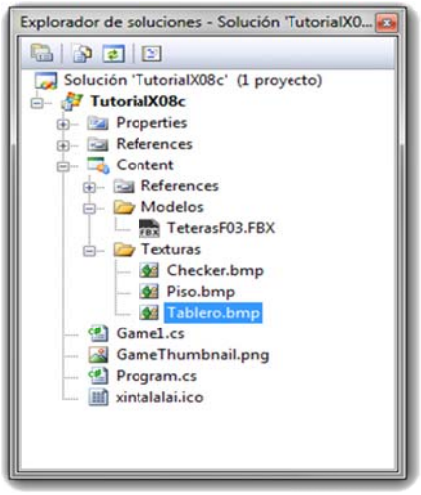

Ilustración 6-4 Agregando texturas en el Content

Todos los modelos los haremos rotar alrededor de los ejes locales de cada uno de ellos, por medio del teclado. Así que comenzaremos por agregar las variables de instancia en el archivo **Game1.cs** que creó Visual Studio.

```
Model modelo;
Single rotacionX, rotacionY, rotacionZ;
const Single angulo = 2.0F * (Single)Math.PI / 180.0F;
Single escala = 1.0F;
Matrix[] transformaciones;
```

Las variables **rotacionX**, **rotacionY** y **rotacionZ** son los ángulos a rotar alrededor de los ejes **x**, **y** y **z** respectivamente. Recordando que los ángulos se deben pasar en radianes, la constante angulo será el ángulo de dos grados por el que se incrementará las variables **rotacionX**, **rotacionY** y **rotacionZ** cada vez que oprima una de las siguientes teclas.

Tabla 6-1

Tecla	Acción
Left	Incrementa la variable rotacionY.
Right	Decrementa la variable rotacionY.
Up	Incrementa la variable rotacionX.
Down	Decrementa la variable rotacionX.
PageDown	Incrementa la variable rotacionZ.
PageUp	Decrementa la variable rotacionZ.
Z	Incrementa la variable escala.
C	Decrementa la varibale escala.

La variable **escala** es el valor en punto flotante que se usará para aumentar o disminuir el tamaño de todas las figuras geométricas. El arreglo **Matrix** guardará todos los **BoneTransforms** del modelo.

En el constructor de la clase **Game1** se cambian las ya dichosas propiedades siguientes:

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);

    Content.RootDirectory = "Content";
    this.Window.Title = "TutorialX08c";
    this.Window.AllowUserResizing = true;
    this.IsMouseVisible = true;
}
```

En el método **LoadContent** de la clase **Game1** se comienza por cargar el contenido del archivo del modelo tridimensional, al objeto **modelo**. Inmediatamente se copian las matrices de transformación de cada figura geométrica del objeto **modelo**

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    modelo = Content.Load<Model>(Content.RootDirectory + @"\Modelos\TeterasF03");

    transformaciones = new Matrix[modelo.Bones.Count];
    modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
}
```

Hasta este punto no cambia nada al ejemplo anterior, pero nótese que hasta el momento no existe ninguna instancia de la clase **BasicEffect**.

En el método **Update**, de la clase **Game1**, hay que agregar todas las acciones descritas en la tabla 6-1.

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (Keyboard.GetState().IsKeyDown(Keys.Left))
        rotacionY += angulo;
    if (Keyboard.GetState().IsKeyDown(Keys.Right))
        rotacionY -= angulo;
    if (Keyboard.GetState().IsKeyDown(Keys.Up))
        rotacionX += angulo;
    if (Keyboard.GetState().IsKeyDown(Keys.Down))
        rotacionX -= angulo;
    if(Keyboard.GetState().IsKeyDown(Keys.PageDown))
        rotacionZ +=angulo;
    if(Keyboard.GetState().IsKeyDown(Keys.PageUp))
        rotacionZ -=angulo;
    if (Keyboard.GetState().IsKeyDown(Keys.Z))
        escala += 0.1F;
    if (Keyboard.GetState().IsKeyDown(Keys.C))
    {
        if((escala-=0.1F)<1.0F)
        escala = 1.0F;
    }
    base.Update(gameTime);
}
```

Cada variable se incrementa o decrementa con un valor constante, cada vez que se registra una tecla oprimida. En el caso de la variable escala, hay que verificar que no tenga un menor a uno. En realidad puede tener un valor menor a uno, lo que haría es encoger los modelos geométricos; sin embargo, debe ser mayor.

Para el método **Draw** se crean dos instrucciones **foreach**, una de ellas está anidada; la primera de ellas va a mapear los **ModelMesh** que contiene el arreglo **Meshes** del objeto **modelo**. El segundo bucle va a mapear los efectos de cada mesh que contiene el arreglo **Effects** del **ModelMesh** obtenido del **foreach** anterior.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    foreach (ModelMesh mesh in modelo.Meshes)
    {
        foreach (BasicEffect efecto in mesh.Effects)
        {
            efecto.World = Matrix.CreateRotationX(rotacionX) *
                Matrix.CreateRotationY(rotacionY)
                    * Matrix.CreateRotationZ(rotacionZ) * Matrix.CreateScale(escala)
                    * transformaciones[mesh.ParentBone.Index];
            efecto.View = Matrix.CreateLookAt(new Vector3(-50.0F, 70.0F, 75.0F),
                    new Vector3(0.0F, 0.0F, 0.0F), Vector3.Up);
            efecto.Projection = Matrix.CreatePerspectiveFieldOfView(1,
                    GraphicsDevice.Viewport.AspectRatio, 1.0F, 10000.0F);
            efecto.EnableDefaultLighting();
            efecto.PreferPerPixelLighting = true;
        }// fin de foreach
        mesh.Draw();
    }// fin de foreach

    base.Draw(gameTime);
}
```

Véase que en el **foreach** anidado se utiliza la clase **BasicEffect** para cargar en el **GraphicsDevice** todas las propiedades, como es el material, la iluminación, texturas, etcétera.

En el cuerpo del **foreach** anidado se le pasan los valores a las matrices **World**, **View** y **Projection**. Aquí lo importante es la matriz **World** de **BasicEffect**, pues es la que hará que las figuras roten o escalen. Tomé en cuenta el orden en que están dispuestas las operaciones, recuerde que es una multiplicación de matrices, y por ende su propiedad no es conmutativa. Así que como tarea vea que lo que pasa al alterar el orden de las matrices de transformación.

La propiedad **PreferPerPixelLighting** de **BasicEffect**, indica que se habilita la iluminación por píxel, siempre y cuando su Unidad de Procesamiento Gráfico o GPU (Graphics Processing Unit) soporte **Pixel Shader Model 2.0**. Si no es así, comente la línea o ponga el valor a false.

Terminando el **foreach** anidado, se manda a llamar al método **Draw** de la clase **ModelMesh**, ésta manda a dibujar todos los **ModelMeshPart** del mesh, usando el efecto actual.

Corra el ejemplo con **F5**, y vera algo similar a la Ilustración 6-5.

Ilustración 6-5 Multiples meshes

Oprima las teclas indicadas para las acciones mencionadas en la Tabla 6-1, y verá cómo se mueven todas la teteras al unísono. Note que todas las teteras tienen diferentes propiedades de materiales, textura y sus combinaciones. Sin embargo, algunas de ellas son trasparentes, como lo muestra la Ilustración 6-6.

Ilustración 6-6 Render tomado desde un modelador 3D.

6.4.1 Blending

En realidad el efecto de las teteras menos opacas o más transparentes es una unión de píxeles que da la sensación de material translúcido. Este proceso se le llama blend y la fórmula que se utiliza en el ejemplo se le llama ecuación de blend.

$$PixelSalisa = (PixelFuente \times FactorBlendFuente) + (PixelDestino \times FactorBlendDestino)$$

PixelSalida: es valor de píxel, resultado de la suma de las multiplicaciones de fuente y destino.

PixelFuente: es el color del píxel que se pretende sea el translúcido.

FactorBlendFuente: es un factor por el que se multiplica a PixelFuente.

PixelDestino: es el color del píxel que se desea ver a través del PixelFuente.

FactorBlendDestino: factor por el que se multiplica a PixelDestino.

Los factores indican cuánto y cómo contribuye cada píxel al cálculo final del color.

Los factores blend se componen por Red, Green, Blue y Alpha (r, g, b, a). Y cada uno de ellos se multiplica por su contraparte en el píxel destino y/o fuente. La siguiente tabla muestra los factores de la numeración **Blend** que proporciona XNA.[20]

Tabla 6-2

Nombre del miembro	Descripción
Zero	Cada componente del píxel es multiplicado por (0, 0, 0, 0), por lo que elimina.
One	Cada componente del píxel es multiplicado por (1, 1, 1, 1), por lo que no se ve afectado.
SourceColor	Cada componente del píxel es multiplicado por el color en el píxel fuente. Éste puede ser representado como (R_s, G_s, B_s, A_s).
InverseSourceColor	Cada componente del píxel es multiplicado por la resta de 1 menos el valor del componente del píxel fuente. Éste puede ser representado como $(1 - R_s, 1 - G_s, 1 - B_s, 1 - A_s)$.
SourceAlpha	Cada componente del píxel es multiplicado por el valor alfa del píxel fuente. Éste puede ser representado como (A_s, A_s, A_s, A_s).
InverseSourceAlpha	Cada componente del píxel es multiplicado por el valor obtenido de la resta de 1 menos el valor que contiene el píxel fuente en el campo del canal alfa. Éste puede ser representado como $(1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)$
DestinationAlpha	Cada componente del píxel es multiplicado por el valor alfa del píxel destino. Éste puede ser representado como (A_d, A_d, A_d, A_d).
InverseDestinationAlpha	Cada componente del píxel es multiplicado por el valor obtenido de la resta de 1 menos el valor que contiene el píxel destino en el campo del canal alfa. Éste puede ser representado como $(1 - A_d, 1 - A_d, 1 - A_d, 1 - A_d)$.
SourceAlphaSaturation	Cada componente del píxel fuente o destino se multiplica por el valor más pequeño entre alfa del fuente o uno menos alfa del destino. El componente alfa no sufre cambios. Éste puede ser representado como $(f, f, f, 1)$, donde $f = min(A_s, 1 - A_d)$.
BothInverseSourceAlpha	Aplica sólo a la plataforma Win32. Cada componente del píxel fuente es multiplicado por el resultado de la resta 1 menos el valor que contiene el píxel fuente en el campo del canal alfa, y cada componente del píxel destino es multiplicado por el valor alfa del píxel fuente. Éstos pueden ser representados como $(1 - A_s, 1 - A_s, 1 - A_s, 1 - A_s)$, y para el blend destino (A_s, A_s, A_s, A_s); el blend destino se inválida. De este modo sólo es soportado por el SourceBlend de RenderState.

[20] Para mayor información visite la siguiente página:
http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.blend.aspx

BlendFactor	Cada componente del píxel es multiplicado por la propiedad RenderState.BlendFactor[21].
InverseBlendFactor	Cada componente del píxel es multiplicado por el valor obtenido de la resta de 1 menos la propiedad BlendFactor. Éste método es soportado solo sí SupportsBlendFactor es verdadero en las propiedades SourceBlendCapablities o DestinationBlendCapabilities.

Estos factores se pueden ocupar tanto para el blend fuente como en el destino, a menos que se diga lo contrario.

En este caso el canal alfa se involucra en las operaciones, y es quien nos indica la opacidad, por lo que un valor grande es un material más opaco. Los valores que puede tomar el canal alfa es entre 0 y 1. Este valor de alfa puede provenir del material de la figura geométrica o de la textura, pero esto ya lo tiene registrado los elementos del arreglo **Effects** del mesh, por lo que no hay necesidad de agregarlo directamente en el código. Para habilitar el efecto de transparencia, se le debe indicar al dispositivo gráfico el estado de procesamiento, en éste caso es el blending.

Tomando el ejemplo anterior, escriba las siguientes líneas de código, entre la llave de apertura del foreach anidado y la asignación de la matriz de mundo de efecto.

Código 6-2

```
1.      if (efecto.Alpha != 1.0F)
2.      {
3.          efecto.GraphicsDevice.RenderState.AlphaBlendEnable = true;
4.          efecto.GraphicsDevice.RenderState.SourceBlend = Blend.One;
5.          efecto.GraphicsDevice.RenderState.DestinationBlend = Blend.One;
6.          efecto.GraphicsDevice.RenderState.BlendFunction = BlendFunction.Add;
7.      }
8.      else
9.          efecto.GraphicsDevice.RenderState.AlphaBlendEnable = false;
```

No todos los mesh del archivo contienen el mismo efecto de transparencia, así que hay que tener cuidado de habilitar y deshabilitar el blending, pues si no se hace, el efecto afectaría a todos los demás meshes siguientes del primero que lo activo. Como se había indicado anteriormente, el canal alfa es la clave del blending, pues nos indica que tan opaco es el material o la textura. En la línea 1 del enlistado anterior, verá que se verifica que la propiedad **Alpha** del objeto efecto sea diferente de 1.0F para activar el efecto de blending.

Para activar el blend se establece en verdadero a la propiedad **AlphaBlendEnable** del **RenderState**. Para las líneas 4 y 5 se establece el factor de blend en **One** de la enumeración **Blend**, a **SourceBlend** y **DestinationBlend**. En la línea 6 se aplica la combinación de colores por medio de la ecuación de blend, que nos ofrece la enumeración **BlendFunction**, como **Add**.

En el cuerpo de else se deshabilita el efecto con sólo poner en falso la propiedad **AlphaBlendEnable** del **RenderState**.

Corra el ejemplo y verá algo similar a la Ilustración 6-7. Cada tetera tiene asignado diferentes materiales y texturas, además se habilito el blend que contiene contiene cada una de ellas.

[21] BlendFactor es una propiedad que obtiene o establece el color usado por un factor constante-blend durante el blending. El valor por default es Color.White. Para mayor información visite la página siguiente:
http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.renderstate.blendfactor.aspx

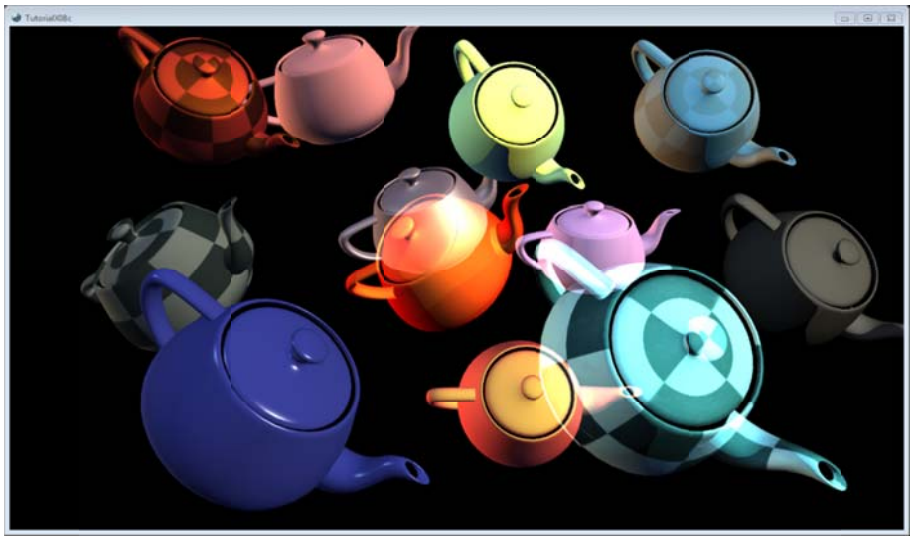

Ilustración 6-7 Blending

Mueva con las teclas vistas en la tabla 6.-1 las teteras, para que vea el resultado final y esperado del archivo.

Juegue con los valores de los factores de blend y los diferentes valores de la propiedad **BlendFunction**[22], para que corrobore los distintos efectos que logra con sus combinaciones.

[22] Para mayor información visite la siguiente página:
http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.blendfunction.aspx

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.

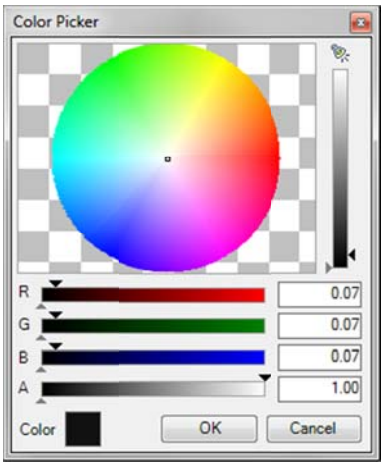

Ilustración 7-1 Color Picker

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**.

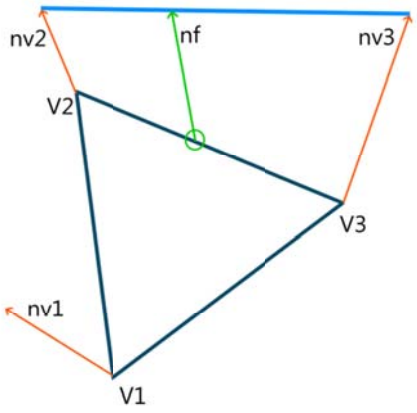

Ilustración 7-2 Interpolación de vertex shader a pixel shader

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

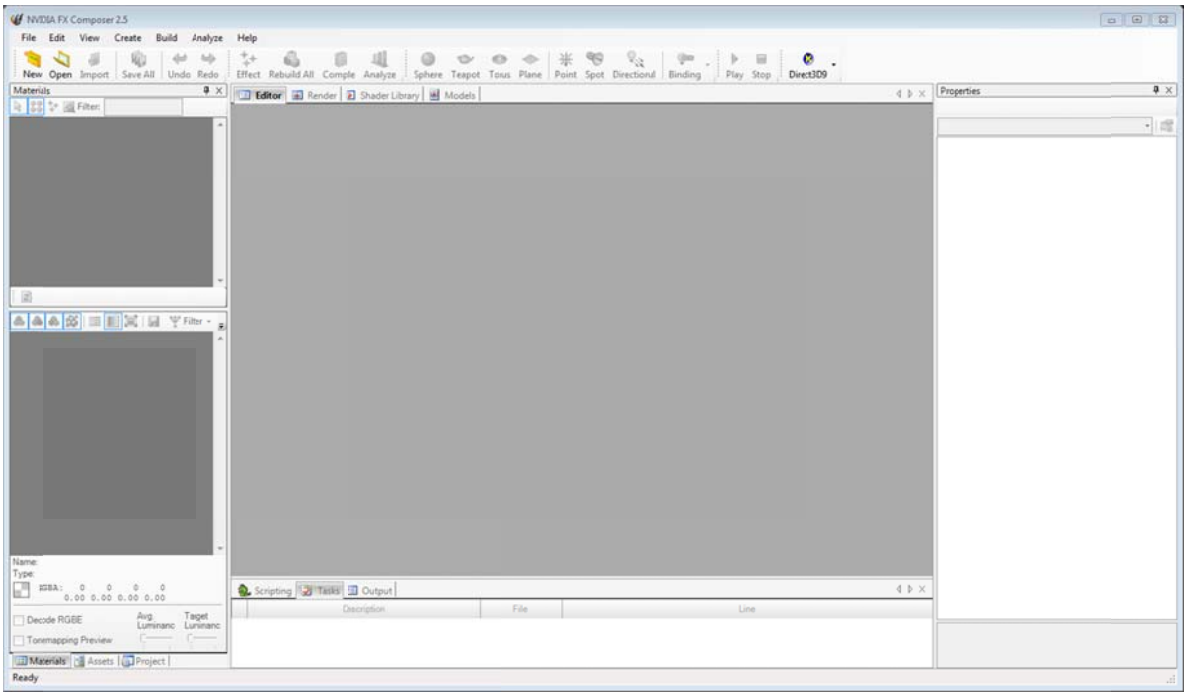

Ilustración 7-3 GUI de Fx Composer 2.5

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.

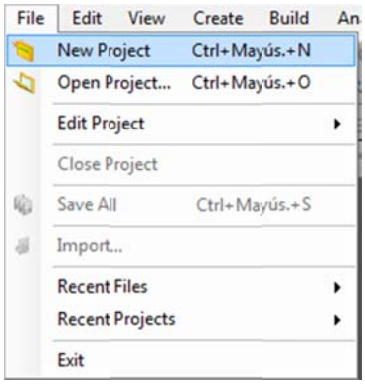

Ilustración 7-4 Nuevo proyecto

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.

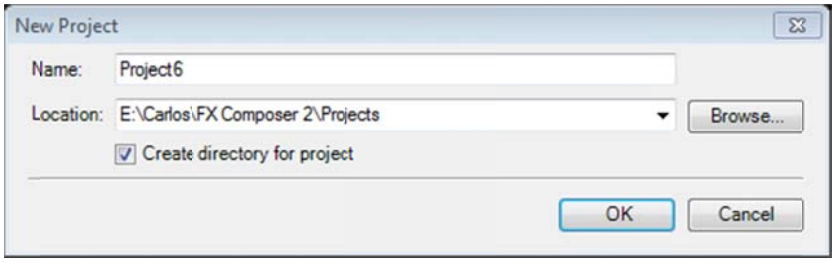

Ilustración 7-5 Ventana de diálogo

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.

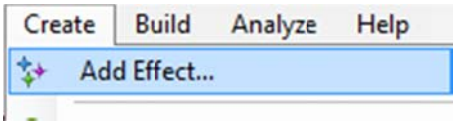

Ilustración 7-6 Añadir nuevo efecto

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.

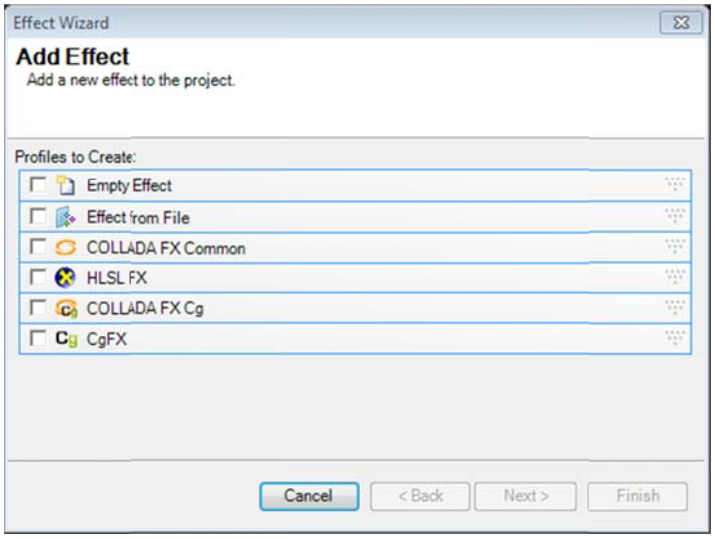

Ilustración 7-7 Ventana de diálogo para añadir nuevo efecto

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.

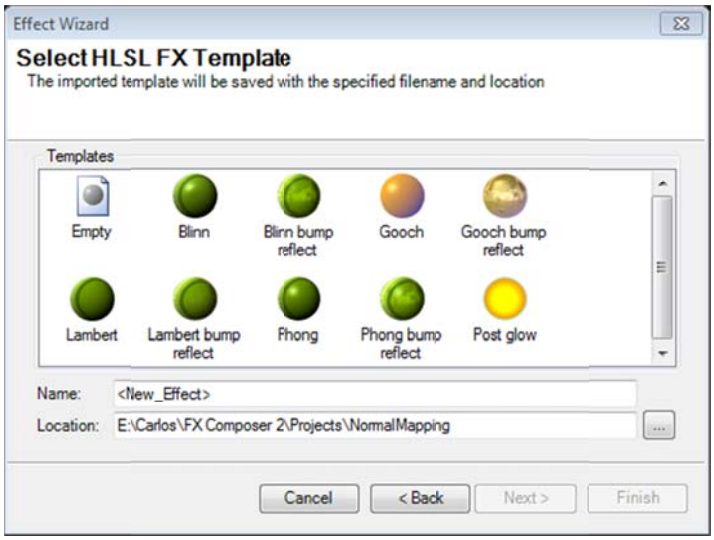

Ilustración 7-8 Plantillas para HLSL

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.

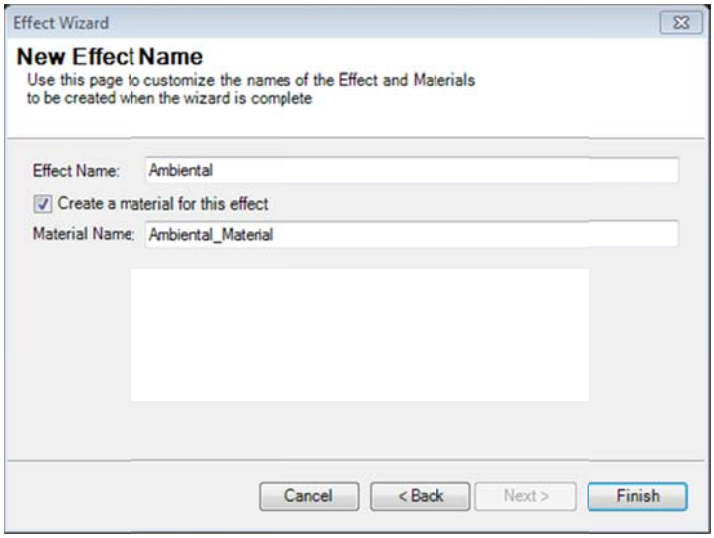

Ilustración 7-9 Ventana de diálogo para el nombre del efecto

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.

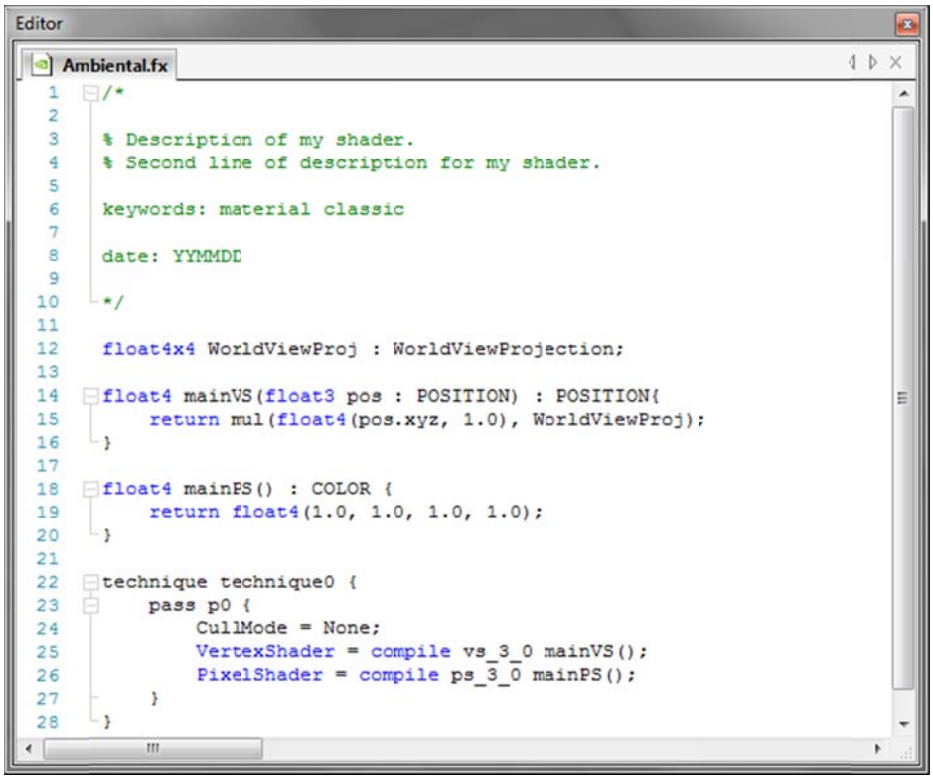

Ilustración 7-10 Editor de texto de FX Composer

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.

