Desarrollo de aplicaciones inteligentes para el movimiento del cuerpo humano

Armandina J. Leal
Eduardo A. Torres Alejandro
Eduardo L. Vázquez Nieto

EDITORIAL DIGITAL
TECNOLÓGICO DE MONTERREY

Primera edición
De venta en: Amazon Kindle, Apple Books, Google Books y Amazon.
Fragmento editado, diseñado, publicado y distribuido por el Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 |
Monterrey, Nuevo León | México.

Acerca de este eBook

Desarrollo de aplicaciones inteligentes para el movimiento del cuerpo humano

Armandina J. Leal Flores | Eduardo A. Torres Alejandro | Eduardo L. Vázquez Nieto

El Tecnológico de Monterrey presenta su colección de eBooks de texto para programas de nivel preparatoria, profesional y posgrado. En cada título se integran conocimientos y habilidades que utilizan diversas tecnologías de apoyo al aprendizaje.

El objetivo principal de este sello es el de divulgar el conocimiento y experiencia didáctica de los profesores del Tecnológico de Monterrey a través del uso innovador de los recursos. Asimismo, apunta a contribuir a la creación de un modelo de publicación que integre en el formato de eBook, de manera creativa, las múltiples posibilidades que ofrecen las tecnologías digitales.

Con la Editorial Digital, el Tecnológico de Monterrey confirma su vocación emprendedora y su compromiso con la innovación educativa y tecnológica en beneficio del aprendizaje de los estudiantes dentro y fuera de la institución.

ebookstec@itesm.mx

Acerca de los autores

Armandina Juana Leal Flores

Profesora del Tecnológico de Monterrey (Campus Monterrey). Egresó de la Ingeniera en Sistemas Electrónicos posteriormente cursó la Maestría en Ciencias Computacionales con especialidad en Sistemas Expertos y la Maestría en Administración en Tecnologías de Información con especialidad en Tecnología Educativa por la cual recibió la Mención Honorífica de Excelencia. Actualmente, aparte de ser profesora titular, se desempeña como Directora Regional Asociada del Departamento de Computación Región Norte del Tecnológico de Monterrey, Campus Monterrey. Se distingue por sus trabajos de innovación educativa, presentados en congresos nacionales e internacionales. Recientemente fue reconocida como Profesor Inspirador 2017 en su campus. Promueve en sus alumnos la innovación y la responsabilidad social a través del tutorado de equipos que han alcanzado los primeros lugares a nivel nacional del concurso Microsoft Imagine Cup. Entre sus áreas de especialización se encuentra la enseñanza de la programación.

Eduardo Alan Torres Alejandro

Es originario de Monterrey, Nuevo León. Impulsado por su padre y por la innovación y el

desarrollo tecnológico, está cursando la carrera de Ingeniería en Mecatrónica en el Tecnológico de Monterrey, Campus Monterrey. Es asistente del Departamento de Computación. Ha impartido cursos a niños y es parte de la Coordinación General del servicio social de Mini robótica para Adolescentes desde el 2016 hasta la fecha. Además, fue miembro de unos de los cinco equipos ganadores a nivel nacional del concurso Microsoft Imagine Cup, con lo que obtuvo una experiencia enriquecedora.

Eduardo Luis Vázquez Nieto

Egresó de la Ingeniería en Tecnologías Electrónicas del Tecnológico de Monterrey. Es originario de Papantla, Veracruz. En su grado universitario trabajó como asistente en el Departamento de Ciencias Computacionales, actualmente es desarrollador *full-stack* en Territorium Life, en el área de “Desarrollo de *Software* a la Medida”.

Durante la preparatoria obtuvo un grado técnico en Mecánica y maquinas herramienta en el Centro de Bachillerato Tecnológico Industrial y de Servicios #78 (CBTIS 78). Desde entonces se apasionó por el desarrollo de aplicaciones de *software*, haciendo uso de lenguajes de programación desde alto a bajo nivel, trabajando igualmente con diversos dispositivos programables y en el desarrollo de sistemas de *hardware* de aplicación específica. Siempre ha tenido gran interés en las áreas de programación embebida, aplicaciones *desktop* y desarrollo *web*.

Índice

Introducción

¿Te interesa aprender a desarrollar aplicaciones? ¿Quieres conocer los fundamentos para la construcción de juegos? ¿Deseas aprender a programar el Kinect? ¿Quieres saber cómo establecer comunicación desde tu programa con un Arduino? Aún si conoces muy poco de programación, este libro es para ti.

Desde los primeros capítulos, a través de explicaciones sencillas y muchos ejemplos irás aprendiendo a desarrollar programas básicos en C# que te permitirán manipular imágenes, agregar audio y crear tus propias animaciones para hacer juegos o cualquier otro tipo de herramienta. También podrás agregar los elementos requeridos para que tus programas se comuniquen con Kinect, y así, desarrollar aplicaciones interesantes relacionadas con el movimiento del cuerpo o expresiones faciales e incluso tomar fotografías y reconocer comandos de voz.

La organización de este libro hace sencilla la tarea de encontrar explicaciones y ejemplos en específico. Si ya cuentas con conocimientos de C# será posible empezar por cualquier capítulo, o bien, revisar solamente los ejemplos del tipo de Kinect que te interesen.

Los ejemplos fueron elaborados a través de Visual Studio C# en WPF (Windows Presentation Foundation) para que puedas crear interfaces gráficas vistosas gracias a sus múltiples elementos multimedia acompañadas de funciones pre construidas que permiten manipularlos fácilmente.

No podría faltar un capítulo que incluya lo básico para establecer comunicación con el Arduino desde un programa en C# ya que esto permitirá crear aplicaciones mucho más interesantes que controlen dispositivos externos a la computadora.

Acompaña al libro una sección de anexos que muestra programas completos (plantillas), estos se pueden tomar como base para desarrollar aplicaciones. También encontrarás la solución de todos los ejemplos del libro de tal manera que no te pierdas la oportunidad de verlos en ejecución.

Capítulo 1. Imágenes y figuras en movimiento

01

Al construir aplicaciones relacionadas con el movimiento del cuerpo, es común que se empleen imágenes y figuras cuya presentación y animación produzcan efectos visuales acorde al movimiento. El lenguaje XAML y C# proporcionan elementos que facilitan la expresión de dichos efectos. Este capítulo aborda herramientas y estrategias que en combinación pueden ser utilizadas para esto.

1.1 Componentes de un programa

Para construir una aplicación en el lenguaje C#, primero hay que seleccionar cuál tipo de proyecto se va a emplear. Windows Presentation Foundation (WPF) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - Graphical User Interfaces) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo dicho proceso.

Un proyecto WPF está conformado por archivos separados en varias carpetas. Algunos de ellos son:

- **MainWindow.xaml**: contiene el código en el lenguaje **XAML** (Extensible Application Markup Language) el cual se utiliza para diseñar la apariencia de la aplicación. El programador define los objetos como las imágenes, figuras, audio y video que serán mostrados en la pantalla así como sus **propiedades** (características). También especificará los **eventos** que son las acciones que deben ser detectadas cuando el proyecto está en ejecución.
- **MainWindow.xaml.cs**: incluye las instrucciones en el lenguaje C# que emplea el programador para definir el comportamiento de la aplicación acorde a lo diseñado en XAML. Este programa hace uso de las bibliotecas y referencias ya existentes, lo cual facilita la creación de la una aplicación vistosa y eficiente.
- **Recursos**: son archivos como imágenes y audio que empleará la aplicación. Se agregan al proyecto para hacer referencia a ellos dentro de las instrucciones tanto de XAML como de C#.

1.2 Paneles en XAML

Inicialmente el programa en XAML está compuesto por una ventana <Window>…</Window> la cual contiene un panel de tipo <Grid>…</Grid>.

```
<Window x:Class="Ejemplo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:local="clr-namespace:Ejemplo"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

Los elementos del programa son marcas encerradas por los símbolos “<” y “>”. Cada objeto (también denominado control o elemento) que se quiera colocar en la ventana debe tener dos marcas: una para definir el inicio de la descripción del objeto, por ejemplo **<Window>**, y otra para dar por terminada dicha descripción, **</Window>**.

Las marcas van acompañadas de propiedades y los eventos que tiene el objeto que se está definiendo. Las propiedades dan forma al objeto ya que establecen características como su color y tamaño. Los eventos, por su parte, establecen acciones relacionadas con métodos del programa en C# que deben ser ejecutadas cuando ocurre algo como el clic de un botón, el movimiento del cursor o cuando se presiona una tecla, por mencionar algunos.

La construcción del programa en XAML se logra al escribir cada una de las propiedades y eventos de los objetos; sin embargo, es muy útil aprovechar las herramientas que proporciona Visual Studio (como el cuadro de herramientas y la ventana de propiedades) ya que al utilizarlas en el programa se va escribiendo automáticamente.

Inicialmente el programa en XAML contiene solo dos objetos **<Window>** y **<Grid>**. **<Window>** indica las características de la ventana que se muestra al ejecutar el programa. XAML proporciona diferentes tipos de paneles, el que aparece por defecto es el panel **<Grid>**. Este se recomienda para acomodar objetos dentro de la ventana.

Otro tipo de panel es el **<Canvas>**. Este es muy útil cuando se desea que los objetos

cambien de lugar durante la ejecución de la aplicación. Si se desea mover los objetos que están dentro de él, es fundamental que el **Canvas** reciba un nombre x:Name y se especifique su ancho Width y su alto Height. Si no se definen las dimensiones, al momento de ejecutar la aplicación no se realizará el movimiento de los objetos, aun cuando las instrucciones de C# lo soliciten.

```
<Window x:Class="Ejemplo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" xmlns:local="clr-namespace:Ejemplo"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
       <Grid>
        <Canvas x:Name ="MainCanvas" Height="300" Width="250">
        </Canvas>
       </Grid>
</Window>
```

1.3 Ubicar objetos dentro de Canvas

Dentro del Canvas es posible ubicar cualquier tipo de objeto (elipses, rectángulos, imágenes, etc.) y asignarle propiedades y eventos.

Los objetos que se agregan deben tener un nombre x:Name, un ancho Width y un alto Height. La siguiente imagen muestra un ejemplo de cómo son representadas las dimensiones del panel y de la elipse en el código XAML de la izquierda.

La elipse está rodeada de un rectángulo; este no se ve, pero debe ser tomado en cuenta ya que las dimensiones de los objetos son rectangulares.

```
<Grid>
  <Canvas x:Name="MainCanvas"
```

```
        Height="250"
        Width="300">
    <Ellipse x:Name="circulo"
        Height="45"
        Width="45"/>
  </Canvas>
</Grid>
```

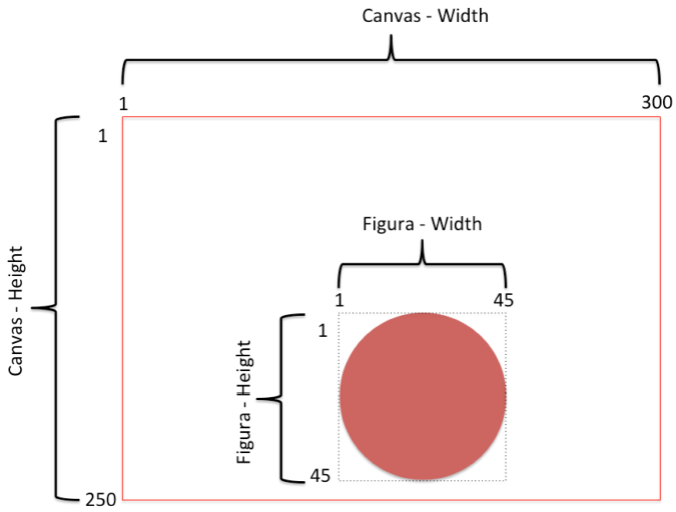


Figura 1.1

Los objetos también deben ubicarse dentro del panel. Las propiedades Canvas.Left y Canvas.Top indican el lugar donde se localiza la esquina superior izquierda de la figura dentro del **Canvas**. Es importante mirar que las posiciones dentro del **Canvas** se numeran a partir de 1.

```
<Grid>
  <Canvas x:Name="MainCanvas"
        Height="250"
        Width="300">
  <Ellipse x:Name="circulo"
        Height="45"
        Width="45"
```

```
            Canvas.Left="100"
            Canvas.Top="150" />
    </Canvas>
</Grid>
```

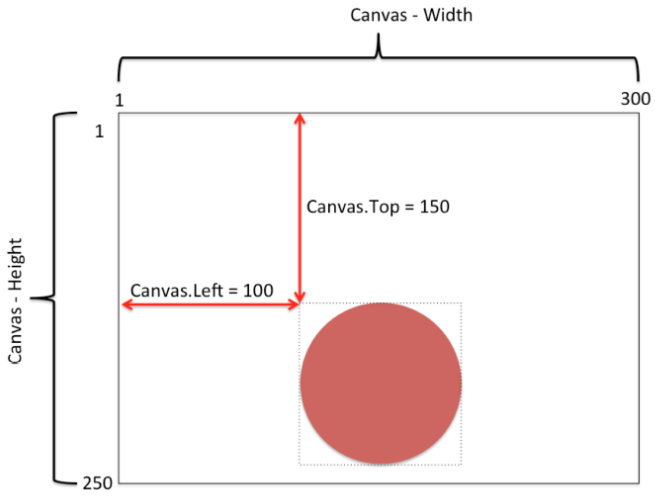


Figura 1.2

Movimiento de objetos

Un objeto dentro del **Canvas** puede cambiar de lugar al definir un evento que indique el momento en el que se debe mover y escribiendo código, ya sea en XAML o en C#, que determine la forma en la que se va a realizar.

Si lo que se desea es reubicar el objeto con instrucciones en C#, lo primero que se debe hacer es enfocar el Canvas para que pueda recibir eventos y hacer modificaciones a su contenido.

El siguiente ejemplo, muestra la forma de enfocar el panel cuyo nombre es **MainCanvas**, cambiando la propiedad Focusable para que permita el enfoque y después encauzarlo con el método Focus. Es importante observar que estas instrucciones están colocadas dentro del método MainWindow el cual es el primero que se ejecuta cuando se corre la aplicación.

```
public MainWindow ()
{
InitializeComponent();
```

```
//Establecer que el Canvas puede ser enfocado
MainCanvas.Focusable = true;
    MainCanvas.Focus ();
```

El método GetValue se emplea para obtener la posición (**dPosX**, **dPosY**) de la esquina superior izquierda del objeto dentro del **Canvas**. En el siguiente ejemplo, la variable dPosX guardará la cantidad de píxeles que hay a la izquierda de la figura círculo y la variable dPosY, la cantidad de píxeles que se encuentran arriba de la misma figura.

```
double dPosX = (double) circulo.GetValue(Canvas.LeftProperty);
double dPosY = (double) circulo.GetValue(Canvas.TopProperty);
```

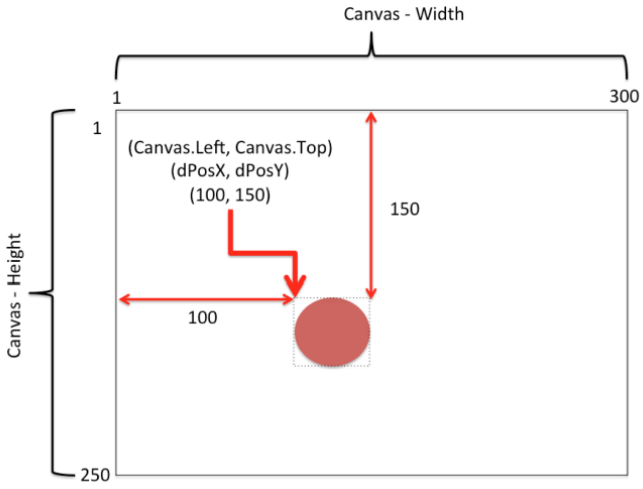


Figura 1.3

Para cambiar de lugar la figura se utiliza el método SetValue indicando la nueva ubicación. Por ejemplo, para moverlo cinco píxeles a la derecha y cinco abajo de la posición anterior se haría lo siguiente:

```
//Mover la figura círculo 5 píxeles a la derecha y 5 píxeles hacia abajo
circulo.SetValue (Canvas.LeftProperty , dPosX + 5);
circulo.SetValue(Canvas.TopProperty , dPosY + 5);
```

Para desplazar el objeto hacia la izquierda o hacia la derecha se debe modificar el valor de la propiedad Canvas.LeftProperty. De la misma forma, para mover el objeto hacia arriba o hacia abajo, la propiedad Canvas.TopProperty es la que debe cambiar de valor.

```
//Mover la figura círculo cinco píxeles a la izquierda
double dPosX = (double) circulo.GetValue(Canvas.LeftProperty) ;
```

```
dPosX = dPosX - 5;
circulo.SetValue (Canvas.LeftProperty , dPosX);
//Mover la figura círculo cinco píxeles a la derecha
double dPosX = (double) circulo.GetValue(Canvas.LeftProperty) ;
dPosX = dPosX + 5;
circulo.SetValue(Canvas.LeftProperty , dPosX);
//Mover la figura círculo cinco píxeles hacia arriba
double dPosY = (double) circulo.GetValue(Canvas.TopProperty) ;
dPosY = dPosY - 5;
circulo.SetValue(Canvas.TopProperty , dPosY);
//Mover la figura círculo cinco píxeles hacia abajo
double dPosY = (double) circulo.GetValue(Canvas.TopProperty) ;
dPosY = dPosY + 5;
circulo.SetValue(Canvas.TopProperty , dPosY);
```

1.4 Controlar el movimiento

Cuando se cambia de lugar un objeto, este puede exceder el tamaño de la ventana. Hay que agregar al programa instrucciones que detecten el tamaño, para lograrlo, es necesario contar con las dimensiones del **Canvas**. Aunque sea conocido el ancho y alto del panel, lo ideal es utilizar las propiedades para obtener sus dimensiones al momento de ejecución, ya que de esta forma, si el **Canvas** cambia de tamaño, los ajustes se realizarán de manera automática. Si el panel se llama **MainCanvas**, se haría de la siguiente forma:

```
//Obtener las dimensiones del Canvas
double dAnchoCanvas = MainCanvas.Width ;
double dAltoCanvas = MainCanvas.Height ;
```

Ya conociendo la cantidad de píxeles que tiene el panel, lo siguiente es verificar si el objeto permanecerá dentro de él antes de hacer el movimiento. Si el objeto se va a mover hacia la izquierda se debe evitar posicionarlo en un valor menor a uno (este sería la orilla izquierda del **Canvas**). Por lo tanto, si la posición de la esquina superior izquierda del

objeto (dPosX) menos la cantidad de píxeles a mover hacia la izquierda (iPixeles) todavía da un valor mayor a cero, quiere decir que aún hay espacio para realizar el movimiento:

```
//Verificar si se puede mover a la izquierda
if (dPosX - iPixeles > 0)
{
    //Mover la figura círculo a la izquierda iPixeles.
    dPosX = dPosX – iPixeles;
    circulo.SetValue(Canvas.LeftProperty, dPosX);
}
```

Ahora bien, lo mismo sucede si el movimiento es hacia arriba, donde la cifra uno también representa a la orilla superior del Canvas. Entonces, cuando la posición de la esquina superior izquierda del objeto (dPosY) menos la cantidad de píxeles a mover hacia arriba (iPixeles) aún da un valor mayor a cero quiere decir que hay espacio para realizar el movimiento:

```
//Verificar si se puede mover hacia arriba
if (dPosY - iPixeles > 0)
{
    //Mover la figura círculo hacia arriba iPixeles.
    dPosY = dPosY - iPixeles;
    circulo.SetValue(Canvas.TopProperty, dPosY);
}
```

Hay que verificar que la figura permanezca dentro del **Canvas** cuando se mueva hacia la derecha o hacia abajo. Esto requiere más cálculos ya que ahora se debe considerar el ancho y el alto del objeto. Por ejemplo, si el movimiento se debe hacer hacia la derecha, hay que evitar que el objeto salga del panel es decir, si a la ubicación de su esquina superior izquierda (dPosX) se le suma el ancho (circulo.Width) y la cantidad de píxeles que se quiere mover (iPixeles) y todavía genera un valor menor al ancho del **Canvas** entonces sí será posible realizar el movimiento.

```
//Verificar si se puede mover a la derecha
if (dPosX + circulo.Width + iPixeles < iAnchoCanvas)
```

```
{
    //Mover la figura círculo a la derecha iPixeles.
    dPosX = dPosX + iPixeles;
    circulo.SetValue(Canvas.LeftProperty, dPosX);
}
```

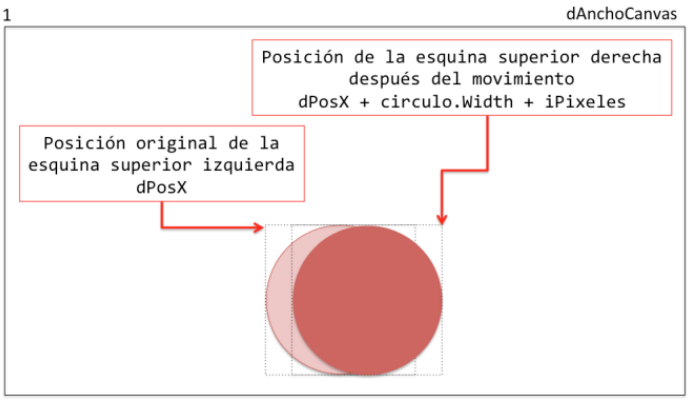


Figura 1.4

Cuando el movimiento es hacia abajo se tiene una situación similar. Para evitar que el objeto desaparezca por la parte inferior de la ventana, el resultado de la suma de la posición de su esquina superior izquierda (dPosY) más su altura (circulo.Height) más la cantidad de píxeles que se van a mover (iPixeles) debe ser un valor menor a la altura del Canvas.

```
//Verificar si se puede mover hacia abajo
if (dPosY + circulo.Height + iPixeles < iAltoCanvas)
{
    //Mover la figura círculo hacia abajo iPixeles.
    dPosY = dPosY + iPixeles;
    circulo.SetValue(Canvas.TopProperty, dPosY);
}
```

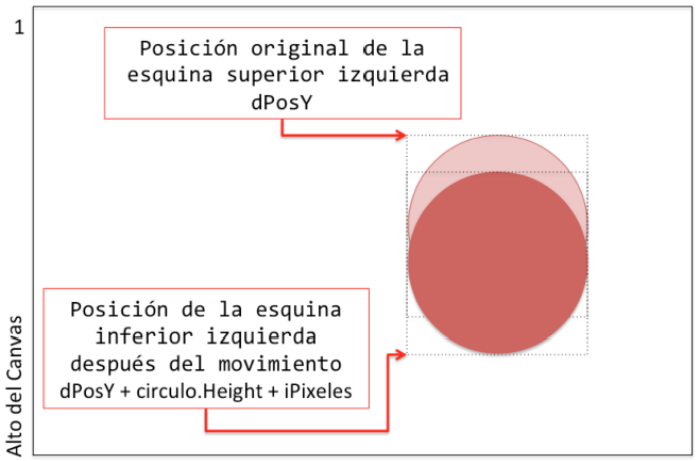


Figura 1.5

1.5 Ejemplo: mover una figura con las teclas flecha

Este programa permite al usuario mover una figura en todas direcciones a través de las teclas flecha. La aplicación limita el movimiento de la figura al área del **Canvas**.

Requerimientos

- Desarrollado en WPF

Código en XAML

El archivo **MainWindow.xaml** muestra al panel de nombre MainCanvas con un ancho de 300 y una altura de 250. Como las dimensiones del **Canvas** son menores a las de la ventana, se agregó color al fondo para que se distingan más fácilmente los límites del área por la que se movería la figura.

También incluye el evento KeyDown el cual se activa cuando el usuario presiona una tecla, haciendo que se ejecute el método MainCanvas_KeyDown que está dentro del programa en C#.

