# 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.

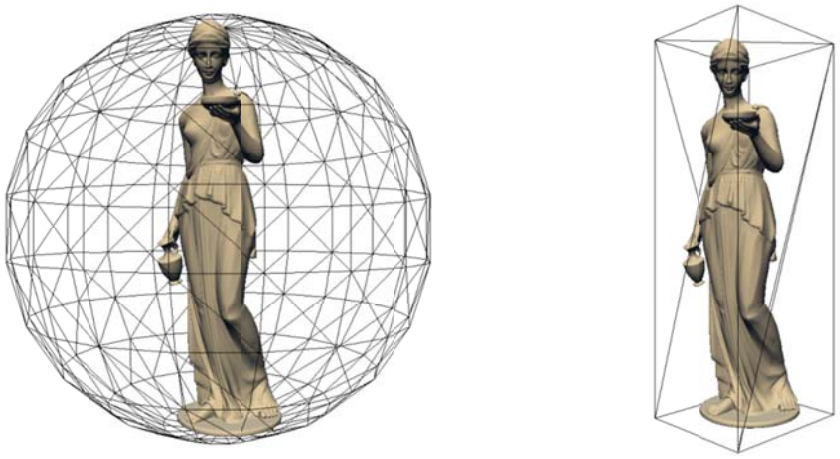

**Ilustración 9-1 Envolvente de colisión**

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.

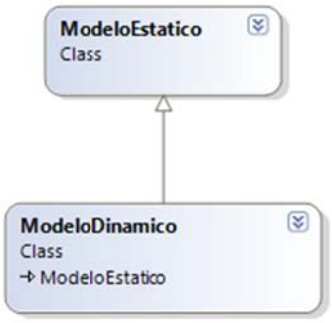


Ilustración 9-2 Diagrama de clase. Modelos estático y dinámicos.

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 $\bar{c}_A$.

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

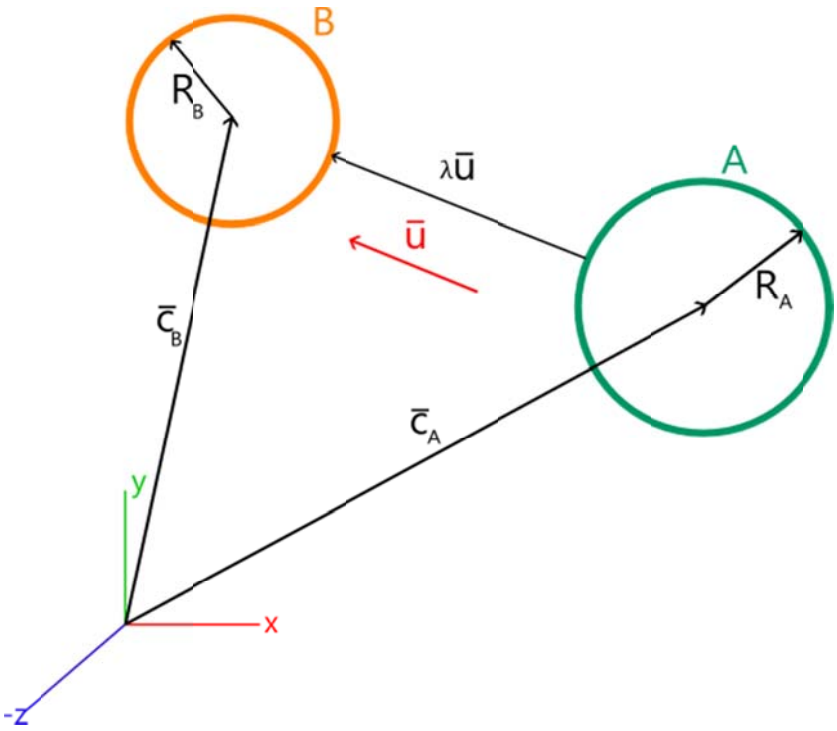


Ilustración 9-3 Distancia entre dos esferas

El valor de $\lambda$ 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.

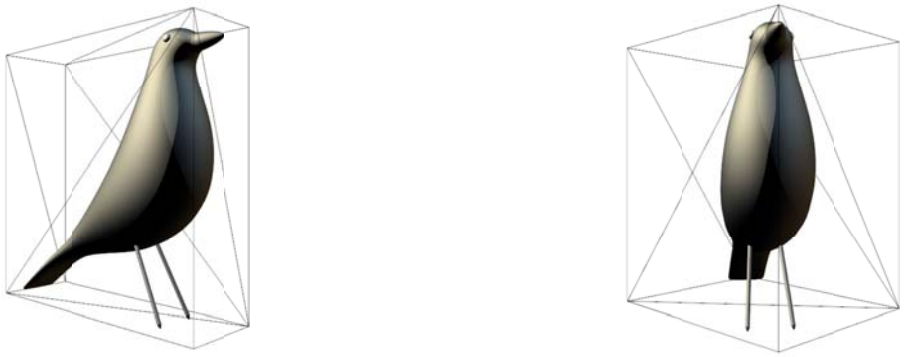

Ilustración 9-5 AABB

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**.