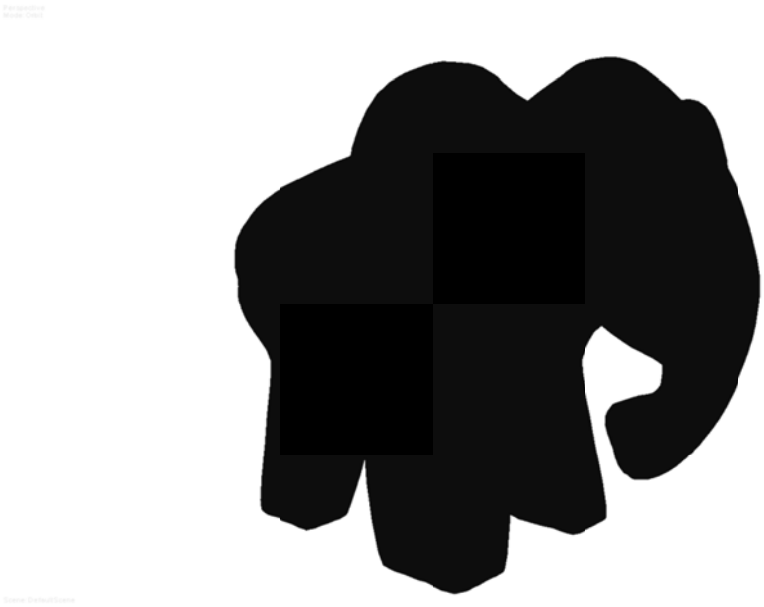

Ilustración 7-11 Elefante con iluminación ambiental

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.

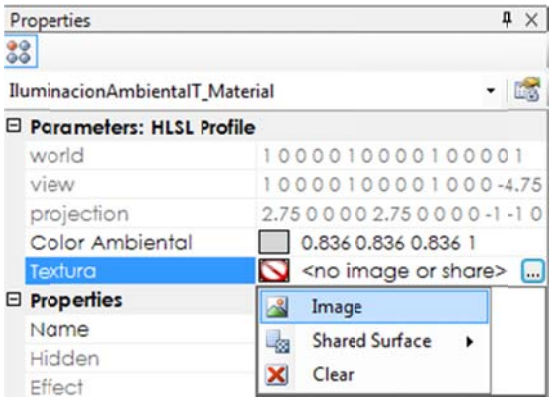

Ilustración 7-12 Campo Textura

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.

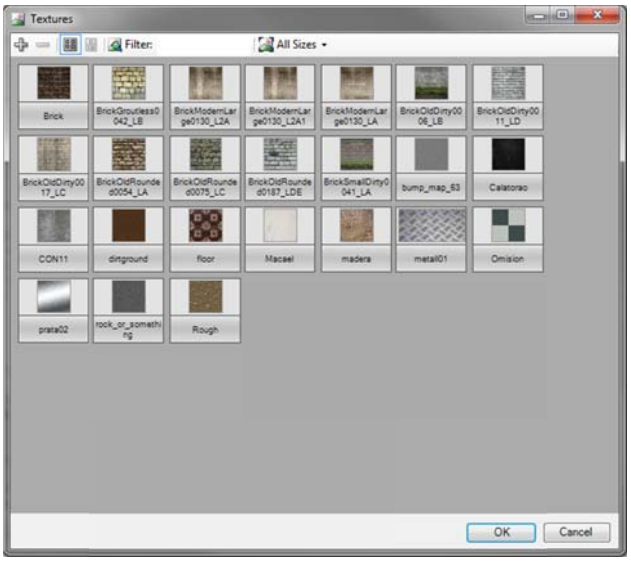

Ilustración 7-13 Texturas

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.

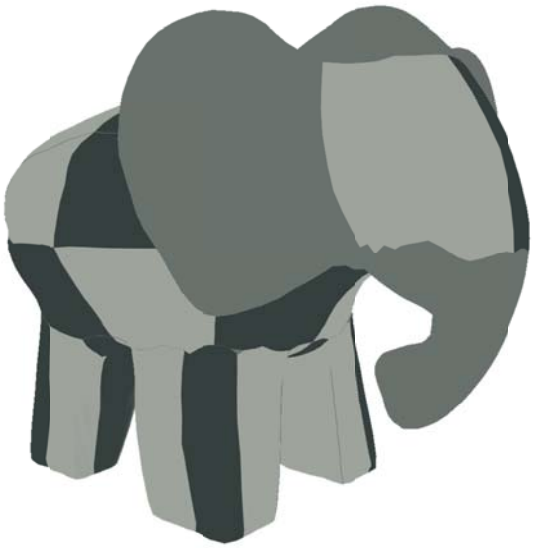

Ilustración 7-14 Ambiental con textura

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.

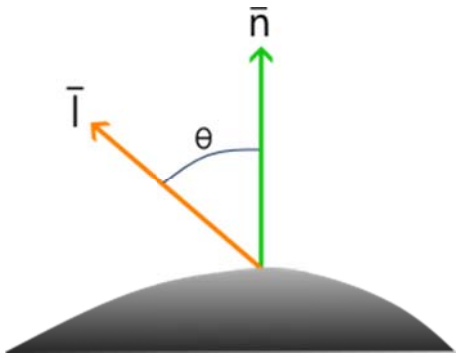

Ilustración 7-15 Ángulo formado entre el haz de luz y la normal de la superficie

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.

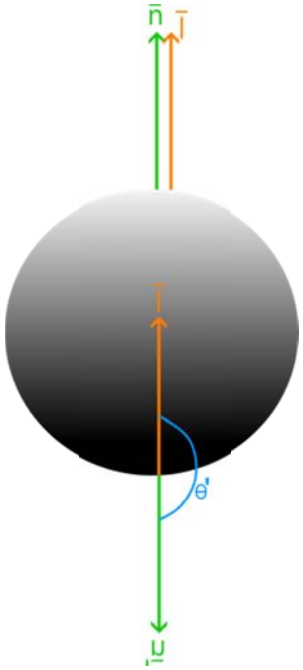

Ilustración 7-16 Autoocluyente

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.

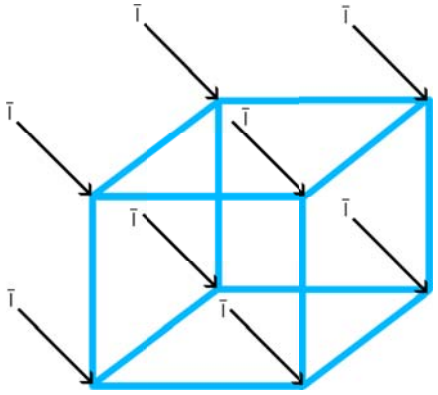

Ilustración 7-17 Iluminación direccional

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.

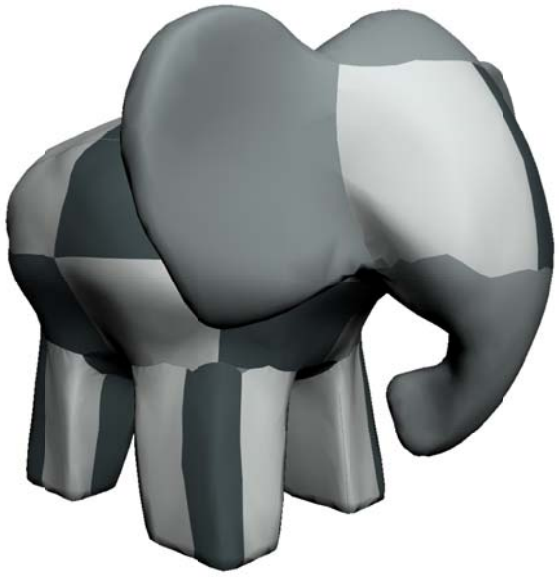

Ilustración 7-18 Difuso con textura

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.

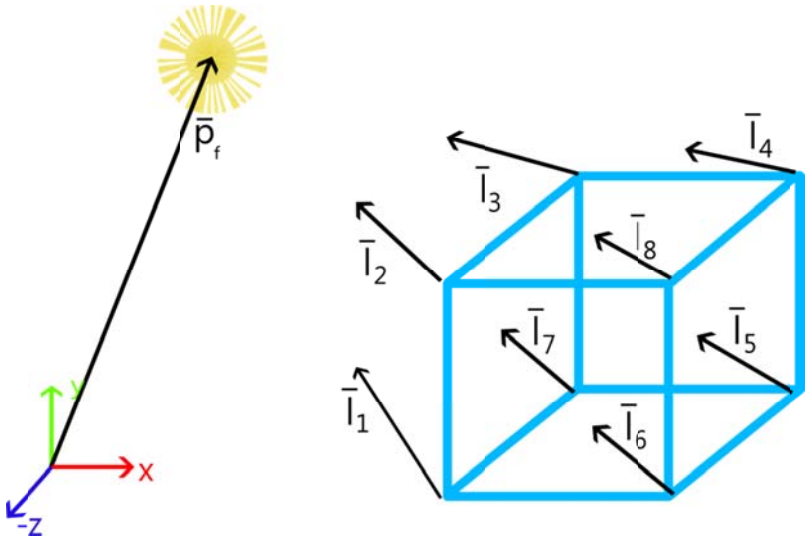

Ilustración 7-20 Fuente puntual

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.

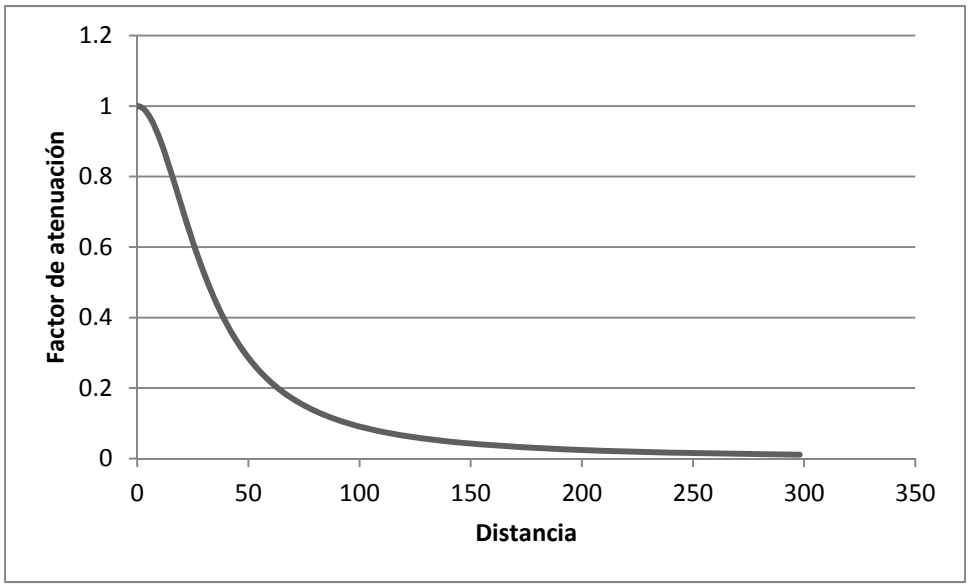

Ilustración 7-21 Factor de atenuación de iluminación. c1= 1, c2 = 0, c3 = 0.001

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
```

```
112.            // será de cero.
113.            if(distancia <= rango)
114.                atenuacion = min(1 / (c1 + (c2 * distancia) +
115.                (c3 * pow(distancia, 2.0F))), 1.0F);
116.            else
117.                atenuacion = 0;
118.            // color final
119.            color *= atenuacion;
120.        }// fin del if
121.        // color final
122.        color += colorMaterialAmbiental;
123.
124.        // texturizado
125.        if(texturizado)
126.        {
127.            float4 texturaColor = tex2D(modeloTexturaMuestra,
128.                entrada.CoordenadaTextura);
129.            color *= texturaColor;
130.        }
131.
132.        // color final
133.        color.a = 1.0F;
134.        return color;
135.    }// fin pixel shader MainPs
136.
137.    technique Texturizado
138.    {
139.        pass P0
140.        {
141.            VertexShader = compile vs_3_0 MainVS();
142.            PixelShader = compile ps_3_0 MainPS(true);
143.        }
144.    }
145.
146.    technique Material
147.    {
148.        pass P0
149.        {
150.            VertexShader = compile vs_3_0 MainVS();
151.            PixelShader = compile ps_3_0 MainPS(false);
152.        }
153.    }
```

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.

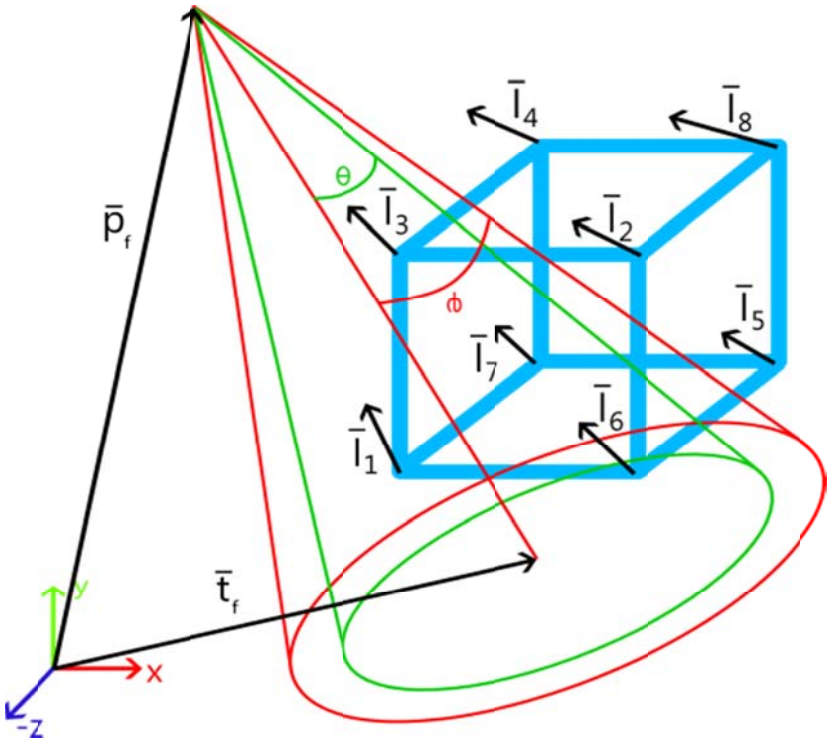

Ilustración 7-23 Fuente de iluminación spot

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

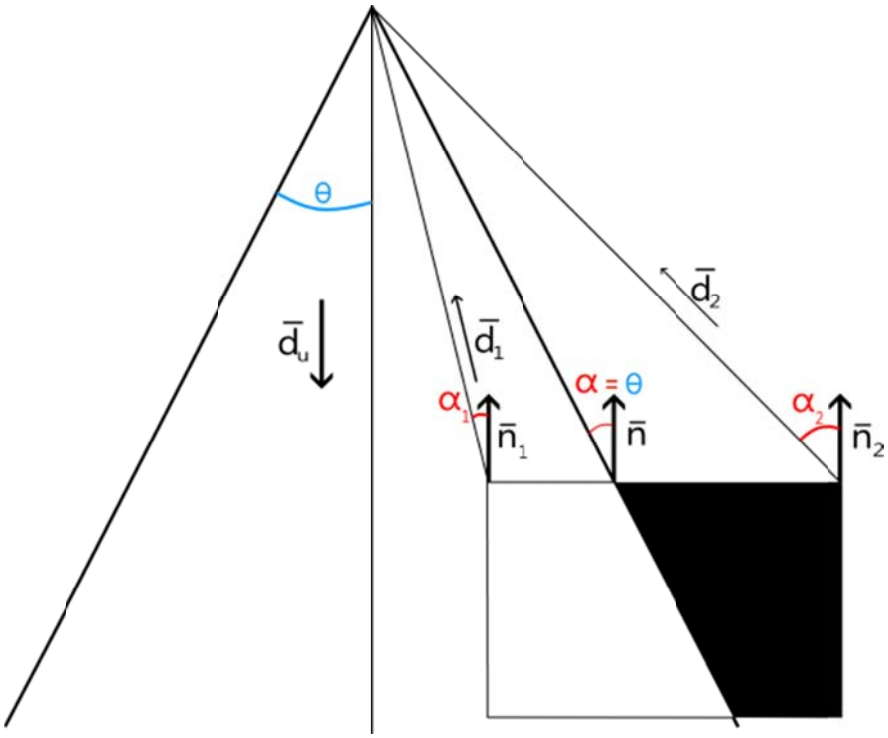

Ilustración 7-24 Fuente spot

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.

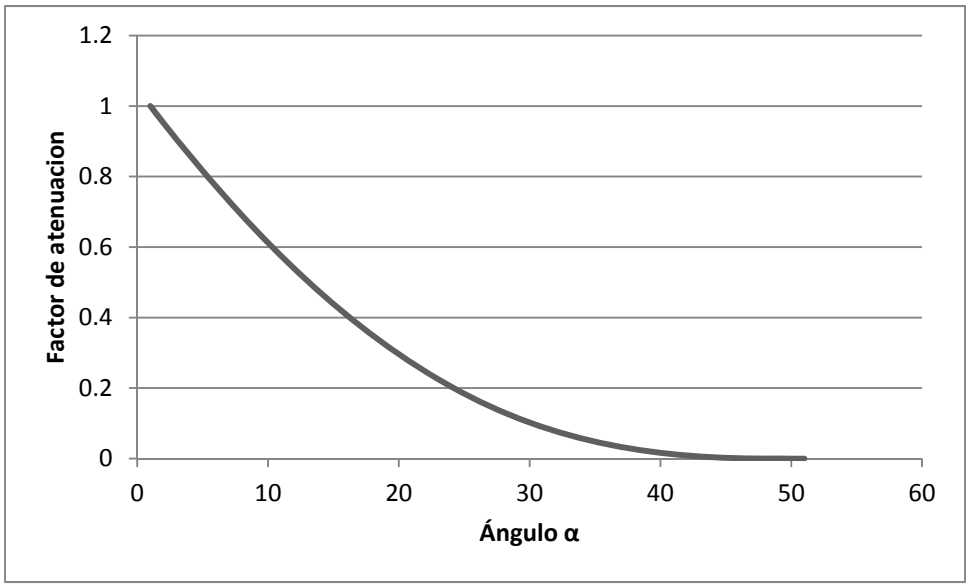

Ilustración 7-25 Factor de atenuación angular. falloff = 3, θ = 10°, ϕ = 15°

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.

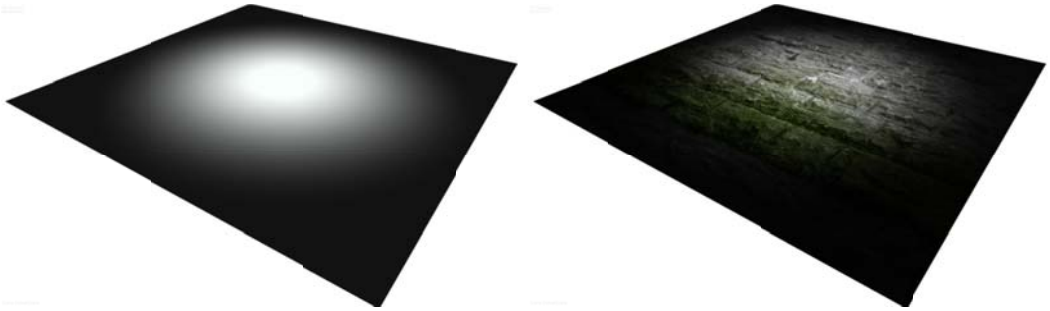

Ilustración 7-26 Fuente Spot

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.

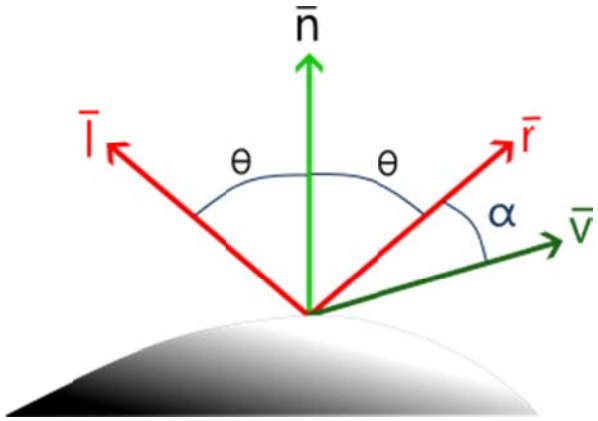

Ilustración 7-27 Reflexión especular

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 $\bar{n}$.

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

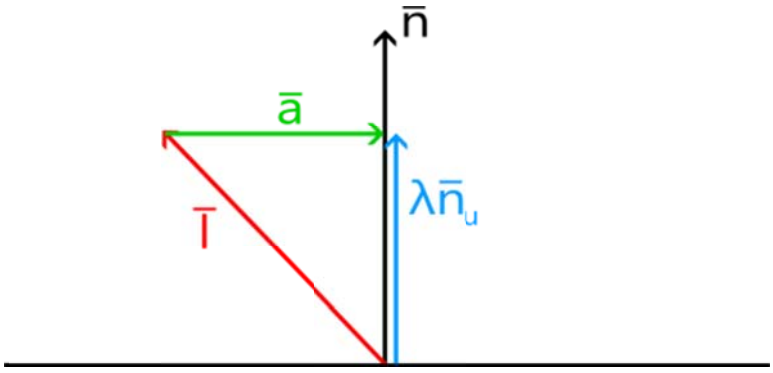

Ilustración 7-28 Vector auxiliar

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.

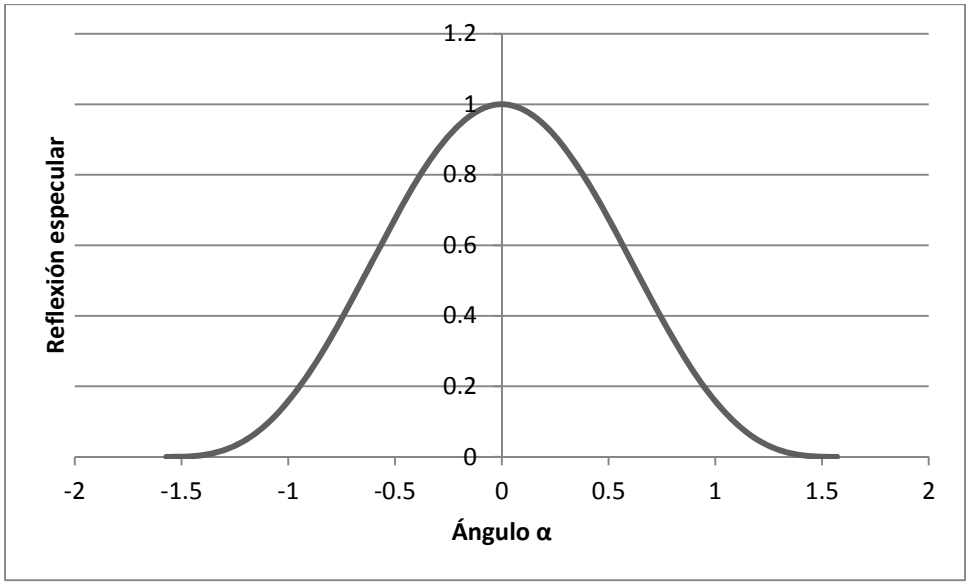

Ilustración 7-29 Reflexión especular n = 3

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.

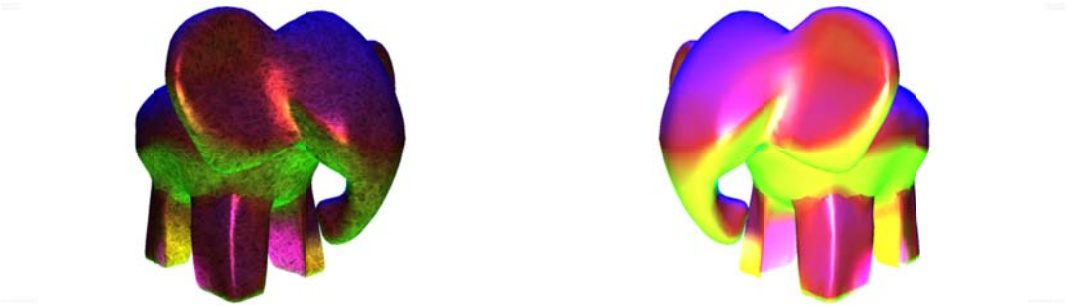

Ilustración 7-33 Multi pass rendering

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.

8 Cómo agregar un efecto en XNA

En este capítulo se muestra cómo cargar un efecto proveniente de un archivo **fx**, en específico serán algunos de los ejemplos vistos anteriormente sobre el lenguaje HLSL, y se espera que el lector pueda cargar cualquier otro shader.

El **ContentManager** de XNA realiza la mayor parte del trabajo, sin embargo, la manera en cómo se le pasarán los datos de XNA a HLSL dependerá del programador, pues si en el shader no se restringieron sus variables **extern uniform**, en C# se pueden limitar; además se debe tener cuidado en la asignación de datos.

La pregunta es ¿cómo reconocer los tipos de datos correspondientes del **.Net Framework Class Library**(CL) y del **Framework de XNA** a HLSL? En sí, la mayoría de los datos comunes o no compuestos como los enteros, flotantes y boleanos tienen nombres similares entre CL y HLSL. La manera de reconocer los tipos de datos compuestos o estructuras como **float4x4**, **float3** es haciendo una comparación "manual" entre sus campos y los que constituyen aquellos que pueden parecerse en el Framework de XNA, por lo tanto hay que conocer los tipos de datos de ambos, lo cual se deja al lector como estudio, y que en ocasiones será intuitivo esta asignación de tipos de datos.

8.1 Efecto ambiental

En este aparatado se muestran dos formas de cargar un efecto proveniente de un archivo **fx**. El primero muestra cómo cambiar las instancias de la clase **Effect** del objeto **Model** por la instancia **Effect** asociada al efecto ambiental; esto permite mandar a llamar el método **Draw** de la clase **ModelMesh**, permitiendo una programación más sencilla, pero a costa de cambiar todas las propiedades de material de la instancia **Model**.

La segunda forma no cambia las propiedades del material asociadas a la instancia **Model**. Para poder dibujar la geometría con el efecto proveniente de un archivo **fx**, se utiliza el método **DrawIndexedPrimitives** de la clase **GraphicsDevice**.

8.1.1 Clonación de efecto

En este primer ejemplo se utiliza el método **Effect.Clone** para copiar los valores de un objeto existente a otro, por lo que este método arroja una excepción cuando se tratan de copiar los valores de un objeto nulo, así que no se recomienda el uso del signo de asignación, =.

Aclarado el uso del método **Clone** y el signo de asignación, creé un nuevo proyecto de XNA en Visual Studio. En la clase **Game1** escriba las siguientes variables de instancia:

Código 8-1

```
1.      private Model modelo;
2.      private Matrix[] transformaciones;
3.      private Texture2D textura;
4.      Single tiempo;
```

La declaración de la variable modelo es para el modelo tridimensional, línea 1, Código 8-1; que en este caso fue **ElefanteC.fbx** y que puede ser cualquiera que tengan a la mano. En seguida se declara un arreglo de matrices que almacenaran las transformaciones del modelo, línea 2.

El efecto **Ambiental.fx** tiene dos tipos de técnicas, **Material** y **Texturizado**, para este último se ha declarado a textura como tipo **Texture2D**, línea 3. La variable tiempo servirá como ángulo de rotación alrededor del eje **y**, línea 4.

Antes de continuar con el código, creé tres carpetas en el **Content** en el **Explorador de soluciones** de Visual Studio, los nombres que se utilizaron para las carpetas fueron: **Modelos**, **Shaders** y **Texturas**.

En cada una de ellas se añadieron el modelo **ElefanteC.fbx**, el shader **Ambiental.fx** y la imagen **Omision.png**, respectivamente.

Una vez que el **ContentManager** les ha asignado el **Asset** a cada elemento, en el método **LoadContent** de la clase **Game1** escriba las siguientes líneas.

Código 8-2