```
<Grid>
  <Canvas x:Name="MainCanvas " KeyDown
 ="MainCanvas_KeyDown "
        Width="300 " Height="250 " Background="#FFF2F3F3">
    <Ellipse x:Name="circulo " Fill="#FFF74A4A"
```

```
            Height="45 " Width="45 "
            Canvas.Left="5 " Canvas.Top="200 "/>
    </Canvas>
</Grid>
```

Dentro del panel se encuentra una elipse que se distingue por el nombre círculo la cual tiene un ancho y alto de 45. La esquina superior izquierda de la figura está localizada a 5 píxeles de la orilla izquierda y a 200 píxeles de la parte superior del **Canvas**.

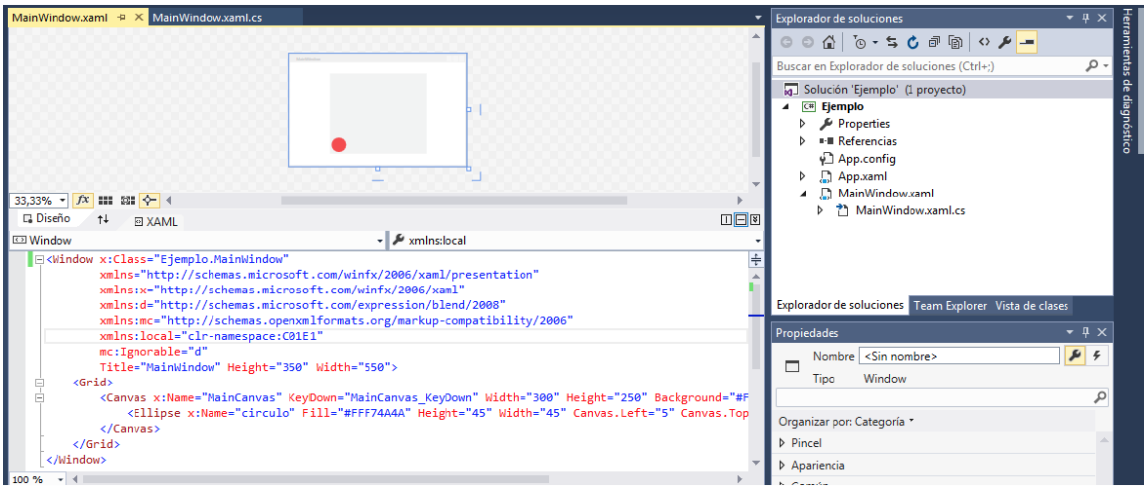


Figura 1.6

Código en C#

El código del archivo **MainWindow.xaml.cs** usa la variable **iPixeles** para determinar la cantidad de píxeles que se moverá la figura cada vez que se presione una tecla flecha. El valor que se le asigne a esta variable depende de cuánto se desea que avance el objeto; se recomienda probar con diferentes valores para verificar cuál de ellos refleja un movimiento más fluido.

También incluye la declaración de las variables que funciona para almacenar el ancho y el alto del **Canvas**. Estas variables se declaran dentro de la clase antes del método **MainWindow** para que puedan ser utilizadas en cualquiera de los métodos del proyecto.

```
public partial class MainWindow : Window
{
    //Cantidad de píxeles que se mueve la elipse en cada ocasión
    int iPixeles = 5;
```

```
        //Variables que almacenarán las dimensiones del Canvas
        double dAnchoCanvas, dAltoCanvas;
    public MainWindow()
    {
```

Como primera instrucción, justo después de inicializar el componente, se tiene que establecer la propiedad **Focus** del panel. Este enfoque permitirá realizar eventos. Hecho lo anterior, se pueden obtener las dimensiones del **Canvas**.

```
    public MainWindow()
    {
        InitializeComponent();
        //Enfocar el Canvas
        MainCanvas.Focusable = true;
        MainCanvas.Focus();
        //Obtener el ancho y la altura del Canvas
        dAnchoCanvas = MainCanvas.Width;
        dAltoCanvas = MainCanvas.Height;
```

El método MainCanvas_KeyDown se ejecuta cada vez que el usuario presiona cualquier tecla. Lo primero que hace es obtener la ubicación de la figura **círculo** donde la variables dPosX y dPosY corresponden a la posición de la esquina superior izquierda del objeto. Posteriormente, se emplea el valor del argumento e para determinar cuál tecla se presionó: e.Key. Si lo que se presionó fue alguna de las flechas (Key.Left, Key.Right, Key.Up y Key.Down) se calcula la nueva posición que debe tener la figura siempre y cuando la elipse se muestre completa dentro del **Canvas**. Finalmente, se realiza el movimiento de la figura hacia la nueva coordenada.

```
    private void MainCanvas_KeyDown (object sender, KeyEventArgs e )
    {
        //Coordenada de la esquina superior izquierda de la elipse
        double dPosX = (double) circulo .GetValue(Canvas.LeftProperty);
        double dPosY = (double) circulo .GetValue(Canvas.TopProperty);
```

```
switch (e.Key )
{
    //Determina nueva coordenada hacia la izquierda
    case Key.Left :
    if (dPosX - iPixeles > 0)
    dPosX -= iPixeles;
    break;
    //Determina nueva coordenada hacia la derecha
    case Key.Right :
    if (dPosX + iPixeles + circulo.Width < dAnchoCanvas)
    dPosX += iPixeles;
    break;
    //Determina nueva coordenada hacia arriba
    case Key.Up :
    if (dPosY - iPixeles > 0)
    dPosY -= iPixeles;
    break;
    //Determina nueva coordenada hacia abajo
    case Key.Down :
    if (dPosY + iPixeles + circulo.Height < dAltoCanvas)
    dPosY += iPixeles;
    break;
}
//Mueve la elipse a la nueva coordenada
circulo.SetValue(Canvas.LeftProperty, dPosX);
```

```
        circulo.SetValue(Canvas.TopProperty, dPosY);
    }
```

Ejecución

Al ejecutar el programa se observa que cuando se usan las teclas flecha, el círculo llega a los extremos del **Canvas** (que es el cuadro del centro) pero no se sale de él.

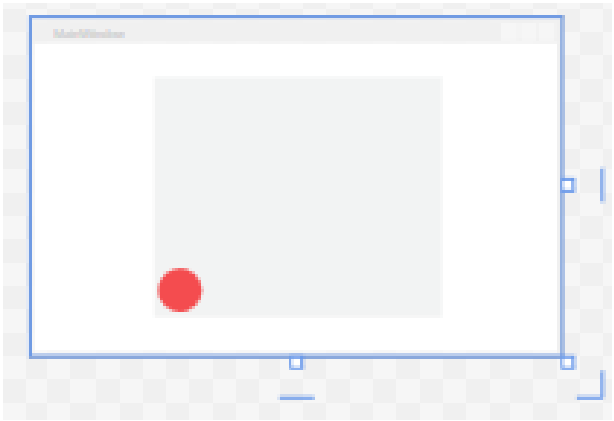


Figura 1.7

1.6 Añadir imágenes

Para trabajar una aplicación con imágenes se debe hacer lo siguiente:

1. Agregar imágenes al proyecto.
2. Agregar el control (objeto) **image** al **Canvas**.
3. Especificar qué imagen debe mostrar el control.

Hay diversas formas de agregar una imagen al proyecto, algunas de ellas se describen a continuación:

- Ejecutar el comando **Agregar elemento existente** del menú **Proyecto** y seleccionar las imágenes. Se observará en el **Explorador de soluciones** las imágenes añadidas incluso se puede agregar una carpeta al proyecto con estas.
- Dentro del **Explorador de Soluciones**, dar clic con el botón derecho del cursor sobre el nombre del proyecto para ejecutar el comando **Propiedades**. Después, sobre **Recursos**, definir el tipo de **imágenes** y agregarlas. En este caso se observará en el **Explorador de Soluciones** que se creó la carpeta **Recursos** y dentro de ella se encuentran la o las imágenes agregadas.

- Arrastrar y soltar las imágenes dentro del proyecto.

Un objeto tipo **image** posee características muy similares a las de la figura. La imagen debe tener un nombre x:Name para poder hacer referencia a ella desde el programa en C#. También es apropiado especificar un alto y un ancho adecuados a las dimensiones de la imagen y no olvidar incluir las propiedades Canvas.Left y Canvas.Top para poder modificar su posición.

Además de lo anterior, es necesario especificar cuál imagen debe aparecer inicialmente en el objeto ya que durante la ejecución del programa esta puede llegar a cambiar. Dado que la imagen ya está incluida en el proyecto, lo que resta por hacer es encontrar la propiedad **Source** del objeto image dentro de la ventana, y seleccionar la **imagen** que se desea emplear.

```
<Image x:Name ="rueda" Height="100 " Width="100 "
    Canvas.Left="10 " Canvas.Top="210 "
    Source="disco.png "/>
```

Nota: si la imagen fue añadida correctamente, el nombre del archivo que la contiene debe mostrarse dentro de la propiedad; de lo contrario, se sugiere utilizar alguno de los otros métodos.

1.7 Crear animaciones modificando propiedades

La animación es el proceso que se utiliza para dar la sensación de movimiento. Una manera de animar objetos es modificar sus propiedades, por ejemplo, se puede cambiar la imagen, modificar las dimensiones de un objeto, o bien, se puede aparecer y desaparecer un elemento momentáneamente. A continuación, se describe la forma de realizar dichas acciones desde el programa en C#.

Cambiar la imagen mostrada

Para mostrar una imagen diferente se debe modificar la propiedad **Source** del objeto **image**. Para lograrlo, lo primero que se debe hacer es incluir al programa la biblioteca que permite el uso de variables de tipo **BitmapImage**.

```
using System.Windows.Media.Imaging;
```

Posteriormente, se declara una variable de tipo BitmapImage a la cual se le asigna la imagen. En el siguiente ejemplo, la variable bmUno recibe la imagen disco.png la cual debió haber sido añadida a los recursos del proyecto para poder ser reconocida.

```
BitmapImage bmUno = new BitmapImage(new Uri(@"disco.png",
                                    UriKind.RelativeOrAbsolute));
```

Para hacer el cambio de la imagen, se le asigna a la propiedad Source, el contenido de la variable de tipo BitmapImage. En el siguiente ejemplo, el objeto tipo imagen de nombre rueda mostrará un elemento almacenado en el archivo disco.png (que fue el que se le asignó a bmUno en el ejemplo anterior).

```
rueda.Source = bmUno ;
```

Cambiar las dimensiones del objeto

Para modificar el tamaño de un objeto basta con especificar el nuevo alto y ancho que debe tener. Por ejemplo, para aumentar tres veces el tamaño de una imagen de nombre rueda, se podría hacer lo siguiente:

```
rueda.Height = rueda.Height * 3;
rueda.Width = rueda.Width * 3;
```

Aparecer o desaparecer un objeto

Cada objeto agregado al programa posee la propiedad de visibilidad. Cambiar este valor permite esconder o desaparecer objetos durante la ejecución del programa.

Esta propiedad puede tener uno de los siguientes valores:

Visible	Cuando el objeto tiene asignado el valor Visible, este se muestra en la ventana.
Hidden	En este caso, el objeto no se despliega, pero se conserva el espacio en el que está ubicado dentro del panel.
Collapsed	Un objeto colapsado no se muestra en la ventana, pero tampoco conserva su espacio, es decir, que momentáneamente otros objetos pueden ocupar su lugar.

Tabla 1.1

El siguiente ejemplo, colapsa el objeto de nombre **rueda**:

```
rueda. Visibility = Visibility.Collapsed ;
```

Si se desea aparecer nuevamente la imagen, se modifica la propiedad:

```
rueda. Visibility = Visibility.Visible ;
```

1.8 Ejemplo: simulación de un disco girando

Esta aplicación muestra un disco de colores girando que el usuario puede mover hacia la derecha o hacia la izquierda gracias a las teclas flechas que lo dirigen.

Requerimientos

- Desarrollado en **WPF**
- Biblioteca: **System.Windows.Media.Imaging**
- Imágenes: cuatro archivos que formen una secuencia.

Recursos

Para dar la sensación de que el disco está rotando, se intercalan varias imágenes para hacer el efecto de giro hacia delante o giro hacia atrás.

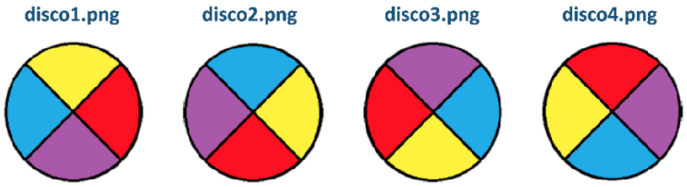


Figura 1.8

Las imágenes deben ser añadidas al proyecto. A continuación se muestra al **Explorador de soluciones** del proyecto que contiene a las cuatro imágenes: **disco1.png**, **disco2.png**, **disco3.png** y **disco4.png**.

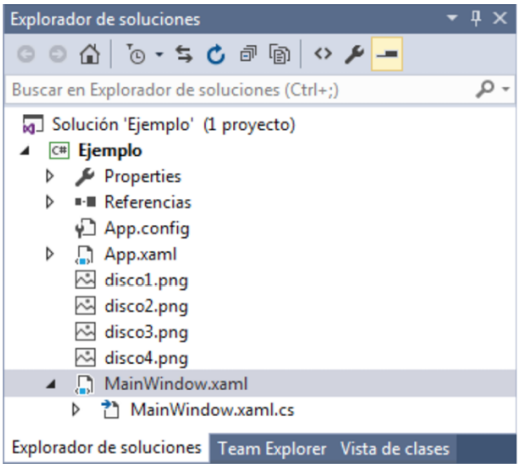


Figura 1.9

Código en XAML

El archivo **MainWindow.xaml** muestra al **Canvas** titulado MainCanvas el cual tiene un ancho de 500 píxeles con una altura de 320, al que se le ha establecido el evento KeyDown, que se activa cuando el usuario presiona una tecla haciendo que se ejecute el método MainCanvas_KeyDown que está localizado dentro del código en C#. También incluye una imagen nombrada disco, la cual tiene una altura y un ancho de 100 píxeles y se encuentra ubicada en la parte inferior de la pantalla (su esquina inferior izquierda está 10 píxeles a la izquierda del marco de **Canvas** y 210 píxeles por debajo de la orilla superior del mismo). A este objeto se le ha asignado la imagen disco1.png.

```
<Grid>
  <Canvas Name="MainCanvas" Height="320" Width="500"
    KeyDown="MainCanvas_KeyDown">
  <Image x:Name="disco " Height="100 " Width="100 "
    Canvas.Left="10 " Canvas.Top="210 "
    Source="disco1.png "/>
  </Canvas>
</Grid>
```

Código en C#

En el archivo **MainWindow.xaml.cs**, dentro de la clase y antes del método **MainWindow**, se encuentra declarada la variable **iPixeles** que indica la cantidad de píxeles que se moverá el **disco** cada vez que se presione una tecla flecha. Incluye también la declaración de las variables que se utilizan para almacenar el ancho y el alto del **Canvas**.

Para lograr que la animación muestre el disco girando, cada imagen es asignada a una variable de tipo **BitmapImage** (bmUno, bmDos, bmTres y bmCuatro). También se incluye la declaración de la variable iNumImagen cuyo contenido indica el número de la imagen que está apareciendo en cada ocasión.

```
public partial class MainWindow : Window
{
  //Cantidad de píxeles que se mueve el elemento en cada ocasión
```

```
int iPixeles = 5;
//Indicador cuyo valor es el número de la imagen a mostrar
//Solamente toma valores del 1 al 4 ya que son 4 imágenes
int iNumImagen = 1;
//Variables que almacenarán las dimensiones del Canvas
double dAnchoCanvas, dAltoCanvas;
//Imágenes para crear el efecto de giro del disco
BitmapImage bmUno = new BitmapImage(new Uri(@"disco1.png ",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmDos = new BitmapImage(new Uri(@"disco2.png ",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmTres = new BitmapImage(new Uri(@"disco3.png ",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmCuatro = new BitmapImage(new Uri(@"disco4.png
",
                                    UriKind.RelativeOrAbsolute));
```

Como primera instrucción y justo después de iniciar el componente, se establece la propiedad **Focus** del **Canvas** para que se pueda activar el evento relacionado con él. Hecho lo anterior, se obtienen las dimensiones del panel.

```
public MainWindow()
{
  InitializeComponent();
  //Enfocar el Canvas
  MainCanvas.Focusable = true;
  MainCanvas.Focus();
  //Obtener el ancho y la altura del Canvas
```

```
dAnchoCanvas = MainCanvas.Width;
dAltoCanvas = MainCanvas.Height;
```

El método **MainCanvas_KeyDown**, que se ejecuta cada vez que el usuario presiona una tecla, es similar al que se utilizó para mover la figura del ejemplo anterior. Lo primero que hace es obtener la ubicación del disco donde las variables dPosX y dPosY corresponden a la posición de la esquina superior izquierda de la imagen. Posteriormente, se emplea el valor del argumento e para determinar cuál de las flechas (**Key.Left**, **Key.Right**, **Key.Up** y **Key.Down**) se presionó. Para recrear la animación de rodar el disco, se añade a las instrucciones para moverse a la izquierda o hacia la derecha la llamada a un método (RotaIzquierda o RotaDerecha respectivamente). Esto es útil para cambiar la propiedad **Source** de la rueda asignándole la siguiente imagen en la secuencia.

```
private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
{
  //Coordenada de la esquina superior izquierda del disco
  double dPosX = (double) disco.GetValue(Canvas.LeftProperty);
  double dPosY = (double) disco.GetValue(Canvas.TopProperty);
  switch (e.Key)
  {
    //Determina nueva coordenada hacia la izquierda
    case Key.Left:
        if (dPosX - iPixeles > 0)
        {
            RotaIzquierda (); //Cambiar la imagen para girarla
            dPosX -= iPixeles;
        }
        break;
    //Determina nueva coordenada hacia la derecha
    case Key.Right:
```

```
            if (dPosX + iPixeles + disco.Width < dAnchoCanvas)
            {
                RotaDerecha (); //Cambiar la imagen para girarla
                dPosX += iPixeles;
            }
            break;
        //Determina nueva coordenada hacia arriba
        case Key.Up:
            if (dPosY - iPixeles > 0)
                dPosY -= iPixeles;
            break;
        //Determina nueva coordenada hacia abajo
        case Key.Down:
            if (dPosY + iPixeles + disco.Height < dAltoCanvas)
                dPosY += iPixeles;
            break;
    }
    //Mueve el disco a la nueva coordenada
    disco.SetValue(Canvas.LeftProperty, dPosX);
    disco.SetValue(Canvas.TopProperty, dPosY);
}
```

Para animar el movimiento hacia la derecha, de la imagen 1 se pasa a la 2 y de la 2 a la 3 y de la 3 para la 4; después de la 4 se regresa nuevamente a la 1.

```
private void RotaDerecha ()
{
    //Cambiar la imagen
```

```
    switch (iNumImagen)
    {
        case 1: disco.Source = bmDos; break;
        case 2: disco.Source = bmTres; break;
        case 3: disco.Source = bmCuatro; break;
        case 4: disco.Source = bmUno; break;
    }
    //Pasar el indicador a la siguiente imagen
    iNumImagen++;
    if (iNumImagen == 5) iNumImagen = 1;
}
```

Para animar el movimiento hacia la izquierda, de la imagen 4 se pasa a la 3 y de la 3 a la 2 y de la 2 para la 1 después de la 1 se regresa nuevamente a la 4.

```
private void RotaIzquierda ()
{
    //Cambiar la imagen
    switch (iNumImagen)
    {
        case 1: disco.Source = bmCuatro; break;
        case 2: disco.Source = bmUno; break;
        case 3: disco.Source = bmDos; break;
        case 4: disco.Source = bmTres; break;
    }
    //Pasar el indicador a la siguiente imagen
    iNumImagen--;
    if (iNumImagen == 0) iNumImagen = 4;
```

}

Ejecución

Al ejecutar el programa se puede observar que los colores del disco van cambiando hasta dar la sensación de que está girando. Se recomienda mirar la diferencia entre la secuencia de las imágenes en movimientos hacia la derecha y la que se utiliza para cuando avanza hacia la izquierda. Esta animación se puede realizar a cualquier altura del **Canvas**.

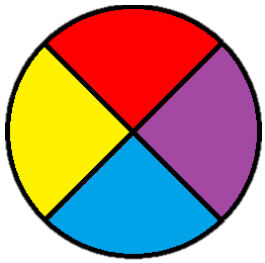


Figura 1.10

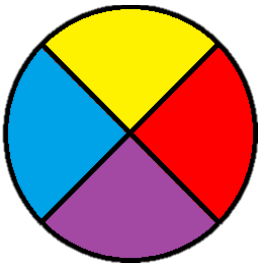


Figura 1.11

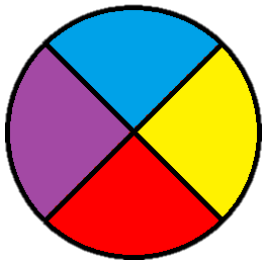


Figura 1.12

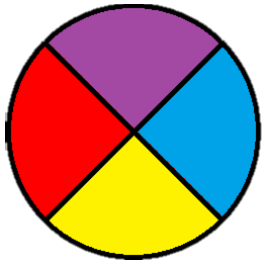


Figura 1.13

1.9 Ejemplo: reacción

La aplicación maneja la estrategia de cambio de los valores de las propiedades para recrear la animación de un personaje al que se denomina Yupi el cual observa en el cielo al personaje Malo y reacciona de acuerdo con su proximidad.

Requerimientos

- Desarrollado en **WPF**
- Biblioteca: **System.Windows.Media.Imaging**
- Imágenes: trece archivos.

Recursos

Para realizar la animación, se cuentan con diez imágenes de Yupi, dos de Malo y un fondo. Todas las imágenes de los personajes tienen un fondo transparente para que al ser colocadas encima del escenario no se vean como un rectángulo (por esa razón son de tipo png). Hay diferentes paquetes de *software* para hacer transparente el fondo; uno de ellos es el **Paint**.

Malo se puede mover hacia la derecha o hacia la izquierda; cuando va hacia la derecha se utiliza la imagen maloD.png y cuando va en el otro sentido se emplea maloI.png. Conforme Malo se desplaza por la ventana, Yupi lo va siguiendo con los ojos. Al principio se presenta con las imágenes Inicial.png; conforme Malo va bajando por la ventana, cambia a Azul.png y posteriormente a Verde.png. Llega un momento en el que está tan cerca que pasa a Final.png para finalmente desaparecer.

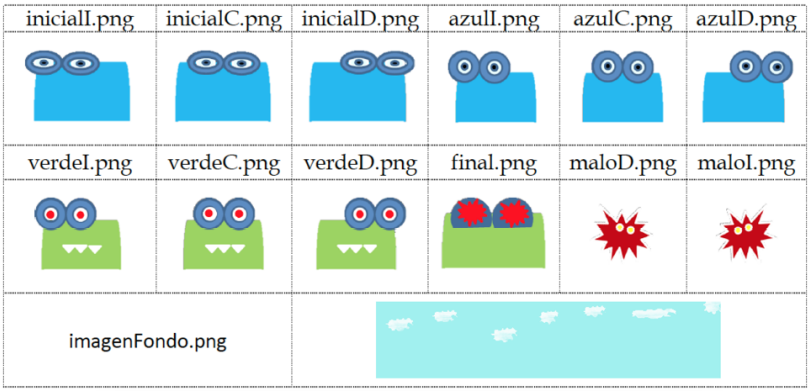


Figura 1.14

Para organizar el contenido del proyecto, se agregó una carpeta llamada **Imágenes** para colocar dentro de ella todos los archivos.

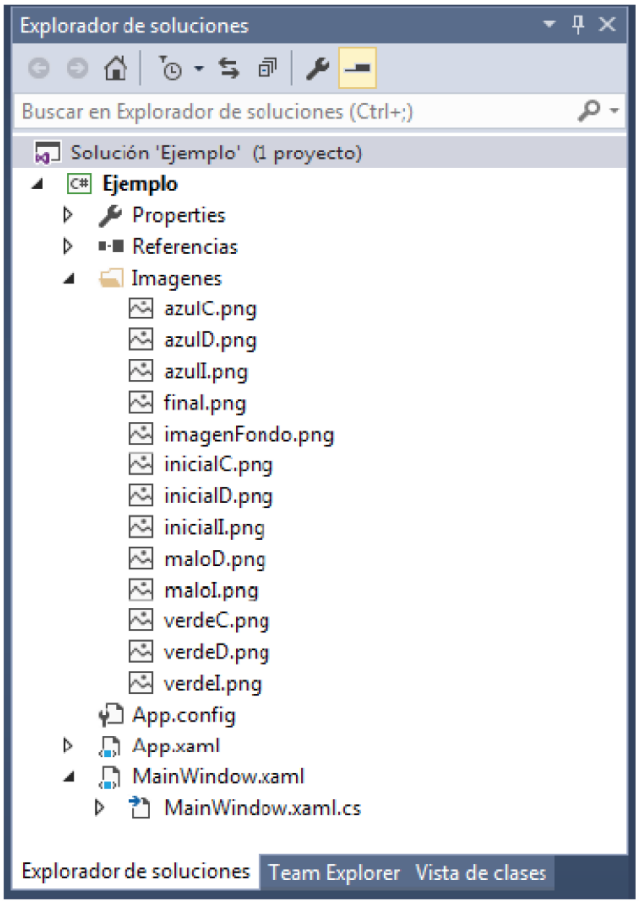


Figura 1.15

Código en XAML

El archivo **MainWindow.xaml** muestra al **Canvas** denominado MainCanvas con sus dimensiones especificadas para que los objetos que contiene se puedan manipular desde el programa en C#. También se establece el evento KeyDown que para mover de lugar a Malo.

El panel incluye tres imágenes. La primera es el fondo para el cual se empleó la propiedad UniformToFill para que la imagen se extienda por toda el área dedicada al objeto con el propósito de cubrir toda la ventana. Aparte, se incluye un objeto para colocar a Yupi y otro para poner a Malo.

