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.

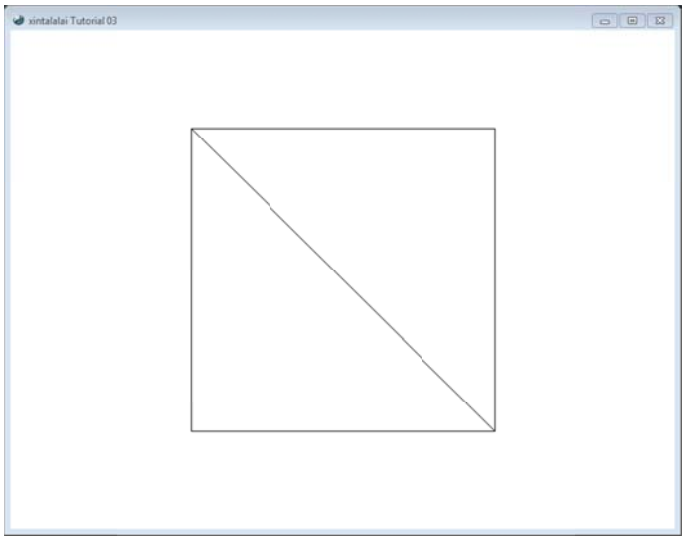

Ilustración 3-1 Cubo

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.

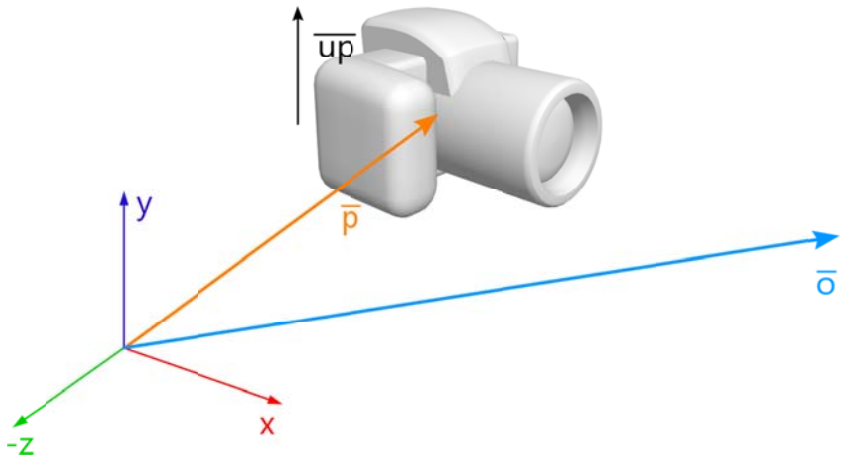

Ilustración 3-2 Parámetros del método CreateLookAt

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.

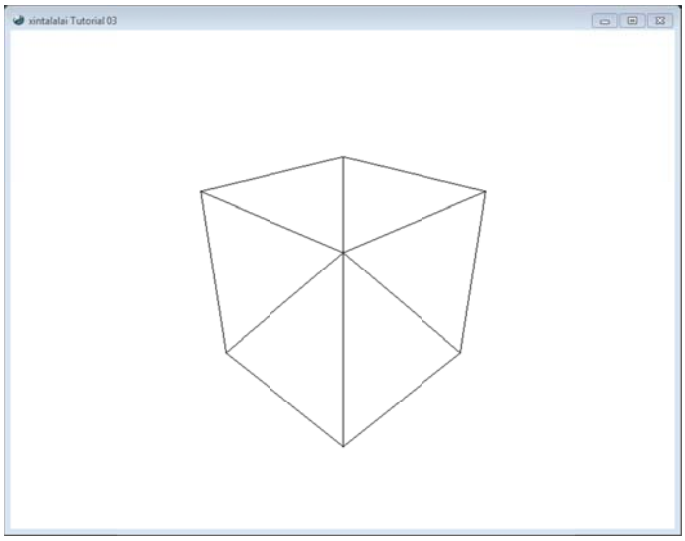

Ilustración 3-4 Cambio de posición

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.