```
1.      protected override void LoadContent()
2.      {
3.          spriteBatch = new SpriteBatch(GraphicsDevice);
4.
5.          modelo = Content.Load<Model>(@"Modelos\ElefanteC");
6.          Effect efecto = Content.Load<Effect>(@"Shaders\Ambiental");
7.          textura = Content.Load<Texture2D>(@"Texturas\Omision");
8.          transformaciones = new Matrix[modelo.Bones.Count];
9.          modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
10.
11.         foreach (ModelMesh mesh in modelo.Meshes)
12.            foreach (ModelMeshPart meshPart in mesh.MeshParts)
13.                 meshPart.Effect = efecto.Clone(GraphicsDevice);
14.     }
```

De la línea 5 a la 7, Código 8-2, el método Load genérico de la clase **ContentManager** crea las instancias respectivas a su tipo de dato de entrada, y se los asigna a modelo, efecto y textura. Luego se copian las matrices de transfomación del modelo al arreglo transformaciones.

En las líneas 11 a 13, se copian los datos de efecto a la propiedad **Effect** del **ModelMeshpart** de cada **ModelMesh** del modelo.

Para rotar el modelo alrededor del eje **y** se ha declarado la variable **tiempo**, que servirá como ángulo. En el método **Update** de la clase **Game1,** se actualizará tiempo cada segundo con el total de tiempo que ha transcurrido la apliación desde que inició. En la línea 6, Código 8-3, se sebe convertir el tipo **double** a **float**.

Código 8-3

```
1.      protected override void Update(GameTime gameTime)
2.      {
3.          if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
4.              this.Exit();
5.
6.          tiempo = (Single)gameTime.TotalGameTime.TotalSeconds;
7.
8.          base.Update(gameTime);
9.      }
```

Por último, en el método **Draw** se dibuja la geometría como se vio en el Cómo cargar modelos 3D desde un archivo X y FBX, con la diferencia en que el **foreach** anidado, líneas 7 a 22, Código 8-4, tiene como elemento de interación una instancia **Effect**.

En la línea 9 del método **Draw**, se establece la técnica del shader con la propiedad **CurrentTechnique** de la clase **Effect** y se obtiene a partir de la colección **Techniques** del objeto **effect**. Se puede utilizar como índice de selección un entero, sin embargo, el **string** con el nombre de la técnica es más útil para el mantenimiento del programa.

Después de seleccionar la técnica del shader, se establecen los valores de las variables **uniform extern** del shader, con la ayuda del método **SetValue**. La instancia de la clase **Effect** tiene una colección de parámetros que representan estas variables del shader. La forma de selección es por medio de un tipo entero o un **string**. Este último se utiliza como preferente, por legibilidad del programa.

El método **SetValue** está sobrecargado 18 veces[38] para aceptar diferentes tipos de datos, sin embargo, no verifica en tiempo de compilación la integridad de éstos.

[38] Para consultar todas las sobrecargas del método consulte la siguiente página: http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=EN-US&k=k(MICROSOFT.XNA.FRAMEWORK.GRAPHICS.EFFECTPARAMETER.SETVALUE);k(DevLang-CSHARP)&rd=true

La línea 10 del método **Draw**, establece la matriz de mundo a partir del elemento del arreglo transformaciones y la matriz de rotación alrededor del eje **y**. En la línea 13 se establece la matriz de vista a partir del método **CreateLookAt**. Siguiendo con la asignación del color del material ambiental, línea 19 – 20, se debe convertir el tipo de dato **Color** por un **Vector4**, que contiene cuatro elementos de tipo flotante. Por último, se asigna la textura al shader en la línea 21.

Código 8-4

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(Color.White);
4.
5.          foreach (ModelMesh mesh in modelo.Meshes)
6.          {
7.              foreach (Effect effect in mesh.Effects)
8.              {
9.                  effect.CurrentTechnique = effect.Techniques["Material"];
10.                 effect.Parameters["world"].SetValue(
11.                     transformaciones[mesh.ParentBone.Index] *
12.                     Matrix.CreateRotationY(tiempo));
13.                 effect.Parameters["view"].SetValue(Matrix.CreateLookAt(
14.                     new Vector3(0, 0, 300), Vector3.Zero, Vector3.Up));
15.                 effect.Parameters["projection"].SetValue(
16.                     Matrix.CreatePerspectiveFieldOfView(
17.                     MathHelper.PiOver4, GraphicsDevice.Viewport.AspectRatio,
18.                     1.0F, 1000.0F));
19.                 effect.Parameters["colorMaterialAmbiental"].SetValue(
20.                     Color.Gray.ToVector4());
21.                 effect.Parameters["texturaModelo"].SetValue(textura);
22.             }// fin del foreach
23.             mesh.Draw();
24.         }// fin del foreach
25.
26.         base.Draw(gameTime);
27.     }
```

Hasta ahora no ha sido tan complicada la asingación de valores al shader, sin embargo, esta manera de utilizar el efecto no es la más conveniente, porque si quisiéramos pintar este mismo objeto **Model** con otros efectos de iluminación se tendría que clonar una vez más la instancia **Effect** del **ModelMesh**, ocupando cuatro bucles **foreach**. En los siguientes ejemplos de este capítulo se utiliza una manera más eficaz pero no tan sencilla o por lo menos no tan corta.

Corra el programa para ver un elefante, o el modelo tridimensional de su preferencia, pintado de un sólo color, véase Ilustración 8-1. Cambie la técnica **Material** por **Texturizacion**, en la línea 9 del método **Draw**.

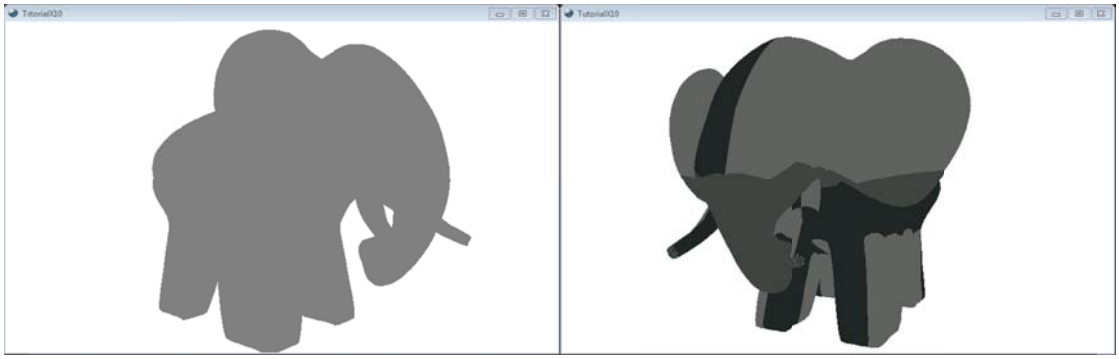

Ilustración 8-1 Integración de shader en XNA

8.1.2 Interfaz para el shader

El subtítulo de este apartado no debe confundirse con la palabra reservada interface de C#, es algo más general. Es la manera de comunicación entre XNA y el shader. Y eso es lo que se programara para dibujar la geometría con el efecto deseado, sin necesidad de cambiar sus propiedades de material.

A partir de este ejemplo y lo queda de este capítulo se reutilizarán los enlistados, con el fin de aprovechar las ventajas de la programación orientada a objetos.

Comience por crear un nuevo proyecto en Visual Studio de XNA Game. Vuelva a elaborar las tres carpetas **Modelos**, **Shaders** y **Texturas** en el **Content**, en la parte del explorador de soluciones. En la carpeta **Modelos**, añada cualquier modelo tridimensional que pueda manejar el **ContentManager**. En la carpeta **Shader** vuelva a añadir el efecto **Ambiental.fx** que se vio en el capítulo previo a éste. Asegúrese que la textura que agregue en la carpeta **Texturas**, sea compatible con los formatos que el **ContentManager** controla.

Con un diagrama de clases se puede visualizar mejor el software que se contruirá, para las interfaces de los efectos. En la ilustración 8 – 2 se muestran las interfaces **IFx**, **IDifuso** e **IEspecular**. Cada una de ellas servirá para el polimorfismo y herencia de las clases que interactúen con el shader.

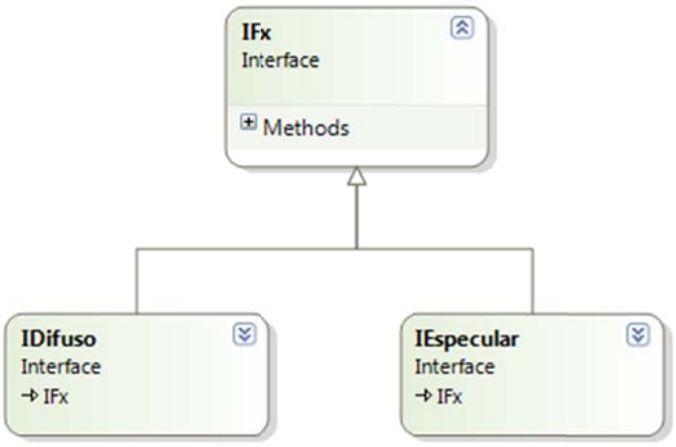

Ilustración 8-2 Diagrama de clases. Interfaces

Para crear una interfaz en Visual Studio seleccione el nombre de la solución en el explorador de soluciones. Haga clic derecho y seleccione **Añadir nuevo elemento**. Inmediatamente se abre una ventana de diálogo para seleccionar una plantilla de cualquiera de las categorías. Seleccione la plantilla Interface de la categoría **Código**. El nombre que le asigne a cualquier interfaz debe comenzar con la letra i mayúscula, seguida del nombre que la describe.

Una vez creado el archivo de la interfaz **IFx** escriba el método **DrawModelMeshPart**, dentro de las llaves de la interfaz **IFx**.

```
public interface IFx
{
    /// <summary>
    /// Método que dibuja un ModelMeshPart.
    /// </summary>
    void DrawModelMeshPart(Microsoft.Xna.Framework.Graphics.ModelMeshPart modelMeshPart,
        Microsoft.Xna.Framework.Graphics.GraphicsDevice graphicsDevice);
}
```

Este método dibujará el **ModelMeshPart** que compone al **ModelMesh** de la clase **Model**. Cabe aclarar que cada **ModelMeshPart** se distingue de otro por sus diferentes materiales.

La interfaz **IDifuso**, hereda de la interfaz IFx, y añade una propiedad que se llama **ColorDifuso**. Creé una nueva interfaz con el mismo nombre **Idifuso** y escriba las siguientes líneas.

```
public interface IDifuso : IFx
{
    /// <summary>
    /// Propiedad que obtiene o establece el color difuso.
    /// </summary>
    Color ColorDifuso { get; set; }
}
```

La creación de esta interfaz es para separar los modelos de iluminación difusa del especular. Por lo que la siguiente interfaz es para el especular.

Agregue una interfaz al proyecto con el nombre **IEspecular** y escriba en ella el siguiente enlistado.

```
{
    /// <summary>
    /// Propiedad que obtiene o establece el coeficiente especular.
    /// </summary>
    Single Ks { get; set; }

    /// <summary>
    /// Propiedad que obtiene o establece la potencia especular.
    /// </summary>
    Single N { get; set; }

    /// <summary>
    /// Propiedad que obtiene o establece el color especular.
    /// </summary>
    Color ColorEspecular { get; set; }

    /// <summary>
    /// Propiedad que habilita o inhabilita el modelo especular.
    /// </summary>
    Boolean HabilitarEspecular { get; set; }

    /// <sumary>
    /// Propiedad que obtiene o establece la posición de la cámara.
    /// </sumary>
    Vector3 PosicionCamara { get; set;}

}
```

Esta interfaz **IEspecular** tiene las propiedades del coeficiente especular, la potencia especular, el color especular, la posición de cámara y la opción de habilitar o inhabilitar el modelo de iluminación especular.

8.1.2.1 Clase FxAmbiental.

La clase **FxAmbiental**, que a continuación se describe, hereda de la interfaz **IFx**, y define el método **DrawModelMeshPart**. Además se definen los campos y propiedades que ayudan a la comunicación entre XNA y el efecto **Ambiental.fx**.

Creé una nueva clase en la solución, con el nombre **FxAmbiental**, y en la parte de directivas escriba las siguientes.

```
using System;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework;
```

Dentro del cuerpo de la clase escriba las variables instancia, líneas 3 – 9, Código 8-5. La única variable cuyo modificador de acceso es diferente a private es **effect**, línea 9. Esto es porque de esta clase se heredan

las siguientes clases para el shader **Direccional.fx** y **MulDirec.fx**, este último es una versión reducida del shader **Multilliminación** visto en el capítulo anterior. Véase que cada una de estas variables, excepto **habilitarTextura** y **effect**, corresponden a las variables **uniform** del shader **Ambiental.fx**.

En el constructor, líneas 11 – 16, se obtiene una instancia **Effect** como parámetro y el cuerpo inicializa el color ambiental e inhabilita la técnica de **Texturizacion**.

Cada una de las propiedades obtiene o establece el valor correspondiente a la variable de instancia de la clase, y para ser legible las propiedades tienen el mismo nombre que la variables, pero diferenciándolas por tener la primera letra en mayúsculas.

El descriptor de acceso **set** de cada propiedad establece el valor a la variable de instancia de la clase, antes de pasar el dato al método **SetValue**.

En casi todas las propiedades la asignación de los datos hacia el efecto es igual al ejemplo anterior, exceptuando en **HabilitarTextura**, línea35 – 44. El valor de ésta es de tipo boleano, sirve como selector entre las técnicas. Para ello se utiliza el operador ternario **? :**. Si el valor de la variable **habilitarTextura** es verdadero, se selecciona el **string "Texturizacion"**, en caso contrario será **"Material"**.

El método **DrawModelMeshPart** utiliza el modificador virtual para que otras clases derivadas de **FxAmbiental** puedan cambiar el cuerpo, pero no así la firma. La manera en que se manda a dibujar la geometría es las misma que se vio en el Vertex Buffer, en donde se envuelve el método **DrawIndexedPrimitives** entre los métodos **Begin** y **End** del objeto **effect**, líneas 103 – 110. En este caso el shader Ambiental contiene una pasada y se toma el primer elemento de la colección **Passes**. Todos los parámetros que necesita este método se obtienen del **ModelMeshPart**.

Código 8-5

```
1.      public class FxAmbiental : IFx
2.      {
3.          private Matrix view;
4.          private Matrix projection;
5.          private Matrix world;
6.          private Color colorAmbiental;
7.          private Texture2D textura2D;
8.          private Boolean habilitarTextura;
9.          protected Effect effect;
10.
11.         public FxAmbiental(Effect effect)
12.         {
13.             this.effect = effect;
14.             ColorAmbiental = Color.Black;
15.             HabilitarTextura = false;
16.         }
17.
18.         /// <summary>
19.         /// Propiedad que obtiene o establece el color ambiental del material.
20.         /// </summary>
21.         public virtual Color ColorAmbiental
22.         {
23.             get { return colorAmbiental; }
24.             set
25.             {
26.                 colorAmbiental = value;
27.                 effect.Parameters["colorMaterialAmbiental"].SetValue(
28.                 colorAmbiental.ToVector4());
29.             }
30.         }// fin de la propiedad ColorAmbiental
31.
32.         /// <summary>
33.         /// Propiedad que habilita o inhabilita la textura.
34.         /// </summary>
35.         public virtual Boolean HabilitarTextura
36.         {
37.             get { return habilitarTextura; }
38.             set
```

```
39.             {
40.                 habilitarTextura = value;
41.                 effect.CurrentTechnique = effect.Techniques[
42.                 (value) ? "Texturizado" : "Material"];
43.             }
44.         }// fin de la propiedad HabilitarTextura
45.
46.         /// <summary>
47.         /// Propiedad que obtiene o establece la matriz projection.
48.         /// </summary>
49.         public virtual Matrix Projection
50.         {
51.             get { return projection; }
52.             set
53.             {
54.                 projection = value;
55.                 effect.Parameters["projection"].SetValue(projection);
56.             }
57.         }// fin de la propiedad Projection
58.
59.         /// <summary>
60.         /// Propiedad que obtiene o establece la textura.
61.         /// </summary>
62.         public virtual Texture2D Textura2D
63.         {
64.             get { return textura2D; }
65.             set
66.             {
67.                 textura2D = value;
68.                 effect.Parameters["texturaModelo"].SetValue(textura2D);
69.             }
70.         }// fin de la propiedad Textura2D
71.
72.
73.         /// <summary>
74.         /// Propiedad que obtiene o establece la matriz view.
75.         /// </summary>
76.         public virtual Matrix View
77.         {
78.             get { return view; }
79.             set
80.             {
81.                 view = value;
82.                 effect.Parameters["view"].SetValue(view);
83.             }
84.         }// fin de la propiedad View
85.
86.         /// <summary>
87.         /// Propiedad que obtiene o establece la matriz world.
88.         /// </summary>
89.         public virtual Matrix World
90.         {
91.             get { return world; }
92.             set
93.             {
94.                 world = value;
95.                 effect.Parameters["world"].SetValue(world);
96.             }
97.         }// fin de la propiedad World
98.         #region IFx Members
99.
100.        public virtual void DrawModelMeshPart(ModelMeshPart modelMeshPart,
101.            GraphicsDevice graphicsDevice)
102.        {
103.            effect.Begin();
104.            effect.CurrentTechnique.Passes[0].Begin();
105.            graphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
106.                modelMeshPart.BaseVertex, 0,
107.                modelMeshPart.NumVertices, modelMeshPart.StartIndex,
108.                modelMeshPart.PrimitiveCount);
109.            effect.CurrentTechnique.Passes[0].End();
```

```
110.            effect.End();
111.        }// fin del método DrawModelMeshPart
112.
113.        #endregion
114.    }// fin de la clase FxAmbiental
```

Queda claro que la nueva forma de asignar un efecto a la geometría es más complicada, o más grande que la anterior, sin embargo, se obtiene un mejor control de los valores de éste.

La implementación del modelo de iluminación ambiental por shader, se hará en la clase **Game1**. Como primer paso escriba las siguientes variables de instancia de la clase **Game1**.

```
GraphicsDeviceManager graphics;
Model elefante;
Matrix[] transformaciones;
FxAmbiental fxAmbiental;
Texture2D textura;
Matrix view, projection;
```

Luego hay que cambiar el tamaño del back – buffer, esto con la finalidad de seguir algunas de las recomendaciones que se hace en la documentación acerca de tamaños de pantalla para ser representados en cualquier dispositivo visual en el que se conecte el Xbox 360. El tamaño recomendado es de 1280 para el ancho y 720 para el alto.

En el constructor de la clase **Game1**, después de la asignación de la ruta del **Content**, escriba las siguientes líneas.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    graphics.PreferredBackBufferHeight = 720;
    graphics.PreferredBackBufferWidth = 1280;
}
```

La propiedad **PreferredBackBufferHeight** obtiene o establece el alto del búfer, y **PreferredBackBufferWidth** hace lo mismo, pero para el alto del búfer.

En el método **Initialize**, inicalice las matrices **view** y projection, como se muestra en el siguiente enlistado, y que se deja como opción los valores propuestos aquí, pues es para visualizar toda la geometría que se dibujará en este ejemplo y los dos más que le siguen.

```
protected override void Initialize()
{
    view = Matrix.CreateLookAt(new Vector3(120.0F, 96.0F, 473.0F),
            new Vector3(75.0F, 104.0F, 7.0F), Vector3.Up);
    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            GraphicsDevice.Viewport.AspectRatio,
            1.0F, 1000.0F);

    base.Initialize();
}
```

En el método **LoadContent** de la clase **Game1**, se cargan los datos de textura, el modelo trdimensional y el shader, los cuales no deben ser desconcidos para el lector, pues en anteriores ejemplos se utilizaron. Luego de leer los datos del archivo **Ambiental.fx** y asignarlos al objeto **fxAmbiental**, se modifican las propiedades **ColorAmbiental** y **Textura2D**.

```
protected override void LoadContent()
```

```
{
    elefante = Content.Load<Model>(@"Modelos\Elefante");
    transformaciones = new Matrix[elefante.Bones.Count];
    elefante.CopyAbsoluteBoneTransformsTo(transformaciones);

    fxAmbiental = new FxAmbiental(
            Content.Load<Effect>(@"Shaders\Ambiental"));
    fxAmbiental.ColorAmbiental = Color.Brown;
    fxAmbiental.Textura2D = textura;

    textura = Content.Load<Texture2D>(@"Texturas\Omision");
}
```

Para concluir con este ejemplo, el método **Draw** de la clase **Game1** retoma parte del primer ejemplo del Cómo cargar modelos 3D desde un archivo X y FBX, en dónde se dibuja un **ModelMeshPart** de la colección **ModelMesh** de la instancia de la clase **Model**. Sin embargo, en esta ocasión se mandarán a dibujar todos los **ModelMeshPart**.

En las líneas 5 y 6, Código 8-6, se establecen las matrices de vista y proyección del objeto **fxAmbiental**. Sin duda, estas propiedades pueden no estar aquí, pero se ha dejado así porque son algunas de las variables que pueden cambiar en una aplicación, por no decir las comunes.

El primer **foreach**, líneas 8 – 22, mapea cada elemento de la colección **Meshes** del objeto **elefante**. Enseguida se establece el valor de la propiedad **World** del objeto **fxAmbiental** con la matriz de transformación obtenida del objeto **elefante**.

Se debe establecer el búfer de índices en el dispositivo gráfico, línea 12, a partir de la instancia **mesh** que el primer **foreach** utiliza como elemento de mapeo sobre la colección **Meshes** del objeto **elefante**. La propiedad **IndexBuffer** de la clase **ModelMesh** solo obtiene dicho búfer.

En el **foreach** anidado, líneas 14 – 21, se utiliza el objeto meshPart para hacer referencia a cada elemento de la colección **MeshPart** del objeto mesh que utiliza el primer **foreach**. Éste es menester utilizarlo para establecer el búfer de vértices y la declaración de estos, líneas 16 – 18, las propiedades **VertexBuffer**, **StreamOffset**, **VertexStride** y **VertexDeclaration** son la clave para dicho éxito.

Antes de la llave que cierra el **foreach** anidado se invoca el método **DrawModelMeshPart** del objeto **fxAmbiental**, y cuyos parámetros son el **mehsPart** y el **GraphicsDevice**, línea 20.

Código 8-6

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(Color.White);
4.
5.          fxAmbiental.View = view;
6.          fxAmbiental.Projection = projection;
7.
8.          foreach (ModelMesh mesh in elefante.Meshes)
9.          {
10.             fxAmbiental.World = transformaciones[mesh.ParentBone.Index];
11.
12.             GraphicsDevice.Indices = mesh.IndexBuffer;
13.
14.             foreach (ModelMeshPart meshPart in mesh.MeshParts)
15.             {
16.                 GraphicsDevice.Vertices[0].SetSource(mesh.VertexBuffer,
17.                         meshPart.StreamOffset, meshPart.VertexStride);
18.                 GraphicsDevice.VertexDeclaration = meshPart.VertexDeclaration;
19.
20.                 fxAmbiental.DrawModelMeshPart(meshPart, GraphicsDevice);
21.             }
22.         }
23.
24.         base.Draw(gameTime);
```

```
25.     }
```

Ejecute el programa con la tecla **F5** y verá algo parecido a la ilustración 8 -3; corriga cualquier error de compilación si es el caso y vuleva a intentar.

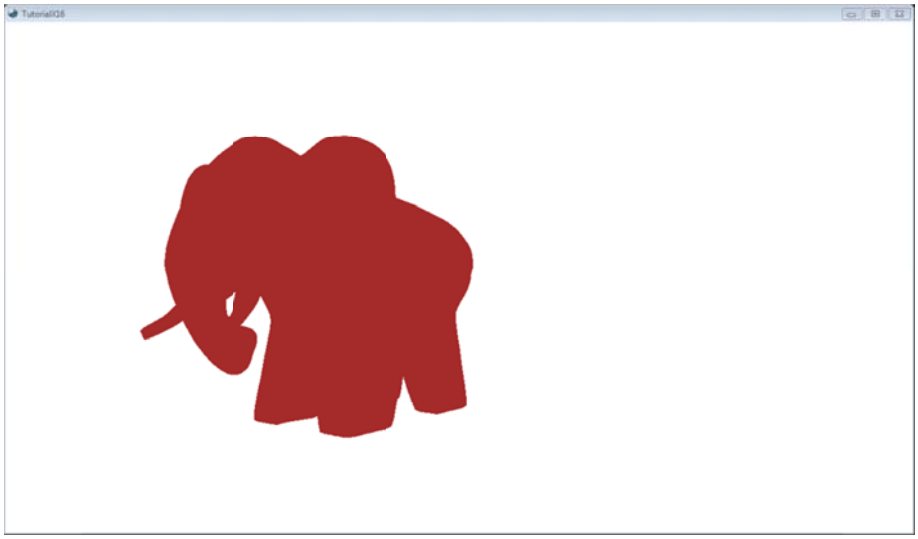

Ilustración 8-3 Integración de shader en XNA

8.2 Luz direccional

Para este ejemplo se usa el shader que implementa una luz de tipo direccional que se estudió en el capítulo anterior, empero se le agregó todas las variables **uniform** necesarias para el modelo de iluminación especular, como no es el punto de este capítulo explicar el siguiente enlistado, se deja como ejercicio entender su funcionamiento.

Escriba el siguiente código en Fx Composer y luego agréguelo en la carpeta **Shaders** de la solución anterior, en la que se creó la clase **FxAmbiental**.

Código 8-7

```
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.     float3 posicionCamara
20.     <
21.         string UIName = "Posición cámara";
22.     >;
23.     float ks
24.     <
25.         string UIWidget = "slider";
26.         float UIMin = 0.0F;
27.         float UIMax = 1.0F;
```

```
28.         float UIStep = 0.05F;
29.         string UIName = "Specular";
30.     > = {0.05F};
31.
32.     float n
33.     <
34.         string UIWidget = "slider";
35.         float UIMin = 0.0F;
36.         float UIMax = 128.0F;
37.         float UIStep = 1.0F;
38.         string UIName = "Specular pow";
39.     > = 5.0F;
40.
41.     float4 colorMaterialAmbiental
42.     <
43.         string UIName = "Material ambiental";
44.         string UIWidget = "Color";
45.     > = {0.05F, 0.05F, 0.05F, 1.0F};
46.     float4 colorMaterialDifuso
47.     <
48.         string UIName = "Material difuso";
49.         string UIWidget = "Color";
50.     > = {0.24F, 0.34F, 0.39F, 1.0F};
51.     float4 colorMaterialEspecular
52.     <
53.         string UIName = "Material especular";
54.         string UIWidget = "Color";
55.     > = {1.0F, 1.0F, 1.0F, 1.0F};
56.
57.     texture texturaModelo
58.     <
59.         string UIName = "Textura";
60.     >;
61.
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 VertexShaderEntrada
73.     {
74.         float4 Posicion : POSITION;
75.         float3 Normal : NORMAL;
76.         float2 CoordenadaTextura : TEXCOORD0;
77.     };
78.
79.     struct VertexShaderSalida
80.     {
81.         float4 Posicion : POSITION;
82.         float2 CoordenadaTextura : TEXCOORD0;
83.         float3 WorldNormal : TEXCOORD1;
84.         float3 WorldPosition : TEXCOORD2;
85.     };
86.
87.     struct PixelShaderEntrada
88.     {
89.         float2 CoordenadaTextura : TEXCOORD0;
90.         float3 WorldNormal : TEXCOORD1;
91.         float3 WorldPosition : TEXCOORD2;
92.     };
93.     bool modIluEsp
94.     <
95.         string UIName = "Modelo Iluminación Especular";
96.     > = false;
97.
98.     VertexShaderSalida MainVS(VertexShaderEntrada entrada)
```

```
99.     {
100.        VertexShaderSalida salida;
101.        float4x4 wvp = mul(world, mul(view, projection));
102.        salida.Posicion = mul(entrada.Posicion, wvp);
103.        salida.WorldNormal = mul(entrada.Normal, world);
104.        float4 worldPosition = mul(entrada.Posicion, world);
105.        salida.WorldPosition = worldPosition / worldPosition.w;
106.        salida.CoordenadaTextura = entrada.CoordenadaTextura;
107.
108.        return salida;
109.    }// fin del vertex shader MainVS
110.
111.    float4 MainPS(PixelShaderEntrada entrada,
112.    uniform bool habilitarTextura) : COLOR
113.    {
114.        // modelo de iluminación difusa
115.        float3 dirLuz = normalize(direccionLuz);
116.        float intensidadDifusa = max(dot(-dirLuz, entrada.WorldNormal), 0.0F);
117.        float4 difuso = colorMaterialDifuso * intensidadDifusa;
118.        float4 color;
119.        // color final
120.        color = colorMaterialAmbiental + difuso;
121.
122.        // modelo de iluminación especular
123.        if(modIluEsp)
124.        {
125.            // iluminación especular
126.            float3 reflexion = normalize(2 * entrada.WorldNormal *
127.            max(dot(-dirLuz, entrada.WorldNormal), 0.0F) + dirLuz);
128.            float3 direccionCamara = normalize(posicionCamara -
129.            entrada.WorldPosition.xyz);
130.            float intensidadEspecular = pow(max(dot(reflexion,
131.                direccionCamara), 0.0F), n);
132.            float4 especular = ks * intensidadEspecular * colorMaterialEspecular;
133.
134.            color += especular * intensidadDifusa;
135.        }// fin del if
136.
137.        if(habilitarTextura)
138.        {
139.            float4 colorTextura = tex2D(modeloTexturaMuestra,
140.                entrada.CoordenadaTextura);
141.            color *= colorTextura;
142.        }
143.        // color final
144.        color.a = 1.0F;
145.        return color;
146.    }// fin pixel shader MainPS
147.
148.    technique Texturizado
149.    {
150.        pass P0
151.         {
152.            VertexShader = compile vs_3_0 MainVS();
153.            PixelShader = compile ps_3_0 MainPS(true);
154.        }
155.    }
156.
157.    technique Material
158.    {
159.        pass P0
160.        {
161.            VertexShader = compile vs_3_0 MainVS();
162.            PixelShader = compile ps_3_0 MainPS(false);
163.        }
164.    }
```

La clase **FxDireccional** es la clase que servirá como “interfaz” entre el shader anterior y XNA. Esta clase necesita las propiedades de las interfaces **IDifuso**, **IEspecular** y las hereda de la clase **FxAmbiental**; en la

Ilustración 8-4 se muestra el diagrama de clase. Creé una nueva clase en la solución anterior, llamada **FxDireccional** y escriba el Código 8-8.