```
<Grid>
  <Canvas Name="MainCanvas" Width="510" Height="320"
    KeyDown ="MainCanvas_KeyDown" Margin="0,0,0,0">
    <Image x:Name="fondo " Height="320" Width="517"
```

```
            Source="Imagenes/imagenFondo.png" Stretch="UniformToFill
            "/>
        <Image x:Name="yupi " Height="70" Width="56"
            Canvas.Left="219" Canvas.Top="265"
            Source="Imagenes/inicialC.png"/>
        <Image x:Name="malo " Height="70" Width="56"
            Canvas.Left="1" Canvas.Top="1"
            Source="Imagenes/maloD.png" Stretch="UniformToFill"/>
    </Canvas>
</Grid>
```

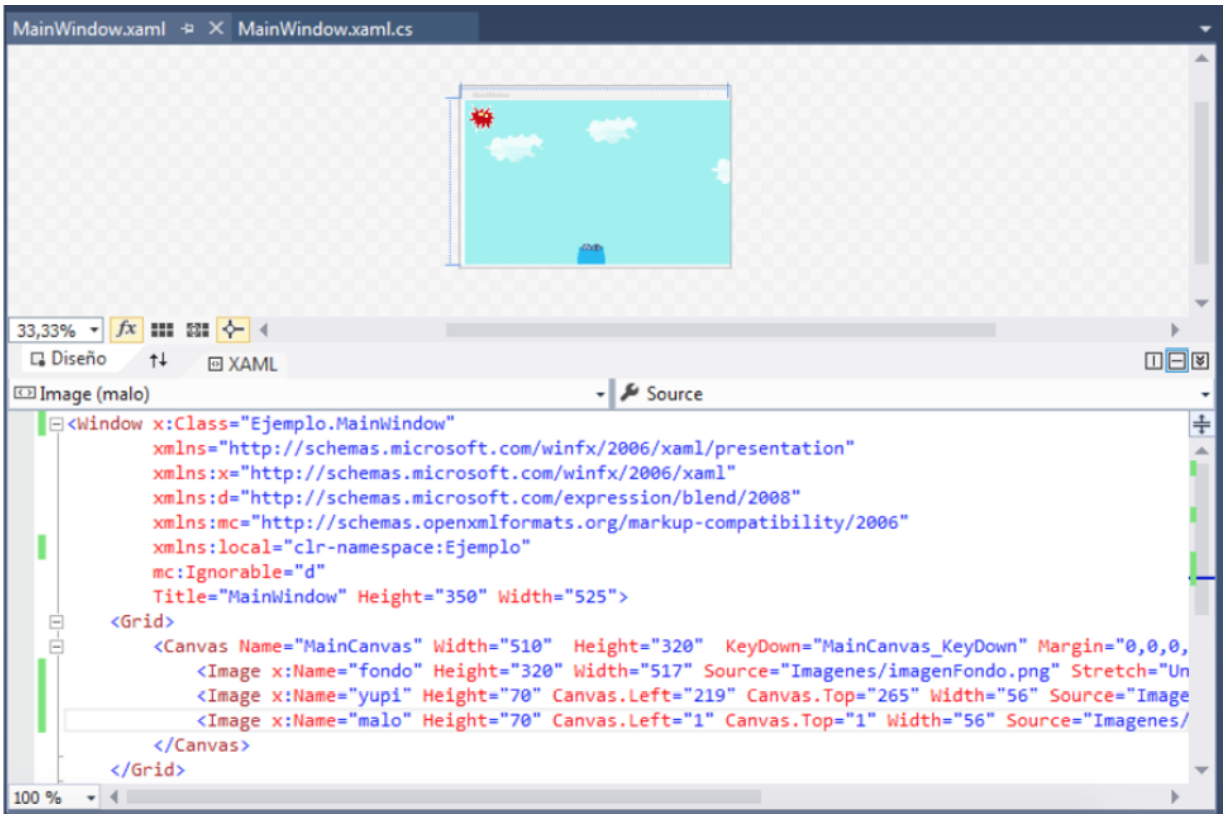


Figura 1.16

Código en C#

En el archivo **MainWindow.xaml.cs**, dentro de la clase y antes del método

MainWindow, se encuentra declarada la variable **iPixeles** para especificar la cantidad de píxeles que se moverá el **malo** cada vez que se presione una tecla flecha. También se declaran las variables para las dimensiones del **Canvas** y las variables de tipo **BitmapImage** (una para cada imagen que se manipula desde C#).

```
public partial class MainWindow : Window
{
  //Cantidad de píxeles que se mueve Malo en cada ocasión
  int iPixeles = 5;
  //Variables que almacenarán las dimensiones del Canvas
  double dAnchoCanvas, dAltoCanvas;
  //Variables que almacenarán las características de Yupi
  double dAltoYupi, dAnchoYupi, dPosXYupi, dPosYYupi;
  //Definición de imágenes a utilizar
  BitmapImage bmInicialI = new BitmapImage(new
Uri(@"Imagenes/inicialI.png",
                                UriKind.RelativeOrAbsolute));
  BitmapImage bmInicialC = new BitmapImage(new
Uri(@"Imagenes/inicialC.png",
                                UriKind.RelativeOrAbsolute));
  BitmapImage bmInicialD = new BitmapImage(new
Uri(@"Imagenes/inicialD.png",
                                UriKind.RelativeOrAbsolute));
  BitmapImage bmAzulI = new BitmapImage(new
Uri(@"Imagenes/azulI.png",
                                UriKind.RelativeOrAbsolute));
  BitmapImage bmAzulC = new BitmapImage(new
Uri(@"Imagenes/azulC.png",
                                UriKind.RelativeOrAbsolute));
```

```
BitmapImage bmAzulD = new BitmapImage(new Uri(@"Imagenes/azulD.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmVerdeI = new BitmapImage(new Uri(@"Imagenes/verdeI.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmVerdeC = new BitmapImage(new Uri(@"Imagenes/verdeC.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmVerdeD = new BitmapImage(new Uri(@"Imagenes/verdeD.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmFinal = new BitmapImage(new Uri(@"Imagenes/final.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmMaloI = new BitmapImage(new Uri(@"Imagenes/maloI.png",
                                    UriKind.RelativeOrAbsolute));
BitmapImage bmMaloD = new BitmapImage(new Uri(@"Imagenes/maloD.png",
                                    UriKind.RelativeOrAbsolute));
```

Dentro del método **MainWindow** se han colocado las instrucciones requeridas para enfocar el **Canvas** permitiendo así trabajar con su contenido. También se realizan las instrucciones para obtener valores que son fijos durante todo el programa como las dimensiones del panel, de Yupi y la posición.

```
public MainWindow()
{
  InitializeComponent();
  //Enfocar el Canvas
```

```
MainCanvas.Focusable = true;
MainCanvas.Focus();
//Obtener el ancho y la altura del Canvas
dAnchoCanvas = MainCanvas.Width;
dAltoCanvas = MainCanvas.Height;
//Obtener la posición y las dimensiones de Yupi
dPosXYupi = (double)yupi.GetValue(Canvas.LeftProperty);
dPosYYupi = (double)yupi.GetValue(Canvas.TopProperty);
dAltoYupi = yupi.Height;
dAnchoYupi = yupi.Width;
```

El método **MainCanvas_KeyDown** contiene las instrucciones para mover a Malo hacia la posición que se indique con la tecla flecha presionada. También se encarga de modificar las propiedades de Yupi con relación a la posición en la que se encuentra Malo.

Para lograrlo, el método inicia restableciendo la ubicación, las dimensiones y la visibilidad de Yupi debido a que estas pudieron haber sido modificadas con el evento anterior. Después, empleando la posición actual de Malo y la tecla que se presionó, se determina la nueva ubicación de Malo para luego colocarlo ahí. Finalmente, se solicita la ejecución del método ActualizaYupi al cual se le envía la nueva posición de Malo con el fin de modificar las propiedades de Yupi.

```
private void MainCanvas_KeyDown (object sender, KeyEventArgs e)
{
  //Restablecer las propiedades originales de Yupi
  yupi.SetValue(Canvas.LeftProperty, dPosXYupi);
  yupi.SetValue(Canvas.TopProperty, dPosYYupi);
  yupi.Height = dAltoYupi;
  yupi.Width = dAnchoYupi;
  yupi.Visibility = Visibility.Visible;
  //Coordenada de la esquina superior izquierda de Malo
```

```
double dPosX = (double)malo.GetValue(Canvas.LeftProperty);
double dPosY = (double)malo.GetValue(Canvas.TopProperty);
switch (e.Key)
{
    //Determina nueva coordenada de Malo hacia la izquierda
    case Key.Left:
        if (dPosX - iPixeles > 0)
            dPosX -= iPixeles;
        malo.Source = bmMaloI;
        break;
    //Determina nueva coordenada de Malo hacia la derecha
    case Key.Right:
        if (dPosX + iPixeles + malo.Width < dAnchoCanvas)
            dPosX += iPixeles;
        malo.Source = bmMaloD;
        break;
    //Determina nueva coordenada de Malo hacia arriba
    case Key.Up:
        if (dPosY - iPixeles > 0)
            dPosY -= iPixeles;
        break;
    //Determina nueva coordenada de Malo hacia abajo
    case Key.Down:
        if (dPosY + iPixeles + malo.Height < dAltoCanvas)
            dPosY += iPixeles;
```

```
            break;
        }
        //Mueve a Malo a la nueva coordenada
        malo.SetValue(Canvas.LeftProperty, dPosX);
        malo.SetValue(Canvas.TopProperty, dPosY);
        //Actualiza la imagen de Yupi de acuerdo a la posición de Malo
        ActualizaYupi (dPosX, dPosY);
    }
```

El método **ActualizaYupi** se encarga de modificar sus propiedades en relación a la posición de Malo. La ventana se divide a lo largo de seis secciones. La primera sección que va del píxel 1 al 39, presenta a Yupi inicial enfocando sus ojos hacia la dirección de Malo (izquierda, central o derecha). Si Malo se encuentra entre los píxeles 40 y 79, se muestra a Yupi de color azul y con los ojos muy abiertos. Cuando Malo está a partir del píxel 80 y hasta el 119, se pone verde. Si Malo se aproxima más entre los píxeles 120 y 159, Yupi crece tres veces su tamaño para lo cual se reubica (para que la imagen enorme quede dentro del Canvas) y se modifica el ancho y alto de la imagen. Finalmente, ya cuando Malo está muy cerca, se muestra a Yupi con la imagen final la cual, después de un tiempo es colapsada para ocultarlo.

```
    private void ActualizaYupi(double dPosX, double dPosY)
    {
        if (dPosY < 40) //Yupi Inicial
        {
            if (dPosX < 150) yupi.Source = bmInicialI;
            else if (dPosX < 300) yupi.Source = bmInicialC;
            else yupi.Source = bmInicialD;
        }
        else if (dPosY < 80) //Yupi Azul
        {
            if (dPosX < 150) yupi.Source = bmAzulI;
```

