# 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**.

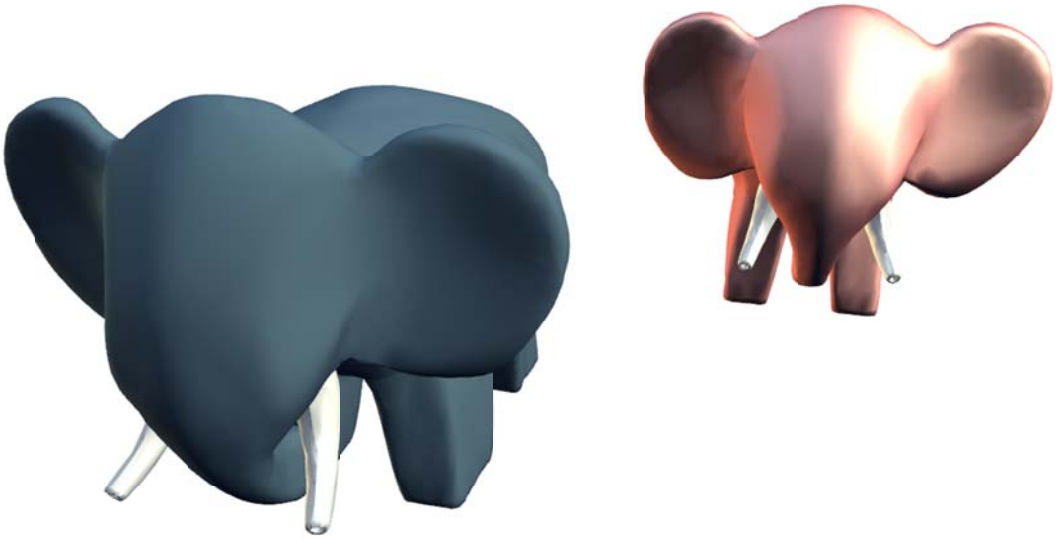

**Ilustración 6-1 Tres ModelMeshpart diferentes**

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.

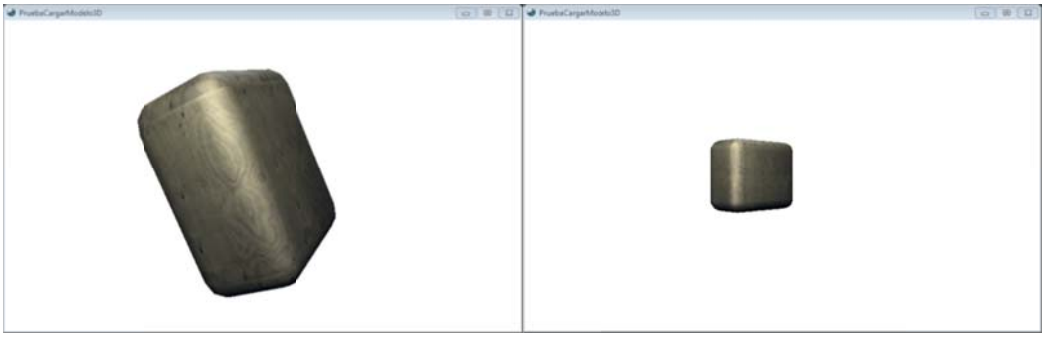

Ilustración 6-2 Posición Correcta e incorrecta

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.

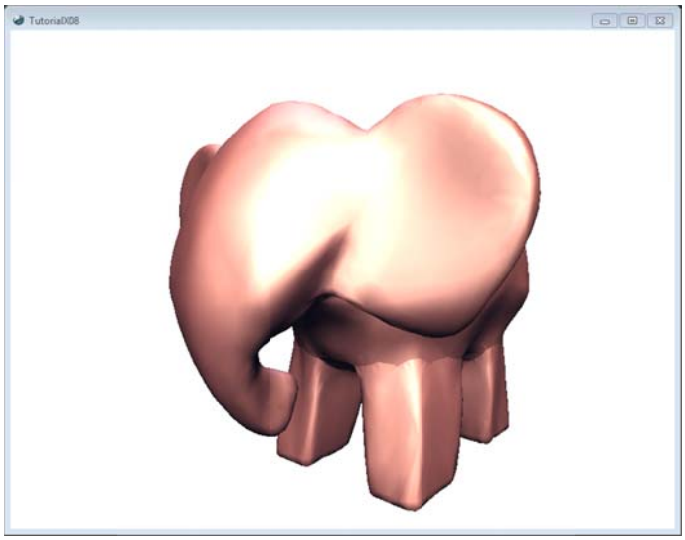


**Ilustración 6-3 Un mesh**

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.

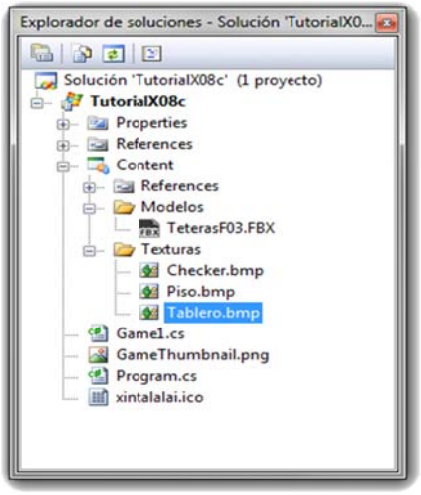


Ilustración 6-4 Agregando texturas en el Content

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

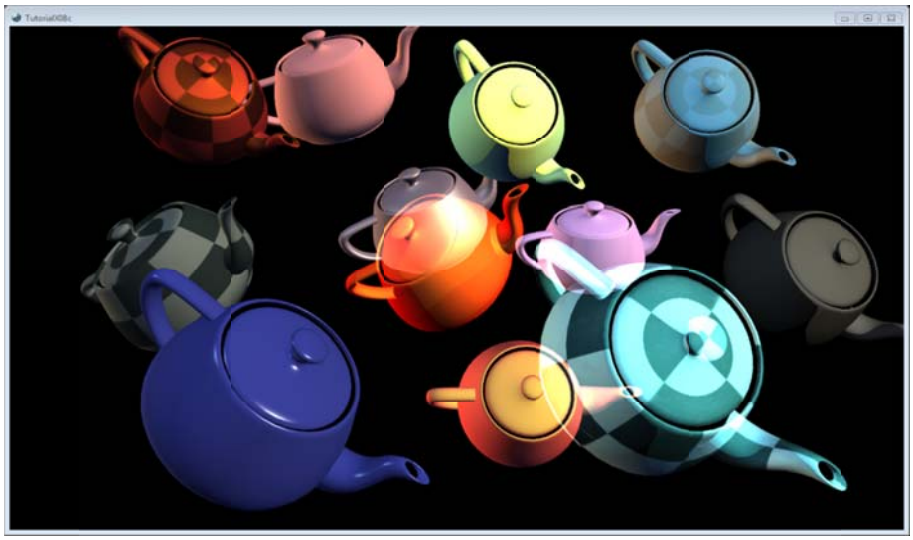

Ilustración 6-7 Blending

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