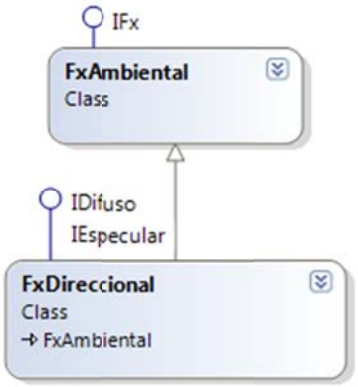

Ilustración 8-4 Diagrama de clases. FxDireccional.

Entre las líneas 3 – 9, se encuentran las variables de instancia que sirven de intermediarios entre los datos que recibe del usuario de la clase **FxDireccional** y el shader.

Vea la equivalencia entre estas variables y aquellas a las que se quiere asociar al shader, excepto en aquellas que se han declarado como estructuras **Color**, líneas 5 y 6.

En el constructor de la clase, líneas 11 – 19, se inicializan las variables de instancia a partir de sus propiedades, esto con la finalidad de darle los valores a cada elemento de la colección de **Parameters** del efecto. Cada propiedad se encargará de establecer dicho valor a cada variable **uniform extern**.

Para la propiedad **Ks**, líneas 26 – 34, coeficiente especular, no se tiene ninguna restricción en el momento de establecer el valor de la variable de instancia **ks**, que varía entre cero y uno. Esto se ha dejado así para que se pueda visualizar qué pasa con valores mayores a uno y para valores negativos. Recuerde que primero se debe asignar el dato a la variable de instancia, y luego ésta se le establece al elemento de la colección correspondiente.

La propiedad potencia especular **N**, líneas 39 – 47, tampoco restringe el valor que generalmente es entre 1 y 128. El objetivo, al igual que la propiedad **Ks**, es que se pueda visualizar con valores fuera de lo común.

Las propiedades **ColorEspecular**, líneas 52 – 60, y **ColorDifuso**, líneas 96 – 105, tienen que convertir la estructura **Color** a una estructura **Vector4** con el fin de garantizar la integración de los datos.

HabilitarEspecular, líneas 66 – 74, y **PosicionCamara**, líneas 79 – 87, solo pasan el valor correspondiente al elemento de la colección **Parameters**.

La propiedad **DireccionLuz**, líneas 112 – 121, evalúa el valor antes de asignar el dato a la variable **direccionLuz**, línea 117; si el valor es diferente de **Vector3.Zero** se le asigna a la variable **direccionLuz**, en caso contrario se le da un valor por default. Esto garantiza que siempre tendrá una dirección.

Como esta clase hereda de la clase **FxAmbiental**, las propiedades **ColorAmbiental**, **HabilitarTextura**, **Projection**, **Textura2D**, **View** y **World** deberían de sobrescribirse, debido a que el índice que se utiliza para seleccionar el elemento de la colección **Parameters** del objeto **Effect** podría cambiar. Esto depende del nombre de las variables **uniform extern** de los shaders, en caso que no coincida, el error no se detectará en tiempo de compilación sino en ejecución. Sin embargo, en todos los ejemplos de shaders se les dio un mismo nombre a cada una de las variables para permitir una programación más sencilla. La mejor práctica es agregar el modificador virtual a las propiedades base para poder cambiarlas en las clases que las hereden.

Código 8-8

```
1.      public class FxDireccional : FxAmbiental, IEspecular, IDifuso
2.      {
```

```
3.          private Single ks;
4.          private Single n;
5.          private Color colorDifuso;
6.          private Color colorEspecular;
7.          private Vector3 direccionLuz;
8.          private Boolean habilitarEspecular;
9.          private Vector3 posicionCamara;
10.
11.         public FxDireccional(Effect effect)
12.             : base(effect)
13.         {
14.             ColorDifuso = Color.WhiteSmoke;
15.             ColorEspecular = Color.WhiteSmoke;
16.             DireccionLuz = new Vector3(-1.0F, -1.0F, -1.0F);
17.             Ks = 20.0F;
18.             N = 128.0F;
19.         }// fin del constructor
20.
21.         #region IEspecular Members
22.
23.         /// <summary>
24.        /// Propiedad que obtiene o establece el coeficiente especular.
25.        /// </summary>
26.        public float Ks
27.        {
28.            get { return ks; }
29.            set
30.            {
31.                 ks = value;
32.                 base.effect.Parameters["ks"].SetValue(ks);
33.             }
34.         }// fin de la propiedad Ks
35.
36.         /// <summary>
37.         /// Propiedad que obtiene o establece la potencia
38.         /// </summary>
39.         public float N
40.         {
41.             get { return n; }
42.             set
43.             {
44.                 n = value;
45.                 base.effect.Parameters["n"].SetValue(n);
46.             }
47.         }// fin de la propiedad N
48.
49.         /// <summary>
50.         /// Propiedad que obtiene o establece el color especular del material.
51.         /// </summary>
52.         public Color ColorEspecular
53.         {
54.             get { return colorEspecular; }
55.             set
56.             {
57.                 colorEspecular = value;
58.                 base.effect.Parameters["colorMaterialEspecular"].SetValue(
59.                     colorEspecular.ToVector4());
60.             }
61.         }// fin de la propiedad ColorEspecular
62.
63.         /// <summary>
64.         /// Propiedad que habilita o inhabilita el modelo iliminación especular.
65.         /// </summary>
66.         public Boolean HabilitarEspecular
67.         {
68.             get { return habilitarEspecular; }
69.             set
70.             {
71.                 habilitarEspecular = value;
72.                 base.effect.Parameters["modIluEsp"].SetValue(habilitarEspecular);
73.             }
```

```
74.         }// fin de la propiedad DireccionLuz
75.
76.         /// <summary>
77.         /// Propiedad que obtiene o establece la posición de la cámara.
78.         /// </summary>
79.         public Vector3 PosicionCamara
80.         {
81.             get { return posicionCamara; }
82.             set
83.             {
84.                 posicionCamara = value;
85.                 effect.Parameters["posicionCamara"].SetValue(posicionCamara);
86.             }
87.         }// fin de la propiedad PosicionCamara
88.
89.         #endregion
90.
91.         #region IDifuso Members
92.
93.         /// <summary>
94.         /// Propiedad que obtiene o establece el color difuso del material.
95.         /// </summary>
96.         public Color ColorDifuso
97.         {
98.             get { return colorDifuso; }
99.             set
100.            {
101.                colorDifuso = value;
102.                base.effect.Parameters["colorMaterialDifuso"].SetValue(
103.                colorDifuso.ToVector4());
104.            }
105.        }// fin de la propiedad ColorDifuso
106.
107.        #endregion
108.
109.        /// <summary>
110.        /// Propiedad que obtiene o establece la dirección de luz.
111.        /// </summary>
112.        public Vector3 DireccionLuz
113.        {
114.            get { return direccionLuz; }
115.            set
116.            {
117.                direccionLuz = (value != Vector3.Zero) ? value :
118.                    new Vector3(-1.0F, -1.0F, -1.0F);
119.                base.effect.Parameters["direccionLuz"].SetValue(direccionLuz);
120.            }
121.        }// fin de la propiedad DireccionLuz
122.    }// fin de la clase FxDireccional
```

La implementación de la clase **FxAmbiental** se realiza en la clase **Game1**. Abra el archivo **Game1.cs** y escriba dos variables de instancia nuevas.

```
FxDireccional fxDireccional;
Vector3 posicion;
```

La variable **posicion** sirve para establecer el lugar de la cámara en la propiedad **PosicionCamara** del objeto **fxDireccional**.

En el método **Initialize** se inicializa la estructura posicion para luego dársela al método **CreateLookAt** como uno de sus parámetros.

```
protected override void Initialize()
{
    posicion = new Vector3(-120.0F, 96.0F, 473.0F);
    view = Matrix.CreateLookAt(posicion,
        new Vector3(75.0F, 104.0F, 7.0F), Vector3.Up);
```

```
    projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
        GraphicsDevice.Viewport.AspectRatio,
        1.0F, 1000.0F);

    base.Initialize();
}
```

En el método **LoadContent** se crea la instancia **fxAmbiental** y se inicializan algunas de sus propiedades, véase el siguiente código.

```
protected override void LoadContent()
{
    elefante = Content.Load<Model>(@"Modelos\Elefante");
    transformaciones = new Matrix[elefante.Bones.Count];
    elefante.CopyAbsoluteBoneTransformsTo(transformaciones);

    fxAmbiental = new FxAmbiental(
            Content.Load<Effect>(@"Shaders\Ambiental"));
    fxAmbiental.ColorAmbiental = Color.Brown;
    fxAmbiental.Textura2D = textura;

    textura = Content.Load<Texture2D>(@"Texturas\Omision");

    fxDireccional = new FxDireccional(Content.Load<Effect>(@"Shaders\Direccional"));
    fxDireccional.Textura2D = textura;
    fxDireccional.ColorAmbiental = Color.Black;
    fxDireccional.ColorDifuso = new Color(5, 2, 2);
    fxDireccional.ColorEspecular = Color.Brown;
    fxDireccional.DireccionLuz = new Vector3(0, -1, -1);
    fxDireccional.HabilitarEspecular = true;
    fxDireccional.HabilitarTextura = true;
    fxDireccional.Ks = 10;
    fxDireccional.N = 120;
    fxDireccional.PosicionCamara = posicion;
}
```

En el método **Draw**,Código 8-9, se actualizan las propiedades **View**, línea 8; **Projection**, línea 9; y **World**, líneas 13 y 24; del objeto **fxDireccional**.

Antes de llamar al método **DrawModelMeshPart** de la instancia **FxDireccional**, línea 25, se traslada la geometría menos cien unidades sobre el eje **x** y menos doscientas unidades sobre el eje **z**. Corra el ejemplo con la tecla **F5** y verá algo similar a la Ilustración 8-5.

Código 8-9

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(Color.White);
4.
5.          fxAmbiental.View = view;
6.          fxAmbiental.Projection = projection;
7.
8.          fxDireccional.View = fxAmbiental.View;
9.          fxDireccional.Projection = fxAmbiental.Projection;
10.         foreach (ModelMesh mesh in elefante.Meshes)
11.         {
12.             fxAmbiental.World = transformaciones[mesh.ParentBone.Index];
13.             fxDireccional.World = fxAmbiental.World;
14.
15.             GraphicsDevice.Indices = mesh.IndexBuffer;
16.
17.             foreach (ModelMeshPart meshPart in mesh.MeshParts)
18.             {
19.                 GraphicsDevice.Vertices[0].SetSource(mesh.VertexBuffer,
20.                         meshPart.StreamOffset, meshPart.VertexStride);
21.                 GraphicsDevice.VertexDeclaration = meshPart.VertexDeclaration;
```

```
22.
23.                 fxAmbiental.DrawModelMeshPart(meshPart, GraphicsDevice);
24.                 fxDireccional.World *= Matrix.CreateTranslation(-100, 0, -200);
25.                 fxDireccional.DrawModelMeshPart(meshPart, GraphicsDevice);
26.             }
27.         }
28.
29.         base.Draw(gameTime);
30.     }
```

Si ocurre un error de compilación o de ejecución revisé de nueva cuenta el código, y vuelva intentar.

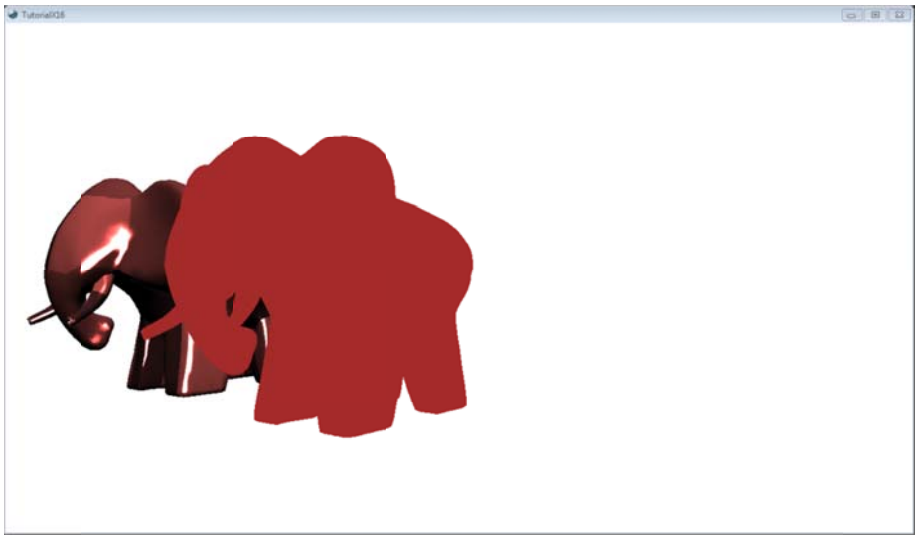

Ilustración 8-5 Integración de los shaders Ambiental y Direccional en XNA

8.3 Efecto multiluces

Para este último ejemplo se utiliza una versión reducida del shader **MultiLuces** que se vio en el capítulo anterior. Este nuevo shader solo contiene un tipo de fuente: **la direccional**. Esto es con el fin que se entienda cómo se crea la clase que comunica los datos de XNA hacia el efecto, y se deja como ejercicio al lector crear la clase correspondiente para la versión completa de **MultiLuces**.

En el listado siguiente se muestra la versión reducida de **MultiLuces**, y se espera que el lector pueda interpretar con facilidad cada línea, pues la explicación ya fue comentada en el capítulo anterior.

Abra Fx Composer y creé un nuevo proyecto para escribir las siguientes líneas en un archivo **fx**.

Código 8-10

```
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.
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.     int numLuces = 4;
95.
96.     VertexShaderSalida MainVS(float3 posicion : POSITION,
97.         float3 normal : NORMAL, float2 coordenadaTextura : TEXCOORD0)
98.     {
99.         VertexShaderSalida salida;
100.        float4x4 mvp = mul(world, mul(view, projection));
101.        salida.Posicion = mul(float4(posicion, 1.0F), mvp);
102.        salida.WorldNormal = mul(normal, world);
103.        salida.WorldPosition = mul(float4(posicion, 1.0F), world);
104.        salida.CoordenadaTextura = coordenadaTextura;
105.
106.        return salida;
107.    }// fin del vertexshader MainVS
108.
109.    float4 MainPSAmbiental(PixelShaderEntrada entrada,
110.        uniform bool habiTextura) : COLOR
111.    {
112.        float4 color = colorMaterialAmbiental;
113.        if(habiTextura)
114.        {
115.            color *=  tex2D(modeloTexturaMuestra, entrada.CoordenadaTextura);
116.        }
117.        return color;
118.    }// fin del pixelshader MainPSAmbiental
119.
120.    // Función que calcula la reflexión especular.
121.    float4 ReflexionEspecular(float3 direccionLuz, float3 worldPosicion,
122.        float3 worldNormal, float4 colorLuz)
123.    {
124.        float3 reflexion = normalize(2 * worldNormal *
125.        max(dot(direccionLuz, worldNormal), 0.0F) - direccionLuz);
126.
127.        float3 direccionCamara = normalize(posicionCamara - worldPosicion);
128.        float reflexionEspecular = pow(max(dot(reflexion, direccionCamara), 0.0F), n);
129.
130.        return (ks * reflexionEspecular) * (colorLuz * colorMaterialEspecular);
131.    }// fin de la función ReflexionEspecular
132.
133.    float4 Phong(float3 direccionLuz, float3 worldPosicion,
134.        float3 worldNormal, float4 colorLuz, float4 colorMater)
135.    {
136.        float reflexionDifusa = max(dot(direccionLuz, worldNormal), 0.0F);
137.        float4 phong = reflexionDifusa * colorLuz * colorMater;
138.        // activación de la reflexión especular
139.        if(habiEspec)
140.        {
141.            phong += ReflexionEspecular(direccionLuz, worldPosicion,
142.                worldNormal, colorLuz) * colorMater * reflexionDifusa;
143.        }// fin del if
144.
145.        return phong;
146.    }// fin de la función Phong
147.
148.    // Función que calcula la luz tipo direccional
149.    float4 FuenteDireccional(LuzDireccional luz, float3 worldPosicion,
150.        float3 worldNormal, float4 colorMater)
151.    {
152.        return Phong(-normalize(luz.Direccion), // dirección de luz
153.            worldPosicion,
154.            worldNormal, luz.Color,
155.            colorMater);
156.    }// fin de la función FuenteDireccional
157.
158.    LuzDireccional direccionales[20];
159.
160.    float4 MainPS(PixelShaderEntrada entrada, uniform bool texturizado) : COLOR
```

```
161.    {
162.        float4 color = 0;
163.        float4 colorDifuso = colorMaterialDifuso;
164.
165.        if(texturizado)
166.        {
167.            colorDifuso *= tex2D(modeloTexturaMuestra, entrada.CoordenadaTextura);
168.        }// fin del if
169.
170.        for(int i = 0; i < numLuces; i++)
171.        {
172.            if(direccionales[i].Encender != 0)
173.            {
174.                color += FuenteDireccional(direccionales[i], entrada.WorldPosition,
175.                    entrada.WorldNormal, colorDifuso);
176.            }
177.        }// fin del for
178.        color.a = 1.0F;
179.        return color;
180.    }// fin del pixelShader mainPS
181.
182.    technique Texturizado
183.    {
184.        pass Ambiental
185.        {
186.            VertexShader = compile vs_3_0 MainVS();
187.            PixelShader = compile ps_3_0 MainPSAmbiental(true);
188.        }
189.        pass MultiLuces
190.        {
191.            VertexShader = compile vs_3_0 MainVS();
192.            PixelShader = compile ps_3_0 MainPS(true);
193.        }
194.    }// fin de la técnica Texturizado
195.
196.    technique Material
197.    {
198.        pass Ambiental
199.        {
200.            VertexShader = compile vs_3_0 MainVS();
201.            PixelShader = compile ps_3_0 MainPSAmbiental(false);
202.        }
203.
204.        pass MultiLuces
205.        {
206.            VertexShader = compile vs_3_0 MainVS();
207.            PixelShader = compile ps_3_0 MainPS(false);
208.        }
209.    }// fin de la técnica Texturizado
```

Continuando con el proyecto de Visual Studio se deben agregar dos clases, llamadas **FxMultiluces** y **LuzDireccional**. La primera representa la clase que comunicará los datos desde XNA hacia el shader, y la segunda representará los campos de la estructura **LuzDireccional** del shader.

En la Ilustración 8-6, se muestra el diagrama de clase para **FxMultiluces**. Ésta hereda de la clase **FxAmbiental** y de las interfaces **IDifuso** e **IEspecular**. Lo que hace pensar que dicha herencia de clase pudo haberse dado de **FxDireccional** hacia **FxMultiluces**. Pero la propiedad **DireccionLuz** de **FxDireccional** no se encuentra como miembro de la clase **FxMultiluces**, y por eso se consideran como clases diferentes.

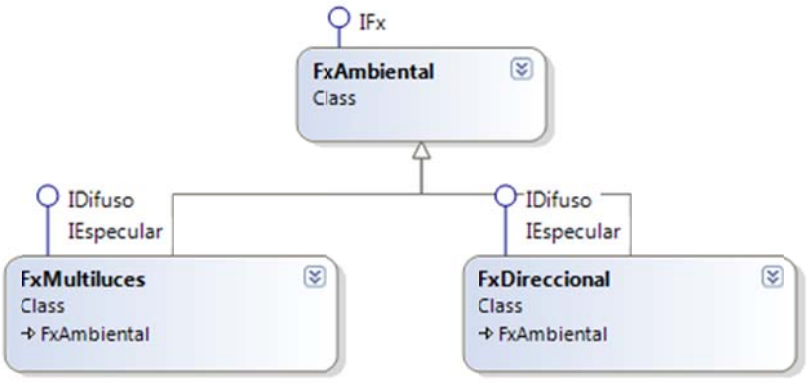

Ilustración 8-6 Diagrama de clases. FxMultiluces

Antes de escribir la clase **FxMultiluces** hay que definir los campos de la clase **LuzDireccional** que utiliza **FxMultiluces** como dependencia.

En las líneas 3 a 6, Código 8-11, se declaran las variables de instancia **color**, **dirección**, **encender** y **estrucutraLuz**. Las tres primeras corresponden a la estructura que **LuzDireccional** del shader **Multluces.fx**. Pero si uno observa el campo **Encender** de dicha estructura en el shader no es del tipo booleano, sino entera. Para conservar la integridad de los datos, en las propiedades, la asignación de dichos valores se hace adecuadamente y dependiendo de la información.

La variable **estructuraLuz**, línea 6, representa un elemento del parámetro direccionales del shader, éste es un arreglo de veinte estrucuturas **LuzDireccional**, así que cada una tiene tres elementos a los cuales se les puede asignar un valor.

El constructor de la clase **LuzDireccional**, líneas 8 – 14, establece la dirección de la luz, el color e inhabilita la luz. Por default todas las luces estarán apagadas.

Para la propiedad **Color**, líneas 19 – 28, se transforma la estructura **Color** por **Vector4**. En la propiedad **DireccionLuz** se evita que se establezca un valor tipo **Vector3.Zero**.

En el descriptor de acceso **set**, líneas 50 – 55 de la propiedad **Encender**, se establece el valor a uno si el valor de la variable encender es **true** y en caso contrario sería cero.

Código 8-11

```
1.      public class LuzDireccional
2.      {
3.          private Color color;
4.          private Vector3 direccion;
5.          private Boolean encender;
6.          private EffectParameter estructuraLuz;
7.
8.          public LuzDireccional(EffectParameter effectParameter, Vector3 direccion)
9.          {
10.             estructuraLuz = effectParameter;
11.             Direccion = direccion;
12.             Color = Color.WhiteSmoke;
13.             Encender = false;
14.         }
15.
16.         /// <summary>
17.         /// Propiedad que obtiene o establece el color de la luz.
18.         /// </summary>
19.         public Color Color
20.         {
21.             get { return color; }
22.             set
23.             {
24.                 color = value;
```

```
25.                 estructuraLuz.StructureMembers["Color"].SetValue(
26.                     color.ToVector4());
27.             }
28.         }// fin de la propiedad Color
29.
30.         /// <summary>
31.         /// Propiedad que obtiene o establece el vector de dirección de la luz.
32.         /// </summary>
33.         public Vector3 Direccion
34.         {
35.             get { return direccion; }
36.             set
37.             {
38.                 direccion = (value != Vector3.Zero) ? value : new Vector3(-1.0F,
39.                     -1.0F, -1.0F);
40.                 estructuraLuz.StructureMembers["Direccion"].SetValue(direccion);
41.             }
42.         }// fin de la propiedad Direccion
43.
44.         /// <summary>
45.         /// Propiedad que enciende o apaga la luz.
46.         /// </summary>
47.         public Boolean Encender
48.         {
49.             get { return encender; }
50.             set
51.             {
52.                 encender = value;
53.                 estructuraLuz.StructureMembers["Encender"].SetValue(
54.                     (encender) ? 1 : 0);
55.             }
56.         }// fin de la propiedad Encender
57.     }// fin de la clase LuzDireccional
```

Agregue una nueva clase en la solución en Visual Studio cuyo nombre sea **FxMultiluces**, y escriba el Código 8-12.

Cada una de las variables del efecto tienen su álter ego en las variables de instancia, líneas 3 – 10. Como se ha manejado anteriormente no todas las variables corresponden al mismo tipo de dato, sin embargo la integridad se conserva en las propiedades.

En el constructor se inicializan los colores del material, el coeficiente especular y su potencia; se inhabilita la textura y se le asigna el parámetro correspondiente al arreglo que tiene las estructuras de la luz direccional, línea 21.

La definición de las propiedades, en su mayoría, son las mismas que en la clase **FxDireccional**, así que solo se explicara el método **DrawModelMeshPart**, líneas 131 – 168

Este shader contiene dos pasadas por técnica y deben ser llamadas una a una en el método **DrawModelMeshPart** de la clase **FxAmbiental**, para sumar los efectos de ambas pasadas. Para lograr dicho efecto se debe inhabilitar la escritura sobre el **DepthBuffer** y habilitar el canal alfa; luego se deben regresar a un estado anterior, para no afectar a las demás geometrías. El cambio debe hacerse solo para la clase **FxMultiLuces**, así que se deberá reescribir el método **DrawModelMeshPart** que se hereda de la clase **FxAmbiental**, líneas 131 – 168.

En el método **DrawModelMeshPart** se hace llamar dos veces el método **DrawIndexedPrimitives** de la instancia **graphicsDevice**, líneas 138 -141, y líneas 158 – 161, cada una de ellas se encuentra entre los métodos **Begin** y **End** de cada elemento **EffectPass** que corresponde a **Ambiental** y **Multiluces** del shader.

Despúes de llamar por primera vez al método **DrawIndexedPrimitives**, se habilita la transparencia, línea 145, y junto con ella todas las propiedades que van de la mano del **blending** .

Enseguida se llama el método **DrawIndexedPrimitives**, no sin antes propagar el efecto anterior al dispositivo gráfico antes del renderizado. Para lograr dicho fin, se llama el método **CommitChanges**, línea 153.

El método **CommitChanges**, debe llamarse dentro del par de métodos **Begin** y **End** de la pasada actual, y antes de varios métodos **DrawPrimitives**, que sirven como propagadores de los cambios hacia el dispositivo gráfico.

Antes de llamar el método **End**, línea 164, se inhabilita el **blending** para no afectar las goemetrias posteriores al llamado del método **DrawModelMeshPart**.

Código 8-12

```
1.      public class FxMultiluces : FxAmbiental, IEspecular, IDifuso
2.      {
3.          private Color colorDifuso;
4.          private Color colorEspecular;
5.          private Boolean habilitarEspecular;
6.          private Single ks;
7.          private Single n;
8.          private Int16 numeroLuces;
9.          private EffectParameter direccionales;
10.         private Vector3 posicionCamara;
11.
12.         public FxMultiluces(Effect effect)
13.             : base(effect)
14.         {
15.             ColorDifuso = Color.Black;
16.             ColorDifuso = Color.WhiteSmoke;
17.             ColorEspecular = Color.WhiteSmoke;
18.             Ks = 10.0F;
19.             N = 20.0F;
20.             HabilitarTextura = false;
21.             direccionales = effect.Parameters["direccionales"];
22.         }
23.
24.         #region IEspecular Members
25.
26.         /// <summary>
27.         /// Propiedad que obtiene o establece el coeficiente especular.
28.         /// </summary>
29.         public float Ks
30.         {
31.             get { return ks; }
32.             set
33.             {
34.                 ks = value;
35.                 base.effect.Parameters["ks"].SetValue(ks);
36.             }
37.         }// fin de la propiedad Ks
38.
39.         /// <summary>
40.         /// Propiedad que obtiene o establece la potencia especular.
41.         /// </summary>
42.         public float N
43.         {
44.             get { return n; }
45.             set
46.             {
47.                 n = value;
48.                 base.effect.Parameters["n"].SetValue(n);
49.             }
50.         }// fin de la propiedad N
51.
52.         /// <summary>
53.         /// Propiedad  que obtiene o establece el color especular.
54.         /// </summary>
55.         public Color ColorEspecular
56.         {
57.             get { return colorEspecular; }
58.             set
59.             {
60.                 colorEspecular = value;
61.                 base.effect.Parameters["colorMaterialEspecular"].SetValue(
```