```
            else if (dPosX < 300) yupi.Source = bmAzulC;
            else yupi.Source = bmAzulD;
        }
        else if (dPosY < 120) //Yupi Verde
        {
            if (dPosX < 150) yupi.Source = bmVerdeI;
            else if (dPosX < 300) yupi.Source = bmVerdeC;
            else yupi.Source = bmVerdeD;
        }
        else if (dPosY < 160) //Yupi enorme
        {
            yupi.SetValue(Canvas.LeftProperty, dPosXYupi - dAnchoYupi);
            yupi.SetValue(Canvas.TopProperty, dPosYYupi - dAltoYupi);
            yupi.Height = dAltoYupi * 3;
            yupi.Width = dAnchoYupi * 3;
        }
        else if (dPosY < 200) //Yupi Final
            yupi.Source = bmFinal;
        else //Yupi Colapsado
            yupi.Visibility = Visibility.Collapsed;
    }
```

Ejecución

La aplicación muestra en la ventana a Malo en la parte superior y a Yupi en la parte inferior. Hay que utilizar las teclas flechas para mover a Malo hacia la izquierda, derecha, arriba y abajo. Las siguientes imágenes muestran algunos momentos en los que Yupi cambia de forma con respecto a la posición de Malo.

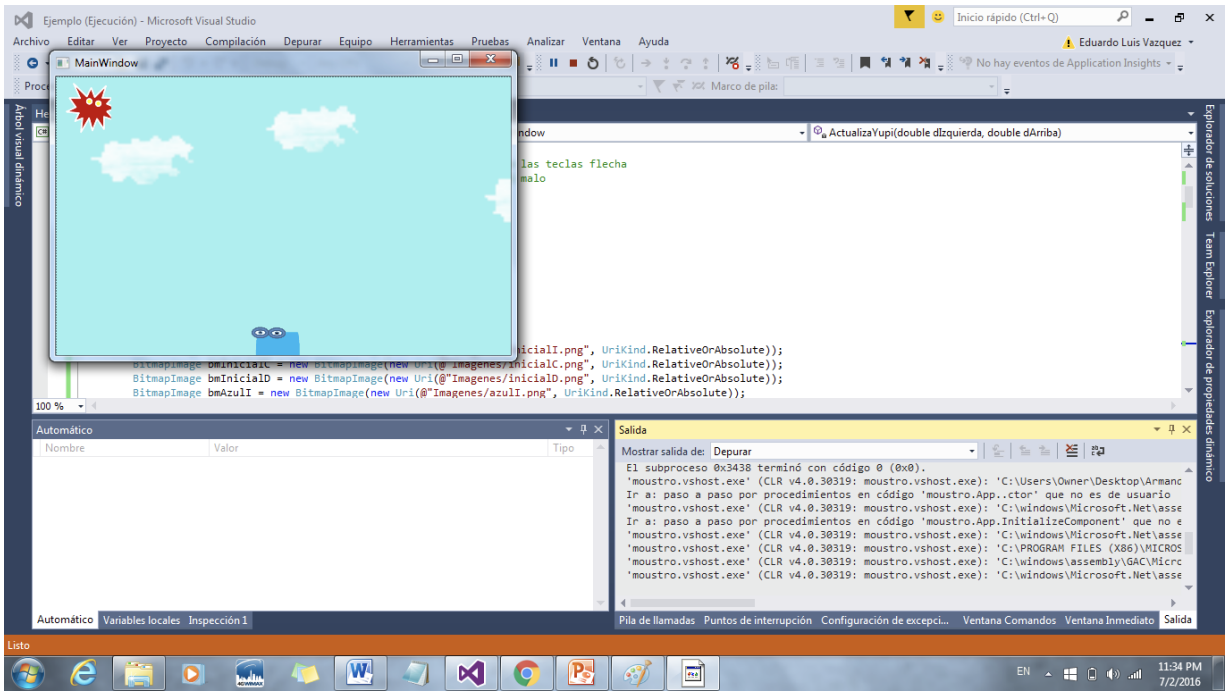


Figura 1.17

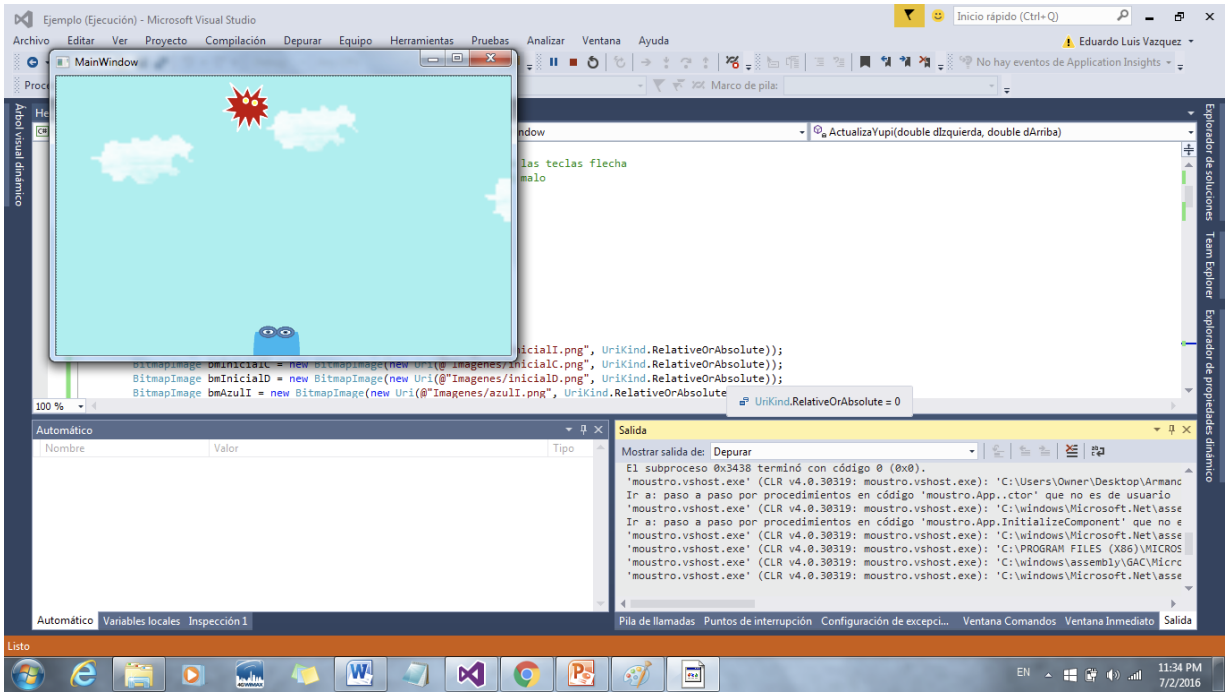


Figura 1.18

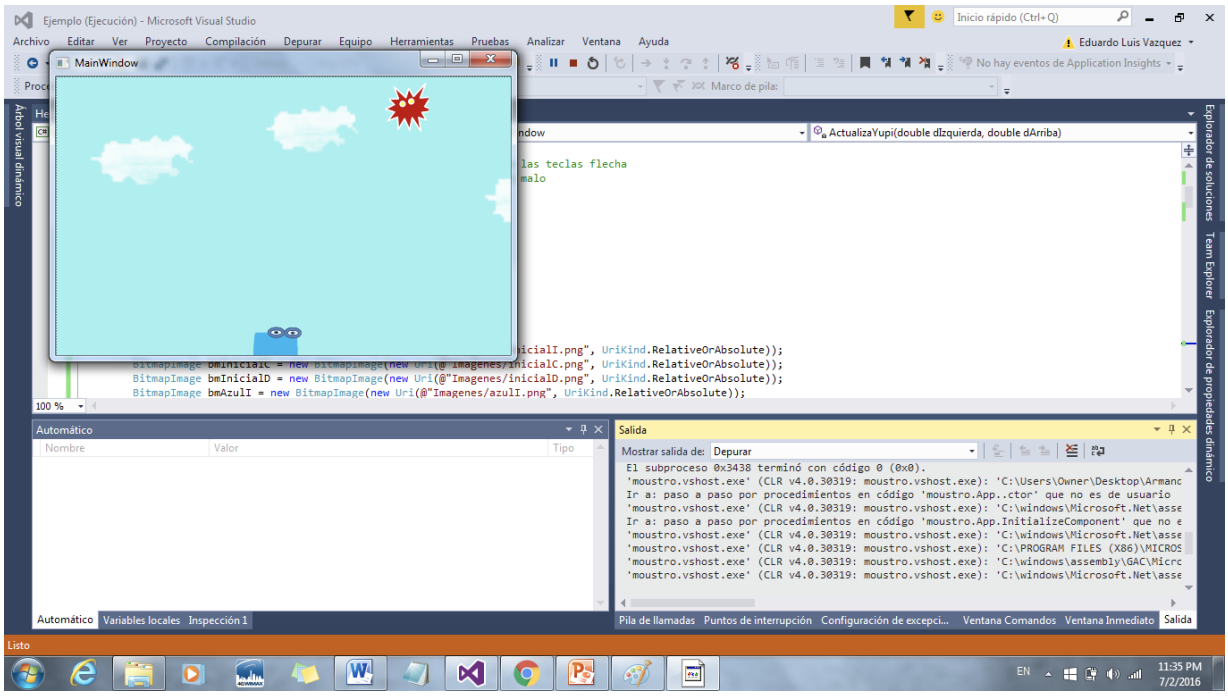


Figura 1.19

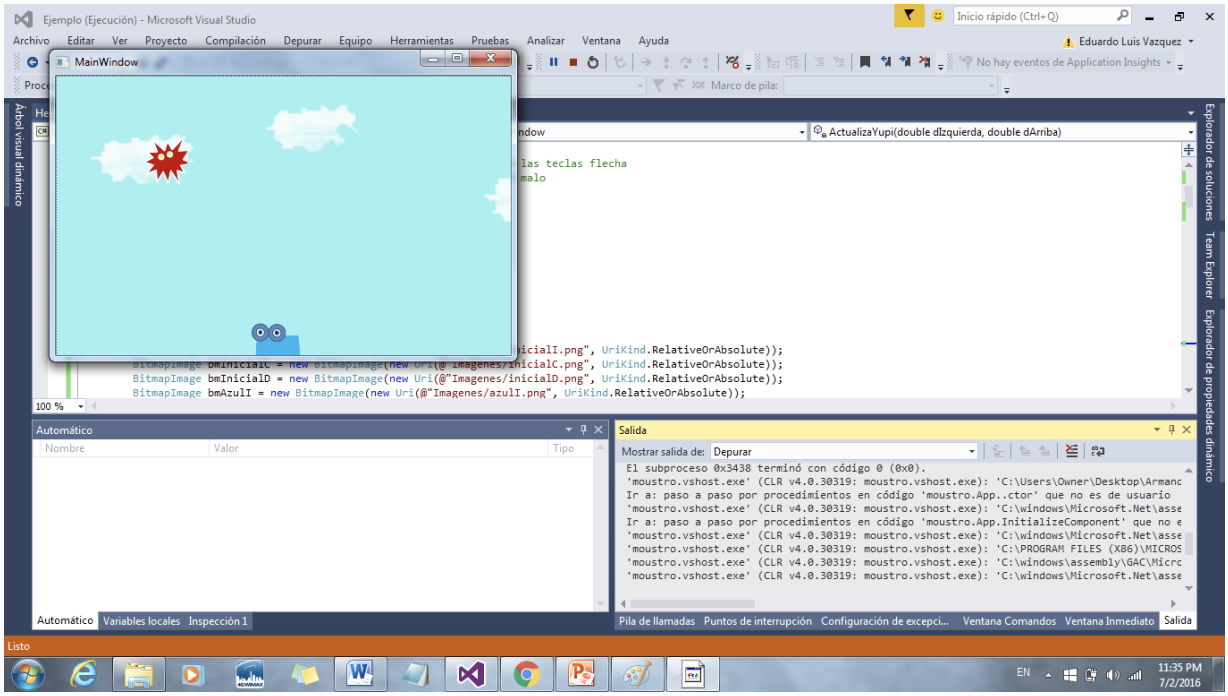


Figura 1.20

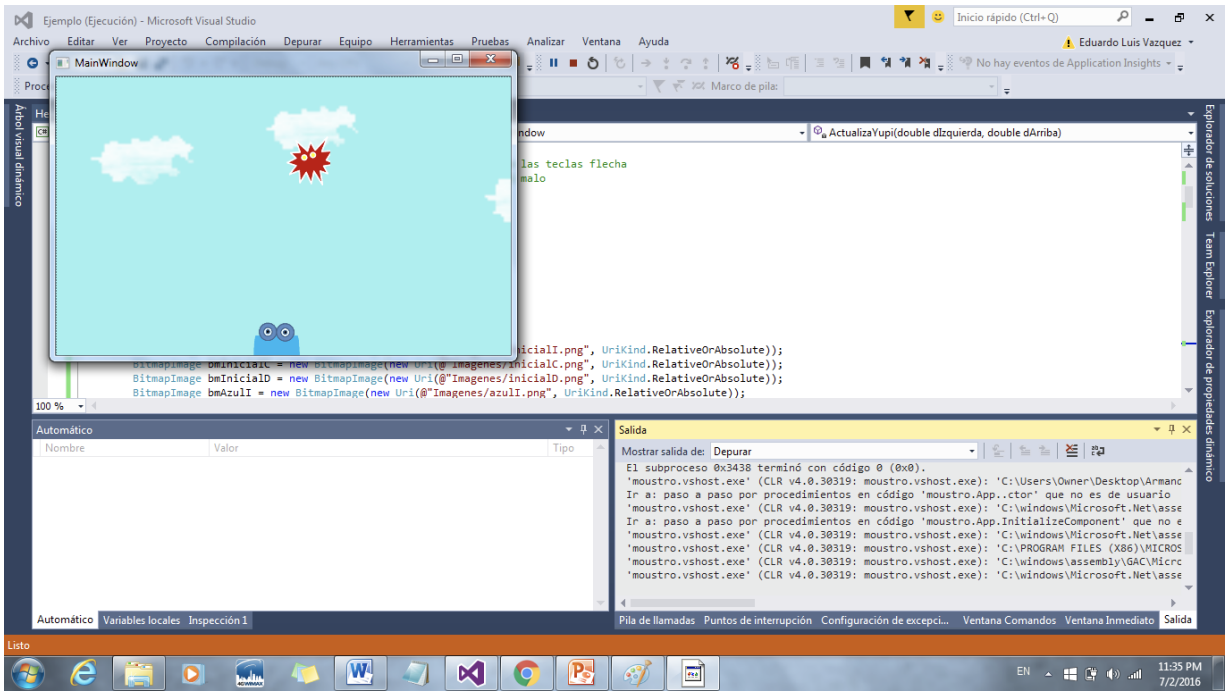


Figura 1.21

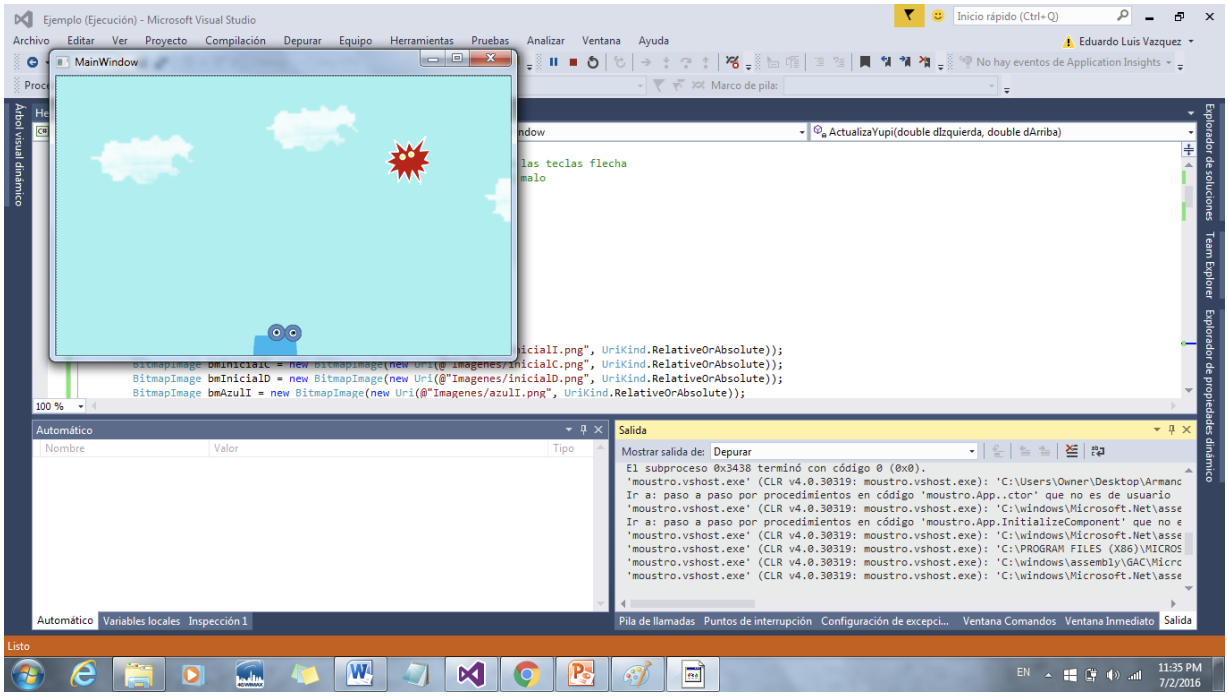


Figura 1.22

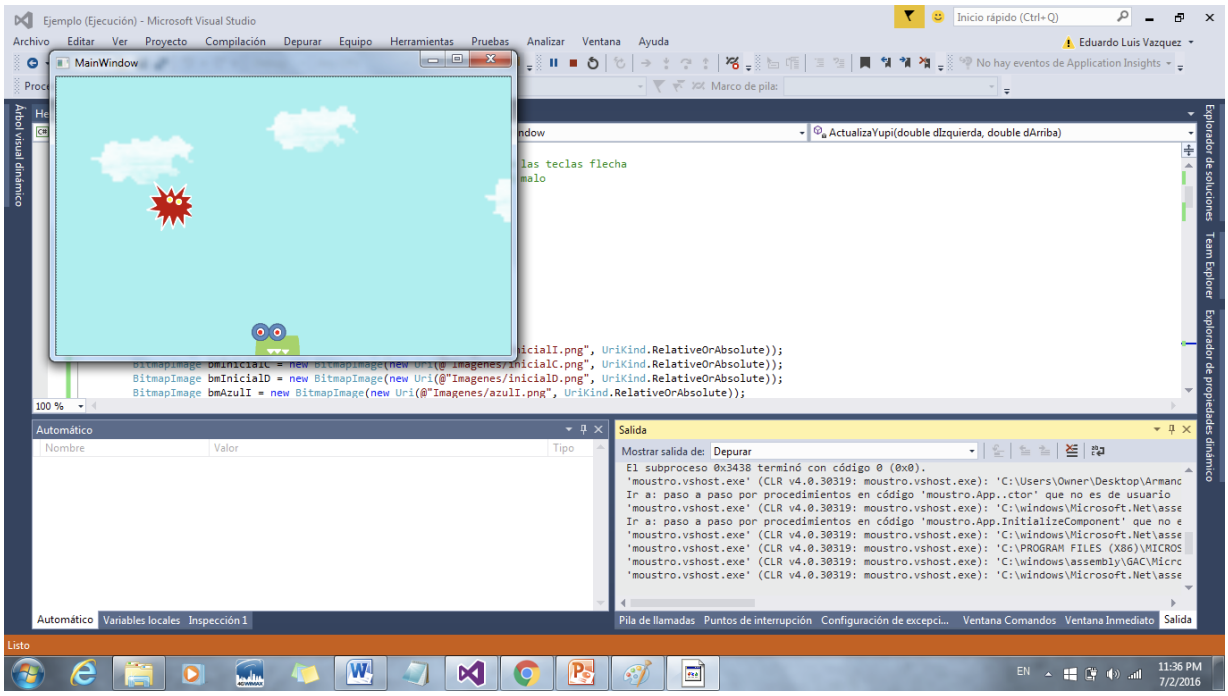


Figura 1.23

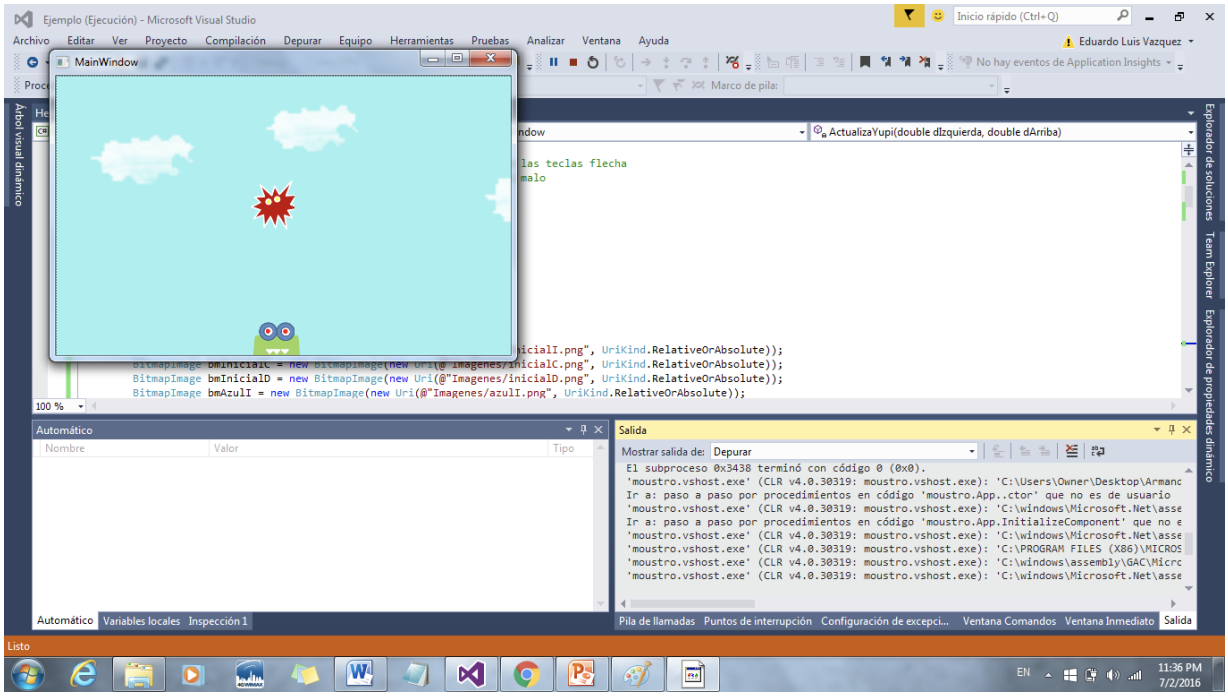


Figura 1.24

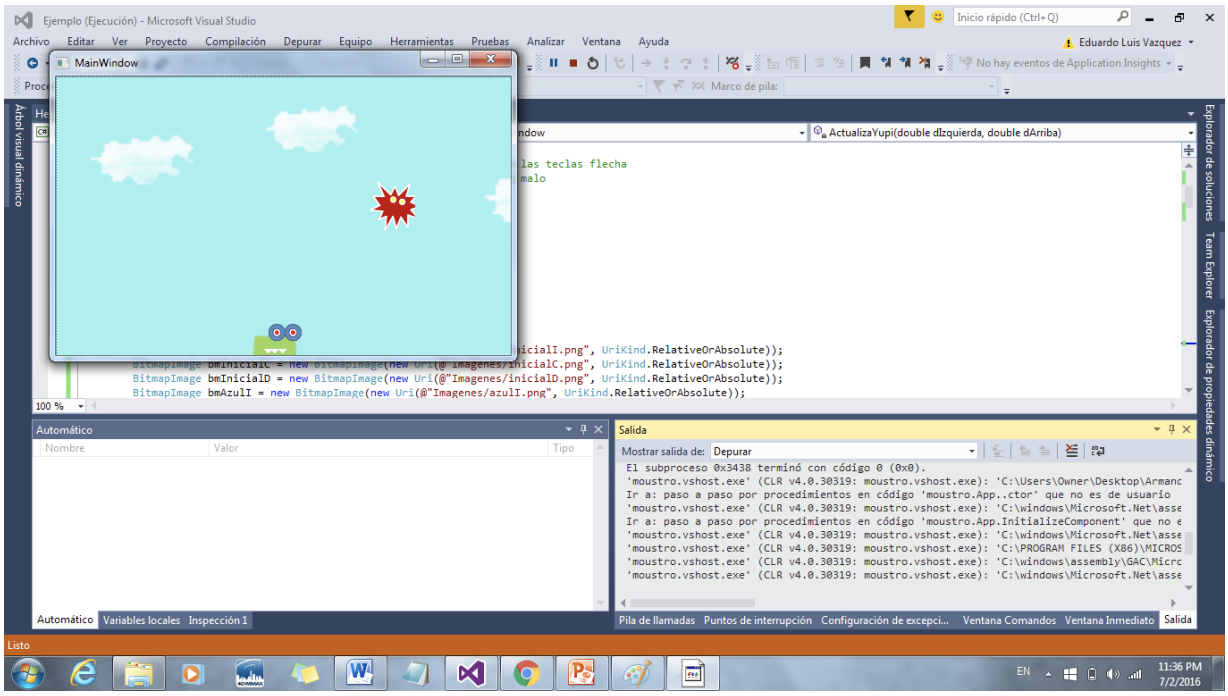


Figura 1.25

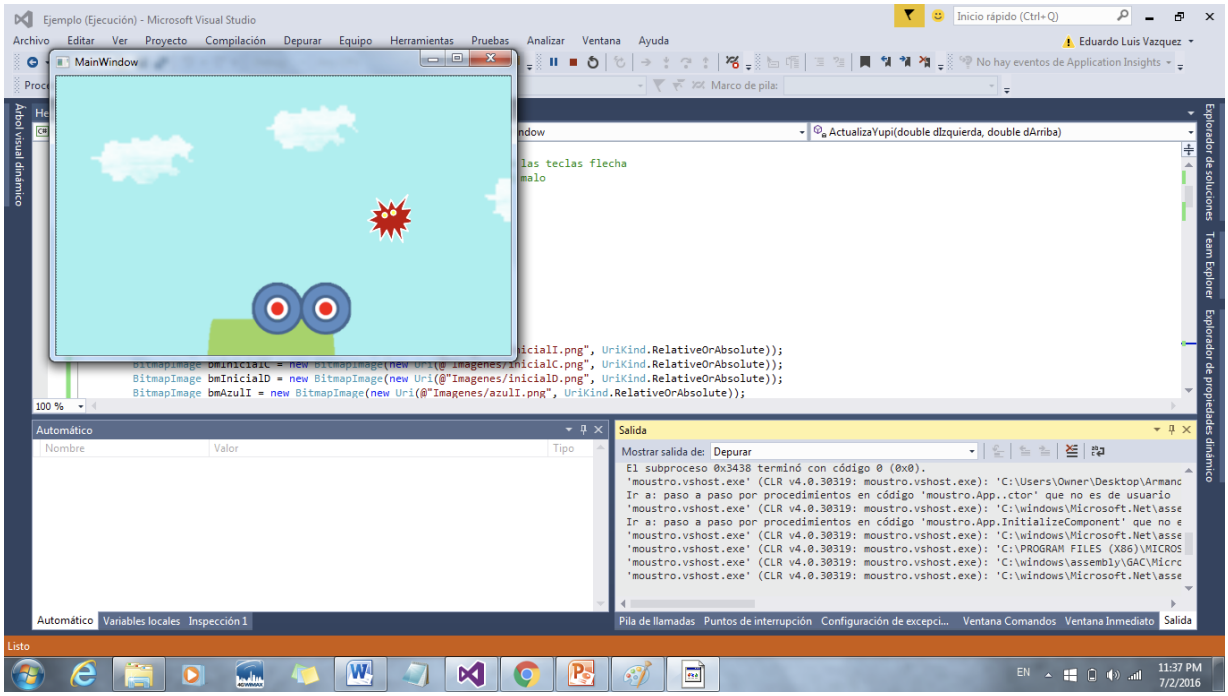


Figura 1.26

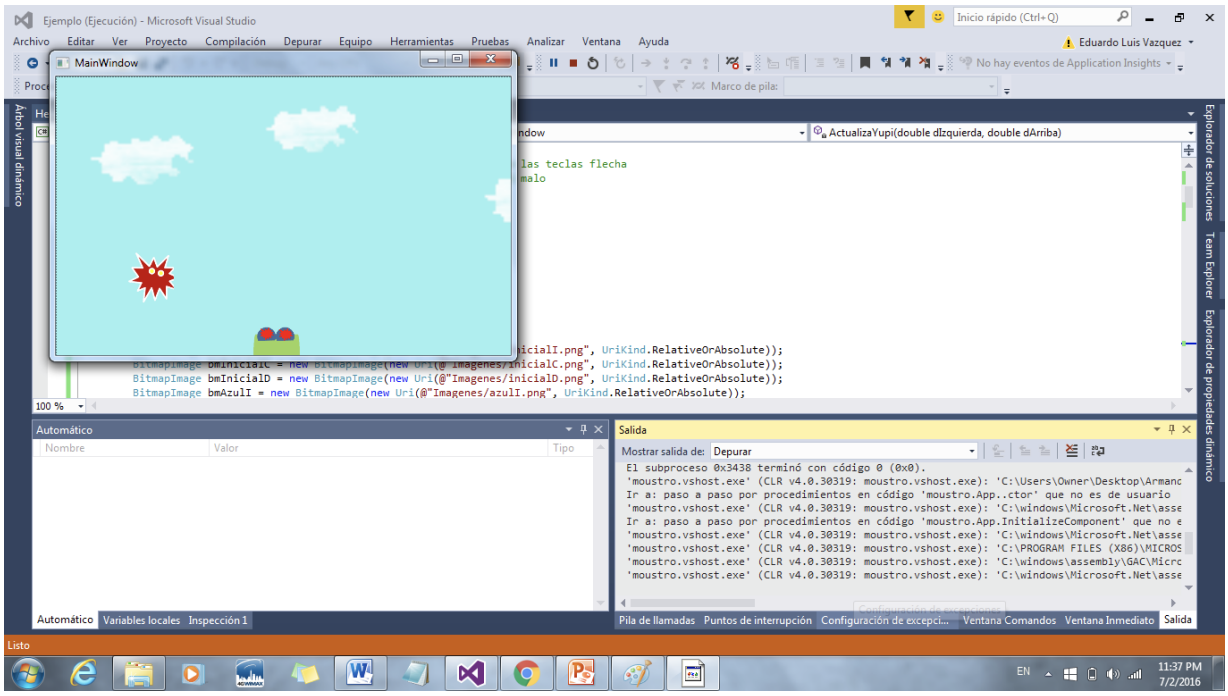


Figura 1.27

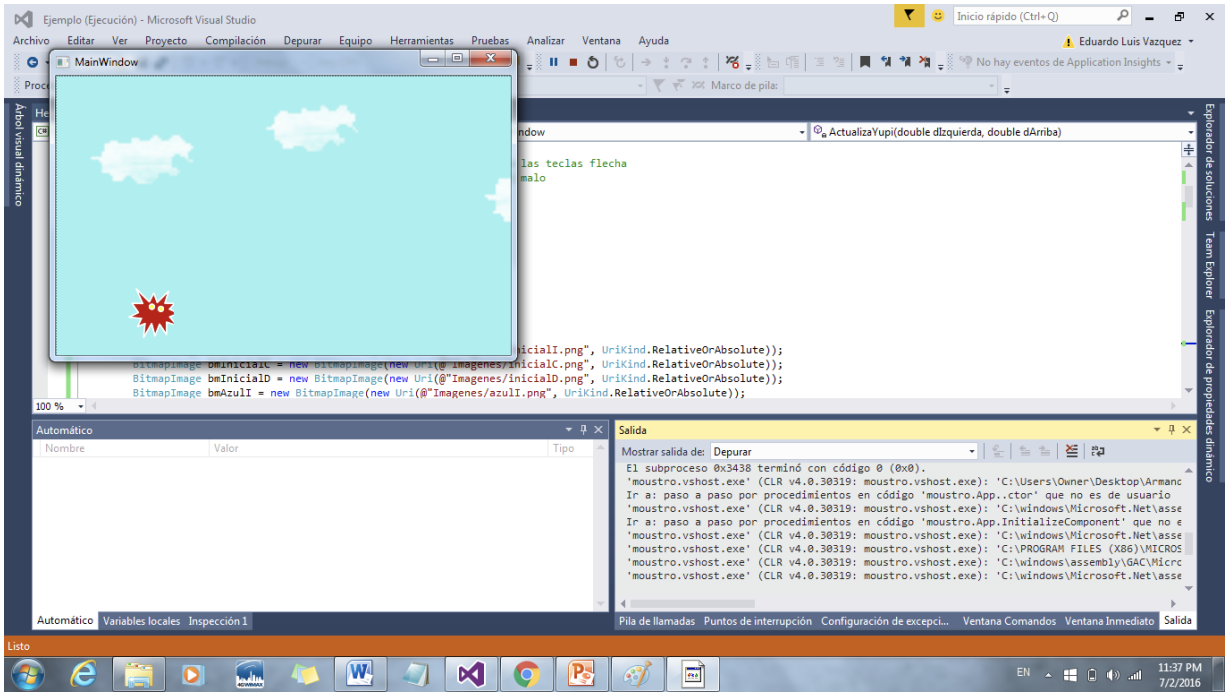


Figura 1.28

Capítulo 2. Movimientos automáticos

El estilo de programación para construir proyectos WPF está orientado a eventos, donde el orden en que se realizan las instrucciones del programa depende de los sucesos que van sucediendo cuando se ejecuta.

Estos sucesos o eventos pueden ser provocados por acciones que realiza el usuario, por ejemplo, cuando presiona una tecla o da un clic.

Otra forma de crear eventos es incluyendo **Timers** en el programa. Un **Timer** es un módulo especial que se ejecuta una y otra vez cada determinado tiempo. Su ventaja es que el resto del programa está en ejecución de manera simultánea. Un ejemplo del uso de **Timers** son las aplicaciones relacionadas con el movimiento del cuerpo. Si el programa está esperando que el usuario realice un gesto al mismo tiempo que se están llevando a cabo otras tareas, no es válido detener todo para quedar en espera del gesto, por el contrario, todo debe continuar y cada cierto tiempo, mientras se están realizando las demás actividades, se puede verificar si el usuario realizó o no el ademán.

En este capítulo, se mostrará la forma en la que se pueden combinar diferentes tipos de eventos para crear aplicaciones más animadas y con actividades que se realizarán de manera automática.

2.1 Creación de eventos que se ejecutan periódicamente

Para crear módulos (conjunto de instrucciones) que se ejecuten cada determinado tiempo, se cuenta con la clase **DispatcherTimer**. Esta permite agregar eventos Tick que se activan cada cierto tiempo (definido por el programador), tienen acceso completo a los objetos añadidos en XAML y la ventaja de no iniciar hasta que se ejecuta su método **Start**, o bien, hasta que su propiedad **IsEnable** sea puesta en verdadero.

Para utilizar la clase DispatcherTimer es necesario incluir la biblioteca:

```
using System.Windows.Threading;
```

Para definir un **Timer**, lo primero que se debe hacer es crear un objeto de tipo **DispatcherTimer**. La siguiente instrucción muestra la creación de un **DispatcherTimer** de nombre timer.