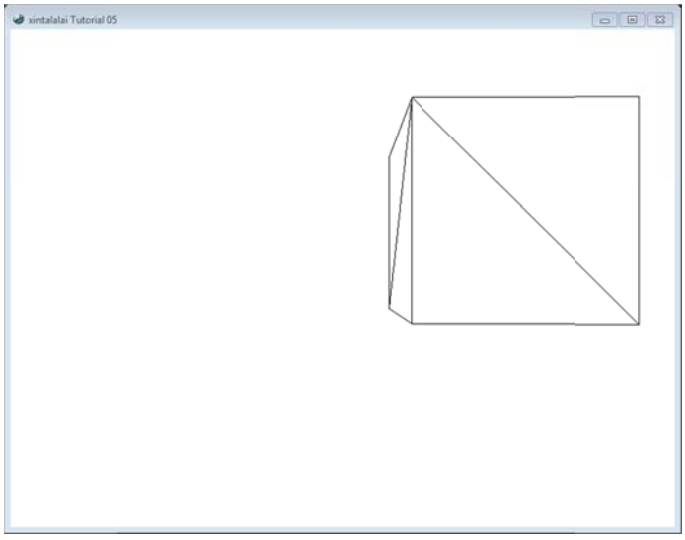

Ilustración 3-5 Traslación

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.

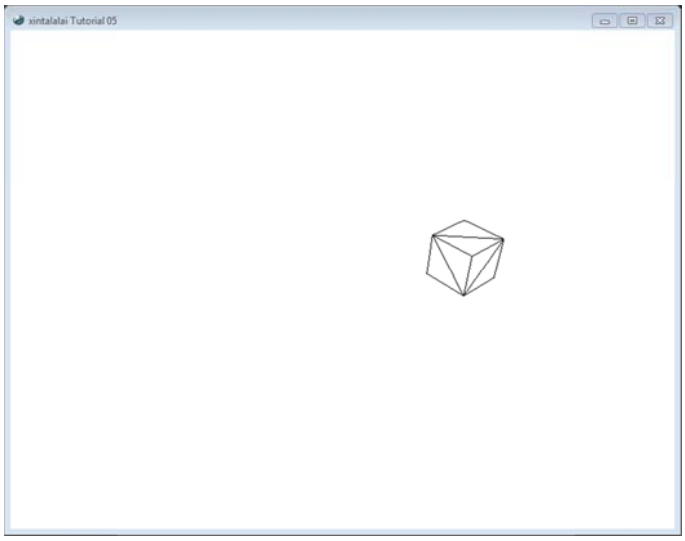

Ilustración 3-6 Escala

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.

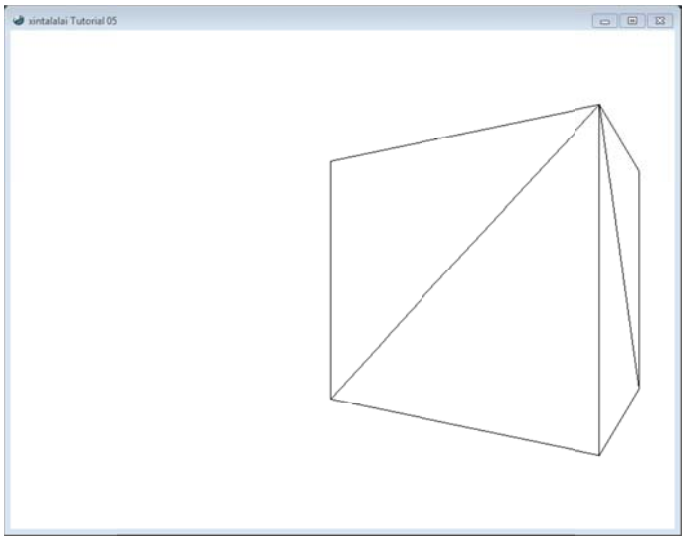

Ilustración 3-7 Rotar y trasladar

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.

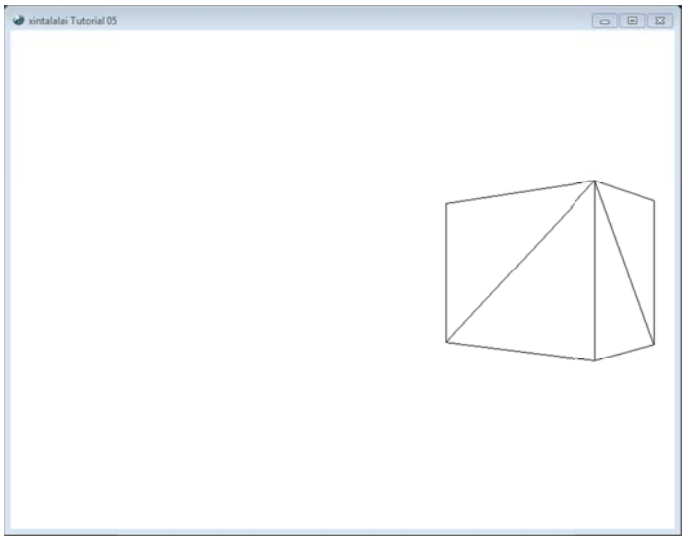

Ilustración 3-8 Rotar y trasladar

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.