```
62.                     colorEspecular.ToVector4());
63.             }
64.         }// fin de la propiedad ColorEspecular
65.
66.         /// <summary>
67.         /// Propiedad que habilita o inhabilita el modelo de iluminación especular.
68.         /// </summary>
69.         public bool HabilitarEspecular
70.         {
71.             get { return habilitarEspecular; }
72.             set
73.             {
74.                 habilitarEspecular = value;
75.                 effect.Parameters["habiEspec"].SetValue(habilitarEspecular);
76.             }
77.         }// fin de la propiedad HabilitarEspecular
78.
79.         /// <summary>
80.         /// Propiedad que obtiene o establece la posicion de la cámara.
81.         /// </summary>
82.         public Vector3 PosicionCamara
83.         {
84.             get { return posicionCamara; }
85.             set
86.             {
87.                 posicionCamara = value;
88.                 effect.Parameters["posicionCamara"].SetValue(posicionCamara);
89.             }
90.         }// fin de la propiedad PosicionCamara
91.
92.         #endregion
93.
94.         #region IDifuso Members
95.
96.         /// <summary>
97.         /// Propiedad que obtiene o establece el color difuso.
98.         /// </summary>
99.         public Color ColorDifuso
100.        {
101.            get { return colorDifuso; }
102.            set
103.            {
104.                colorDifuso = value;
105.                base.effect.Parameters["colorMaterialDifuso"].SetValue(
106.                    colorDifuso.ToVector4());
107.            }
108.        }// fin de la propiedad ColorDifuso
109.
110.        #endregion
111.
112.        /// <summary>
113.        /// Propiedad que obtiene o establece el número de luces.
114.        /// </summary>
115.        public Int16 NumeroLuces
116.        {
117.            get { return numeroLuces; }
118.            set
119.            {
120.                numeroLuces = value;
121.                base.effect.Parameters["numLuces"].SetValue(numeroLuces);
122.            }
123.        }// fin de la propiedad NumeroLuces
124.
125.        /// <summary>
126.        /// Propiedad que obtiene el parámetro de la estructura
127.        /// que corresponde a las luces direccionales.
128.        /// </summary>
129.        public EffectParameter Direccionales { get { return direccionales; } }
130.
131.        public override void DrawModelMeshPart(ModelMeshPart modelMeshPart,
132.            GraphicsDevice graphicsDevice)
```

```
133.        {
134.            effect.Begin();
135.            #region Effect.Begin
136.
137.            effect.CurrentTechnique.Passes["Ambiental"].Begin();
138.            graphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
139.                modelMeshPart.BaseVertex, 0,
140.                modelMeshPart.NumVertices, modelMeshPart.StartIndex,
141.                modelMeshPart.PrimitiveCount);
142.            effect.CurrentTechnique.Passes["Ambiental"].End();
143.
144.            // se habilita el blending
145.            graphicsDevice.RenderState.AlphaBlendEnable = true;
146.            graphicsDevice.RenderState.BlendFunction =
147.                BlendFunction.Add;
148.            graphicsDevice.RenderState.DestinationBlend = Blend.One;
149.            graphicsDevice.RenderState.SourceBlend = Blend.One;
150.
151.            effect.CurrentTechnique.Passes["MultiLuces"].Begin();
152.
153.            effect.CommitChanges();
154.
155.            graphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList,
156.                modelMeshPart.BaseVertex, 0,
157.                modelMeshPart.NumVertices, modelMeshPart.StartIndex,
158.                modelMeshPart.PrimitiveCount);
159.            effect.CurrentTechnique.Passes["MultiLuces"].End();
160.
161.             // valor por default de XNA
162.            graphicsDevice.RenderState.AlphaBlendEnable = false;
163.            #endregion
164.            effect.End();
165.        }// fin del método DrawModelMeshPart
166.    }// fin de la clase FxMultiluces
```

Para implementar esta última clase, se debe crear el objeto de la clase **Multiluces** y las luces de tipo direccional. En la clase **Game1** de la solución, escriba las siguientes variables de instancia que indican dichos objetos.

```
FxMultiluces fxMultiluces;
LuzDireccional[] lucesDireccionales;
```

Y en el método **LoadContent**, de la clase **Game1**, se carga el shader en el constructor de la clase **FxMultiluces**, línea 1, Código 8-13. Luego se inicializan las propiedades del objeto **fxMultilices**, líneas 2 – 10.

La creación de las luces direccionales se hace dentro del mismo método **LoadContent**, pues su constructor necesita la dirección de memoria de cada elemento que indica una luz del shader, líneas 12 – 24. Luego se inicializan las propiedades de cada fuente de iluminación, líneas 26 – 37.

Código 8-13

```
1.      fxMultiluces = new FxMultiluces(Content.Load<Effect>(@"Shaders\MulDirec"));
2.      fxMultiluces.NumeroLuces = 6;
3.      fxMultiluces.Textura2D = textura;
4.      fxMultiluces.HabilitarEspecular = true;
5.      fxMultiluces.HabilitarTextura = true;
6.      fxMultiluces.ColorDifuso = new Color(3, 3, 3);
7.      fxMultiluces.ColorAmbiental = Color.Black;
8.      fxMultiluces.Ks = 10;
9.      fxMultiluces.N = 20;
10.     fxMultiluces.PosicionCamara = posicion;
11.
12.     lucesDireccionales = new LuzDireccional[fxMultiluces.NumeroLuces];
13.     lucesDireccionales[0] = new LuzDireccional(
14.     fxMultiluces.Direccionales.Elements[0], new Vector3(0, -1, 0));
15.     lucesDireccionales[1] = new LuzDireccional(
16.     fxMultiluces.Direccionales.Elements[1], new Vector3(0, 1, 0));
```

```
17.     lucesDireccionales[2] = new LuzDireccional(
18.     fxMultiluces.Direccionales.Elements[2], new Vector3(0, 0, 1));
19.     lucesDireccionales[3] = new LuzDireccional(
20.     fxMultiluces.Direccionales.Elements[3], new Vector3(0, 0, -1));
21.     lucesDireccionales[4] = new LuzDireccional(
22.     fxMultiluces.Direccionales.Elements[4], new Vector3(1, 0, 0));
23.     lucesDireccionales[5] = new LuzDireccional(
24.     fxMultiluces.Direccionales.Elements[5], new Vector3(-1, 0, 0));
25.
26.     lucesDireccionales[0].Encender = true;
27.     lucesDireccionales[0].Color = Color.Green;
28.     lucesDireccionales[1].Encender = true;
29.     lucesDireccionales[1].Color = Color.LawnGreen;
30.     lucesDireccionales[2].Encender = true;
31.     lucesDireccionales[2].Color = Color.Indigo;
32.     lucesDireccionales[3].Encender = true;
33.     lucesDireccionales[3].Color = Color.Brown;
34.     lucesDireccionales[4].Encender = true;
35.     lucesDireccionales[4].Color = Color.Chocolate;
36.     lucesDireccionales[5].Encender = true;
37.     lucesDireccionales[5].Color = Color.LemonChiffon;
```

Para concluir con este ejemplo, en el método **Draw** de la clase **Game1**, Código 8-14, se establecen las matrices de mundo, línea 17; vista, línea 10; y proyección, línea 11; del objeto **fxMultiluces**. Y se llama al método **DrawModelMeshPart** de la instancia **fxMultiluces**, línea 32.

Ejecute el programa, y corrija cualquier error de compilación que surgiese y vuelva a intentarlo, verá algo parecido a la ilustración 8 – 7.

Código 8-14

```
1.      protected override void Draw(GameTime gameTime)
2.      {
3.          GraphicsDevice.Clear(Color.White);
4.
5.          fxAmbiental.View = view;
6.          fxAmbiental.Projection = projection;
7.
8.          fxDireccional.View = fxAmbiental.View;
9.          fxDireccional.Projection = fxAmbiental.Projection;
10.         fxMultiluces.View = fxAmbiental.View;
11.         fxMultiluces.Projection = fxAmbiental.Projection;
12.
13.         foreach (ModelMesh mesh in elefante.Meshes)
14.         {
15.             fxAmbiental.World = transformaciones[mesh.ParentBone.Index];
16.             fxDireccional.World = fxAmbiental.World;
17.             fxMultiluces.World = fxAmbiental.World;
18.
19.
20.             GraphicsDevice.Indices = mesh.IndexBuffer;
21.
22.             foreach (ModelMeshPart meshPart in mesh.MeshParts)
23.             {
24.                 GraphicsDevice.Vertices[0].SetSource(mesh.VertexBuffer,
25.                         meshPart.StreamOffset, meshPart.VertexStride);
26.                 GraphicsDevice.VertexDeclaration = meshPart.VertexDeclaration;
27.
28.                 fxAmbiental.DrawModelMeshPart(meshPart, GraphicsDevice);
29.                 fxDireccional.World *= Matrix.CreateTranslation(-100, 0, -200);
30.                 fxDireccional.DrawModelMeshPart(meshPart, GraphicsDevice);
31.                 fxMultiluces.World *= Matrix.CreateTranslation(100, 0, 200);
32.                 fxMultiluces.DrawModelMeshPart(meshPart, GraphicsDevice);
33.             }
34.         }
35.
36.         base.Draw(gameTime);
37.     }
```

Este último ejemplo demuestra que tan complicado se puede hacer la comunicación entre los datos de XNA hacia el shader, además representa un gran número de líneas de código en comparación con el primer ejemplo que se vió en este capítulo, pero que sin duda es mejor.

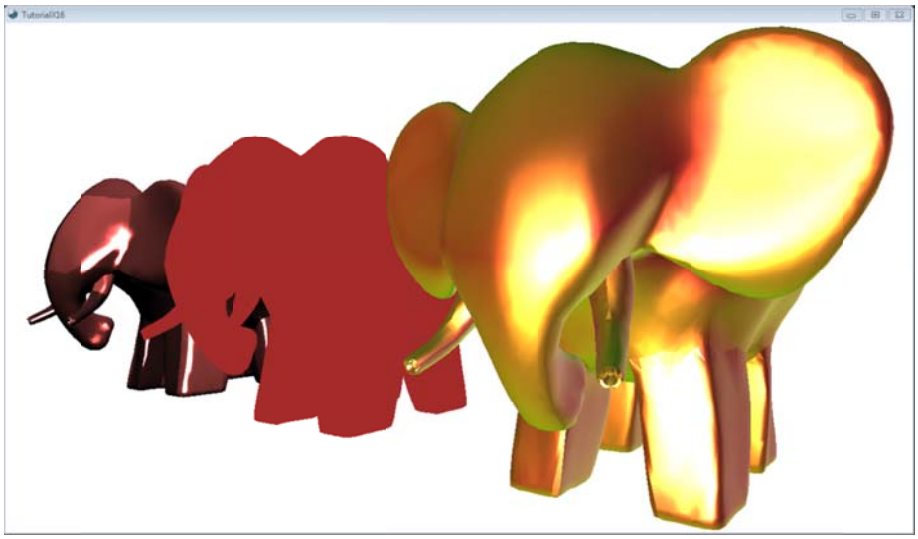

Ilustración 8-7 Integración del shader Multiluces en XNA

Como ejercicio para el lector se deja crear la clase que comunique los datos desde XNA hacia el shader MultiLuces que se vio en el apartado Iluminación, en donde se tienen diferentes fuentes de iluminación.

9 Colisión

Una colisión es una configuración en la que dos objetos ocupan parte de una misma porción del espacio al mismo tiempo.[39]

La anterior definición sobre lo que es colisión en computación gráfica rompe la propiedad de la impenetrabilidad de la física, y es este mismo que se toma en cuenta en los siguientes ejemplos. Los cuales muestran la impenetrabilidad con diferentes tipos de envolventes de colisión.

Un volumen envolvente es una primitiva simple que encierra a una forma más compleja y al que se le puede hacer una prueba de intersección más rápido en términos computacionales.[40]

Es decir, toda geometría compleja puede ser sustituida por una geometría más sencilla que envuelva a la primera, permitiendo cálculos más sencillos a un costo de la impenetrabilidad. Existen varias envolventes de colisión como son:

- Esfera envolvente (Bounding Sphere, BS)
- Caja envolvente alineada con los ejes (Axis Aligned Bounding Box, AABB)
- Caja envolvente orientada (Oriented Boundig Box, OBB)
- Politopo de Orientación Discreta (Discrete Orientation Polytope, k-DOP)

En la Ilustración 9-1 se muestra la figura de Hebe envuelta en una esfera y en una caja. La primera cubre un mayor volumen que la estatua, lo que no permitiría a otra acercarse lo suficiente. La segunda envolvente esta más pegada a Hebe, lo que permite a otra acercarse lo suficiente como para no atravesarla.

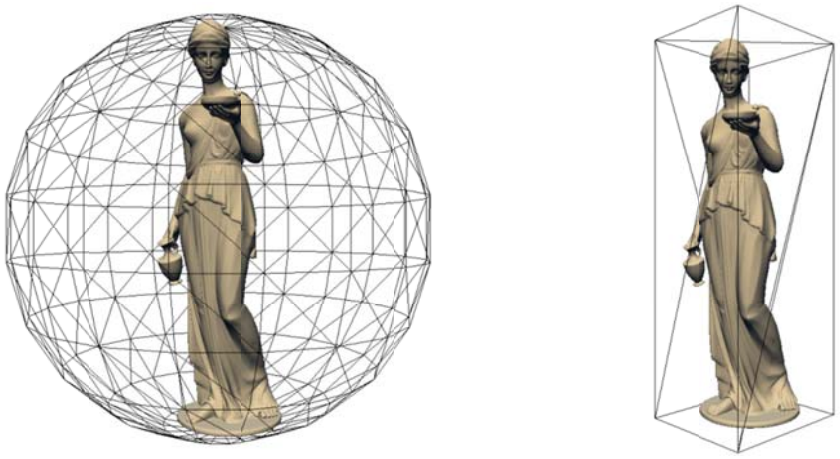

Ilustración 9-1 Envolvente de colisión

La selección de qué tipo de envolvente es la más adecuada, es aquella que sea lo más cercana a las dimensiones de la figura, tomando en cuenta el costo computacional, pues las pruebas de intersección que se hacen con cada envolvente se hace cada vez que se actualiza el dibujo o se atiende una interrupción.

La manipulación de la colisión se puede ver en tres pasos:

- Detección de colisión
- Determinación de colisión
- Repuesta a la colisión

La **detección de colisión** es un valor lógico que indica si dos o más objetos han colisionado. Para conocer si existe o no se determina la colisión con los cálculos de las intersecciones. Y por último se toman acciones en respuesta a la colisión.

[39] "Detección de Colisiones mediante Recubrimientos Simpliciales". D. Juan José Jimenéz Delgado. 2006 Universidad de Granada. p. 11.
[40] Ídem p. 20

Este texto no mostrará todas las envolventes de colisión, ni tampoco todas las combinaciones que puedan existir entre cada una de ellas para determinar una, esto sale del alcance del texto. XNA ofrece algunos métodos que devuelven un valor lógico cuando sucede una colisión. Sin embargo, no ofrece tipos de acciones que se deben llevar a cabo una vez detectada, esto depende del problema.

Es por eso, que en los siguientes cuatro ejemplos se muestra el uso de dichos métodos de detección, de determinación y respuesta; estos dos últimos son implementaciones que el autor determinó, pero no indican que sean las mejores.

En cada ejemplo se manejan modelos tridimensionales con un solo **ModelMesh**; quedaría como ejercicio para el lector, crear el programa que pueda manejar más de un **ModelMesh** para las colisiones.

9.1 Bounding Sphere vs. Bounding Sphere

La envolvente de colisión **Bounding Sphere** (BS) es una esfera que encierra a la geometría principal. Esta envolvente ofrece dos propiedades para los cálculos: radio y centro. XNA ya ofrece métodos que calculan la envolvente a partir de los vectores de posición de los vértices y a partir de otras envolventes. Por otra parte ofrece métodos que indican el valor lógico si encuentra una colisión con otra envolvente. A partir de los ejemplos se hará mención de cada uno de estos métodos.

Para este primer ejemplo, se hace mover una figura por medio del teclado o el gamepad, para hacerlo colisionar contra otra figura. Cada una estará envuelta en una esfera.

Una vez determinada la colisión, se realizan cálculos para hacer retroceder la envolvente y aparentar un choque, es decir, solo se detendrá en la dirección del movimiento cuando exista colisión.

Para mostrar en tiempo real los cambios numéricos que sufren las propiedades de las envolventes de colisión se dibujará con un SpriteFont en pantalla, las propiedades de cada envolvente.

Comience por crear un nuevo proyecto en Visual Studio y seleccione la plantilla de **Windows Game (3.1)**. Agregue una nueva clase llamada **Texto**, Código 9-1, La clase **Texto** ayudará a dibujar los valores numéricos de cada envolvente. Las tres variables de instancia, líneas 9 – 11, sirven para dibujar el **string** en pantalla, cada una de ellas se explicó en el Texto. La parte importante de la clase es el método **Draw**, líneas 20 – 26, el cual recibe un **string** llamado **mensaje** y el color para la fuente, línea 20. Después de mandar a dibujar el texto, se habilita el **DephBuffer**, línea 25, porque en la llamada al método **DrawString**, línea 23, se inhabilita el búfer.

Código 9-1

```
1.      using System;
2.      using Microsoft.Xna.Framework.Graphics;
3.      using Microsoft.Xna.Framework;
4.
5.      namespace TutorialX12
6.      {
7.          public class Texto
8.          {
9.              GraphicsDevice graphicsDevice;
10.             SpriteBatch spriteBatch;
11.             SpriteFont spriteFont;
12.
13.             public Texto(SpriteFont spriteFont, GraphicsDevice graphicsDevice)
14.             {
15.                 this.graphicsDevice = graphicsDevice;
16.                 spriteBatch = new SpriteBatch(graphicsDevice);
17.                 this.spriteFont = spriteFont;
18.             }// fin del constructor
19.
20.             public void Draw(String mensaje, Color color)
21.             {
22.                 spriteBatch.Begin();
23.                 spriteBatch.DrawString(spriteFont, mensaje, Vector2.Zero, color);
24.                 spriteBatch.End();
```

```
25.                 graphicsDevice.RenderState.DepthBufferEnable = true;
26.             }// fin del método Draw
27.         }// fin de la clase Texto
28.     }// fin del namespace
```

La Ilustración 9-2 es un diagrama UML que muestra dos clases: **ModeloEstatico** y **ModeloDinamico**, este último hereda del primero. Para los ejemplos siguientes se seguirá utilizando este diagrama de clases. La primera clase obtiene la envolvente de colisión de esfera, pinta el modelo tridimensional del archivo **fbx** o **x** y regresa el nombre de la figura, el centro y el radio de la esfera.

La clase **ModeloEstatico**, Código 9-2, tiene cuatro variables de instancia: **modelo**, línea 9, que representa el modelo obtenido a partir de un archivo **fbx** o **x**; el arreglo transformaciones, línea 10, guarda todas las matrices de transformación del modelo; **boundingSphere**, línea 11, representa la envolvente de colisión esfera; y nombre es un **string** que guarda el nombre de la figura 3D.

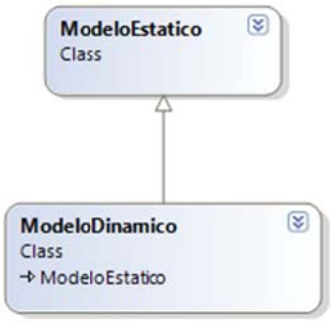

Ilustración 9-2 Diagrama de clase. Modelos estático y dinámicos.

El constructor, líneas 14 – 26, toma como parámetro un objeto **Model** e inicializa el arreglo de transformaciones, líneas 17 – 18. Cada objeto **ModelMesh** del **Model** contiene su propia envolvente de esfera; además se sabe que el modelo del archivo contiene un solo **ModelMesh**, así que solo se le asigna a la variable **boundingSphere**, línea 19. La esfera de colisión debe ser transformada por medio de una de matriz del modelo que indica la posición correcta y tamaño del radio de la envolvente. El objeto **boundingSphere** tiene un método llamado **Transform** que sirve para trasladar o escalar el BS, líneas 21 – 23. El método **Transform** tiene dos sobrecargas, la primera toma una matriz como parámetro y devuelve un BS; la segunda sobrecarga toma dos parámetros de entrada, el primero es la matriz de transformación como referencia, y la segunda es el BS como salida.

```
public void Transform (ref Matrix matrix, out BoundingSphere result)
```

Para extraer el nombre de la figura, se asigna la propiedad **Name**, del primer **ModelMesh** del modelo a la variable **nombre**, línea 25.

La propiedad **EnvolventeEsfera**, línea 31, obtiene la **BoundingSphere** para hacer pruebas de colisión. El método **Draw**, líneas 33 – 49, dibuja el **Model** extraído del archivo del modelo; este mismo método se vio en el Cómo cargar modelos 3D desde un archivo X y FBX.

Por último, el método **ToString**, líneas 55 – 61, devuelve un string con el nombre del primer **ModelMesh** del modelo 3D, el centro y radio de la esfera de colisión.

Código 9-2

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Graphics;
4.
5.      namespace TutorialX12
6.      {
7.          public class ModeloEstatico
8.          {
9.              private Model modelo;
```

```
10.             private Matrix[] transformaciones;
11.             protected BoundingSphere boundingSphere;
12.             string nombre;
13.
14.             public ModeloEstatico(Model modelo)
15.             {
16.                 this.modelo = modelo;
17.                 transformaciones = new Matrix[modelo.Bones.Count];
18.                 modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
19.                 boundingSphere = modelo.Meshes[0].BoundingSphere;
20.                 // traslación del centro de la envolvente
21.                 boundingSphere.Transform(
22.                     ref transformaciones[modelo.Meshes[0].ParentBone.Index],
23.                     out boundingSphere);
24.
25.                 nombre = modelo.Meshes[0].Name;
26.             }// fin del constructor
27.
28.             /// <summary>
29.             /// Propiedad que obtiene la envolvente esfera.
30.             /// </summary>
31.             public BoundingSphere EnvolventeEsfera { get { return boundingSphere; } }
32.
33.             public virtual void Draw(ref Matrix world, ref Matrix view,
34.                 ref Matrix projection)
35.             {
36.                 foreach (ModelMesh mesh in modelo.Meshes)
37.                 {
38.                     foreach (BasicEffect basicEffect in mesh.Effects)
39.                     {
40.                         basicEffect.World = transformaciones[mesh.ParentBone.Index]
41.                             * world;
42.                         basicEffect.View = view;
43.                         basicEffect.Projection = projection;
44.                         basicEffect.EnableDefaultLighting();
45.                         basicEffect.PreferPerPixelLighting = true;
46.                     }// fin del foreach
47.                     mesh.Draw();
48.                 }// fin del foreach
49.             }// fin del método Draw
50.
51.             /// <summary>
52.             /// Propiedad que devuelve información de la envolvente esfera.
53.             /// </summary>
54.             /// <returns></returns>
55.             public override string ToString()
56.             {
57.                 return string.Format("{0}\nBoundingSphere.Center {1}\nRadio {2}",
58.                     nombre,
59.                     boundingSphere.Center.ToString(),
60.                     boundingSphere.Radius);
61.             }// fin del método ToString
62.         }// fin de la clase ModeloEstatico
63.     }// fin del namespace
```

La clase **ModeloDinamico**, **¡Error! No se encuentra el origen de la referencia.**, traslada la geometría con ayuda de las teclas de navegación o el gamepad. En caso de encontrar una colisión en contra de una envolvente de esfera, el movimiento del objeto se detendrá en esa dirección de desplazamiento, evitando que se traslapen las figuras.

En esta clase se utilizan dos matrices de traslación, líneas 11 – 12, la primera sirve para trasladar todo el modelo tridimensional y la segunda matriz es para trasladar el centro de la esfera. La distancia máxima que puede desplazarse la figura será dada por la constante **step**, línea 10.

La Tabla 9-1 enlista las teclas, ejes del stick derecho, y los triggers que se usan para desplazar el objeto tridimensional.

Tabla 9-1

Tecla	GamePad	Movimiento
Up	ThumbSticks.Right.Y	Desplazamiento negativo sobre el eje Z.
Down		Desplazamiento positivo sobre el eje Z.
Right	ThumbSticks.Right.X	Desplazamiento positivo sobre el eje X.
Left		Desplazamiento negativo sobre el eje X.
PageUp	Triggers.Right	Desplazamiento positivo sobre el eje Y.
PageDown	Triggers.Left	Desplazamiento negativo sobre el eje Y.

El cuerpo del constructor no contiene ninguna inicialización, líneas 14 – 17, tan solo se pasa el parámetro modelo al constructor de la clase base.

La actualización de la información del desplazamiento se genera en el método **Update**, líneas 19 – 81. La primera parte corresponde al teclado, líneas 21 – 52, en donde, en cada cuerpo del condicional **if** se crea la matriz de traslación para cambiar el centro de la envolvente de esfera. Enseguida se llama al método **DetenerMovimiento** que actualiza la posición de la figura y el de la envolvente. La segunda parte corresponde al gamepad, líneas 54 – 80; en la creación de cada matriz de traslación se multiplica la constante **step** por el cambio en el eje del stick o por el cambio en los triggers, luego se llama al método **DetenerMovimiento**.

El método **DetenerMovimiento**, líneas 87 – 117, determina si el movimiento de la envolvente del objeto **ModeloDinamico** continua o se debe detener. El método recibe una envolvente de esfera de colisión, para buscar la colisión con su envolvente. En la línea 90, se traslada la envolvente del objeto **ModeloDinamico** con el método **Transform**.

La instancia del **BoundingSphere** contiene el método **Intersects**, línea 92, que comprueba la colisión entre la envolvente actual y alguna en específico, en este caso la del objeto estático. El método **Intersects** está sobrecargado para las diferentes envolventes que ofrece XNA.[41]

```
public bool Intersects (BoundingSphere sphere)
```

También existe el método **Contains**, que comprueba si el **BoundingSphere** actual contiene a la envolvente dada, y al igual que **Intersects**, ésta también se encuetra sobrecargada.

```
public ContainmentType Contains (BoundingSphere sphere)
```

Una vez que se ha detectado la intersección entre las envolventes, se recorre el centro del **BundingSphere** del modelo dinámico, líneas 96 -98, a una posición anterior a la colisión; esto se logra

[41] Las diferentes sobrecargas se pueden revisarse en la siguente página de Internet: http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=ES-ES&k=k(MICROSOFT.XNA.FRAMEWORK.BOUNDINGSPHERE.INTERSECTS)&rd=true

multiplicando la matriz de traslación de la envolvente actual por menos uno y después multiplandola por el centro de la esfera actual.