```
DispatcherTimer timer = new DispatcherTimer();
```

Posteriormente, se debe especificar la frecuencia con la que se debe ejecutar el evento.

Se construye una estructura del tipo **TimeSpan**, útil para expresar el intervalo en término de días, horas, minutos, segundos y milisegundos. Por ejemplo, la siguiente instrucción asigna el intervalo de tiempo al **DispatcherTimer** timer creado anteriormente para indicar que el evento debe ser ejecutado cada 20 milisegundos.

```
timer.Interval = new TimeSpan(0, 0, 0, 0, 20);
```

Después de lo anterior, hay que indicar el nombre del método que contiene las instrucciones que deben ser ejecutadas cada vez que ocurra el intervalo de tiempo. La siguiente línea de código muestra la manera de añadir al **DispatcherTimer** timer, un Tick (método) de nombre Timer_Tick (el nombre puede ser diferente).

```
timer .Tick += new EventHandler(Timer_Tick );
```

La instrucción anterior requiere que el programa de C# incluya el método Timer_Tick con el siguiente encabezado:

```
private void Timer_Tick (object sender, EventArgs e)
{
    //Instrucciones que serán ejecutadas cada vez que ocurra el intervalo
}
```

Finalmente, hay que arrancar a la medición de los intervalos al cambiar la propiedad IsEnabled del **DispatcherTimer** timer a verdadero.

```
timer.IsEnabled = true;
```

2.2 Ejemplo: rebote automático de una pelota

La aplicación muestra una pelota (círculo) apareciendo por la parte superior de la ventana que de manera automática, baja y rebota en la parte inferior y se mantiene rebotando ilimitadamente.

Este es un ejemplo del uso del **DispatcherTimer** ya que la pelota se mueve automáticamente, es decir, cada intervalo de tiempo se ejecutan instrucciones que mueven la pelota un poco más hacia abajo o hacia arriba según sea su dirección en ese momento. El movimiento es automático ya que no se requiere la intervención del usuario para dar continuidad al movimiento.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Windows.Threading**.

Código en XAML

El archivo **MainWindow.xaml** muestra al Canvas que contiene una elipse de nombre pelota ubicada en la parte superior del panel. Gracias a la ventana de propiedades se rellenó la elipse con un color.

```
<Grid>
<Canvas x:Name="MainCanvas" Width="505" Height="315">
<Ellipse x:Name="pelota" Height="50" Width="50"
Canvas.Left="130" Canvas.Top="10">
<Ellipse.Fill>
<LinearGradientBrush EndPoint="0.5,1"
StartPoint="0.5,0">
<GradientStop Color="Black" Offset="0"/>
<GradientStop Color="#FF485EE6" Offset="1"/>
</LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
</Canvas>
</Grid>
```

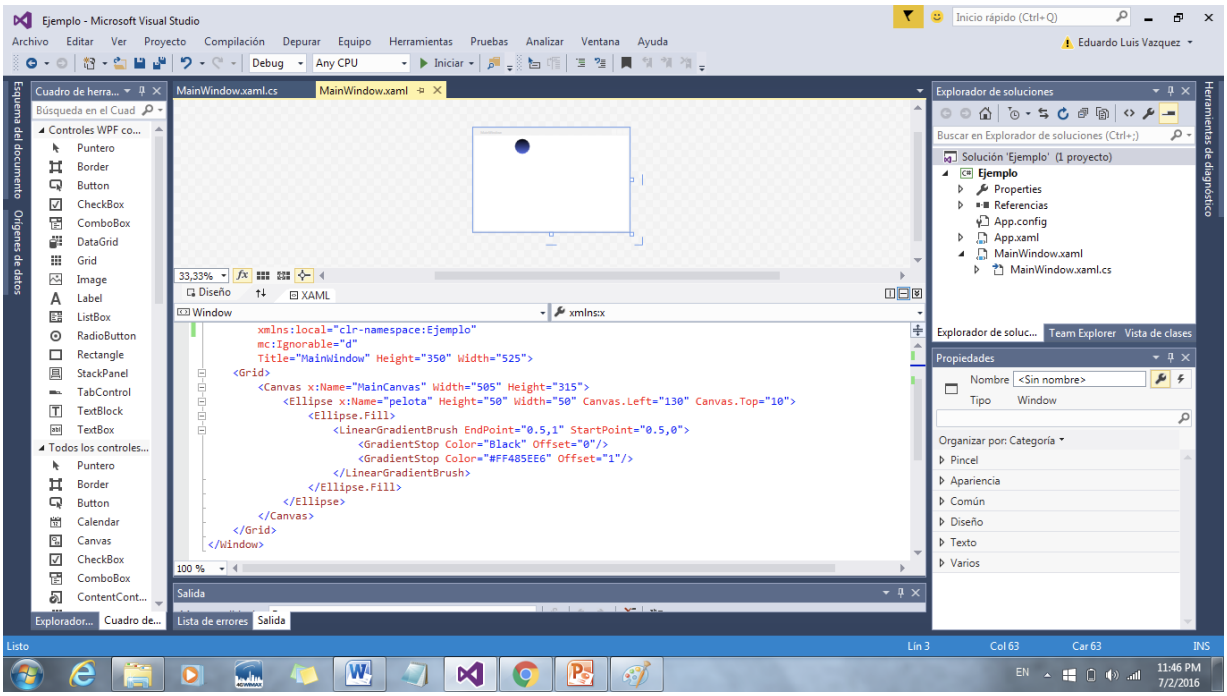


Figura 2.1

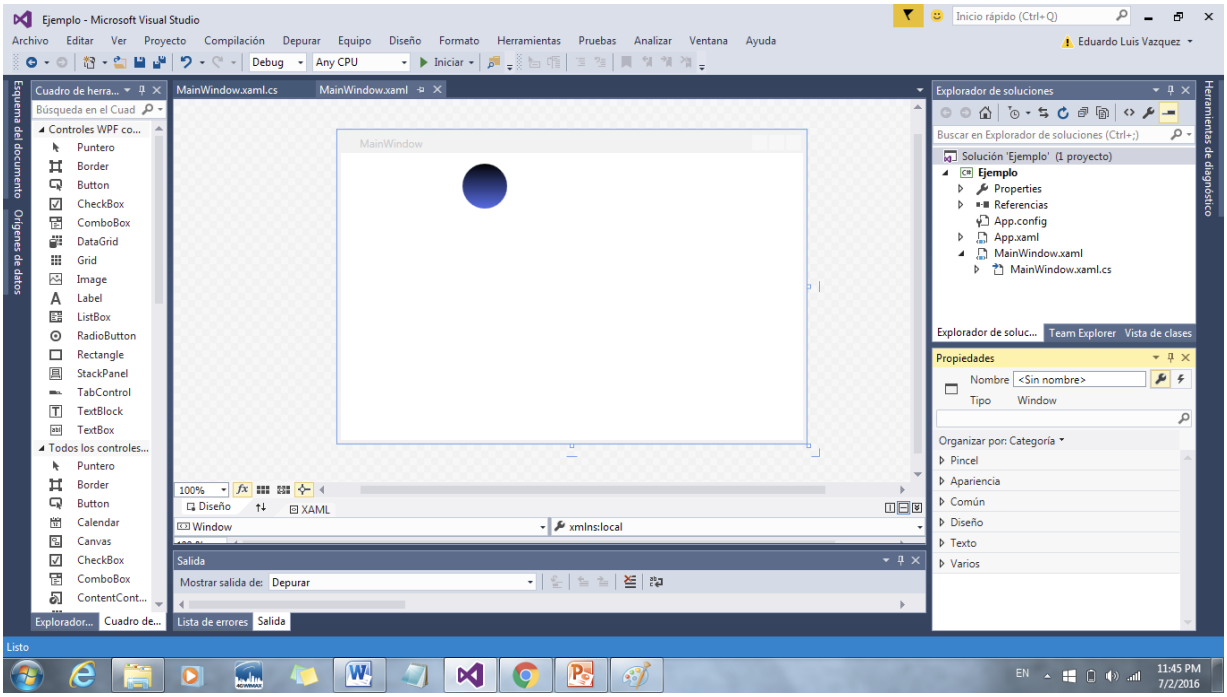


Figura 2.2

Código en C#

En el archivo **MainWindow.xaml.cs** el primer cambio que se debe hacer es agregar la biblioteca requerida para emplear el **DispatcherTimer**.

```
using System.Windows.Threading;
```

Dentro de la clase y antes del método **MainWindow** se declara la variable **timer** de tipo **DispatcherTimer**. También se declaran las variables para almacenar la altura del **Canvas** y la cantidad de píxeles que se moverá la pelota cada vez que se cumpla el intervalo.

```
public partial class MainWindow : Window
{
    //Variable de tipo DispatcherTimer
    DispatcherTimer timer;
    //Variable para almacenar el alto del Canvas
    double dAltoCanvas;
    //Cantidad de pixeles a mover la pelota
    int iPixeles = 5;
```

Dentro del método **MainWindow**, además de enfocar al **Canvas** y obtener la altura del mismo, se colocan las instrucciones para crear el evento. Para lograrlo, primero se crea el **DispatcherTimer** y después se establecen sus características (**Interval**, **Tick** y **IsEnabled**). Con la propiedad Interval, se indica que el evento debe ser ejecutado cada 20 milisegundos (esta cantidad de tiempo se puede ajustar dependiendo del equipo computacional en el que se ejecute el programa, el intervalo determinará la velocidad con la que avanza la pelota).

```
public MainWindow()
{
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener la altura del Canvas
    dAltoCanvas = MainCanvas.Height;
    //Crea el DispatcherTimer
```

```
        timer = new DispatcherTimer();
        //Especificar el intervalo (cada cuánto tiempo se ejecuta el evento)
        timer .Interval = new TimeSpan(0, 0, 0, 0, 20 );
        //Crear el evento
        timer .Tick += new EventHandler(Timer_Tick );
        //Iniciar el evento
        timer .IsEnabled = true;
    }
```

Después, con la propiedad Tick, se crea el evento relacionando el **timer** con el método Timer_Tick. Por último, para dar inicio al movimiento de la pelota se asigna el valor de verdadero a la propiedad IsEnabled.

Como se mencionó anteriormente, el método Timer_Tick se ejecuta cada vez que se cumpla el intervalo de tiempo establecido para el **DispatcherTimer**. Este método se encarga de mover la pelota hacia arriba y hacia abajo de manera automática dentro del **Canvas**. La variable **iPixeles**, que contiene la cantidad de píxeles que se debe mover la pelota en cada ocasión, tendrá un valor positivo o negativo; cuando el valor es positivo, la pelota va hacia abajo, pero cuando es negativo el movimiento de la pelota es hacia arriba. Para determinar el sentido del movimiento se verifica la posición actual de pelota; si no hay espacio para que avance entonces se hace el cambio de dirección (se multiplica por -1 el contenido de iPixeles; recordar que un valor negativo multiplicado por -1 da el mismo valor pero positivo o bien, un valor positivo multiplicado por -1 resulta en el mismo valor pero negativo dando la alternancia requerida para el movimiento).

```
    private void Timer_Tick(object sender, EventArgs e)
    {
        //Coordenada de la esquina superior izquierda de la elipse
        double dPosY = (double)pelota.GetValue(Canvas.TopProperty);
        if ( iPixeles > 0 )
        { //Movimiento hacia abajo
        if (dPosY + pelota.Height + iPixeles < dAltoCanvas)
        dPosY = dPosY + iPixeles;
```

```
            else iPixeles = iPixeles * -1; //Cambio de dirección del movimiento
            }
            else
            { //Movimiento hacia arriba
            if (dPosY + iPixeles > 0)
            dPosY = dPosY + iPixeles;
            else iPixeles = iPixeles * -1; //Cambio de dirección del movimiento
            }
            //Mueve la pelota a la nueva coordenada
            pelota.SetValue(Canvas.TopProperty, dPosY);
    }
```

Ejecución

Cuando se ejecuta el programa, y sin la intervención del usuario, la pelota se mueve hacia abajo, rebota y se regresa en sentido contrario por el mismo camino.

Al llegar nuevamente a la parte superior, vuelve a bajar y continua en ese movimiento hasta que se cierra la ventana. Las imágenes muestran dos momentos de la ejecución.

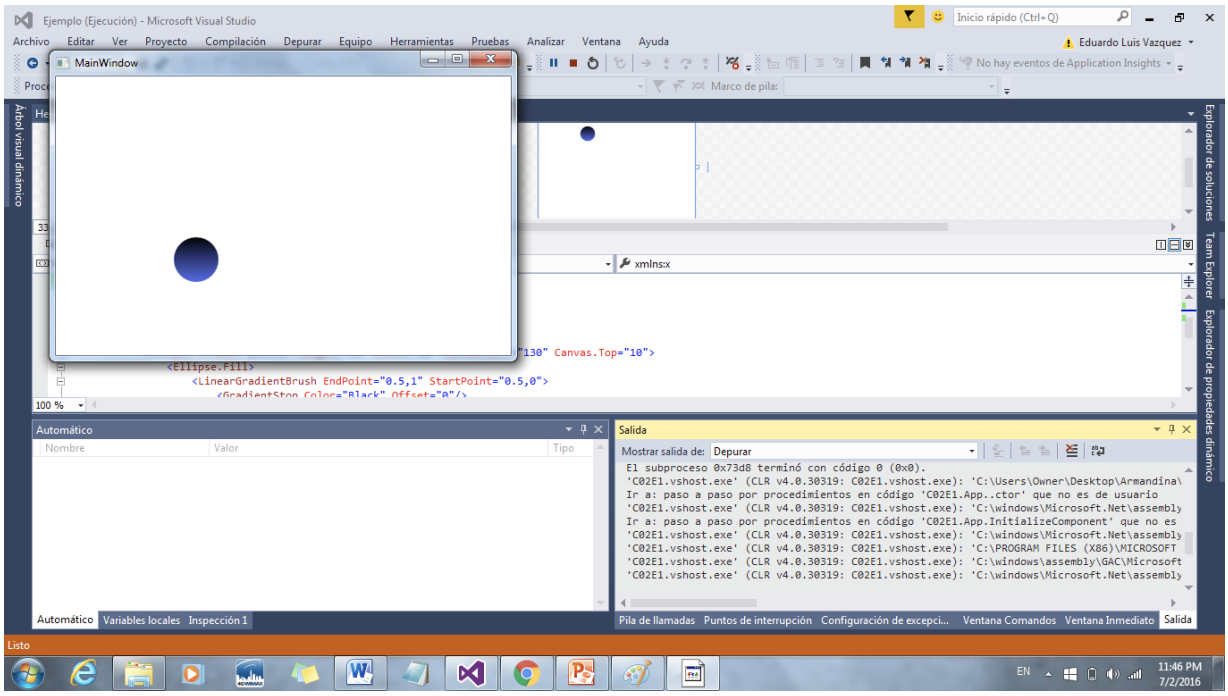


Figura 2.3

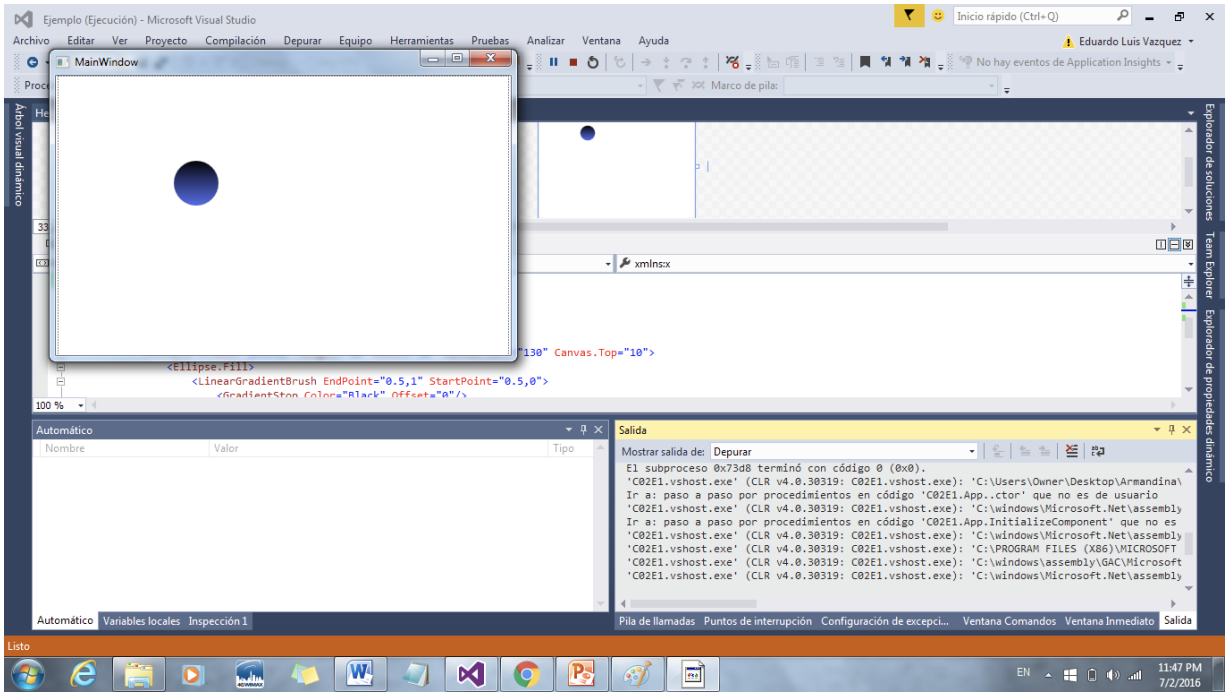


Figura 2.4

2.3 Ejemplo: colapsar rectángulos

Este programa emplea un DispatcherTimer para mover una pelota que choca contra rectángulos que al recibir el impacto colapsan y desaparecen.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Windows.Threading y System.Media**

Código en XAML

El archivo **MainWindow.xaml** muestra al **Canvas** con un StackPanel con tres rectángulos (rect1, rect2, rect3) y una elipse de nombre pelota. Un StackPanel es un panel al igual que el **Grid** y el **Canvas**. Este tipo de panel se emplea para acomodar objetos uno a lado del otro. En este caso, se empleará para que cuando se colapse un rectángulo el que sigue pueda ocupar su lugar.

```
<Grid>
  <Canvas Name="MainCanvas" HorizontalAlignment="Right" Width="517">
    <StackPanel Canvas.Left="417">
      <Rectangle x:Name="rect1" Height="100" Width="100"
          Canvas.Left="406" Canvas.Top="10"
          Stroke="Black" Fill="#FFF52611"
          Visibility="Visible" />
      <Rectangle x:Name="rect2" Height="100" Width="100"
          Canvas.Left="406" Canvas.Top="112"
          Stroke="Black" Fill="#FF1BF111"
          Visibility="Visible"/>
      <Rectangle x:Name="rect3" Height="100" Width="100"
          Canvas.Left="406" Canvas.Top="210"
          Stroke="Black" Fill="#FF1616F7"
          Visibility="Visible"/>
    </StackPanel>
```

```
        <Ellipse x:Name="pelota" Height="30" Width="3x0"
                 Fill="Black" Stroke="Black"
                 Canvas.Top="51" Canvas.Left="10"/>
    </Canvas>
</Grid>
```

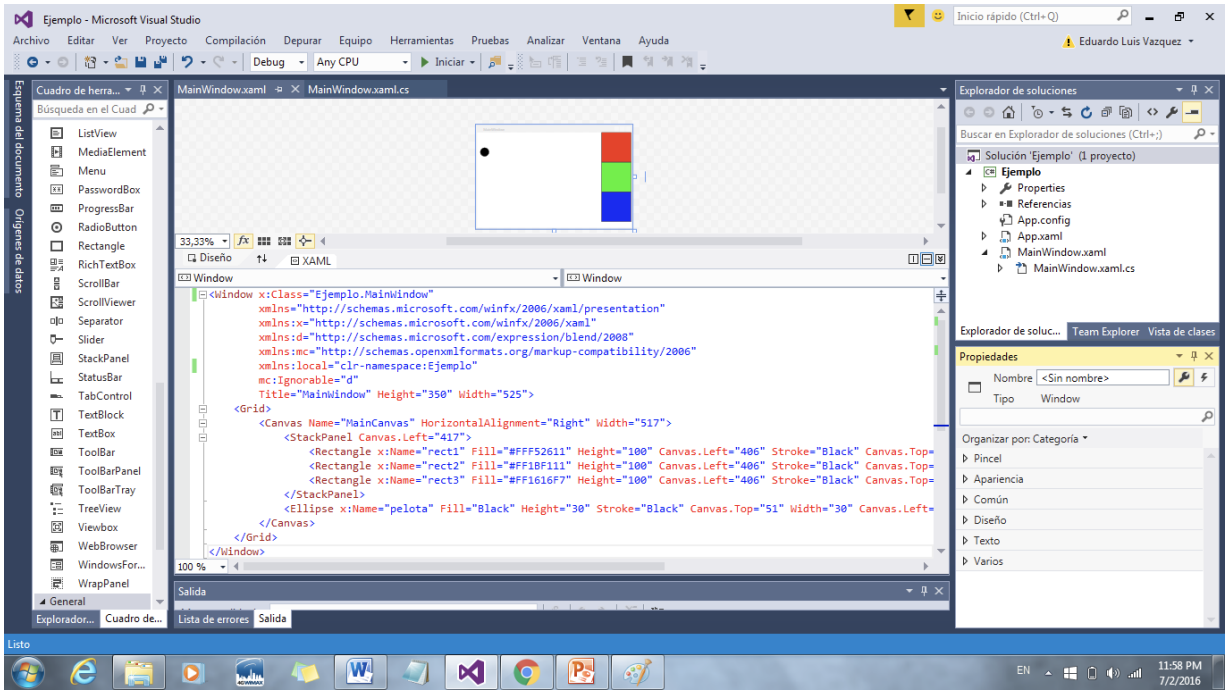


Figura 2.5

Código en C#

Al archivo **MainWindow.xaml.cs** se le añaden dos bibliotecas; la primera de ellas para utilizar el **DispatcherTimer** y la segunda para producir un sonido cada vez que choca la pelota con un rectángulo.

```
using System.Windows.Threading; //Para emplear el DispatcherTimer
using System.Media; //Para emplear sonidos del Sistema
```

Dentro de la clase y antes del método **MainWindow** se declaran las variables que se requieren en todos los métodos del programa. La variable timer de tipo **DispatcherTimer** servirá para crear el evento que moverá la pelota de forma automática. Para guardar los límites de movimiento, se declaran variables para almacenar el ancho del **Canvas** y de la pelota. En la variable dPosInicialPelota, se almacenará la posición inicial en la que se

encuentra la pelota ya que cada vez que la pelota golpea un rectángulo, esta se regresa a su posición inicial. La variable iRectángulo se empleará para indicar cuál rectángulo se colapsará y la variable iPixeles, la cantidad de pixeles que se moverá la pelota cada vez que se cumpla el intervalo.

```
public partial class MainWindow : Window
{
    //Variable para control del evento
    DispatcherTimer timer ;
    //Variables para almacenar el ancho del Canvas y la pelota
    double dAnchoCanvas;
    double dAnchoPelota;
    //Variable para almacenar la posición inicial de la pelota
    double dPosInicialPelota ;
    //Indicador de cuál rectángulo se colapsará
    int iRectangulo = 1;
    //Cantidad de pixeles a mover la pelota cada vez que se cumpla el
    intervalo
    int iPixeles = 5;
```

Dentro del método **MainWindow** además de enfocar el **Canvas** y obtener las dimensiones de los objetos, se obtiene la posición inicial de la pelota para regresarla ahí cada vez que choca con un rectángulo. También se crea un **DispatcherTimer** que ejecuta el método **Timer_TicK** cada 20 milisegundos.