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

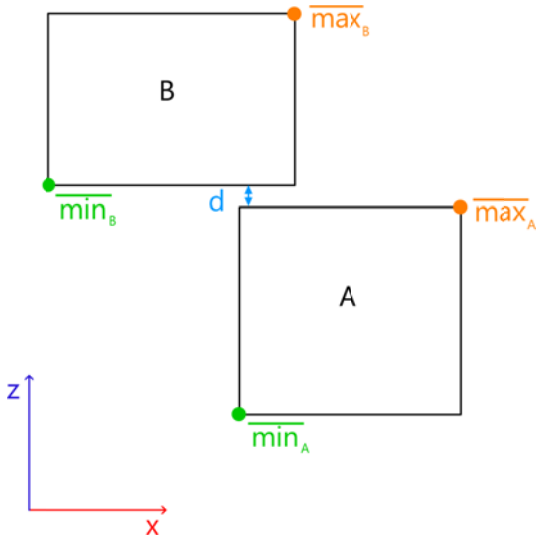


Ilustración 9-6 Distancia entre AABBs

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.

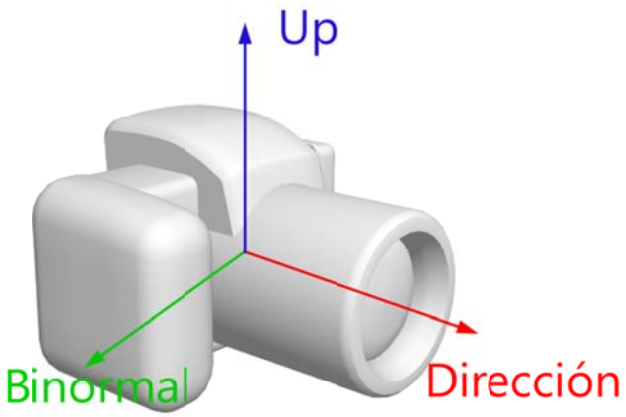


Ilustración 9-8 Vectores de la cámara

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.

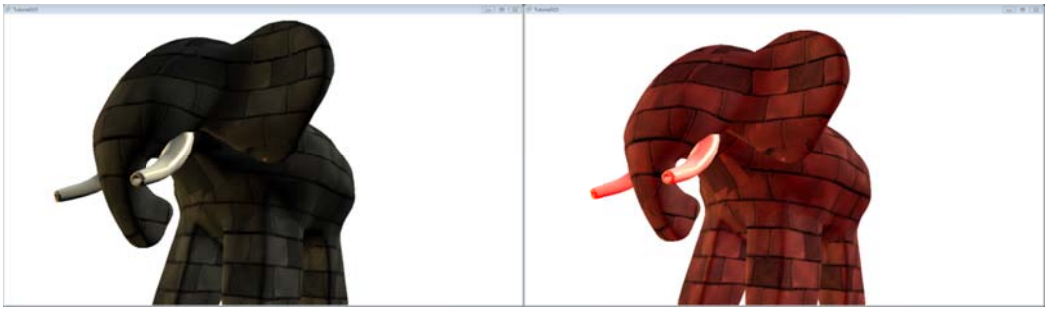

Ilustración 9-10 Cambio de color ambiental

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.

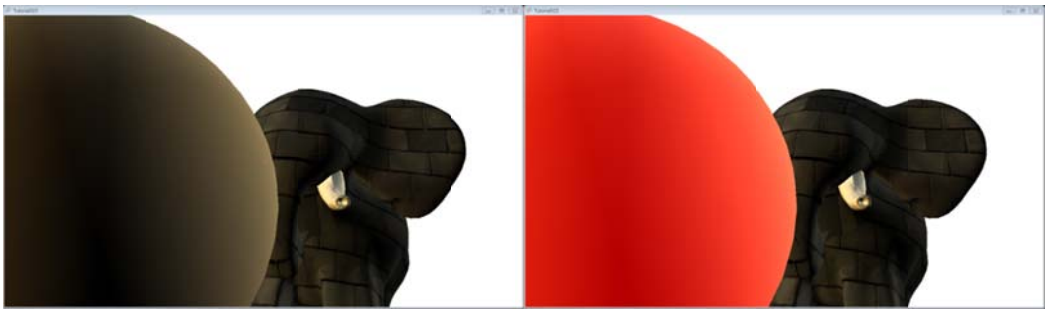

**Ilustración 9-11 La colisión existe para el elemento más cercano a la cámara**