En la Ilustración 9-3 se muestran dos esferas **A** y **B**, la primera representa el modelo dinámico y la segunda el estático. El vector $\bar{u}$ representa el vector unitario paralelo a la resta entre los centros $\bar{c}_B$ y $\bar{c}_A$.

$$\bar{u} = \frac{\bar{c}_B - \bar{c}_A}{|\bar{c}_B - \bar{c}_A|} \qquad (1)$$

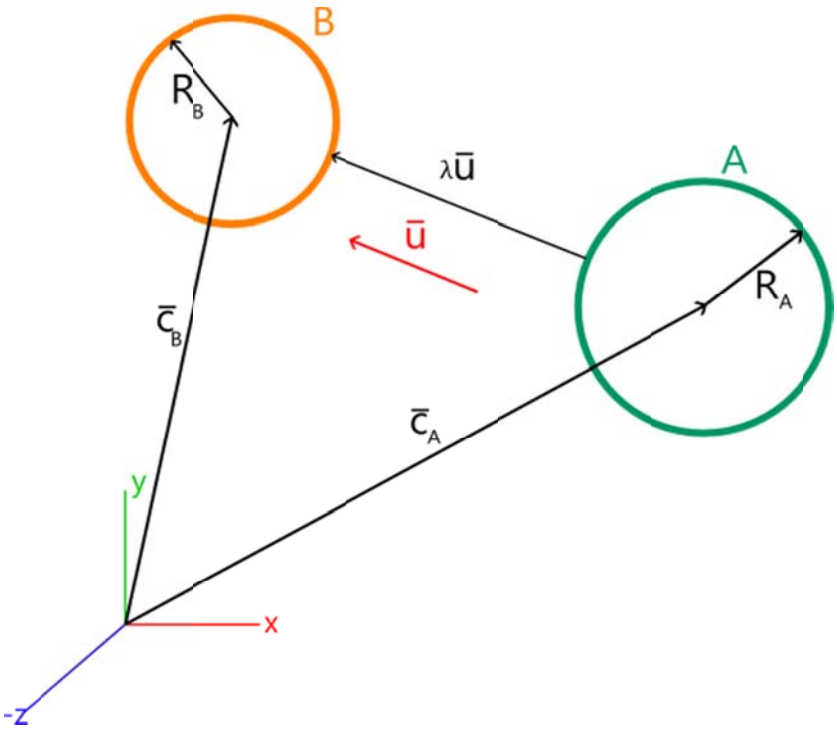

Ilustración 9-3 Distancia entre dos esferas

El valor de λ es: la diferencia entre la distancia de los vectores de posición de los centros de las esferas y la suma de los radios de éstas.

$$\lambda = |\bar{C}_B - \bar{C}_A| - (R_B + R_A) \qquad (2)$$

El vector de traslación para la esfera **A** estará dada por:

$$\lambda\bar{u} = \left(|\bar{C}_B - \bar{C}_A| - (R_B + R_A)\right)\frac{\bar{c}_B - \bar{c}_A}{|\bar{c}_B - \bar{c}_A|} \qquad (3)$$

$$\lambda\bar{u} = (\bar{c}_B - \bar{c}_A)\left(1 - \frac{R_B + R_A}{|\bar{c}_B - \bar{c}_A|}\right) \qquad (4)$$

En las líneas 101 – 105, se calcula el valor de la distancia que falta para que las esferas estén tan cerca como sea posible, aplicando la ecuación (4). Tómese en cuenta que la distancia entre dos vectores cualesquiera, es el módulo entre la diferencia entre ellos. Por eso se utiliza el método estático **Distance**, de la escructura **Vector3**.

Se vuelve a crear la matriz de traslación para el centro de la esfera del modelo dinámico, línea 106, y se cambia la posición de la esfera con el método **Transform**, líneas 109 – 110.

Al final del condicional **if** se multiplica la matriz de transformación de la esfera por la del modelo geométrico, para volvérselo a asignar a este último.

En el método **Draw**, líneas 126 – 131, se multiplica la matriz de traslación de la geometría por la matriz de mundo, línea 129, por si hay alguna modificación desde el exterior sobre la figura. Luego se hace llamar el

método **Draw** de la clase base con la nueva matriz **mtxTraslacion**, como su primer parámetro; los demás pasan sin cambio desde la clase hija.

Código 9-3

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Input;
4.      using Microsoft.Xna.Framework.Graphics;
5.
6.      namespace TutorialX12
7.      {
8.          public class ModeloDinamico : ModeloEstatico
9.          {
10.             private const float step = 1.0F;
11.             private Matrix mtxTraslacion = Matrix.Identity;
12.             private Matrix mtxTraslacionCenter;
13.
14.             public ModeloDinamico(Model modelo)
15.                 : base(modelo)
16.             {
17.             }// fin del constructor
18.
19.             public void Update(BoundingSphere boudingSphereB)
20.             {
21.                 #region Keyboard
22.                 if (Keyboard.GetState().IsKeyDown(Keys.Up))
23.                 {
24.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, 0, -step);
25.                     DetenerMovimiento(boudingSphereB);
26.                 }
27.                 if (Keyboard.GetState().IsKeyDown(Keys.Down))
28.                 {
29.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, 0, step);
30.                     DetenerMovimiento(boudingSphereB);
31.                 }
32.                 if (Keyboard.GetState().IsKeyDown(Keys.Right))
33.                 {
34.                     mtxTraslacionCenter = Matrix.CreateTranslation(step, 0, 0);
35.                     DetenerMovimiento(boudingSphereB);
36.                 }
37.                 if (Keyboard.GetState().IsKeyDown(Keys.Left))
38.                 {
39.                     mtxTraslacionCenter = Matrix.CreateTranslation(-step, 0, 0);
40.                     DetenerMovimiento(boudingSphereB);
41.                 }
42.                 if (Keyboard.GetState().IsKeyDown(Keys.PageUp))
43.                 {
44.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, step, 0);
45.                     DetenerMovimiento(boudingSphereB);
46.                 }
47.                 if (Keyboard.GetState().IsKeyDown(Keys.PageDown))
48.                 {
49.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, -step, 0);
50.                     DetenerMovimiento(boudingSphereB);
51.                 }
52.                 #endregion
53.
54.                 #region Gamepad
55.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y != 0.0F)
56.                 {
57.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, 0, -step *
58.                         GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y);
59.                     DetenerMovimiento(boudingSphereB);
60.                 }
61.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X != 0.0F)
62.                 {
63.                     mtxTraslacionCenter = Matrix.CreateTranslation(step *
64.                         GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X
65.                         , 0, 0);
```

```
66.                     DetenerMovimiento(boudingSphereB);
67.                 }
68.                 if (GamePad.GetState(PlayerIndex.One).Triggers.Right != 0.0F)
69.                 {
70.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, step *
71.                     GamePad.GetState(PlayerIndex.One).Triggers.Right, 0);
72.                     DetenerMovimiento(boudingSphereB);
73.                 }
74.                 if (GamePad.GetState(PlayerIndex.One).Triggers.Left != 0.0F)
75.                 {
76.                     mtxTraslacionCenter = Matrix.CreateTranslation(0, -step *
77.                     GamePad.GetState(PlayerIndex.One).Triggers.Left, 0);
78.                     DetenerMovimiento(boudingSphereB);
79.                 }
80.                 #endregion
81.             }// fin del método Update
82.
83.             /// <summary>
84.             /// Método que detiene el movimiento de la BS, si ésta intersecta con otra.
85.             /// </summary>
86.             /// <param name="boudingSphereB"></param>
87.             private void DetenerMovimiento(BoundingSphere boundingSphereB)
88.             {
89.                 // se traslada el centro de la envolvente del modelo dinámico
90.                 boundingSphere.Transform(ref mtxTraslacionCenter, out boundingSphere);
91.
92.                 if (boundingSphere.Intersects(boundingSphereB))
93.                 {
94.                     // se invierte el vector de traslación para regresar la posición
95.                     // del centro de la esfera, antes de intersecar con boundigSphereB
96.                     mtxTraslacionCenter.Translation = -mtxTraslacionCenter.Translation;
97.                     boundingSphere.Transform(ref mtxTraslacionCenter,
98.                         out boundingSphere);
99.
100.                    // se cálcula la distancia a recorrer la envolvente de esfera
101.                    Vector3 distancia = (boundingSphereB.Center –
102.                        boundingSphere.Center) *
103.                        (1 - (boundingSphere.Radius + boundingSphereB.Radius) /
104.                        Vector3.Distance(boundingSphereB.Center,
105.                        boundingSphere.Center));
106.                    mtxTraslacionCenter = Matrix.CreateTranslation(distancia);
107.
108.                    // se actualiza el centro de la esfera
109.                    boundingSphere.Transform(ref mtxTraslacionCenter,
110.                        out boundingSphere);
111.                }// fin del if
112.
113.                // se multiplica la matriz de traslación del centro
114.                // de la envolvente de la esfera por la matriz de
115.                // traslación de la figura
116.                mtxTraslacion *= mtxTraslacionCenter;
117.
118.            }// fin del método DetenerMovimiento
119.
120.            /// <summary>
121.            /// Método que dibuja la geometría.
122.            /// </summary>
123.            /// <param name="world">Matriz de mundo.</param>
124.            /// <param name="view">Matriz de vista.</param>
125.            /// <param name="projection">Matriz de proyección.</param>
126.            public override void Draw(ref Matrix world, ref Matrix view,
127.                ref Matrix projection)
128.            {
129.                mtxTraslacion *= world;
130.                base.Draw(ref mtxTraslacion, ref view, ref projection);
131.            }// fin del método
132.        }
133.    }
```

Para la clase **Game1**, Código 9-4, se necesitan dos modelos que contengan un **ModelMesh** cada uno, y un **SpriteFont**. En este ejemplo se utilizó una esfera y una tetera para representar a los objetos

ModeloEstatico y **ModeloDinamico**, respectivamente, véase Ilustración 9-4. Cada una de estas figuras se representará por los objetos **modeloEstatico**, línea 20, y **modeloDinamico**, línea 21. La variable **texto**, línea 23, es para mostrar la información de las envolventes de esfera de cada objeto.

En el método **Initialize**, líneas 33 – 46, se inicializan las matrices de mundo, vista y proyección. En el método **LoadContent**, líneas 48 – 58, se crean las instancias de **ModeloEstatico**, **ModeloDinamico** y **Texto**.

Para actualizar el estado de **modeloDinamico** se hace llamar su método **Update** en el mismo de la clase **Game1**, línea 71. El parámetro que toma es la envolvente del **modeloEstatico**.

Ilustración 9-4 BS vs BS

Por último, se presenta el método **Draw**, líneas 76 – 86, que hace llamar a los métodos de dibujo de cada objeto, **modeloEstatico**, **modeloDinamico** y **texto**. El método **Draw** de **texto**, líneas 82, toma como primer parámetro el **string** devuelto por el método **ToString** de **modeloEstatico**, y en la línea 83 toma el **string** de **modeloDinamico**; para diferenciar a cada uno se le da diferentes colores al método **Draw** de texto.

Ejecute el programa y pruebe con el gamepad o el teclado el mover el **modeloDinamico**, verá que la información de ésta cambia y al momento de acercarse tanto a la envolvente de **modeloEstatico**, se detendrá en seco. Esta es una manera de responder al momento de saber que existe una colisión, sin embargo, no es la única. Por ejemplo, se pudieron hacer este tipo de acciones al conocer que existe dicha situación:

- Regresar al punto de inicio el modeloDinamico.
- Desaparecer el modeloDinamico.
- Rebotar el modeloDinamico.
- El modeloDinamico rodea al modeloEstatico.
- Etcétera.

Cada una de esas acciones le corresponde un modelo matemático diferente, o bien se puede crear una física de cuerpos rígidos completa. Lo que por desgracia está fuera del alcance de este texto.

Código 9-4

```
1.      using System;
2.      using System.Collections.Generic;
3.      using System.Linq;
4.      using Microsoft.Xna.Framework;
5.      using Microsoft.Xna.Framework.Audio;
6.      using Microsoft.Xna.Framework.Content;
7.      using Microsoft.Xna.Framework.GamerServices;
8.      using Microsoft.Xna.Framework.Graphics;
9.      using Microsoft.Xna.Framework.Input;
10.     using Microsoft.Xna.Framework.Media;
11.     using Microsoft.Xna.Framework.Net;
12.     using Microsoft.Xna.Framework.Storage;
13.
14.     namespace TutorialX12
15.     {
16.         public class Game1 : Microsoft.Xna.Framework.Game
17.         {
18.             GraphicsDeviceManager graphics;
19.             SpriteBatch spriteBatch;
20.             ModeloEstatico modeloEstatico;
21.             ModeloDinamico modeloDinamico;
22.             Matrix world, view, projection;
23.             Texto texto;
24.
25.             public Game1()
26.             {
27.                 graphics = new GraphicsDeviceManager(this);
28.                 Content.RootDirectory = "Content";
29.                 graphics.PreferredBackBufferHeight = 720;
30.                 graphics.PreferredBackBufferWidth = 1280;
31.             }
32.
33.             protected override void Initialize()
34.             {
35.                 world = Matrix.Identity;
36.                 view = Matrix.CreateLookAt(
37.                     new Vector3(0, 100, 350),
38.                     Vector3.Zero,
39.                     Vector3.Up);
40.                 projection = Matrix.CreatePerspectiveFieldOfView(
41.                     MathHelper.PiOver4,
42.                     GraphicsDevice.Viewport.AspectRatio,
43.                     1.0F, 10000.0F);
44.
45.                 base.Initialize();
46.             }
47.
48.             protected override void LoadContent()
49.             {
50.                 spriteBatch = new SpriteBatch(GraphicsDevice);
51.
52.                 modeloEstatico = new ModeloEstatico(
53.                     Content.Load<Model>("SphereA"));
54.                 modeloDinamico = new ModeloDinamico(
55.                     Content.Load<Model>("SphereB"));
56.
57.                 texto = new Texto(Content.Load<SpriteFont>("Arial"), GraphicsDevice);
58.             }
59.
60.             protected override void UnloadContent()
61.             {
62.                 // TODO: Unload any non ContentManager content here
63.             }
64.
65.             protected override void Update(GameTime gameTime)
66.             {
67.                 if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
68.                     ButtonState.Pressed)
69.                     this.Exit();
```

```
70.
71.                 modeloDinamico.Update(modeloEstatico.EnvolventeEsfera);
72.
73.                 base.Update(gameTime);
74.             }
75.
76.             protected override void Draw(GameTime gameTime)
77.             {
78.                 GraphicsDevice.Clear(Color.White);
79.
80.                 modeloEstatico.Draw(ref world, ref view, ref projection);
81.                 modeloDinamico.Draw(ref world, ref view, ref projection);
82.                 texto.Draw(modeloEstatico.ToString(), Color.Brown);
83.                 texto.Draw("\n\n\n" + modeloDinamico.ToString(), Color.Black);
84.
85.                 base.Draw(gameTime);
86.             }
87.         }
88.     }
```

9.2 Axis Aligned Bounding Box vs. Axis Aligned Bounding Box

La **AABB** por sus siglas en inglés, o Caja Envolvente Alineada con los Ejes, cubre la geometría con una caja alineada a los ejes del mundo. Esto puede causar un inconveniente en el momento de girar la geometría, porque puede cambiar las dimensiones de la envolvente, véase Ilustración 9-5.

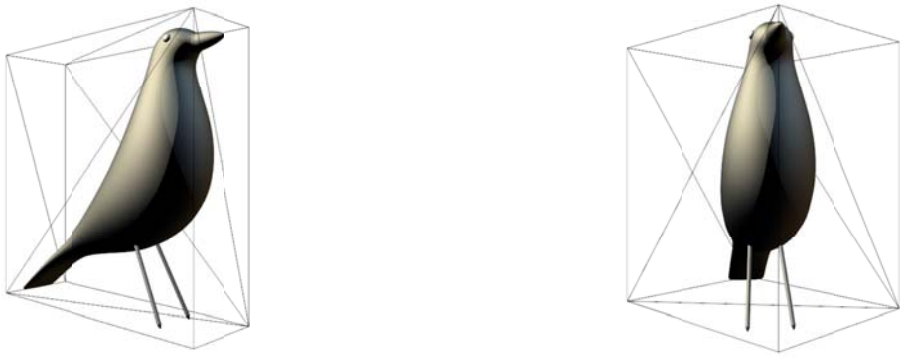

Ilustración 9-5 AABB

La AABB se basa en dos puntos en el espacio llamados máximo y mínimo; cada uno representa en sus coordenadas el valor máximo o mínimo de los vectores de posición de los vértices de la geometría. La obtención de dichos puntos se puede hacer buscando el menor y mayor valor entre las coordenadas de cada vértice de la figura tridimensional.

XNA proporciona métodos para la obtención de la envolvente a partir de las posiciones de los vértices, de una envolvente de esfera y de dos instancias de **BoundingBox** especificadas.

Al igual que el ejemplo de Bounding Sphere vs. Bounding Sphere, se hará mover la instancia de **ModeloDinamico** para mostrar el choque entre dos AABB.

La clase **ModeloEstatico**, Código 9-5, es muy similar al Código 9-2, a excepción de la envolvente **BoundingBox**, línea 11, y el método **ObtenerAABB**, líneas 35 -53. La primera representa un volumen 3D en forma de caja alineado a los ejes, y el segundo es un método privado que se describe más adelante.

En la línea 20 se obtiene la AABB a partir de la información del búfer de vértices del **ModelMesh**. Cabe aclarar que cada **ModelMesh** de un **Model** tiene su propio búfer, por lo tanto, en este ejemplo sólo se ocupa el primer **ModelMesh**.

El método **ObtenerAABB** regresa un **BoundingBox** a partir de un **VertexBuffer**, que recibe como parámetro. En las líneas 37 – 39, se declara un arreglo de **VertexPositionNormalTexture** para almacenar una copia de los vértices del búfer. El tamaño de dicho arreglo se obtiene a partir de la división entre el tamaño en bytes del búfer y el tamaño en bytes del tipo de dato, el resultado es el número de vértices en el búfer. El tipo de dato debe soportar toda la información almacenada en el búfer, de no ser así, una excepción en tiempo de ejecución resultará.

La información almacenada en el búfer de vértices no se puede modificar, pero si se puede leer; para ello se hace una copia de los datos, con el método **VertexBuffer.GetData**, líneas 40 – 41.

```
public void GetData<T> (T[] data)
```

Este método es genérico y está sobrecargado tres veces. Las estructuras definidas por XNA, para los vértices son lo más apropiados para su uso. El arreglo debe ser lo suficientemente grande para hacer la copia y no provocar una excepción en tiempo de ejecución.

Una vez que se tiene la copia de los vértices se debe extraer los vectores de posición, el arreglo **puntos**, línea 43, es de tipo **Vector3** y es de la misma dimensión que el arreglo de vértices.

En el ciclo **for**, líneas 44 – 49, se itera a través del arreglo de vértices, **vertexPositionNormalTexture**, para extraer la posición y multiplicarla por la matriz de transformaciones que se obtiene del modelo, líneas 46 – 48; y almacenarla en el elemento correspondiente al arreglo puntos.

El método estático **BoundingBox.CreateFromPoints**, línea 52, crea el **BoundingBox** a partir de un grupo de puntos. Este método también acepta un **List**[42] como entrada, pues ocupa una interfaz **IEnumerable**.

```
public static BoundingBox CreateFromPoints (IEnumerable<Vector3> points)
```

Código 9-5

```
1.      using System;
2.      using Microsoft.Xna.Framework.Graphics;
3.      using Microsoft.Xna.Framework;
4.
5.      namespace TutorialX13
6.      {
7.          public class ModeloEstatico
8.          {
9.              private Model modelo;
10.             private Matrix[] transformaciones;
11.             protected BoundingBox boundingBox;
12.             string nombre;
13.
14.             public ModeloEstatico(Model modelo)
15.             {
16.                 this.modelo = modelo;
17.                 transformaciones = new Matrix[modelo.Bones.Count];
18.                 modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
19.
20.                 boundingBox = ObtenerAABB(modelo.Meshes[0].VertexBuffer);
21.
22.                 nombre = modelo.Meshes[0].Name;
23.             }// fin del constructor
24.
25.             /// <summary>
26.             /// Propiedad que obtiene la AABB.
27.             /// </summary>
28.             public BoundingBox EnvolventeCaja { get { return boundingBox; } }
29.
30.             /// <summary>
31.             /// Método que obtiene la AABB.
32.             /// </summary>
33.             /// <param name="vertexBuffer">Búfer de vértices.</param>
34.             /// <returns></returns>
```

[42] Representa una lista de objetos.

```
35.             private BoundingBox ObtenerAABB(VertexBuffer vertexBuffer)
36.             {
37.                 VertexPositionNormalTexture[] vertexPositionNormalTexture =
38.                     new VertexPositionNormalTexture[vertexBuffer.SizeInBytes
39.                         / VertexPositionNormalTexture.SizeInBytes];
40.                 vertexBuffer.GetData<VertexPositionNormalTexture>(
41.                     vertexPositionNormalTexture);
42.                 // se copian las posiciones de los puntos en un arreglo de Vector3
43.                 Vector3[] puntos = new Vector3[vertexPositionNormalTexture.Length];
44.                 for (int i = 0; i < vertexPositionNormalTexture.Length; i++)
45.                 {
46.                     puntos[i] = Vector3.Transform(
47.                         vertexPositionNormalTexture[i].Position,
48.                         transformaciones[modelo.Meshes[0].ParentBone.Index]);
49.                 }// fin del for
50.
51.                 // se crea el BoundinBox a partir de un arreglo de puntos
52.                 return BoundingBox.CreateFromPoints(puntos);
53.             }// fin del método ObtenerAABB
54.
55.             public virtual void Draw(ref Matrix world, ref Matrix view,
56.                 ref Matrix projection)
57.             {
58.                 foreach (ModelMesh mesh in modelo.Meshes)
59.                 {
60.                     foreach (BasicEffect basicEffect in mesh.Effects)
61.                     {
62.                         basicEffect.World = transformaciones[mesh.ParentBone.Index]
63.                             * world;
64.                         basicEffect.View = view;
65.                         basicEffect.Projection = projection;
66.                         basicEffect.EnableDefaultLighting();
67.                         basicEffect.PreferPerPixelLighting = true;
68.                     }// fin del foreach
69.                     mesh.Draw();
70.                 }// fin del foreach
71.             }// fin del método Draw
72.
73.             public override string ToString()
74.             {
75.                 return string.Format("{0}\nBoundingBox.Min {1}\nBoundingBox.Max {2}",
76.                     nombre, boundingBox.Min, boundingBox.Max);
77.             }// fin del método ToString
78.
79.         }// fin de la clase ModeloEstatico
80.     }// fin del namespace TutorialX13
```

Al igual que en el ejemplo de Bounding Sphere vs. Bounding Sphere, se tiene una clase **ModeloDinamico** que cambia la posición del modelo tridimensional, con el teclado o el gamepad. La asignación de acciones para el teclado y los elementos del control del Xbox son los mismos que en el subtema anterior.

En el Código 9-6 la variable de instancia **mtxTraslacionPuntos**, línea 11, sirve como una matriz de transformación para los puntos máximo y minímo del AABB. La variable direccion, línea12, es un numerador que presenta las seis posibles traslaciones, la definición de dicha numeración se presenta en el El método **Draw**, líneas 171 – 176, no sufre ningún cambio con relación al visto en el Código 9-3.

Código 9-7.

El método **Update**, líneas 19 – 98, en cada condicional **if** se establece la dirección de hacia donde se mueve la figura, se crea la matrix de traslación y se hace llamar al método **DetenerMovimiento**.

Como en el subtema anterior, se hace mover primero los puntos de la AABB a través de la matriz **mtxTraslacionPuntos**, concebida en el método **Update**, para conocer si existe colisión entre cajas. En caso de que exista, se hace retroceder la caja a una posición anterior y se calcula una distancia lo suficientemente cercana entre los límites de las cajas. Luego se hace mover los puntos, esa distancia, y por último se multiplica la matriz de traslación de puntos por la matriz de traslación de la figura y el resultado es asignado a esta última. Si no existiera dicha colisión, solo se hace este último paso.

El método **DetenerMovimiento**, líneas 104 – 163, calcula las matrices de traslación para la figura tridimensional y para los puntos de la AABB, que en realidad siempre está empujando la envolvente de caja a una posición prudente para que no exista intersección. En la primera parte de este método se multiplican los puntos máximo y mínimo, de la AABB del objeto **ModeloDinamico**, por la matriz **mtxTraslacionPuntos**, líneas 117 – 120. Enseguida se busca la existencia de una intersección entre las envolventes del modelo estático, **boundingBoxB**, y el dinámico, **boundingBox**, línea 112. Si el método **Intersects** devuelve un valor verdadero, se comienza por retroceder los puntos del **boundingBox** a un estado anterior, esto se logra multiplicando la matriz de **mtxTraslacionPuntos** por menos uno, línea 115, para luego aplicarlo sobre los puntos **Min** y **Max** de **boundingBox**, líneas 117 – 120.

Se crea una variable de tipo flotante llamada **dist**, línea 124, que sirve como margen de error para evitar que la distancia entre los límites de las AABB sea cero, permitiendo que se acerque tanto como sea posible para mover el **boundingBox** en otra dirección.

En la instrucción de control **switch**, líneas 125 – 154, se selecciona la dirección en la que se pretende mover el modelo dinámico. Esta selección sirve para realizar la operación correcta entre los puntos máximo y mínimo de las AABB de cada objeto. Por ejemplo, en la Ilustración 9-6, se tienen dos cajas. **A** representa el modelo dinámico y **B** es el modelo estático. Los puntos naranjas indican los puntos máximos, y los verdes son los puntos mínimos de cada envolvente. La caja **A** se mueve en dirección de **z** positivo, lo que indica que su dirección es hacia atrás, y la distancia a recorrer A hacia **B** es el valor absoluto de la resta entre el componente **z** del punto mínimo de **B** menos el componente **z** del punto máximo de **A**. Este mismo razonamiento se aplica en cada posible movimiento de la envolvente **A**.

$$d = |z_{minB} - z_{maxA}| \qquad (1)$$

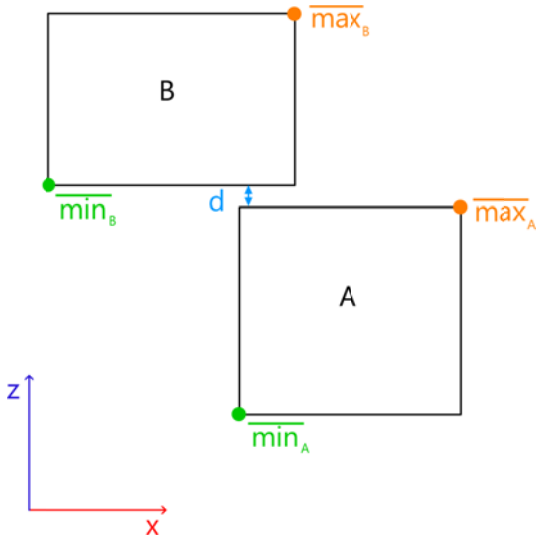

Ilustración 9-6 Distancia entre AABBs

Sin embargo, al hacer la distancia **d** cero, el método **Intersects** del **BoundingBox** seguirá arrogando un valor verdadero, aún si el movimiento siguiente no provoque una colisión. Es por eso que se agrega una constante de error a la ecuación 1. Esta constante debe ser muy pequeña, para no ser percibida en el dibujo final.

$$d = |z_{minB} - z_{maxA}| + k \qquad (2)$$

Continuando con el enlistado, dentro del primer **case**, línea 127, se cálcua la distancia con la ecuación 2, línea 128, enseguida se crea la matriz de traslación de los puntos, línea 129. En cada caso se debe utilizar una ecuación diferente para obtener la distancia correcta a recorrer la figura y la envolvente. Al final del **swtich** se calculan las nuevas posiciones de los puntos del **boundingBox**, líneas 156 – 159.

Independientemente, si el cuerpo del condicional **if** es alcanzado o no, se multiplica la matriz de traslación de los puntos, **mtxTraslacionPuntos**, por la matriz de traslación de la figura, **mtxTraslacion** y el resultado es guardado en esta última, línea 162.

Código 9-6

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Input;
4.
5.      namespace TutorialX13
6.      {
7.          public class ModeloDinamico : ModeloEstatico
8.          {
9.              private const float step = 1.0F;
10.             private Matrix mtxTraslacion = Matrix.Identity;
11.             private Matrix mtxTraslacionPuntos;
12.             private Direccion direccion;
13.
14.             public ModeloDinamico(Microsoft.Xna.Framework.Graphics.Model modelo)
15.                 : base(modelo)
16.             {
17.             }// fin del constructor
18.
19.             public void Update(BoundingBox boundingBoxB)
20.             {
21.                 #region Keyboard
22.                 if (Keyboard.GetState().IsKeyDown(Keys.Up))
23.                 {
24.                     direccion = Direccion.Adelante;
25.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, 0, -step);
26.                     DetenerMovimiento(boundingBoxB);
27.                 }
28.                 if (Keyboard.GetState().IsKeyDown(Keys.Down))
29.                 {
30.                     direccion = Direccion.Atras;
31.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, 0, step);
32.                     DetenerMovimiento(boundingBoxB);
33.                 }
34.                 if (Keyboard.GetState().IsKeyDown(Keys.Right))
35.                 {
36.                     direccion = Direccion.Derecha;
37.                     mtxTraslacionPuntos = Matrix.CreateTranslation(step, 0, 0);
38.                     DetenerMovimiento(boundingBoxB);
39.                 }
40.                 if (Keyboard.GetState().IsKeyDown(Keys.Left))
41.                 {
42.                     direccion = Direccion.Izquierda;
43.                     mtxTraslacionPuntos = Matrix.CreateTranslation(-step, 0, 0);
44.                     DetenerMovimiento(boundingBoxB);
45.                 }
46.                 if (Keyboard.GetState().IsKeyDown(Keys.PageUp))
47.                 {
48.                     direccion = Direccion.Arriba;
49.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, step, 0);
50.                     DetenerMovimiento(boundingBoxB);
51.                 }
52.                 if (Keyboard.GetState().IsKeyDown(Keys.PageDown))
53.                 {
54.                     direccion = Direccion.Abajo;
55.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, -step, 0);
56.                     DetenerMovimiento(boundingBoxB);
57.                 }
58.                 #endregion
59.
60.                 #region Gamepad
61.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y < 0)
62.                 {
63.                     direccion = Direccion.Atras;
64.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, 0, step);
```

```
65.                     DetenerMovimiento(boundingBoxB);
66.                 }
67.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.Y > 0)
68.                 {
69.                     direccion = Direccion.Adelante;
70.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, 0, -step);
71.                     DetenerMovimiento(boundingBoxB);
72.                 }
73.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X < 0)
74.                 {
75.                     direccion = Direccion.Izquierda;
76.                     mtxTraslacionPuntos = Matrix.CreateTranslation(-step, 0, 0);
77.                     DetenerMovimiento(boundingBoxB);
78.                 }
79.                 if (GamePad.GetState(PlayerIndex.One).ThumbSticks.Right.X > 0)
80.                 {
81.                     direccion = Direccion.Derecha;
82.                     mtxTraslacionPuntos = Matrix.CreateTranslation(step, 0, 0);
83.                     DetenerMovimiento(boundingBoxB);
84.                 }
85.                 if (GamePad.GetState(PlayerIndex.One).Triggers.Right != 0)
86.                 {
87.                     direccion = Direccion.Arriba;
88.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, step, 0);
89.                     DetenerMovimiento(boundingBoxB);
90.                 }
91.                 if (GamePad.GetState(PlayerIndex.One).Triggers.Left != 0)
92.                 {
93.                     direccion = Direccion.Abajo;
94.                     mtxTraslacionPuntos = Matrix.CreateTranslation(0, -step, 0);
95.                     DetenerMovimiento(boundingBoxB);
96.                 }
97.                 #endregion
98.             }// fin del método Update
99.
100.            /// <summary>
101.            /// Método que detiene el movimiento de la BS, si ésta intersecta con otra.
102.            /// </summary>
103.            /// <param name="boudingSphereB"></param>
104.            private void DetenerMovimiento(BoundingBox boundingBoxB)
105.            {
106.                // se trasladan los puntos Min y Max de BoundingBox
107.                Vector3.Transform(ref boundingBox.Min, ref mtxTraslacionPuntos,
108.                    out boundingBox.Min);
109.                Vector3.Transform(ref boundingBox.Max, ref mtxTraslacionPuntos,
110.                    out boundingBox.Max);
111.
112.                if (boundingBox.Intersects(boundingBoxB))
113.                {
114.                    // se regresan los puntos a una posición anterior
115.                    mtxTraslacionPuntos.Translation = -mtxTraslacionPuntos.Translation;
116.
117.                    Vector3.Transform(ref boundingBox.Min, ref mtxTraslacionPuntos,
118.                        out boundingBox.Min);
119.                    Vector3.Transform(ref boundingBox.Max, ref mtxTraslacionPuntos,
120.                        out boundingBox.Max);
121.                    // se calcula la distancia que falta para que
122.                    // las envolventes estén tan cerca
123.                    // como para no intersecarse
124.                    float dist = -0.01F;
125.                    switch (direccion)
126.                    {
127.                        case Direccion.Atras:
128.                            dist += Math.Abs(boundingBoxB.Min.Z - boundingBox.Max.Z);
129.                            mtxTraslacionPuntos = Matrix.CreateTranslation(0, 0, dist);
130.                            break;
131.                        case Direccion.Adelante:
132.                            dist += Math.Abs(boundingBox.Min.Z - boundingBoxB.Max.Z);
133.                            mtxTraslacionPuntos = Matrix.CreateTranslation(
134.                                0, 0, -dist);
135.                            break;
```

```
136.                         case Direccion.Derecha:
137.                             dist += Math.Abs(boundingBoxB.Min.X - boundingBox.Max.X);
138.                             mtxTraslacionPuntos = Matrix.CreateTranslation(dist, 0, 0);
139.                             break;
140.                         case Direccion.Izquierda:
141.                             dist += Math.Abs(boundingBox.Min.X - boundingBoxB.Max.X);
142.                             mtxTraslacionPuntos = Matrix.CreateTranslation(
143.                                 -dist, 0, 0);
144.                             break;
145.                         case Direccion.Abajo:
146.                             dist += Math.Abs(boundingBox.Min.Y - boundingBoxB.Max.Y);
147.                             mtxTraslacionPuntos = Matrix.CreateTranslation(
148.                                 0, -dist, 0);
149.                             break;
150.                         case Direccion.Arriba:
151.                             dist += Math.Abs(boundingBoxB.Min.Y - boundingBox.Max.Y);
152.                             mtxTraslacionPuntos = Matrix.CreateTranslation(0, dist, 0);
153.                             break;
154.                     };
155.                     // se trasladan los puntos a la nueva posición
156.                     Vector3.Transform(ref boundingBox.Min, ref mtxTraslacionPuntos,
157.                         out boundingBox.Min);
158.                     Vector3.Transform(ref boundingBox.Max, ref mtxTraslacionPuntos,
159.                         out boundingBox.Max);
160.                 }// fin del if
161.
162.                 mtxTraslacion *= mtxTraslacionPuntos;
163.             }// fin del método DetenerMovimiento
164.
165.             /// <summary>
166.             /// Método que dibuja la geometría.
167.             /// </summary>
168.             /// <param name="world">Matriz de mundo.</param>
169.             /// <param name="view">Matriz de vista.</param>
170.             /// <param name="projection">Matriz de proyección.</param>
171.             public override void Draw(ref Matrix world, ref Matrix view,
172.                 ref Matrix projection)
173.             {
174.                 mtxTraslacion *= world;
175.                 base.Draw(ref mtxTraslacion, ref view, ref projection);
176.             }// fin del método
177.         }// fin de la clase
178.     }// fin del namespace
```