```
public MainWindow()
{
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
```

```
        //Obtener el ancho del Canvas y de la pelota
        dAnchoCanvas = MainCanvas.Width;
        dAnchoPelota = pelota.Width;
        //Obtener la posición inicial de la pelota para regresarla ahí
        //cada vez que choca con un rectángulo
        dPosInicialPelota = (double)pelota.GetValue(Canvas.LeftProperty);
        //Crea un DispatcherTimer que ejecuta el método Timer_TicK cada
        //20 milisegundos
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 0, 20);
        timer.Tick += new EventHandler(Timer_Tick);
        timer.IsEnabled = true;
    }
```

El método Timer_Tick se encarga de mover la pelota horizontalmente hacia la derecha. Lo primero que hace es verificar si la pelota puede continuar avanzando, es decir, aún no choca con un rectángulo; si puede continuar lo mueve y termina la ejecución del método.

Si la pelota alcanza la orilla donde inicia el rectángulo, verifica con cuál de los rectángulos está chocando para colapsarlo lo cual hará que automáticamente desaparezca y los otros rectángulos del **StackPanel** ocupen su lugar. Cada vez que la pelota choca con un rectángulo se reproduce el sonido **Hand** el cual se escucha parecido a un golpe; este es uno de los sonidos predefinidos del sistema y es factible utilizarlo porque se agregó la biblioteca **System.Media**. Cuando ya se colapsaron los tres rectángulos, se restablece el ambiente haciendo visibles todos los rectángulos y reiniciando el valor de la variable **iRectangulo** a 1. Finalmente, después del choque, la pelota regresa a su posición original.

```
    private void Timer_Tick(object sender, EventArgs e)
    {
      //Obtiene la posición actual de la pelota
      double dPosX = (double)pelota.GetValue(Canvas.LeftProperty);
      if (dPosX + dAnchoPelota + iPixeles < dAnchoCanvas – rect1.Width)
```

```
{ //Avanza la pelota a hacia la derecha mientras
    //no choque con el rectángulo
    dPosX = dPosX + iPixeles;
    pelota.SetValue(Canvas.LeftProperty, dPosX);
}
else
{ //Cuando la pelota choca con el rectángulo lo colapsa
    //permitiendo que el siguiente de la secuencia tome su lugar
    if (iRectangulo == 1)
        rect1.Visibility = Visibility.Collapsed;
    else if (iRectangulo == 2)
        rect2.Visibility = Visibility.Collapsed;
    else if (iRectangulo == 3)
        rect3.Visibility = Visibility.Collapsed;
    //Producir un sonido del sistema para indicar el choque
    SystemSounds.Hand.Play();
    //Incrementar el contador de rectángulos colapsados
    iRectangulo++;
    //Si ya se colapsaron los tres rectángulos los aparece de nuevo
    if (iRectangulo == 4)
    {
        iRectangulo = 1;
        rect1.Visibility = Visibility.Visible;
        rect2.Visibility = Visibility.Visible;
        rect3.Visibility = Visibility.Visible;
```

```
        }
        //Regresar la pelota a la posición inicial
        pelota.SetValue(Canvas.LeftProperty, dPosInicialPelota);
    }
}
```

Ejecución

El programa, al ser ejecutado, muestra la pelota moviéndose automáticamente de izquierda a derecha. Cada vez que choca con un rectángulo lo desaparece.

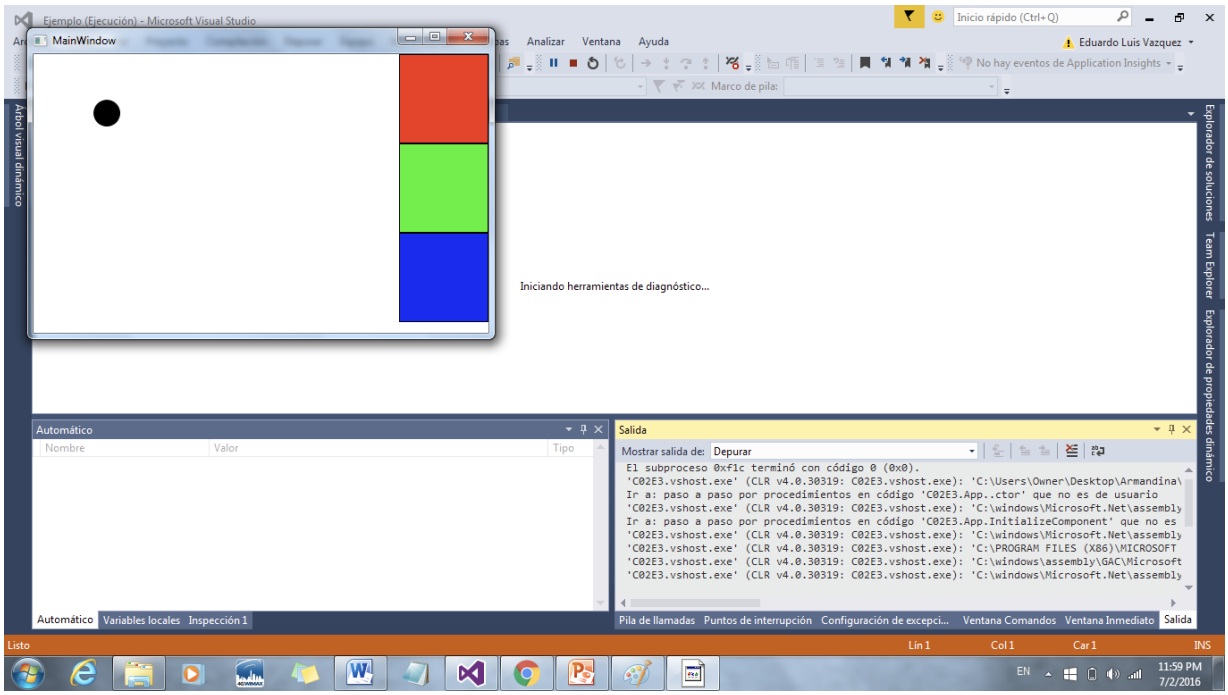


Figura 2.6

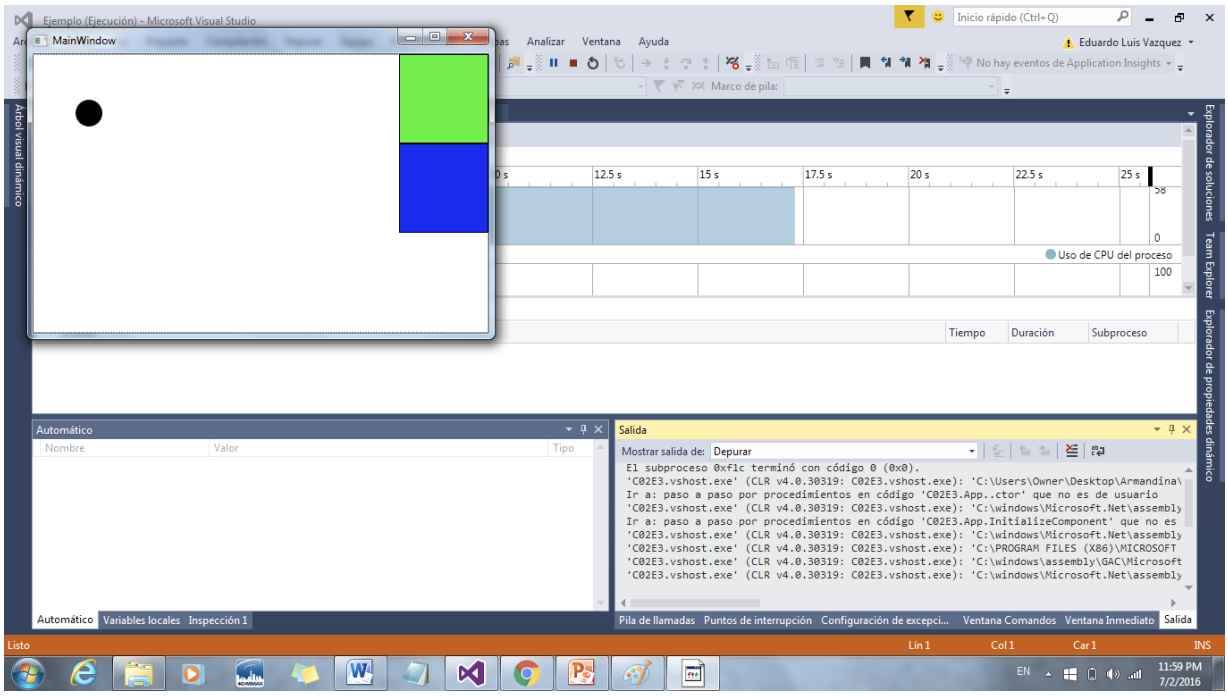


Figura 2.7

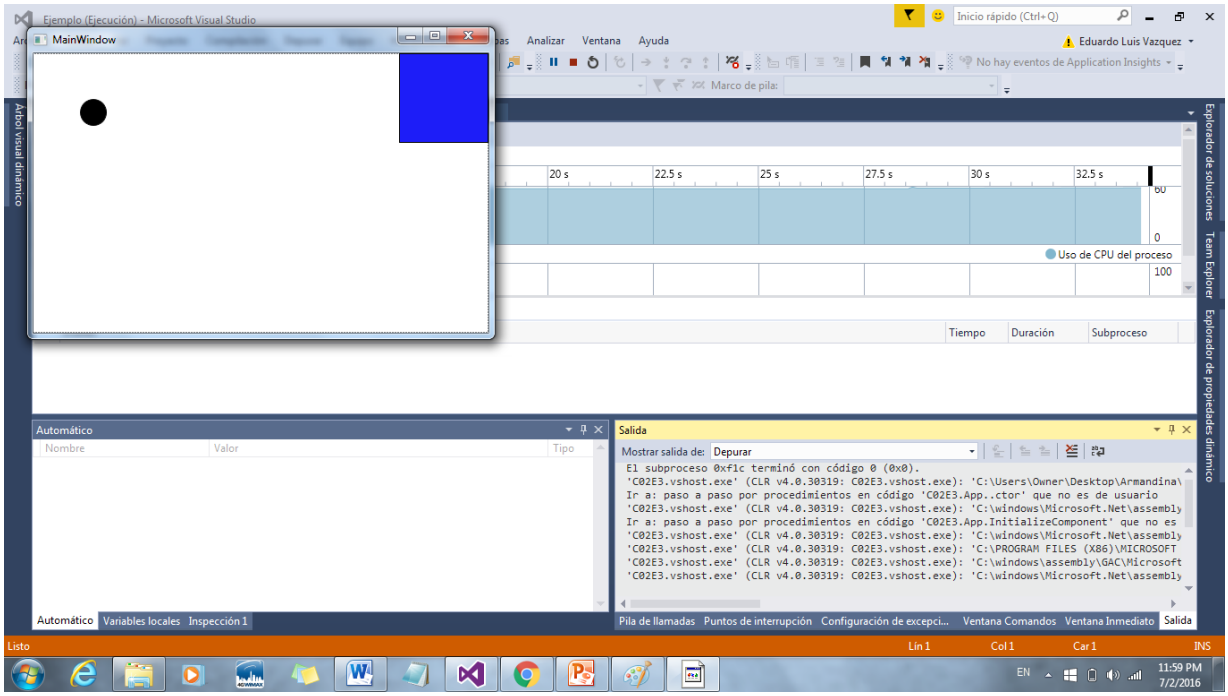


Figura 2.8

2.4 Ejemplo: imagen de fondo en movimiento

Esta aplicación muestra un disco que, aparentemente, se está moviendo de manera horizontal desplazándose a través de un escenario. Aun cuando pareciese que el

disco es el que se mueve, internamente lo que está en movimiento es la imagen que está al fondo.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Windows.Threading**
- Imágenes: un fondo y cuatro imágenes de un disco.

Recursos

La imagen de fondo es ancha para incluir varios escenarios. En este caso, muestra nubes colocadas en distintas posiciones y de diferentes formas.

fondo.png

Figura 2.9

Para dar la sensación de que el disco está rotando, se intercalan varias imágenes para hacer el efecto de giro hacia delante o giro hacia atrás. Dado que el disco se coloca encima del fondo es importantes que la imagen sea transparente (se puede emplear cualquier paquete de dibujo para hacer transparencia).

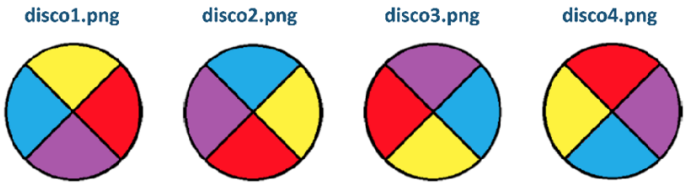


Figura 2.10

Las imágenes deben ser añadidas al proyecto. Con el fin de organizar los archivos contenidos en el proyecto, se creó una carpeta de nombre **Imágenes** donde se colocaron ahí la imagen de fondo y las cuatro imágenes del disco.

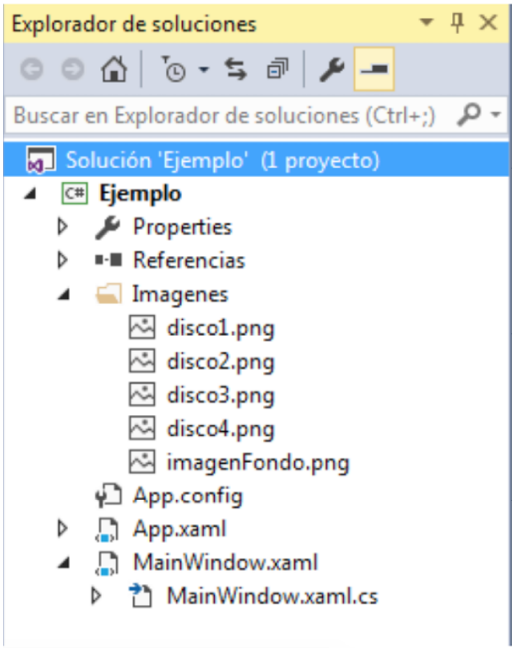


Figura 2.11

Código en XAML

El archivo **MainWindow.xaml** contiene un **Canvas** con dos imágenes. La primera de ellas es la imagen de fondo la cual tiene un ancho mucho mayor al del panel. A esta, se le ha establecido la propiedad **Strech** con el fin de alargar la imagen a las dimensiones del objeto. La segunda imagen es la del disco el cual es colocado en el centro de la parte inferior del Canvas.

```
<Grid>
  <Canvas Name="MainCanvas" Width="520">
    <Image x:Name="fondo" Height="317" Width="1200"
        Source="Imagenes/imagenFondo.png"
        Stretch="UniformToFill"
        Canvas.Left="1" Canvas.Top="1"/>
    <Image x:Name="disco" Height="100" Width="100"
        Source="Imagenes/disco1.png"
        Canvas.Left="211" Canvas.Top="210"/>
  </Canvas>
```

```
</Grid>
```

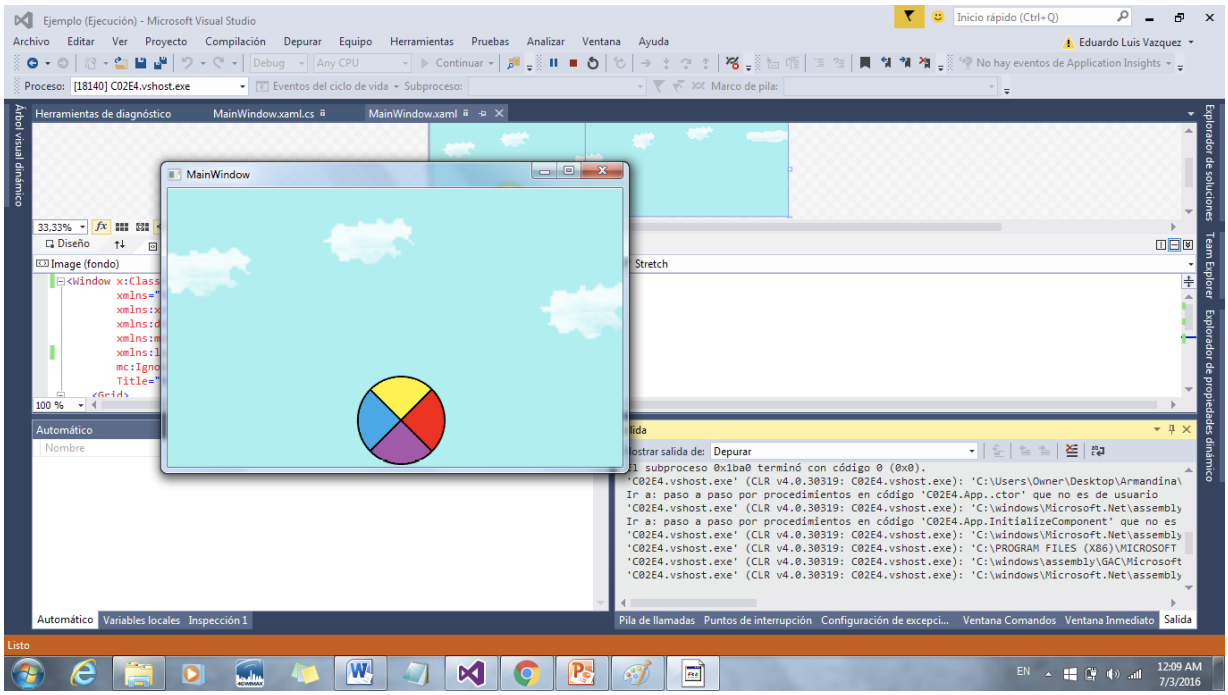


Figura 2.12

Código en C#

El primer cambio a realizar en el archivo **MainWindow.xaml.cs** es incluir la biblioteca para utilizar el **DispatcherTimer**.

```
using System.Windows.Threading; //Para emplear el DispatcherTimer
```

Al inicio del programa se declara la variable para el **DispatcherTimer**, que se empleará para mover la imagen de fondo; también se incluye una variable para almacenar la posición en la que se encuentra el fondo en todo momento. Dado que se emplearán cuatro imágenes para dar la apariencia de que el disco está rotando, se definen cuatro variables de tipo **BitmapImage** y se le asigna a cada una de ellas la imagen correspondiente al mismo tiempo, se tiene la variable **iNumDisco** para contar con un indicador de cuál imagen es la que se está mostrando en ese momento.

```
public partial class MainWindow : Window
{
    //Variable para control del evento
    DispatcherTimer timer;
```

```
//Variables para almacenar el ancho del Canvas y de la imagen de fondo
double dAnchoCanvas, dAnchoFondo;
//Variable para almacenar la posición de la imagen de fondo
double dPosicionFondo;
//Cantidad de pixeles que se mueve el fondo en cada ocasión
int iPixeles = 5;
//Número de la imagen que sigue por mostrar
int iNumDisco = 1;
//Imágenes para crear el efecto de giro de del disco
BitmapImage bmUno = new BitmapImage(new Uri(@"Imagenes/disco1.png ",
UriKind.RelativeOrAbsolute));
BitmapImage bmDos = new BitmapImage(new Uri(@"Imagenes/disco2.png ",
UriKind.RelativeOrAbsolute));
BitmapImage bmTres = new BitmapImage(new Uri(@"Imagenes/disco3.png ",
UriKind.RelativeOrAbsolute));
BitmapImage bmCuatro = new BitmapImage(new Uri(@"Imagenes/disco4.png ",
UriKind.RelativeOrAbsolute));
```

El método **MainWindow** inicia al enfocar el **Canvas**; posteriormente obtiene la dimensión del Canvas y del fondo con el fin de poder controlar cuándo terminar el movimiento de la imagen de fondo. Además se crea un **DispatcherTimer** que ejecuta cada 100 milisegundos el método Timer_Tick que se encarga de mover la imagen de fondo y de rotar el disco.

```
public MainWindow()
```

```
{
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho del Canvas y de la imagen de fondo
    dAnchoCanvas = MainCanvas.Width;
    dAnchoFondo = fondo.Width;
    //Crea un DispatcherTimer que ejecuta el método Timer_TicK cada
    100 ms
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 0, 100 );
    timer.Tick += new EventHandler(Timer_Tick );
    timer.IsEnabled = true; //Inicia la actividad del evento
  }
```

El método Timer_Tick verifica la posición actual de la imagen de fondo; si aún no se ha mostrado toda la imagen, es decir, si aún se puede mover hacia la izquierda sin que por eso quede al descubierto el **Canvas**, se realiza el movimiento y se pide la ejecución del método que se encarga de rotar el disco. Cuando se ha terminado de pasar por la ventana toda la imagen del fondo, se deshabilita el evento ya que no tiene sentido continuar ejecutándolo.

```
private void Timer_Tick(object sender, EventArgs e)
{
    //Obtener la posición actual de la imagen de fondo
    dPosicionFondo = (double)fondo.GetValue(Canvas.LeftProperty);
    //Mover el fondo si aún no se ha mostrado todo
    if (dPosicionFondo + dAnchoFondo + iPixeles > dAnchoCanvas)
    {
```

```
            fondo.SetValue(Canvas.LeftProperty, dPosicionFondo -
            iPixeles);
            RotaDerecha(); //Rotar el disco hacia la derecha
        }
        else
            timer.IsEnabled = false; //Deshabilitar el evento
}
```

El método RotaDerecha se ejecuta cada vez que se mueve la imagen de fondo. Este método se encarga de cambiar la imagen del disco de tal manera que simule que este rota. La variable **iNumDisco** indica el número de la imagen que se debe presentar; después de cambiar la imagen, se actualiza para que contenga el valor adecuado para la siguiente ocasión. Dado que solamente se tienen cuatro imágenes, esta variable toma los valores de 1 al 4. Cuando alcanza el número 5, se corrige al valor 1.