El método **Draw**, líneas 171 – 176, no sufre ningún cambio con relación al visto en el Código 9-3.

Código 9-7

```
1.      using System;
2.
3.      namespace TutorialX13
4.      {
5.          public enum Direccion
6.          {
7.              Adelante,
8.              Atras,
9.              Abajo,
10.             Arriba,
11.             Derecha,
12.             Izquierda,
13.         }
14.     }
```

Para probar la colisión entre las AABBs se reutiliza el Código 9-4, con un ligero cambio en la llamada del método **Update**, línea 71; pues esta vez se pasa como parámetro una **BoundingBox**.

```
modeloDinamico.Update(modeloEstatico.EnvolventeCaja);
```

Corra la aplicación y corrija cualquier error en tiempo de ejecución que pudiera suceder. En la Ilustración 9-7 se muestran dos figuras de elefante colisionando.

Ilustración 9-7 AABB vs. AABB

9.3 Ray vs. Boundign Sphere

A pesar que las envolventes de colisión ayudan en gran medida a conocer si dos o más figuras impactan, reduciendo los cálculos y tiempos para la computadora; a veces existen cosas pequeñas que bien podrían ser abstraídas como partícula, permitiendo un mejor desempeño computacional que si se hubieran manejado como esfera u otra envolvente. Por ejemplo:

La detección de una colisión de un proyectil en contra de un objeto se puede calcular omitiendo el tamaño, la fricción, el peso, temperatura y demás factores que podrían alterar su trayectoria, antes de chocar. Lo que nos queda como estudio es la posición a partir de la cual se lanza y la dirección que toma.

El rayo es un ente que parte de un punto en el espacio e indica una trayectoria a partir de un vector de dirección. **Ray** es la estructura que XNA ofrece para el rayo.

Rayo no es considerado como un volumen de envolvente, pues no existe el conjunto de vértices al cual cubrir. Sin embargo, se pueden tener pruebas de intersección entre el rayo y las envolventes.

En el siguiente ejemplo se muestra un sprite como mira, para orientar la vista y poder seleccionar algún elemento rodeado por una BS. Cuando la mira esté apuntando a alguna envolvente de esfera, cambiará su color original a rojo, indicando que puede ser seleccionada. La figura elegida cambiará su color ambiental negro por rojo, después de un segundo, retornará a su color original. El movimiento de la cámara se hará a través del gamepad.

9.3.1 Cámara libre

Los movimientos de la cámara libre no tienen alguna restricción, permitiendo cualquier observación en cualquier punto, en cualquier modo de orientación. La cámara puede trasladarse sobre sus propios ejes y rotar alrededor de ellos. Los ejes de la cámara que se manejan en el ejemplo son: **Up**, **Binormal** y **Dirección**, véase Ilustración 9-8. El primero permite tener una noción de donde es arriba y abajo; el segundo es perpendicular a **Up** y **Dirección**. El último, es resultado de la resta entre el vector de observación y la posición de la cámara.

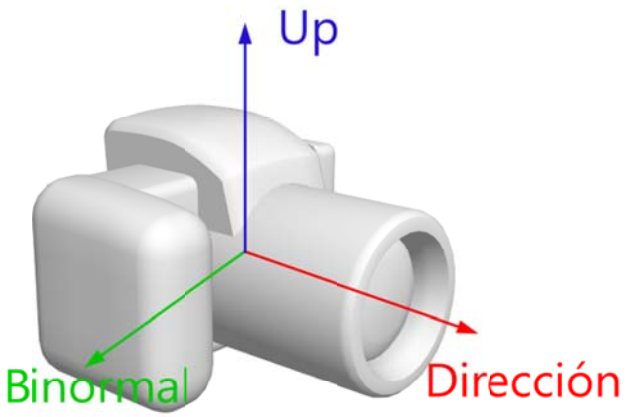

Ilustración 9-8 Vectores de la cámara

En la clase **CamaraLibre**, Código 9-8, las variables de instancia **posicion**, **objetivo** y **up**, línea 9, son las necesarias para obtener la matriz de vista **mtxVista**, línea 10, con el método **CreateLookAt**. La variable **playerIndex**, línea 11, es para seleccionar al jugador que controla la cámara, por medio del gamepad. Los ejes binormal, línea 12, y dirección, línea 13, están representados por las variables de su mismo nombre, y que ayudaran a calcular las transformaciones sobre la matriz de vista. La traslación y rotación de la cámara estarán dadas por las matrices **mtxTraslacion**, línea 14, y **mtxRotacion**, línea 15. La **sensibilidad**, línea 16, denota que tan rápido puede girar la cámara al cambio sobre el stick o triggers del gamepad. La variable **pasoMaximo**, línea 17, le da un valor máximo a los cambios sobre el stick, para trasladar la cámara. La última variable de instancia, **invertirVista**, línea 18, permite cambiar el sentido en que gira la cámara sobre el eje binormal.

El constructor debe recibir los vectores que definen la matriz de vista, el valor de la variable de sólo lectura, **pasoMaximo**; y el número del jugador. En el cuerpo del constructor se inicializan algunas variables de instancia, líneas 32 – 38, y se hace llamar al método **Inicializar**, líneas 72 – 80. En este último, se calculan los ejes de la cámara y la matriz de vista.

El cálculo del vector dirección está dada por:

$$\bar{d} = |\bar{o} - \bar{p}| \qquad (1)$$

Donde $\bar{o}$ es el vector de observación y $\bar{p}$ es el vector de posición de la cámara. El operador – está sobrecargado para las estructuras que definen un vector, línea 75, y matrices. Además estas estructuras contienen métodos que completan las operaciones sobre los vectores o matrices; el método **Normalize**, línea 76, crea un vector unitario.

El vector unitario binormal es el resultado de la normalización del producto cruz entre los vectores dirección y objetivo, tómese en cuenta que el producto cruz no cumple con la propiedad de conmutación.

$$\bar{b} = \left|\bar{d} \times \bar{o}\right| \qquad (2)$$

Donde $\bar{d}$ es el vector dirección y $\bar{o}$ es el vector de observación. El método estático **Cross**, línea 77, obtiene el producto cruz. Estos métodos sobre los vectores o matrices, tienen sobrecargas que permiten seleccionar el tipo de paso[43] de valores a sus parámetros y es cuestión del desarrollador elegir adecuadamente cuál es el mejor.

[43] Existen dos formas de pasar los valores a un método: por valor o por referencia. El paso por valor hace una copia de la información evitando alterar la fuente.

Al final del método Inicializar se crea la matriz de vista, línea 79, unque no es necesario normalizar el vector **up** para el método **CreateLookAt**, si lo será para los cálculos de rotación.

La propiedad **MtxVista**, línea 45, obtiene el valor de la matriz de vista, parte importante para el render; la propiedad **Direccion**, línea 50, obtiene el vector de dirección de la cámara**,** éste sirve para definir el rayo; la propiedad **Posicion**, línea 55, obtiene el vector de posición de la cámara, también importante para crear un rayo. La única propiedad que obtiene o establece valores es **Sensibilidad**, líneas 61 – 70, permitiendo cambiar fuera de la instancia la rapidez con que puede cambiar el giro de la cámara sobre sus ejes. La **Sensibilidad** es el ángulo, expresado en radianes, restringido entre uno a diez grados, en caso que el valor a establecer se encuentre fuera de este rango, se le asigna un grado.

El método **AdelanteAtras**, líneas 82 – 89, hace avanzar la cámara sobre su eje dirección positivamente o negativamente, dependiendo del signo del parámetro de entrada. En el cuerpo de éste, se crea la matriz de traslación, línea 84, a partir del vector de dirección y los escalares **pasoMaximo** y **paso**. Enseguida se multiplica la matriz por los vectores posición y objetivo, líneas 86 – 87, para luego crear la matriz de vista, línea 88.

Para hacer avanzar la cámara en dirección positivo o negativo sobre el eje binormal, el método **DerechaIzquierda**, líneas 91 – 97, utiliza la mismas operaciones que el método **AdelanteAtras**, pero utilizando el vector binormal como eje de traslación, línea 93.

Los giros de la cámara se hacen alrededor de sus ejes, y a cada uno le corresponde un método cuyo nombre corresponde a las distintas superficies de control de un avión, véase Ilustración 9-9.

Ilustración 9-9 Superficies de control del avión

El método **Guiñada**, líneas 99 – 108, hace rotar la cámara alrededor del vector **Up**, cambiando los ejes dirección y up; el parámetro de entrada, así como en los otros dos métodos de giro, es una variable de tipo flotante que representa el ángulo de cambio. El ángulo es multiplicado por la propiedad **Sensibilidad**, y el valor obtenido se almacena en la variable a**ngulo**, línea 101. Luego se crea la matriz de rotación **mtxRotacion** con el método **Matrix.CreateFromAxesAngle**, línea 102. Este método crea una matriz que gira alrededor de un vector arbitrario, por lo tanto su parámetros son el vector y el escalar como ángulo. Una vez obtenido la matriz de rotación, se les multiplica a los vectores de dirección y up para girarlos, líneas 103 – 104. Ahora que se han actualizado los ejes de la cámara, es momento de cambiar el vector **objetivo** de la matriz de vista, línea 105; a partir de la ecuación (1) se obtiene el valor de dicho vector.

$$\bar{o} = \bar{p} + \bar{d} \qquad (3)$$

Nótese que el vector **objetivo** no es un vector unitario, por eso se ha omitido la normalización.

El paso por referencia no hace una copia del valor, sino "apunta" a la dirección sobre la cual se encuentra la información, evitando redundancia, lo que permite cambiar el valor de la fuente.

Los métodos cuyos parámetros son estructuras, por default pasan por valor, a menos que se indique lo contrario por medio de las palabras clave out o ref.

Al final, se actualiza la matriz de vista, creando una nueva con **Matriz.CreateLookAt**, línea 106.

El siguiente método, **Inclinación**, gira la cámara alrededor del vector binormal, afectando los ejes dirección y up. Las operaciones son las mismas que en el método **Guiñada**, se crea la matriz de rotación alrededor del eje binormal, línea112, se multiplica la matriz por los vectores de dirección y up, líneas 113 – 114; se actualiza el vector objetivo, línea 115; y se crea una nueva matriz de vista, línea 116.

El método **Balanceo** gira la cámara alrededor del eje dirección, afectando los vectores binormal y up. Así que primero se crea la matriz de rotación alrededor del vector dirección, línea 122, se multiplica esta matriz por los vectores **binormal** y **up**, líneas 123 – 124; y se actualiza la matriz de vista, línea 125.

En la Tabla 9-2 se muestra la asignación de movimientos en el gamepad para describir el método **Update**, líneas 128 – 158.

Tabla 9-2

Gamepad	Momiviento
ThumbSticks.Right.X	Guiñada
ThumbSticks.Right.Y	Inclinación
ThumbSticks.Left.X	DerechaIzquierda
ThumbSticks.Left.Y	ArribaAbajo
Triggers.Right	Balanceo
Triggers.Left	Balanceo

El método **Update** no se describirá a detalle, pues una vez descrita la tabla anterior, es muy fácil implementarla; a excepción del uso del **ThumbSticks.Richt.Y**, donde el uso de un condicional **if** anidado, líneas 136 – 140, verifica en qué sentido se debe hacer el cambio de la cámara para mirar hacia arriba o hacia abajo. Si el valor de la propiedad **InvertirVista** es verdadero, el cambio sobre el **ThumbSticks.Right.Y** se multiplicará por menos uno y se le pasará como parámetro al método privado **Inclinación**; en caso contrario el valor del **ThumbSitck** no se alterará.

Código 9-8

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Input;
4.
5.      namespace TutorialX15
6.      {
7.          public class CamaraLibre
8.          {
9.              private Vector3 posicion, objetivo, up;
10.             private Matrix mtxVista;
11.             private PlayerIndex playerIndex;
12.             private Vector3 binormal;
13.             private Vector3 direccion;
14.             private Matrix mtxTraslacion;
15.             private Matrix mtxRotacion;
16.             private Single sensibilidad;
17.             private readonly Single pasoMaximo;
18.             public Boolean InvertirVista;
19.
20.             /// <summary>
21.             /// Constructor
```

```
22.             /// </summary>
23.             /// <param name="position">Posición de la cámara.</param>
24.             /// <param name="objetivo">Objetivo de la cámara.</param>
25.             /// <param name="up">Vector Up de la cámara.</param>
26.             /// <param name="pasoMaximo">Paso máximo al que se
27.             /// movera la cámara.</param>
28.             /// <param name="playerIndex">Número de jugador.</param>
29.             public CamaraLibre(Vector3 posicion, Vector3 objetivo,
30.                 Vector3 up, Single pasoMaximo, PlayerIndex playerIndex)
31.             {
32.                 this.posicion = posicion;
33.                 this.objetivo = objetivo;
34.                 this.up = up;
35.                 this.playerIndex = playerIndex;
36.                 this.pasoMaximo = pasoMaximo;
37.                 Sensibilidad = 1.0F;
38.                 InvertirVista = true;
39.                 Inicializar();
40.             }// fin del constructor
41.
42.             /// <summary>
43.             /// Propiedad que obtiene la matriz de vista.
44.             /// </summary>
45.             public Matrix MtxVista { get { return mtxVista; } }
46.
47.             /// <summary>
48.             /// Propiedad que obtiene la dirección de la cámara.
49.             /// </summary>
50.             public Vector3 Direccion { get { return direccion; } }
51.
52.             /// <summary>
53.             /// Propiedad que obtiene la posición de la cámara.
54.             /// </summary>
55.             public Vector3 Posicion { get { return posicion; } }
56.
57.             /// <summary>
58.             /// Propiedad que obtiene o establece la sensibilidad
59.             /// de giro de la cámara.
60.             /// </summary>
61.             public Single Sensibilidad
62.             {
63.                 get { return sensibilidad; }
64.                 set
65.                 {
66.                     sensibilidad = MathHelper.ToRadians(
67.                         (value >= 1 && value <= 10) ?
68.                     value : 1.0F);
69.                 }
70.             }// fin de la propiedad Sensibilidad
71.
72.             private void Inicializar()
73.             {
74.                 up.Normalize();
75.                 direccion = objetivo - posicion;
76.                 direccion.Normalize();
77.                 Vector3.Cross(ref direccion, ref up, out binormal);
78.                 binormal.Normalize();
79.                 mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
80.             }// fin del método Inicializar
81.
82.             private void AdelanteAtras(Single paso)
83.             {
84.                 mtxTraslacion = Matrix.CreateTranslation(pasoMaximo * paso *
85.                     direccion);
86.                 Vector3.Transform(ref posicion, ref mtxTraslacion, out posicion);
87.                 Vector3.Transform(ref objetivo, ref mtxTraslacion, out objetivo);
88.                 mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
89.             }// fin del método AdelanteAtras
90.
91.             private void DerechaIzquierda(Single paso)
92.             {
```

```
93.                 mtxTraslacion = Matrix.CreateTranslation(pasoMaximo * paso * binormal);
94.                 Vector3.Transform(ref posicion, ref mtxTraslacion, out posicion);
95.                 Vector3.Transform(ref objetivo, ref mtxTraslacion, out objetivo);
96.                 mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
97.             }// fin del método DerechaIzquierda
98.
99.             private void Guiñada(Single angulo)
100.            {
101.                angulo *= Sensibilidad;
102.                mtxRotacion = Matrix.CreateFromAxisAngle(up, angulo);
103.                Vector3.Transform(ref direccion, ref mtxRotacion, out direccion);
104.                Vector3.Transform(ref binormal, ref mtxRotacion, out binormal);
105.                objetivo = posicion + direccion;
106.                mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
107.            }// fin del método
108.
109.            private void Inclinacion(Single angulo)
110.            {
111.                angulo *= Sensibilidad;
112.                mtxRotacion = Matrix.CreateFromAxisAngle(binormal, angulo);
113.                Vector3.Transform(ref direccion, ref mtxRotacion, out direccion);
114.                Vector3.Transform(ref up, ref mtxRotacion, out up);
115.                objetivo = posicion + direccion;
116.                mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
117.            }// fin del método
118.
119.            private void Balanceo(Single angulo)
120.            {
121.                angulo *= Sensibilidad;
122.                mtxRotacion = Matrix.CreateFromAxisAngle(direccion, angulo);
123.                Vector3.Transform(ref binormal, ref mtxRotacion, out binormal);
124.                Vector3.Transform(ref up, ref mtxRotacion, out up);
125.                mtxVista = Matrix.CreateLookAt(posicion, objetivo, up);
126.            }// fin del método
127.
128.            public void Update()
129.            {
130.                if (GamePad.GetState(playerIndex).ThumbSticks.Right.X != 0)
131.                {
132.                    Guiñada(-GamePad.GetState(playerIndex).ThumbSticks.Right.X);
133.                }
134.                if (GamePad.GetState(playerIndex).ThumbSticks.Right.Y != 0)
135.                {
136.                    if(InvertirVista)
137.                        Inclinacion(
138.                            -GamePad.GetState(playerIndex).ThumbSticks.Right.Y);
139.                    else
140.                        Inclinacion(GamePad.GetState(playerIndex).ThumbSticks.Right.Y);
141.                }
142.                if (GamePad.GetState(playerIndex).ThumbSticks.Left.X != 0)
143.                {
144.                    DerechaIzquierda(GamePad.GetState(playerIndex).ThumbSticks.Left.X);
145.                }
146.                if (GamePad.GetState(playerIndex).ThumbSticks.Left.Y != 0)
147.                {
148.                    AdelanteAtras(GamePad.GetState(playerIndex).ThumbSticks.Left.Y);
149.                }
150.                if (GamePad.GetState(playerIndex).Triggers.Right != 0)
151.                {
152.                    Balanceo(GamePad.GetState(playerIndex).Triggers.Right);
153.                }
154.                if (GamePad.GetState(playerIndex).Triggers.Left != 0)
155.                {
156.                    Balanceo(-GamePad.GetState(playerIndex).Triggers.Left);
157.                }
158.            }// fin del método Update
159.        }// fin de la clase
160.    }// fin del namespace
```

9.3.2 La clase Mira

La clase **Mira** representa un sprite que es dibujado en medio del viewport, Código 9-9, y cambia de textura cada vez que ha encontrado un BS con el que colisiona un rayo.

El **graphicsDevice**, línea 9, sirve para obtener las dimensiones del viewport y poder crear el **spriteBatch**, línea 10; la texturas **A** y **B**, línea 11, serán los dos posibles sprites a dibujar; la variable **rectangulo**, línea 12, constituye las dimensiones del sprite; y la variable **preparado**, línea 13, selecciona qué textura se dibujará.

El constructor, líneas 15 – 26, tiene como parámetros las dos texturas, el ancho y alto para definir el rectángulo; y el **graphicsDevice**. Dentro del cuerpo se incializan y se crean las variables de instancia. El constructor **Rectangle**, líneas 23 – 26, pide el valor en entero de la coordenada **x** de la posición de la esquina superior izquierda del rectángulo, en el viewport; el valor en entero de la coordenada **y** de la misma esquina, el ancho y alto del rectángulo. Así que para centrar el rectángulo en el viewport se le resta al ancho del viewport el ancho del rectángulo y se le divide entre dos, y lo mismo sucede para el alto; recuérdese que los valores deben ser enteros, y la conversión explicita de **float** a **int**, líneas 24 – 25, es obligatoria.

El método **Update** actualiza el valor de la variable preparado, a partir de la colisión entre un rayo y una envolvente de esfera. Sus parámetros son el rayo y un arreglo de objetos **ModeloEstatico** de donde se obtienen las BS.

Para seleccionar cada BS se hace pasar el arreglo de **ModeloEstatico** por un bucle **foreach**, líneas 30 – 39, y a cada uno se le hace la prueba de intersección, Ray vs. BS, línea 32; la prueba regresa un **Nullable**[44]**<float>**, esto indica que el elemento puede ser del tipo de dato **T**, en este caso **float**, o puede ser **null**. En caso que la prueba haya encontrado una intersección, el método regresa la distancia del punto de posición del rayo a la envolvente, en caso contrario regresa un **null**. El método **Ray.Intersects** está sobrecargado para aceptar otras envolventes.