```
private void RotaDerecha()
{
  //Cambiar la imagen
  switch (iNumDisco)
  {
     case 1: disco.Source = bmDos; break;
     case 2: disco.Source = bmTres; break;
     case 3: disco.Source = bmCuatro; break;
     case 4: disco.Source = bmUno; break;
  }
  //Pasar el indicador a la siguiente imagen
  iNumDisco++;
  if (iNumDisco == 5) iNumDisco = 1;
}
```

Ejecución

Al ejecutar el programa se debe observar que la pelota gira mientras que el paisaje de fondo cambia.

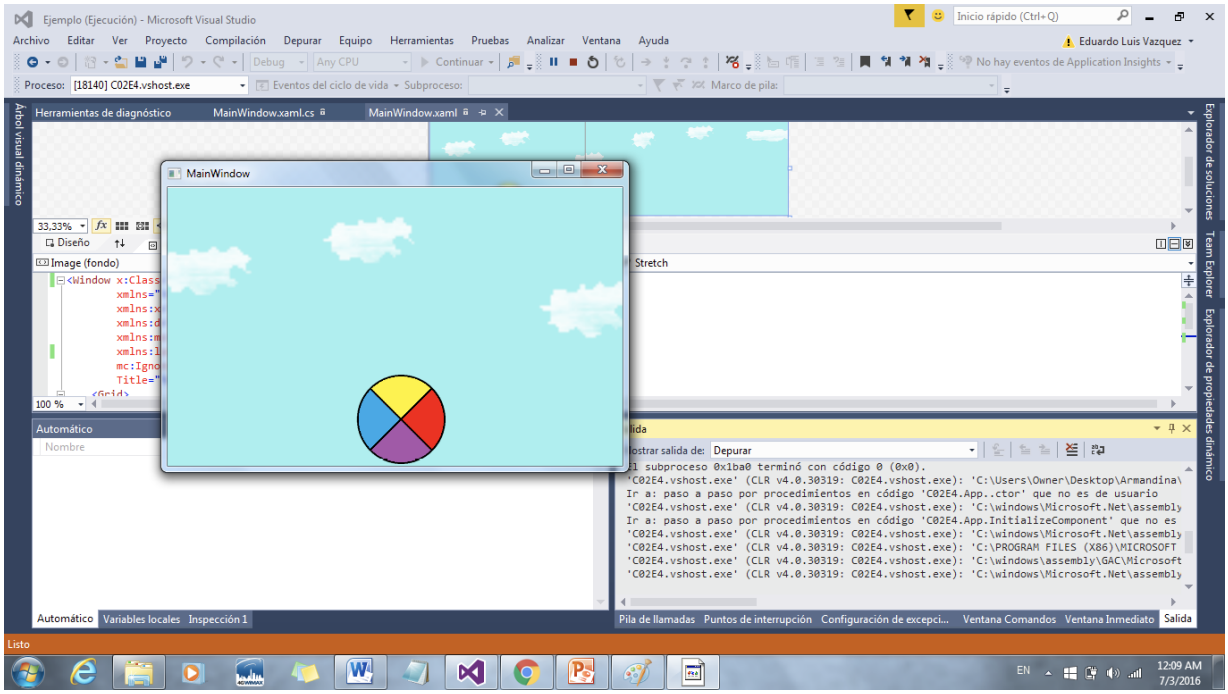


Figura 2.13

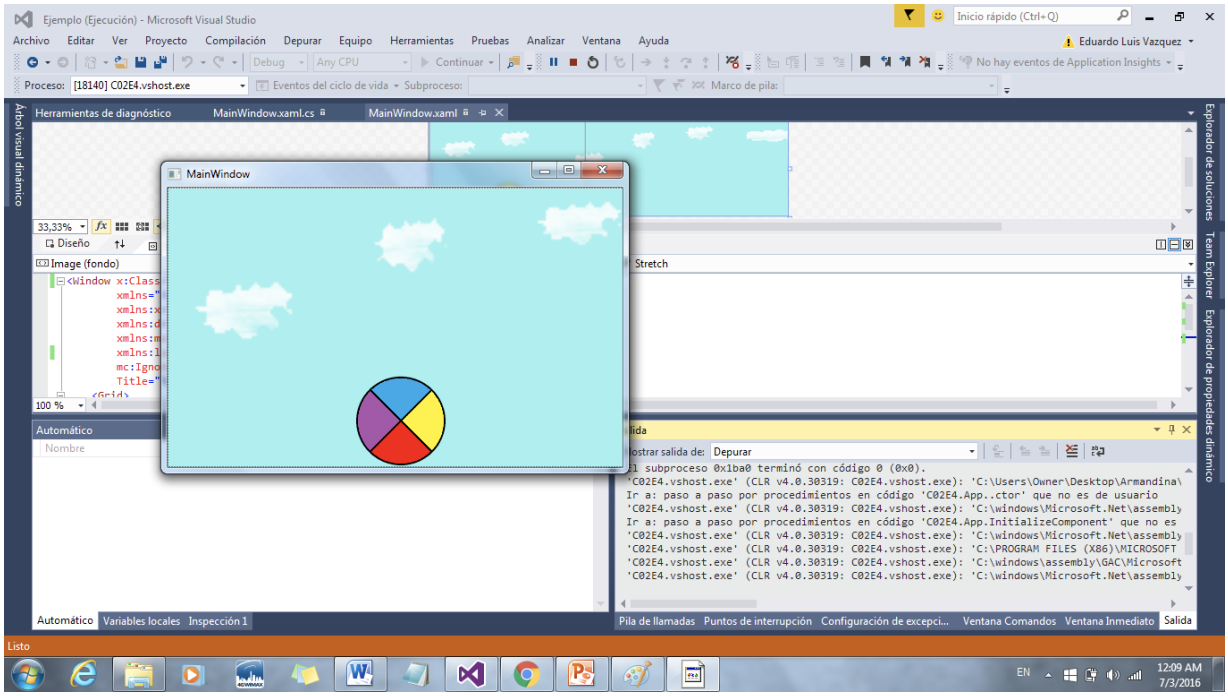


Figura 2.14

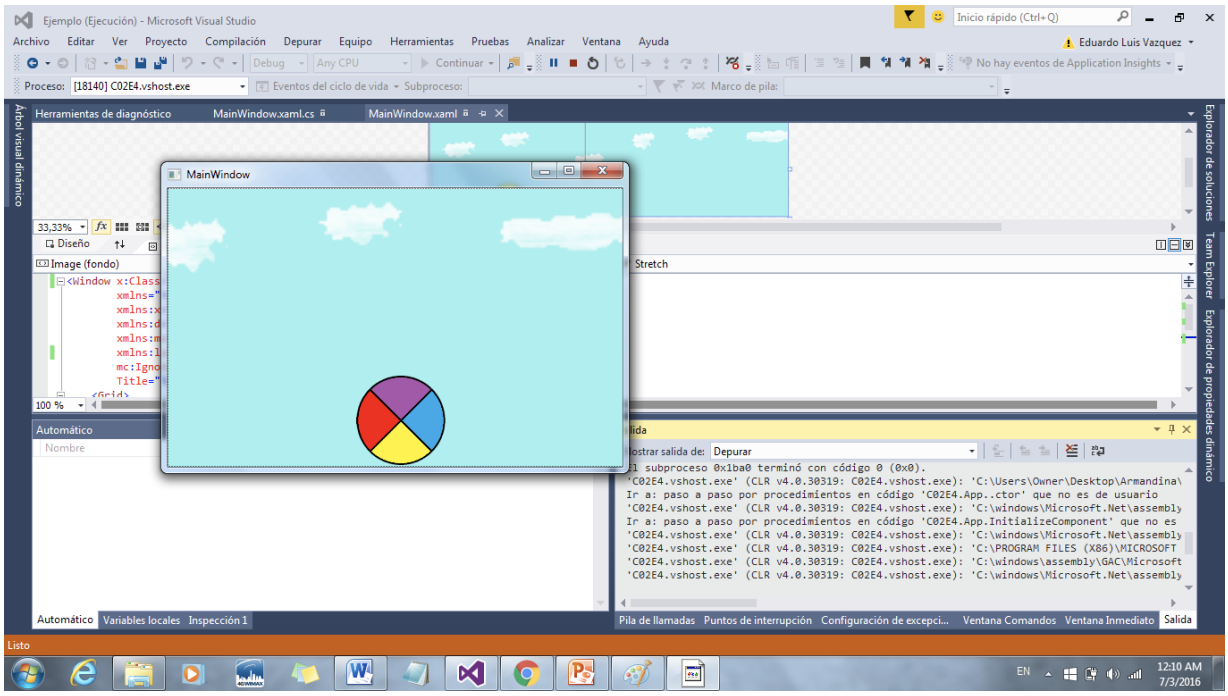


Figura 2.15

2.5 Ejemplo: atrapar una estrella

Este ejemplo combina eventos del ratón con eventos del **DispatcherTimer** para realizar un juego en el que el usuario debe dar clic a una estrella que desaparece y aparece en posiciones al azar. El programa cuenta las veces que el usuario le atina; cuando el contador tiene un múltiplo de 5, la estrella crece momentáneamente.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Windows.Threading** y **System.Media**
- Imágenes: una estrella.

Código en XAML

El contenido del archivo **MainWindow.xaml** define dentro del panel **Grid** un Canvas: dos botones y dos etiquetas. Para este caso, el **Canvas** solo ocupa la parte izquierda de la ventana para poder colocar del lado derecho los botones y las etiquetas.

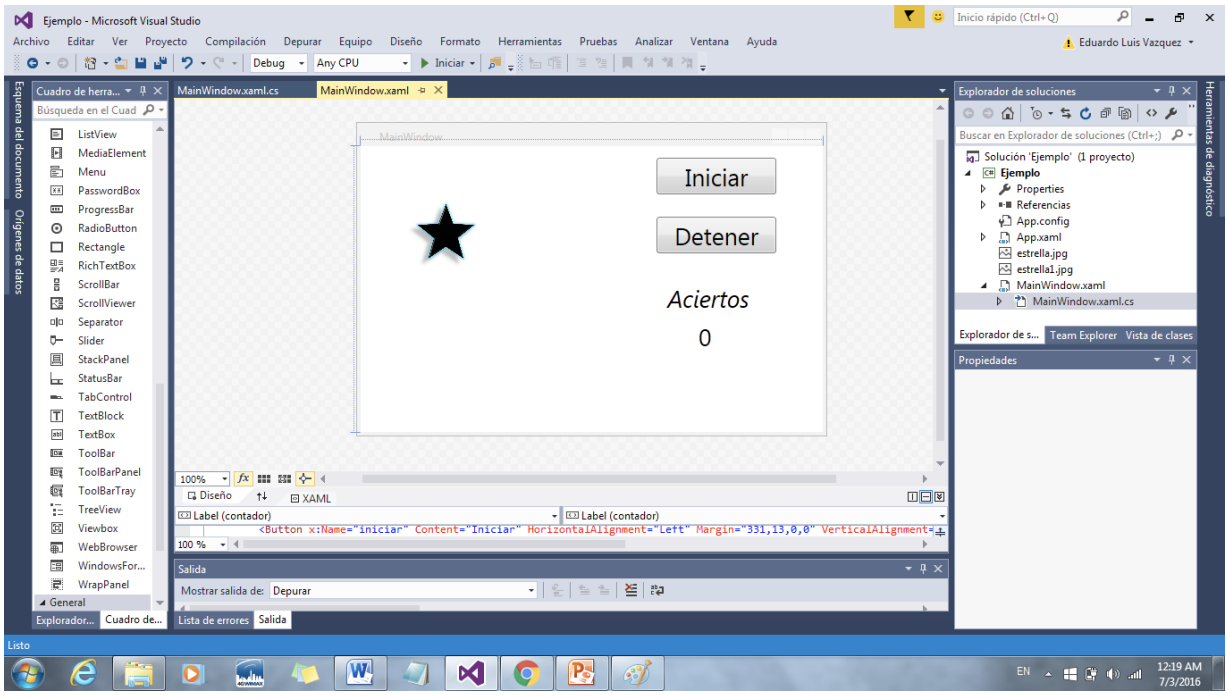


Figura 2.16

El **Canvas** incluye un evento del tipo MouseLeftButtonUp el cual se activa cuando el botón izquierdo del ratón es liberado (es decir, cuando se termina de dar un clic). Este evento ejecutará el método **MainCanvas_MouseLeftButtonUp**, encargado de realizar todas las actividades relacionadas con la movilidad de la estrella. Dentro del **Canvas** se ha colocado la imagen de la estrella.

De lado derecho del **Grid** se encuentran el botón para iniciar y otro para detener la actividad de la estrella; cada uno de ellos tiene asociado un evento de tipo **Click** que se activa cuando el usuario presiona el botón correspondiente.

La etiqueta de nombre contador se utiliza para mostrar al usuario la cantidad de veces que le ha dado clic a la estrella.

```
<Grid>
  <Canvas Name="MainCanvas" Width="300" Height="340"
 Margin="0,0,217,-20"
     MouseLeftButtonUp="MainCanvas_MouseLeftButtonUp" >
     <Image x:Name="estrella" Height="88" Width="84"
          Source="estrella1.jpg" Canvas.Left="48" Canvas.Top="52"/>
  </Canvas>
```

```
    <Button x:Name="iniciar" Content="Iniciar" FontSize="26.667"
        VerticalAlignment="Top" HorizontalAlignment="Left"
        Margin="331,13,0,0" Width="134" Click="iniciar_Click"/>
    <Button x:Name="detener" Content="Detener" FontSize="26.667"
        HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="331,79,0,0" Width="134" Click="detener_Click"/>
    <Label x:Name="contador" Content="0" Width="92"
  FontSize="26.667"
        Margin="373,190,52,65"/>
    <Label x:Name="label" Content=" Aciertos" Width="134"
        FontSize="26.667" FontStyle="Italic"
    Margin="331,147,52,121"/>
  </Grid>
```

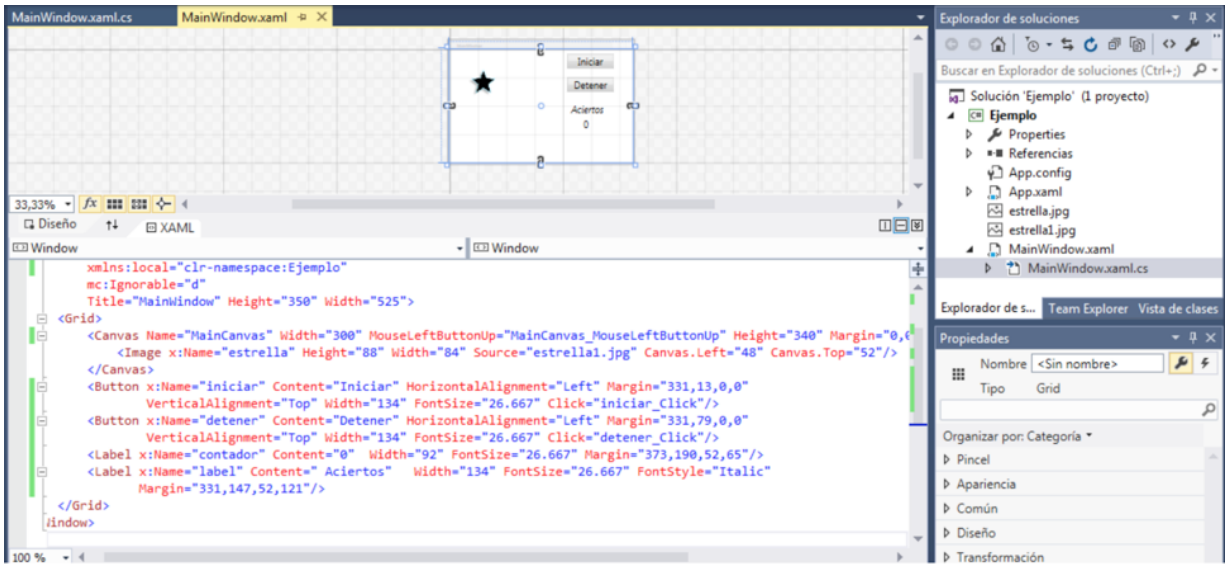


Figura 2.17

Código en C#

Para esta aplicación se requiere añadir la biblioteca para crear eventos con el **DispatcherTimer** y la biblioteca que permite el uso de los sonidos predeterminados del

sistema.

```
using System.Windows.Threading; //Para emplear la clase DispatcherTimer
using System.Media; //Para emplear sonidos del Sistema
```

Las primeras instrucciones a realizar son la declaración de la variable que se empleará para crear el evento relacionado con el tiempo y las que se emplean para almacenar la dimensión tanto del **Canvas** como de la estrella.

Aquí también se crea el objeto random de tipo **Random** el cual permite generar números al azar. Dado que la estrella estará desapareciendo y apareciendo, las posiciones de la estrella serán definidas con los números que se generen. El programa inicializa en este punto con la variable iContador, la cual se emplea para llevar la cuenta de las veces que el usuario dio clic sobre la estrella. Además se inicializa la variable booleana bActividad la cual solamente puede contener el valor **true** (para verdadero) y **false** (para falso). Esta variable se usa para saber si el juego está detenido o no.

```
public partial class MainWindow : Window
{
    //Variable para control del evento
    DispatcherTimer timer;
    //Variables para almacenar dimensiones del Canvas y la estrella
    double dAltoCanvas, dAnchoCanvas;
    double dAltoEstrella, dAnchoEstrella;
    //Generador de números al azar
    Random random = new Random();
    //Contador de aciertos
    int iContador = 0;
    //Indicador de actividad
    bool bActividad = false;
```

Dentro del **MainWindow**, además de enfocar el **Canvas** y obtener las dimensiones de este y de la estrella, se crea un **DispatcherTimer** que ejecuta el método **Timer_Tick** cada segundo.

```
public MainWindow()
{
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas y de la estrella
    dAnchoCanvas = MainCanvas.Width;
    dAltoCanvas = MainCanvas.Height;
    dAnchoEstrella = estrella.Width;
    dAltoEstrella = estrella.Height;
    //Crea un DispatcherTimer que ejecuta el método Timer_TicK cada
    segundo
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 1, 0);
    timer.Tick += new EventHandler(Timer_Tick);
}
```

El método Timer_Tick se encarga únicamente de cambiar la estrella de lugar. Utiliza el método Next para lanzar un número al azar entre 1 y la diferencia entre el tamaño del Canvas y el tamaño de la estrella, dado que esta nueva posición debe ser tal que la estrella se muestre completa. Después de ubicarla en la nueva posición, restablecerá sus dimensiones.

```
private void Timer_Tick (object sender, EventArgs e)
{
    //Obtiene una posición al azar para ubicar al estrella
    double dPosX = random.Next (1, (int)(dAnchoCanvas -
    estrella.Width));
```

```
            double dPosY = random.Next (1, (int)(dAltoCanvas -
            estrella.Height));
            //Mueve el estrella a la nueva coordenada
            estrella.SetValue(Canvas.LeftProperty, dPosX);
            estrella.SetValue(Canvas.TopProperty, dPosY);
            //Ajusta el tamaño del estrella a su valor original
            estrella.Width = dAnchoEstrella;
            estrella.Height = dAltoEstrella;
    }
```

El método MainCanvas_MouseLeftButtonUp se ejecuta cada vez que el usuario da clic sobre la estrella. Si no se ha presionado el botón para detenerlo, el programa se contabiliza y escribe en la ventana la cantidad de aciertos (veces que se le ha dado clic a la estrella). Cada vez que el contador tiene un múltiplo de cinco, hace un sonido y aumenta el tamaño de la estrella tres veces su dimensión original.

```
    private void MainCanvas_MouseLeftButtonUp(object sender,
MouseButtonEventArgs e)
    {
      if (bActividad)
      {
          //Incrementa la cantidad de aciertos
          iContador++;
          //Muestra la cantidad de aciertos en la etiqueta contador
          contador.Content = iContador.ToString();
          if (iContador % 5 == 0)
          {
              //Sonido para indicar que se acumularon 5 clics sobre la
              estrella
              SystemSounds.Exclamation.Play();
```

```
            //Aumenta el estrella 3 veces su tamaño
            estrella.Width = dAnchoEstrella * 3;
            estrella.Height = dAltoEstrella * 3;
    }
        else //Sonido para indicar que se dio clic a la estrella
            SystemSounds.Hand.Play();
      }
    }
```

El programa tiene también el método iniciar_Click que corresponde al evento que se ejecuta cuando se presiona el botón Iniciar. Además de reiniciar el contador de clics sobre la estrella, da inicio al evento del **DispatcherTimer**.

Para terminar, se tiene el método detener_Click que suspende la actividad de la estrella.

```
private void iniciar_Click(object sender, RoutedEventArgs e)
{
  timer.IsEnabled = true; //Inicia la actividad del DispatcherTimer
  iContador = 0; //Inicializa el contador de aciertos
  contador.Content = iContador.ToString(); //Actualizar etiqueta de aciertos
  bActividad = true; //Establece que hay actividad para que
                     //inicie la movilidad de la estrella
}
private void detener_Click(object sender, RoutedEventArgs e)
{
  timer.IsEnabled = false; //Detiene la ejecución del evento
  bActividad = false; //Indica que esta inactivo
}
```

Ejecución

El programa al ser ejecutado muestra una estrella que aparece y desaparece cada cierto tiempo. Cada vez que el usuario da clic sobre esta se incrementa la cantidad de aciertos. Se utilizan los botones Iniciar y Detener para inicializar o pausar la ejecución de la aplicación.

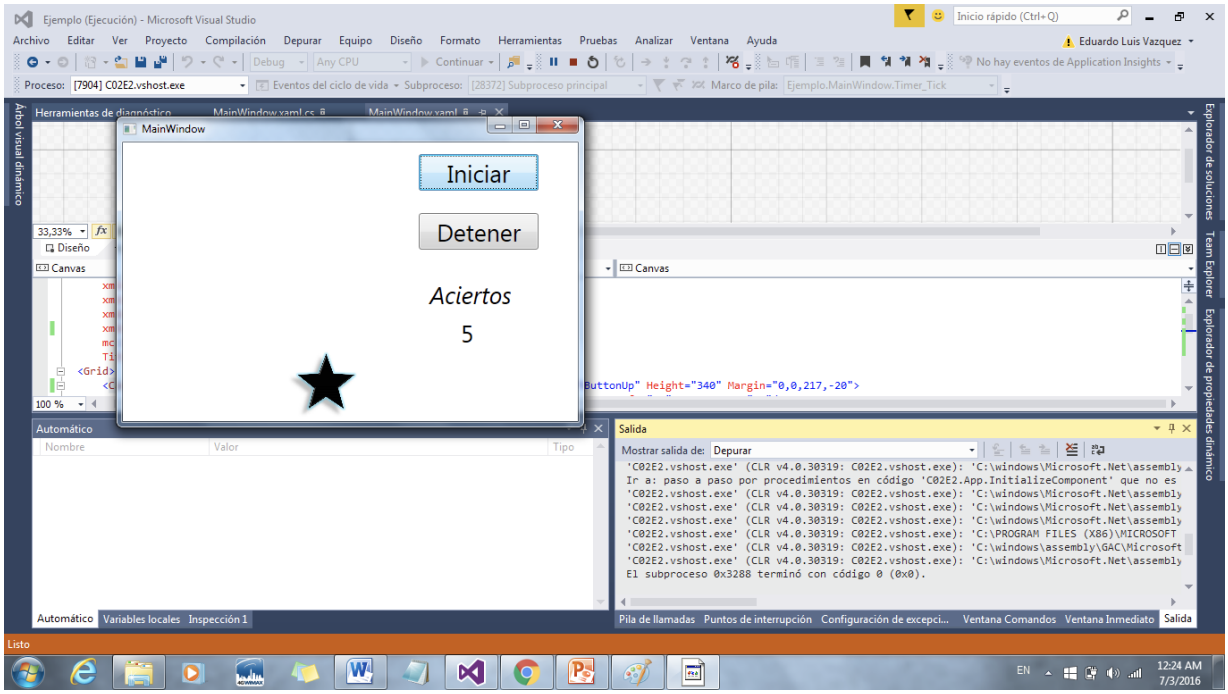


Figura 2.18

Capítulo 3. Manejo de colisiones

Los dispositivos que se emplean para mostrar el movimiento del cuerpo, por lo general arrojan las coordenadas (x, y) en las que se encuentran cada uno de los puntos del cuerpo que este puede detectar. Algunas de estas aplicaciones, acostumbran a relacionar la posición de uno o más puntos del cuerpo, con uno o más objetos mostrados en la pantalla; por ejemplo, cuando la aplicación permite al usuario mover un círculo por toda la pantalla siguiendo el movimiento de su mano derecha.

Cuando se mueven los objetos en la pantalla pueden ocurrir colisiones, es decir, un objeto puede ocupar, en su totalidad o en parte, el mismo espacio que otro objeto. Como se ha analizado anteriormente, todos los objetos (imágenes, figuras, etc.) son enmarcados por un rectángulo invisible del que se conoce la esquina superior izquierda; sin embargo, el contenido de dicho rectángulo puede tener otra forma por ejemplo, la imagen de una pelota ocupa un espacio rectangular aunque la pelota sea circular.

En este capítulo se estudiarán distintas formas para detectar cuándo hay una colisión entre dos objetos con el fin de escoger la que mejor quede para la aplicación que se desea desarrollar.

3.1 Paquete de información de un objeto (*struct*)

Para detectar si dos objetos están colisionando se necesita conocer la posición de cada uno de ellos dentro del **Canvas** y sus dimensiones. A fin de construir un método generalizado que pueda detectar si dos objetos están colisionando, es necesario que este reciba como entrada las características de ambos.

Una estructura (**struct**) es un tipo de dato que permite empaquetar valores manipulables como uno solo. Por ejemplo, la siguiente estructura de nombre Objeto, encapsula cuatro valores: la esquina superior izquierda de un objeto (dPosX, dPosY) y las dimensiones del mismo (dAncho, dAlto).