```
public Nullable<float> Intersects (BoundingSphere sphere)
```

El condicional **if**, línea 32, selecciona cualquier valor diferente de **null**, para cambiar el estado del booleano, preparado, a verdadero y salir del ciclo **foreach** con la instrucción **break**; en caso contrario se le asigna el valor falso a la variable **preparado**.

El método **Update** se hace llamar desde fuera de la clase, y no está sujeto a alguna interrupción de una entrada estándar o el gamepad, así que siempre que se llame a este método se hará recorrer el arreglo de **ModeloEstatico** para buscar la condición para cambiar la textura. La llamada a este método, pudo hacerse dentro del método **Draw** de esta misma clase, pero se ha dejado así para seguir con la separación de tareas entre los métodos **Draw** y **Update** de la clase **Game**.

Con el método **Draw**, líneas 42 – 51, se dibuja el sprite seleccionado por el booleano, líneas 46 – 49, al final del método **spriteBath.End** se cambia la propiedad **DepthBufferEnable** a verdadero, línea 50, ya que el método **spriteBatch.Draw** lo cambia.

Código 9-9

```
1.      using System;
2.      using Microsoft.Xna.Framework.Graphics;
3.      using Microsoft.Xna.Framework;
4.
5.      namespace TutorialX15
6.      {
7.          public class Mira
8.          {
9.              GraphicsDevice graphicsDevice;
10.             SpriteBatch spriteBatch;
11.             Texture2D texturaA, texturaB;
12.             Rectangle rectangulo;
13.             Boolean preparado;
14.
```

[44] http://msdn.microsoft.com/es-es/library/b3h38hb0.aspx

```
15.             public Mira(Texture2D texturaA, Texture2D texturaB, Int32 ancho,
16.                 Int32 alto, GraphicsDevice graphicsDevice)
17.             {
18.                 this.graphicsDevice = graphicsDevice;
19.                 spriteBatch = new SpriteBatch(graphicsDevice);
20.                 this.texturaA = texturaA;
21.                 this.texturaB = texturaB;
22.                 // se crea el rectangulo y se posiciona al centro de la pantalla
23.                 rectangulo = new Rectangle(
24.                     (int)(graphicsDevice.Viewport.Width - ancho) / 2,
25.                     (int)(graphicsDevice.Viewport.Height - alto) / 2, ancho, alto);
26.             }// fin del constructor
27.
28.             public void Update(Ray rayo, ModeloEstatico[] modelosEstaticos)
29.             {
30.                 foreach (ModeloEstatico modelo in modelosEstaticos)
31.                 {
32.                     if (rayo.Intersects(modelo.EnvolventeEsfera) != null)
33.                     {
34.                         preparado = true;
35.                         break;
36.                     }
37.                     else
38.                         preparado = false;
39.                 }// fin del foreach
40.             }// fin del método Update
41.
42.             public void Draw()
43.             {
44.                 spriteBatch.Begin();
45.                 if(preparado)
46.                     spriteBatch.Draw(texturaB, rectangulo, Color.White);
47.                 else
48.                     spriteBatch.Draw(texturaA, rectangulo, Color.White);
49.                 spriteBatch.End();
50.                 graphicsDevice.RenderState.DepthBufferEnable = true;
51.             }// fin del método Draw
52.         }// fin de la clase
53.     }// fin del namespace
```

9.3.3 La clase Proyectil

La clase **Proyectil**, Código 9-10, busca impactar un rayo con alguna envolvente de esfera, cada vez que se presiona el botón **A** del gamepad.

PlayterIndex, línea 9, es para conocer qué jugador ha disparado el proyectil, y distancia, línea 10, es para conocer el blanco con el que se ha impactado primero. Al constructor solo se le hace pasar el **playerIndex** para inicializar la variable de instancia del mismo nombre.

Update, líneas 17 – 22, es el método en donde se busca impactar el proyectil con alguna BS que se encuentre en su paso. Para eso necesita datos del exterior, como la posición de dónde se hizo el disparo, la dirección del móvil y un arreglo de objetos **ModeloEstatico**, para hacer la búsqueda.

Cada vez que se presiona el botón **A** del Gamepad se llama al método **Impactar**, líneas 30 – 35, el cual ya recibe un rayo y el arreglo de modelos como parámetros. La variable de tipo **Nullable<Int32>**, líneas 32, es para conocer el índice del arreglo en donde se ha impactado el proyectil. El método privado **Indice** regresa dicho valor, a partir del rayo y el arreglo. Una vez encontrado el índice, se cambia la propiedad **Seleccionado** del modelo, línea 34, a verdadero.

Un proyectil no puede impactar a más de un modelo, por eso la distancia mínima entre la posición del disparo y la envolvente debe ser la mínima entre aquellos que puede intersecar. La variable **distMIn**, línea 46, representa dicha situación; **unaVez** es para conocer que al menos se ha encontrado una distancia de la prueba **Intersects** e **indice** indica el valor encontrado del arreglo.

En el bucle **for**, líneas 50 – 59, se hace la prueba **Intersects** del objeto **ray** a cada una de las envolventes, línea 52, el valor arrojado por el método se alojará en **distancia**. El primer **if**, línea 54, selecciona cualquier

valor diferente de **null**, o sea cualquier colisión hallada. En el primer **if** anidado, línea 56, se le asigna a **distMin** del valor en flotate de la prueba de intersección, es importante que el valor de la variable distancia no sea **null**, porque arrojaría un error en tiempo de ejecución; luego se cambia la variable **unaVez** a falso, línea 59, y a **indice** se le asigna el número del elemento con el que encontró colisión, línea 60. Para evitar la prueba del segundo **if** anidado, se hace uso de la instrucción **continue**, línea 61, para seguir con la siguiente iteración del bucle **for**.

En el segundo **if** anidado, líneas 63 – 67, se hace la comparación entre la distancia mínima y la distancia actual encontrada, línea 63; en caso que ésta última sea menor, se cambia el valor de la distancia mínima por la actual, línea 65, y el número del índice se actualiza por el del elemento con el que se colisionó, línea 66.

Al final del ciclo **for**, se regresa el número del índice del arreglo con el que se tiene una distancia mínima, en comparación con otra con la que se haya encontrado.

Código 9-10

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Input;
4.
5.      namespace TutorialX15
6.      {
7.          public class Proyectil
8.          {
9.              PlayerIndex playerIndex;
10.             Nullable<Single> distancia;
11.
12.             public Proyectil(PlayerIndex playerIndex)
13.             {
14.                 this.playerIndex = playerIndex;
15.             }// fin del constructor
16.
17.             public void Update(Vector3 posicion, Vector3 direccion,
18.                 ModeloEstatico[] modeloEstatico)
19.             {
20.                 if (GamePad.GetState(playerIndex).Buttons.A == ButtonState.Pressed)
21.                     Impactar(new Ray(posicion, direccion), modeloEstatico);
22.             }// fin del método Update
23.
24.             /// <summary>
25.             /// Método que selecciona el modelo estático
26.             /// sobre el que se ha impactado la bala.
27.             /// </summary>
28.             /// <param name="rayo"></param>
29.             /// <param name="modeloEstatico"></param>
30.             private void Impactar(Ray rayo, ModeloEstatico[] modeloEstatico)
31.             {
32.                 Int32? i = Indice(rayo, modeloEstatico);
33.                 if (i != null)
34.                     modeloEstatico[i.Value].Seleccionado = true;
35.             }// fin del método
36.
37.             /// <summary>
38.             /// Método que devuelve el índice del modelo estático
39.             /// con el que se ha impactado la bala.
40.             /// </summary>
41.             /// <param name="rayo"></param>
42.             /// <param name="modelos"></param>
43.             /// <returns></returns>
44.             private Nullable<Int32> Indice(Ray rayo, ModeloEstatico[] modelos)
45.             {
46.                 Single distMin = 0.0F;
47.                 bool unaVez = true;
48.                 Int32? indice = null;
49.
50.                 for (Int32 i = 0; i < modelos.Length; i++)
51.                 {
```

```
52.                     distancia = rayo.Intersects(modelos[i].EnvolventeEsfera);
53.                     // signifa que existe una intersección
54.                     if (distancia != null)
55.                     {
56.                         if (unaVez)
57.                         {
58.                             distMin = distancia.Value;
59.                             unaVez = false;
60.                             indice = i;
61.                             continue;
62.                         }
63.                         if (distMin > distancia.Value)
64.                         {
65.                             distMin = distancia.Value;
66.                             indice = i;
67.                         }
68.                     }// fin del if
69.                 }// fin del for
70.                 return indice;
71.             }// fin del método Indice
72.
73.             public override string ToString()
74.             {
75.                 return string.Format("Distancia: {0}", distancia);
76.             }// fin del método ToString
77.
78.         }// fin de la clase
79.     }// fin del namespace
```

9.3.4 La clase ModeloEstatico

La clase **ModeloEstatico**, Código 9-11, se ha aumentado el número de líneas de código, al visto en Bounding Sphere vs. Bounding Sphere, Código 9-2. Por lo tanto solo aquellas nuevas líneas se explicaran.

ColorAmbiental es una variable de instancia pública, línea 13, que ayudará a cambiar el color ambiental del shader predeterminado de XNA. El booleano **Seleccionado**, línea 14, es para indicar que la instancia del **ModeloEstatico** ha sido seleccionado para cambiar el color ambiental. Al ser seleccionado la instancia, el cambio de color ambiental dura un tiempo determinado por la variable de instancia **Tiempo**, línea 15.

El único cambio en el constructor, es en la inicialización de la variable **ColorAmbiental** al color negro, línea 29.

En el método **Draw**, líneas 37 – 54, se agrego el cambio del color ambiental, línea 50, para hacerlo actualizar cada vez que se ha seleccionado la instancia; la propiedad **AmbientLightColor** es de tipo **Vector3** por lo que el **ColorAmbiental** debe convertirse explisitamente a este tipo de dato con el método **ToVector3**.

Se ha agregado el método **Update** con el parámero **gameTime** para regresar el color ambiental de rojo a negro, cada vez que ha pasado un segundo después de ser seleccionado. En el primer **if**, líneas 58 – 62, la variable de instancia **Seleccionado** permite cambiar el color ambiental de negro a rojo e iniciando la suma de tiempo transcurrido a partir de la primera vez que el programa entra en el cuerpo de este condicional. Si el tiempo es mayor a un segundo, en el segundo **if**, líneas 63 – 68, se cambia el color ambiental a negro, la variable **tiempo** se inicia en cero y la variable de instancia **Seleccionado** toma el valor de falso, para no continuar con la suma del tiempo.

Código 9-11

```
1.      using System;
2.      using Microsoft.Xna.Framework.Graphics;
3.      using Microsoft.Xna.Framework;
4.
5.      namespace TutorialX15
6.      {
7.          public class ModeloEstatico
8.          {
9.              private Model modelo;
10.             private Matrix[] transformaciones;
```

```
11.             protected BoundingSphere boundingSphere;
12.             private String nombre;
13.             public Color ColorAmbiental;
14.             public bool Seleccionado;
15.             private Single tiempo;
16.
17.             public ModeloEstatico(Model modelo)
18.             {
19.                 this.modelo = modelo;
20.                 transformaciones = new Matrix[modelo.Bones.Count];
21.                 modelo.CopyAbsoluteBoneTransformsTo(transformaciones);
22.                 boundingSphere = modelo.Meshes[0].BoundingSphere;
23.                 // traslación del centro de la envolvente
24.                 boundingSphere.Transform(
25.                     ref transformaciones[modelo.Meshes[0].ParentBone.Index],
26.                     out boundingSphere);
27.
28.                 nombre = modelo.Meshes[0].Name;
29.                 ColorAmbiental = Color.Black;
30.             }// fin del constructor
31.
32.             /// <summary>
33.             /// Propiedad que obtiene la envolvente de esfera.
34.             /// </summary>
35.             public BoundingSphere EnvolventeEsfera { get { return boundingSphere; } }
36.
37.             public virtual void Draw(ref Matrix world, ref Matrix view,
38.                 ref Matrix projection)
39.             {
40.                 foreach (ModelMesh mesh in modelo.Meshes)
41.                 {
42.                     foreach (BasicEffect basicEffect in mesh.Effects)
43.                     {
44.                         basicEffect.World = transformaciones[mesh.ParentBone.Index]
45.                             * world;
46.                         basicEffect.View = view;
47.                         basicEffect.Projection = projection;
48.                         basicEffect.EnableDefaultLighting();
49.                         basicEffect.PreferPerPixelLighting = true;
50.                         basicEffect.AmbientLightColor = ColorAmbiental.ToVector3();
51.                     }// fin del foreach
52.                     mesh.Draw();
53.                 }// fin del foreach
54.             }// fin del método Draw
55.
56.             public void Update(GameTime gameTime)
57.             {
58.                 if (Seleccionado)
59.                 {
60.                     ColorAmbiental = Color.Red;
61.                     tiempo += (Single)gameTime.ElapsedGameTime.TotalSeconds;
62.                 }
63.                 if ((tiempo) > 1.0)
64.                 {
65.                     ColorAmbiental = Color.Black;
66.                     tiempo = 0;
67.                     Seleccionado = false;
68.                 }
69.             }// fin del método Update
70.
71.             public override string ToString()
72.             {
73.                 return string.Format("{0}\nBoundingSphere.Center {1}\nRadio {2}",
74.                     nombre,
75.                     boundingSphere.Center.ToString(),
76.                     boundingSphere.Radius);
77.             }// fin del método ToString
78.
79.         }// fin de la clase
80.     }// fin del namespace
```

9.3.5 Implementación

Ya para concluir este capítulo, en la clase **Game1**, Código 9-12, se implementan las clases descritas anteriormente.

En las variables de instancia, se ha agregado la cámara libre, líneas 15; la mira, línea 16; y el proyectil, línea 17.

Dentro del método **Initialize**, líneas 24 – 41, se crea la cámara, línea 34, y se inicializa la **sensibilidad** de rotación a tres unidades, línea 37. A la matriz **view**, línea 38, se le asigna el valor de la propiedad **MtxVista** de la cámara. El proyectil también se crea en la este método, **Initialize**, línea 39.

Los modelos estáticos y la mira deben crearse en el método **LoadContent**, líneas 43 – 56, por tener parámetros que el **ContenManager** maneja.

En el método **Update** se hacen llamar a los métodos de actualización de: la cámara, de cada elemento del arreglo de modelos estáticos, del proyectil y de la mira. Véase que la bala y la mira, dependen de la cámara, por lo tanto, deben ir después ésta.

En el método **Draw**, líneas 77 – 87, se actualiza la matriz **view** con la propiedad **MtxVista** de la cámara, y luego se llaman los métodos de dibujado de cada modelo estático y de la mira.

Código 9-12

```
1.      using System;
2.      using Microsoft.Xna.Framework;
3.      using Microsoft.Xna.Framework.Content;
4.      using Microsoft.Xna.Framework.Graphics;
5.      using Microsoft.Xna.Framework.Input;
6.
7.      namespace TutorialX15
8.      {
9.          public class Game1 : Microsoft.Xna.Framework.Game
10.         {
11.             GraphicsDeviceManager graphics;
12.             SpriteBatch spriteBatch;
13.             ModeloEstatico[] modelosEsticos;
14.             Matrix world, view, projection;
15.             CamaraLibre camara;
16.             Mira mira;
17.             Proyectil bala;
18.
19.             public Game1()
20.             {
21.                 graphics = new GraphicsDeviceManager(this);
22.                 Content.RootDirectory = "Content";
23.                 graphics.PreferredBackBufferHeight = 720;
24.                 graphics.PreferredBackBufferWidth = 1280;
25.             }
26.
27.             protected override void Initialize()
28.             {
29.                 world = Matrix.Identity;
30.                 projection = Matrix.CreatePerspectiveFieldOfView(
31.                     MathHelper.PiOver4,
32.                     GraphicsDevice.Viewport.AspectRatio,
33.                     1.0F, 10000.0F);
34.                 camara = new CamaraLibre(new Vector3(0, 10, 350),
35.                     new Vector3(0, 10, 0), Vector3.Up, 4.0F,
36.                     PlayerIndex.One);
37.                 camara.Sensibilidad = 3.0F;
38.                 view = camara.MtxVista;
39.                 bala = new Proyectil(PlayerIndex.One);
40.                 base.Initialize();
41.             }
42.
43.             protected override void LoadContent()
44.             {
```

```
45.                 spriteBatch = new SpriteBatch(GraphicsDevice);
46.
47.                 modelosEsticos = new ModeloEstatico[2];
48.                 modelosEsticos[0] = new ModeloEstatico(Content.Load<Model>("EsferaD"));
49.                 modelosEsticos[1] = new ModeloEstatico(
50.                     Content.Load<Model>("ElefanteA"));
51.
52.                 mira = new Mira(
53.                     Content.Load<Texture2D>("Mira"),
54.                     Content.Load<Texture2D>("Mira2"),
55.                     10, 10, GraphicsDevice);
56.             }
57.
58.             protected override void UnloadContent()
59.             {
60.             }
61.
62.             protected override void Update(GameTime gameTime)
63.             {
64.                 if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
65.                     ButtonState.Pressed)
66.                     this.Exit();
67.
68.                 camara.Update();
69.                 modelosEsticos[0].Update(gameTime);
70.                 modelosEsticos[1].Update(gameTime);
71.                 bala.Update(camara.Posicion, camara.Direccion, modelosEsticos);
72.                 mira.Update(new Ray(camara.Posicion, camara.Direccion),
73.                     modelosEsticos);
74.                 base.Update(gameTime);
75.             }
76.
77.             protected override void Draw(GameTime gameTime)
78.             {
79.                 GraphicsDevice.Clear(Color.White);
80.
81.                 view = camara.MtxVista;
82.                 modelosEsticos[0].Draw(ref world, ref view, ref projection);
83.                 modelosEsticos[1].Draw(ref world, ref view, ref projection);
84.                 mira.Draw();
85.
86.                 base.Draw(gameTime);
87.             }
88.         }
89.     }
```

Ejecute el programa y corrija cualquier error que pudiera encontrarse, e intente de nuevo. En la Ilustración 9-10 se muestra la figura del elefante antes y después de ser cambiado su color ambiental.

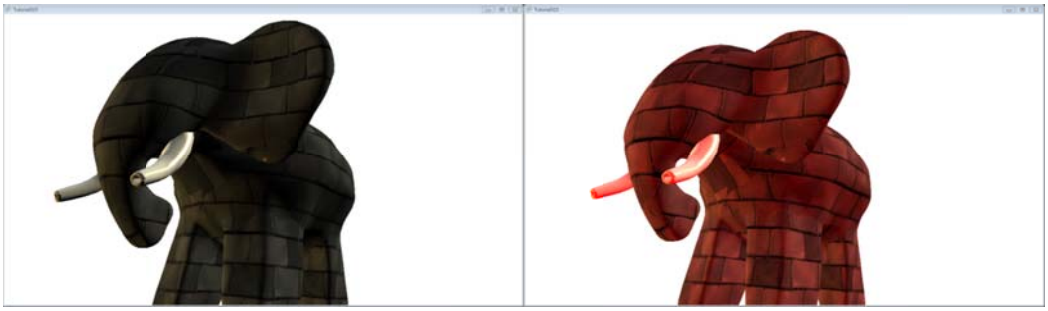

Ilustración 9-10 Cambio de color ambiental

En la Ilustración 9-11 se trato de mostrar que el cambio de color solo afecta a la figura que se encuentre más cerca de la posición de la cámara, de donde se ha disparado el proyectil.

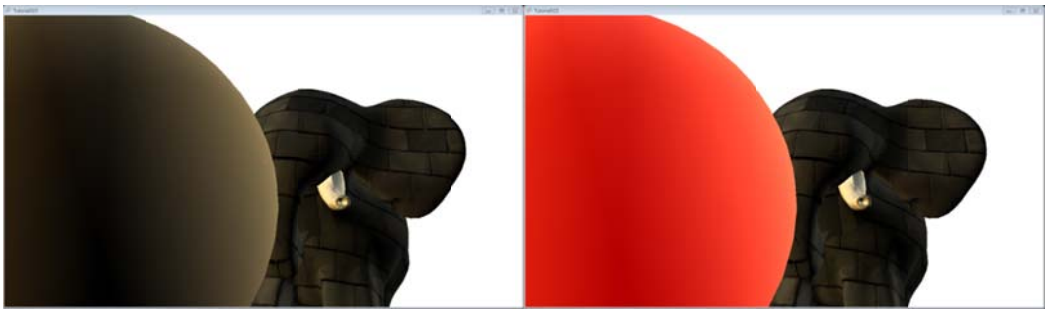

Ilustración 9-11 La colisión existe para el elemento más cercano a la cámara

10 Conclusiones

La elaboración del manual de programación de XNA está cumpliendo su objetivo, ya que desde sus primeras versiones ha sido básico para cumplir con las siguientes metas:

- Impartición de un curso de programación en XNA, dado en el palacio de Minería de la Facultad de Ingeniería, por parte de la División de Educación Continua y a Distancia.
- Demostración de superficies cilíndricas, en plataformas PC, Xbox 360 y Windows Phone 7. Esta se presentó en la semana de Ingeniería, por parte de la División de Educación Continua y a Distancia.
- Visualización de modelos tridimensionales que representan el salón de actos del Palacio de Minería, en plataforma PC y Xbox 360.

Por lo anterior, se demuestra la necesidad de contar con un manual de programación para XNA, que ayude al estudiante en la elaboración de programas interactivos tridimensionales, aplicados en tres dispositivos.

Como parte integral a mi desarrollo profesional, el manual de programación de XNA representó un arduo trabajo de autoaprendizaje, y en medida al desarrollo de los puntos anteriores, me ayudó a conocer las necesidades de los usuarios. En especial de aquellos que están integrándose al mundo de la computación gráfica.

11 Bibliografía

1. **St-Lauret, Sebastien.** *The complete effect and HLSL guide.* Estados Unidos de América : Paradoxal, 2005. ISBN 0-9766132-1-2.

2. **Solis Ubaldo, Rodolfo.** *Geometría Analítica.* México : Limusa, 1992. ISBN 968-837-162-9.

3. **Solar González, Eduardo.** *Álgebra Lineal.* México : Limusa, 2003. ISBN 968-18-5365-2.

4. **Landa Cosio, Nicólas Arrioja.** *DirectX programación de gráficoas 3D.* Buenos Aires, Argentina : Users.code, 2006. ISBN 987-1347-04-9.

5. **Jiménez Delgado, Juan José.** *Detección de Colisiones mediante Recubrimientos Simpliciales.* Granada, España : Universidad de Granada, 2006.

6. **Deitel M., Harvey y Deitel J., Paul.** *Cómo programar C#.* México : Pearson Educación de México, 2007. ISBN 978-970-26-1056-4.

7. **The Competitive Intelligence Unit.** [En línea] [Citado el: 16 de enero de 2011.] http://www.the-ciu.net/ciu_0k/pdf/CIU-Mercado%20de%20Videojuegos%20Mexico%20v01.pdf.

8. **El Economista.** El Economista. [En línea] http://eleconomista.com.mx/tecnociencia/2010/10/22/ser-gamer-cuesta-6818-mexico.

9. **Universidad de Oviedo.** srg. [En línea] http://www.srg.es/files/apendice_tuberia_programable.pdf.

10. **NVIDIA.** nVidia. [En línea] nVidia Corporation, 2011. http://www.nvidia.es/object/IO_20020107_6675.html.

11. **MSDN.** msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-mx/library/bb203925.aspx.

12. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-es/library/bb447756.aspx.

13. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-es/library/b3h38hb0.aspx .

14. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=ES-ES&k=k(MICROSOFT.XNA.FRAMEWORK.BOUNDINGSPHERE.INTERSECTS)&rd=true.

15. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/query/dev10.query?appId=Dev10IDEF1&l=EN-US&k=k(MICROSOFT.XNA.FRAMEWORK.GRAPHICS.EFFECTPARAMETER.SETVALUE);k(DevLang-CSHARP)&rd=true.

16. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/ff471376(v=VS.85).aspx.

17. —. msdn. [En línea] Microsoft Coporation, 2011. ttp://msdn.microsoft.com/en-us/library/bb173347(v=VS.85).aspx.

18. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb509626(v=VS.85).aspx.

19. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb509644(VS.85).aspx.

20. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/ff471376(v=VS.85).aspx.

21. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb509647(v=VS.85).aspx.

22. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb509587(v=VS.85).aspx.

23. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb509706(v=VS.85).aspx.

24. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.blendfunction.aspx.

25. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.renderstate.blendfactor.aspx .

26. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.blend.aspx.

27. —. msdn. [En línea] Microsoft Corporation, 2011. ttp://msdn.microsoft.com/en-us/library/bb174837(VS.85).aspx.

28. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/4z4t9ed1(VS.71).aspx.

29. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb447759.aspx.

30. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.content.contentmanager.aspx.

31. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb197848.aspx.

32. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-es/library/microsoft.windowsmobile.directx.direct3d.texturefilter(VS.85).aspx.

33. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-es/library/microsoft.windowsmobile.directx.direct3d.texturefilter(VS.85).aspx.

34. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.texturefilter.aspx.

35. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.surfaceformat.aspx.

36. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game_members.aspx.

37. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/bb198836.aspx.

38. —. msdn. [En línea] Microsoft Corporation, 2011. http://msdn.microsoft.com/es-es/library/system.aspx.

39. **Nvidia.** Developer zone. [En línea] nVidia, 2008. http://developer.nvidia.com/object/using_sas.html.

40. **3dRender.** 3dRender.com. [En línea] 2006. http://www.3drender.com/light/compositing/index.html.

41. **The Game Development Wiki.** [En línea] http://wiki.gamedev.net/index.php/D3DBook:%28Lighting%29_Direct_Light_Sources.

12 Glosario

Depth Buffer: es un búfer que es de la misma anchura y altura como su render target. Este búfer registra la profundidad de cada píxel que es renderizado. Cuando un píxel es renderizado una segunda vez, como cuando un objeto es renderizado detrás de otro, el Depth Buffer toma cualquier valor anterior de profundidad, o lo reemplaza por el valor de profundidad del segundo píxel. Qué profundidad es preservada y qué profundidad es descartada dependerá de la función de profundidad seleccionada. Por ejemplo, si CompareFunction.LessEqual es la función actual, los valores de profundidad son menores o iguales al valor actual son preservados. Cualquier valor mayor al valor de profundidad actual es descartado. Esto se le llama *depth test*. El depth test se produce cada vez que el píxel es renderizado. Cuando un píxel pasa el depth test, ese color es escrito en el render target y esa profundidad es escrita en el depth buffer.

La profundidad de un píxel se determina basándose en el tamaño de las matrices de vista y proyección seleccionadas para renderizar. Un píxel que toca el plano near de la proyección tiene una profundidad 0. Un píxel que toca el plano far de la proyección tiene una profundidad 1. Como cada uno de los objetos en la escena es renderizado, normalmente los píxeles que estén cerca de la cámara se mantendrán, ya que esos objetos bloquean la visión de los objetos detrás de ellos.

El depth buffer puede contener bits de stencil, por esta razón se le llama a menudo *depth-stencil buffer*. El formato de profundidad describe la composición del depth buffer. El depth buffer siempre es de 32 bits, pero esos bits pueden ser arreglados en diferentes maneras, similar a cómo pueden variar los formatos de textura. Un común formato de profundidad es Depth32, donde todos los 32 bits son reservados para la información de profundidad. Otro formato común es DepthFormat.Depth24Stencil8, donde los 24 bits son reservados para los cálculos de profundidad y 8 bits son usados para el stencil buffer. DepthFormat.Depth24Stencil8Single es un formato inusual donde los 24 bits del depth buffer son dispuestos como un valor de punto flotante.[45]

Renderizado: es la representación final de la imagen, a la que han eliminado las partes ocultas y se le han aplicado modelos de iluminación.

Render Target: Es un búfer donde la tarjeta de video dibuja los píxeles de una escena siendo éste renderizado por un Effect Class.[46]

Shader*:* procedimiento de sombreado e iluminación personalizado que permite al artista o programador especificar el renderizado de un vértice o un píxel.[47]

Sprite: Es un bitmap 2D que es dibujado directamente sin usar las transformaciones del pipeline, luces o efectos.[48]

Stencil Buffer: Es similar al depth buffer. De hecho, éste ocupa parte del depth buffer. El stencil buffer permite a los programadores establecer una función de stencil que probará el valor de referencia stencil, un valor global, contra el valor en el stencil buffer cada momento que el píxel es renderizado.

El resultado del test stencil determina si el valor del color del píxel es escrito en la target render, y si el valor de profundidad de ese píxel es escrito en el depth buffer.[49]

Viewport: Es un rectángulo que define cómo se representa una escena 3D en una venta 2D.[50]

[45] Traducción hecha a partir de la siguiente dirección: http://msdn.microsoft.com/en-us/library/bb976071.aspx
[46] Traducción hecha a partir de la siguiente dirección: http://msdn.microsoft.com/en-us/library/bb976073.aspx
[47] http://www.srg.es/files/apendice_tuberia_programable.pdf
[48] http://msdn.microsoft.com/en-us/library/bb203919.aspx
[49] Traducción hecha a partir de la siguiente dirección: http://msdn.microsoft.com/en-us/library/bb976074.aspx
[50] http://msdn.microsoft.com/es-es/library/microsoft.windowsmobile.directx.direct3d.viewport(VS.90).aspx