```
struct Objeto
{
  public double dPosX;
  public double dPosY;
  public double dAncho;
```

```
    public double dAlto;
}
```

Si se observa, el contenido de la estructura es la declaración de las variables que almacenarán los datos que se están empaquetando. Todas las variables son etiquetadas como **public** para que puedan ser alcanzadas desde cualquier lugar del programa.

Una vez declarada la estructura, es posible definir variables que sean de ese tipo de dato. El siguiente ejemplo muestra la declaración de la variable obA la cual es de tipo **Objeto**.

```
Objeto obA ;
```

Para introducir los datos a la variable de tipo estructura, se usa la notación punto. Antes del punto se define el nombre de la variable tipo estructura y después del punto el nombre de la variable que contiene la misma. Por ejemplo, las siguientes instrucciones muestran la forma de llenar la estructura de nombre obA de tipo **Objeto** con las características del objeto figura.

```
obA.dPosX = (double) figura.GetValue(Canvas.LeftProperty);
obA.dPosY = (double) figura.GetValue(Canvas.TopProperty);
obA.dAncho = figura.Width;
obA.dAlto = figura.Height;
```

Para utilizar el valor almacenado en la estructura se emplea la misma notación; por ejemplo, el siguiente código verifica si el objeto obA puede ser movido **iPixeles** hacia la izquierda.

```
if (obA.dPosX - iPixeles > 0)
{
    obA.dPosX -= iPixeles;
}
```

Cuando se envía una estructura a un método se está mandando un paquete de datos; esto es más práctico que hacerlo por separado. El siguiente ejemplo puede ser el encabezado de un método útil para detectar si dos objetos, ob1 y ob2, cuyas características están almacenadas dentro de sus estructuras están colisionando o no (por eso el resultado del método es bool, para regresar el valor **true** si están colisionando o **false** en caso contrario).

```
    private bool checarColision(Objeto ob1, Objeto ob2)
    {
        //Instrucciones para checar su hay una colisión entre el objeto ob1 y
        ob2
}
```

3.2 Ejemplo: detectar la colisión por el tamaño del objeto

El objetivo de este ejemplo es mostrar una manera elegante de comprobar si dos figuras rectangulares están colisionando. La aplicación muestra dos rectángulos; uno de ellos se ubica en el centro de la venta y permanece sin movimiento durante toda la ejecución; el otro es movido por el usuario empleando las teclas flecha; el objetivo de la aplicación es evitar que el usuario posicione el rectángulo encima del otro. En esta ocasión, se emplean rectángulos pero el código funciona con otro tipo de objetos, por ejemplo las imágenes y las elipses aunque para estas últimas no es la mejor opción como se verá más adelante.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Media**

Código en XAML

Para mostrar la forma de detectar si dos objetos están colisionando, en el archivo **MainWindow.xaml** se definió un **Canvas** con dos rectángulos. El panel tiene asociado un evento de tipo KeyDown para que el usuario pueda mover uno de los rectángulos empleando las teclas flecha. El rectángulo que está en el centro de la ventana objetoA (color azul) permanecerá estático mientras que el otro, objetoR (color rojo) será desplazado por el usuario.

```
<Grid>
    <Canvas Name="MainCanvas" Height="320" Width="500"
        KeyDown="MainCanvas_KeyDown">
        <Rectangle x:Name="objetoA" Fill="#FF0D0DF0"
            Width="70"          Height="70"          Canvas.Left="234"
```

```
            Canvas.Top="116" />
        <Rectangle x:Name="objetoR" Fill="#FFF0220D"
            Width="70"  Height="70"  Canvas.Left="84"
            Canvas.Top="145" />
    </Canvas>
</Grid>
```

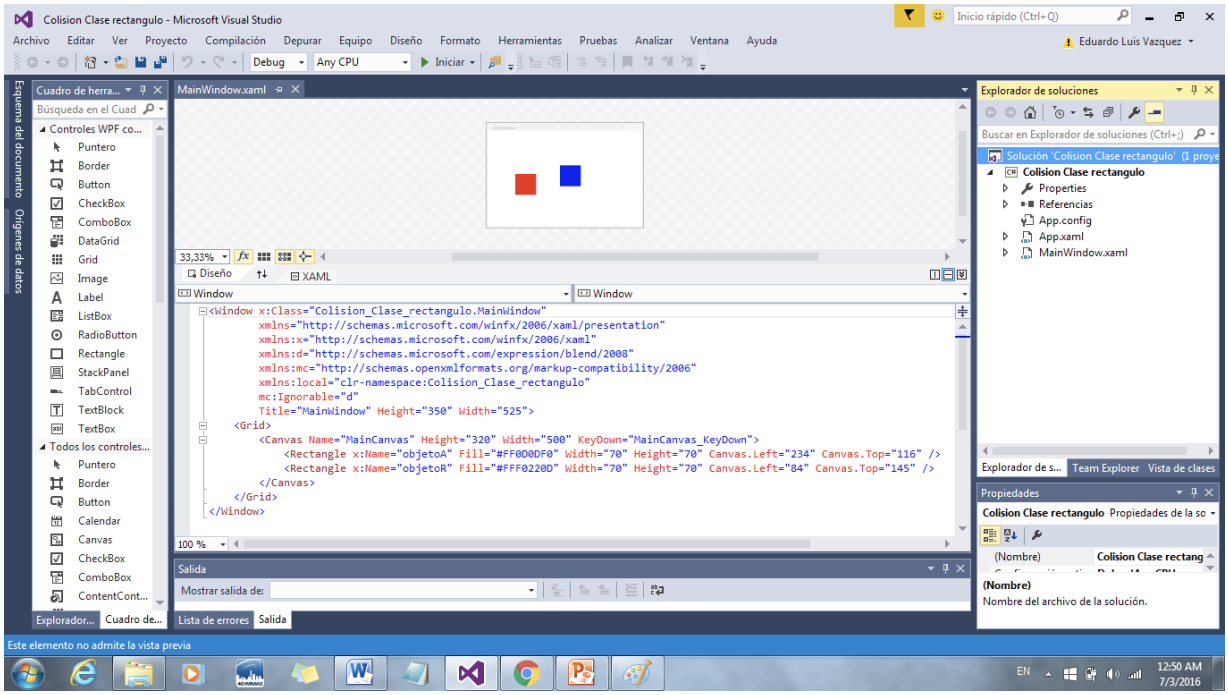


Figura 3.1

Código en C#

La única biblioteca agregada a **MainWindow.xaml.cs** es la que se emplea para emitir sonidos del sistema cuando hay una colisión.

```
using System.Media; //Sonido del Sistema
```

Al inicio del programa, entre otras cosas, se define la estructura que almacenará la información de los dos objetos, que en este caso son los rectángulos incluidos en el código XAML.

```
public partial class MainWindow : Window
{
```

```
    //Cantidad de pixeles que se mueve el elemento en cada ocasión
    int iPixeles = 3;
    //Variables que almacenarán las dimensiones del Canvas
    double iAnchoCanvas, iAltoCanvas;
    // Estructura para almacenar la información del objeto
    struct Objeto
    {
        public double dPosX;
        public double dPosY;
        public double dAncho;
        public double dAlto;
    }
    //Objetos que emplean la aplicación
  Objeto obR, obA;
```

El método **MainWindow** se encarga de inicializar las variables que contienen los datos que se requieren durante toda la ejecución del programa. Aquí es donde se llena la estructura con la información del rectángulo que permanece estático (objetoA) y se introducen los valores que no cambian del objeto en movimiento (objetoR).

```
  public MainWindow()
  {
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas
    iAnchoCanvas = MainCanvas.Width;
```

```
    iAltoCanvas = MainCanvas.Height;
    //Llenar la estructura para el objeto estático
    obA.dPosX = (double) objetoA.GetValue(Canvas.LeftProperty);
    obA.dPosY = (double) objetoA.GetValue(Canvas.TopProperty);
    obA.dAncho = objetoA.Width;
    obA.dAlto = objetoA.Height;
    //Llenar parte de la estructura para el objeto en movimiento
    obR.dAncho = objetoR.Width;
    obR.dAlto = objetoR.Height;
}
```

Dentro del método MainCanvas_KeyDown se realizan todas las acciones para desplazar hacia la derecha, izquierda, arriba o abajo al objetoR según sea la tecla flecha que presiona el usuario. Para esto, se determina su nueva ubicación y se termina el llenado de su estructura (obR).

Antes de cambiar el rectángulo de lugar, se verifica si hay o no una colisión; logra llamando al método checarColision al cual se le envían los dos objetos que pueden haber chocado (obR y obA). El método regresa el valor de verdadero si hay una colisión o el valor de falso en caso contrario. Si hay colisión emite un sonido, si no reubica el rectángulo.

```
private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
{
    //Coordenada de la esquina superior izquierda del objeto a mover
    obR.dPosX = (double)objetoR.GetValue(Canvas.LeftProperty);
    obR.dPosY = (double)objetoR.GetValue(Canvas.TopProperty);
    switch (e.Key)
    {
        //Determina nueva coordenada hacia la izquierda
        case Key.Left:
```

```
            if (obR.dPosX - iPixeles > 0)
                obR.dPosX -= iPixeles;
            break;
        //Determina nueva coordenada hacia la derecha
        case Key.Right:
            if (obR.dPosX + iPixeles + objetoR.Width < iAnchoCanvas)
                obR.dPosX += iPixeles;
            break;
        //Determina nueva coordenada hacia arriba
        case Key.Up:
            if (obR.dPosY - iPixeles > 0)
                obR.dPosY -= iPixeles;
            break;
        //Determina nueva coordenada hacia abajo
        case Key.Down:
            if (obR.dPosY + iPixeles + objetoR.Height < iAltoCanvas)
                obR.dPosY += iPixeles;
            break;
    }
    if (checarColision(obR, obA))
        SystemSounds.Hand.Play(); //Si hay colisión emite un sonido
    else
    { // Si no hay colisión mueve el objeto
        objetoR.SetValue(Canvas.LeftProperty, obR.dPosX);
        objetoR.SetValue(Canvas.TopProperty, obR.dPosY);
```

```
    }
}
```

Las siguientes imágenes muestran las formas en las que el objetoR al aproximarse al objetoA puede provocar una colisión. Hay que observar que el objetoR puede golpear al objetoA cuando su movimiento es hacia la derecha, o hacia arriba, o hacia la izquierda o hacia abajo.

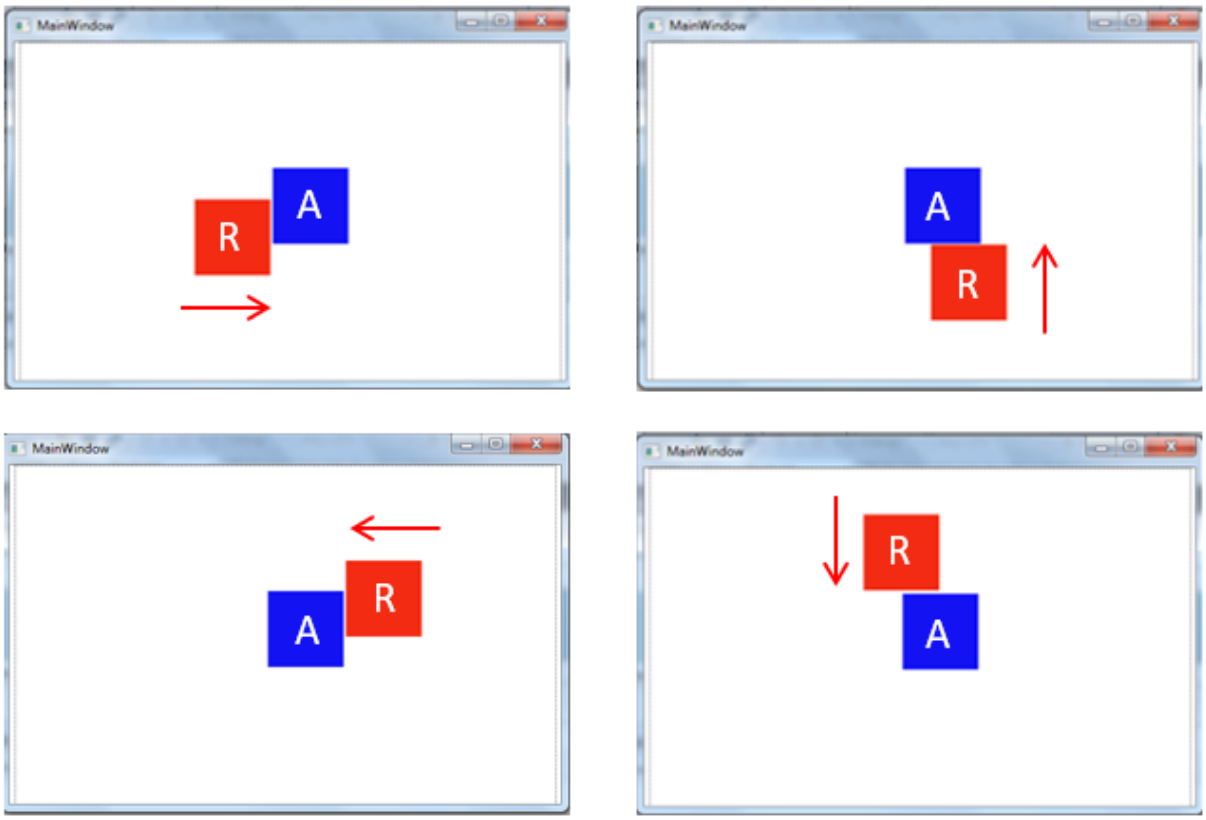


Figura 3.2

Ahora bien, el método checarColision recibe la información de dos objetos (ob1 y ob2) y se encarga de verificar si existe o no colisión entre ellos. Cuando detecta una colisión, regresa el valor verdadero (**true**), en caso contrario da como resultado el valor de falso (**false**).

La estrategia que utiliza es verificar todos los casos en los que se pueda asegurar que no hay colisión los cuales se indican a continuación:

	Si ob1 se está moviendo hacia la derecha, se debe detener su avance cuando su orilla derecha colisione con la izquierda de ob2. Para esto, si la suma de posición X de ob1 más su ancho es menor a la posición X de ob2, entonces no hay una colisión.

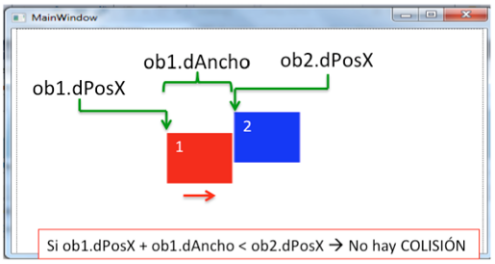
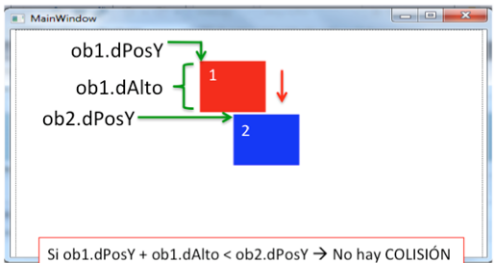
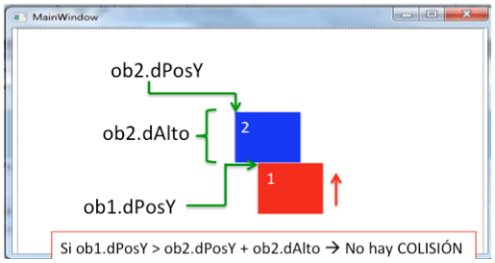
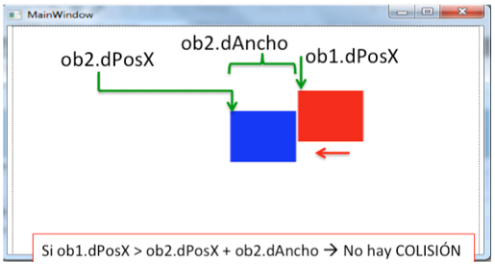
	
	Si ob1 se está moviendo hacia abajo, se debe impedir que su orilla inferior colisione con la superior de ob2. Para esto, si la suma de posición Y de ob1 más su altura es menor a la posición Y de ob2, entonces no hay una colisión.
	Si ob1 se está moviendo hacia la arriba, se debe evitar que su parte superior choque con la inferior de ob2. Para esto, si la posición Y de ob1 es mayor a la suma de la posición en Y de ob2 más su altura, entonces no hay una colisión.
	Si ob1 se está moviendo hacia la izquierda, se debe evitar su orilla izquierda choque con el lado derecho de ob2. Para esto, si la posición X de ob1 es mayor a la suma de la posición en X de ob2 más su ancho, entonces no hay una colisión.

Tabla 3.1

```
private bool checarColision(Objeto ob1, Objeto ob2)
{
  if (ob1.dPosX + ob1.dAncho < ob2.dPosX) //Colisión por la izquierda de ob2
      return false;
  if (ob1.dPosY + ob1.dAlto < ob2.dPosY) //Colisión por arriba de ob2
      return false;
```

```
    if (ob1.dPosY > ob2.dPosY + ob2.dAlto) //Colisión por abajo ob2
        return false;
    if (ob1.dPosX > ob2.dPosX + ob2.dAncho) //Colisión por la derecha ob2
        return false;
    return true;
}
```

Ejecución

Para comprobar el funcionamiento de la aplicación, se debe mover el rectángulo en todas direcciones en torno al objeto que está estático mientras se observa si se produce alguna colisión.

3.3 Ejemplo: detectar la colisión a través de la distancia

Si los objetos del ejemplo anterior hubiesen sido círculos en lugar de rectángulos se observaría que, en algunos puntos, la aplicación revelaría colisión aun si estos estuviesen separados. Esto sucede porque el esquema de colisiones supone que las figuras son rectangulares aun cuando no lo sean.

En esta aplicación se muestra otra manera de detectar la proximidad entre dos objetos. El programa muestra dos círculos: uno estático y el otro en movimiento dirigido por las teclas que presiona el usuario. Nuevamente, la aplicación evita que el usuario coloque un objeto encima del otro.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Media**.

Código en XAML

El archivo **MainWindow.xaml** define un **Canvas** con dos elipses. La elipse objetoA (color azul) permanecerá estática mientras que la elipse objetoR (color rojo) será desplazado por el usuario. El panel incluye el evento KeyDown para que el usuario pueda mover la elipse con las teclas flecha.

```
<Grid>
    <Canvas Name="MainCanvas" Height="320" Width="500"
            KeyDown="MainCanvas_KeyDown">
        <Ellipse x:Name="objetoA" Fill="#FF0D0DF0" Height="70"
        Width="70"
            Canvas.Left="230" Canvas.Top="132"/>
        <Ellipse x:Name="objetoR" Fill="#FFFD0606" Height="70"
        Width="70"
            Canvas.Left="74" Canvas.Top="132"/>
    </Canvas>
</Grid>
```

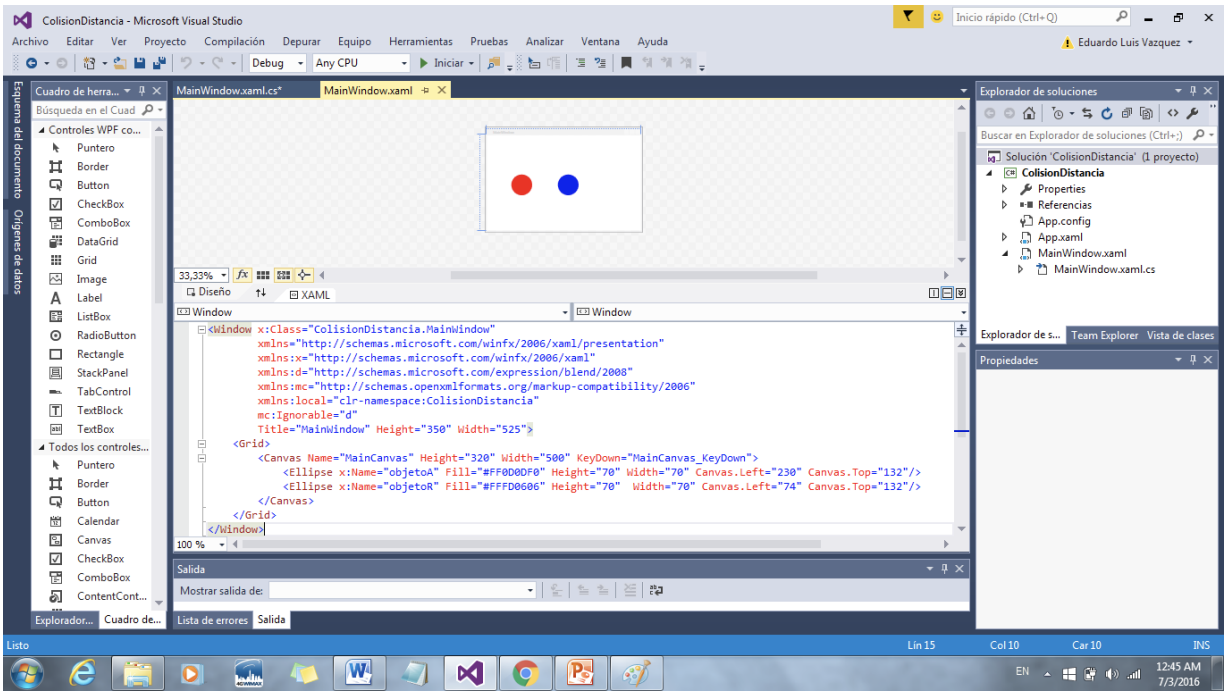


Figura 3.3

Código en C#

El programa **MainWindow.xaml.cs** es similar al del ejemplo anterior; el único método que se modifica es el que verifica si existe o no una colisión.

```
    using System.Media; //Sonido del Sistema
public partial class MainWindow : Window
{
    //Cantidad de pixeles que se mueve el elemento en cada ocasión
    int iPixeles = 3;
    //Variables que almacenarán las dimensiones del Canvas
    double iAnchoCanvas, iAltoCanvas;
    // Estructura para almacenar la información del objeto
    struct Objeto
    {
        public double dPosX;
        public double dPosY;
        public double dAncho;
        public double dAlto;
    }
    //Objetos que emplean la aplicación
    Objeto obR, obA;
public MainWindow()
{
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas
    iAnchoCanvas = MainCanvas.Width;
```

```
    iAltoCanvas = MainCanvas.Height;
    //Llenar la estructura para el objeto estático
    obA.dPosX = (double) objetoA.GetValue(Canvas.LeftProperty);
    obA.dPosY = (double) objetoA.GetValue(Canvas.TopProperty);
    obA.dAncho = objetoA.Width;
    obA.dAlto = objetoA.Height
    //Llenar parte de la estructura para el objeto en movimiento
    obR.dAncho = objetoR.Width;
    obR.dAlto = objetoR.Height;
}
private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
{
    //Coordenada de la esquina superior izquierda del objeto a mover
    obR.dPosX = (double)objetoR.GetValue(Canvas.LeftProperty);
    obR.dPosY = (double)objetoR.GetValue(Canvas.TopProperty);
    switch (e.Key)
    {
        //Determina nueva coordenada hacia la izquierda
        case Key.Left:
            if (obR.dPosX - iPixeles > 0)
              obR.dPosX -= iPixeles;
            break;
        //Determina nueva coordenada hacia la derecha
        case Key.Right:
            if (obR.dPosX + iPixeles + objetoR.Width < iAnchoCanvas)
```

```
                obR.dPosX += iPixeles;
            break;
        //Determina nueva coordenada hacia arriba
        case Key.Up:
            if (obR.dPosY - iPixeles > 0)
                obR.dPosY -= iPixeles;
            break;
        //Determina nueva coordenada hacia abajo
        case Key.Down:
            if (obR.dPosY + iPixeles + objetoR.Height < iAltoCanvas)
                obR.dPosY += iPixeles;
            break;
    }
    if (checarColision(obR, obA))
        SystemSounds.Hand.Play(); //Si hay colisión emite un sonido
    else
    { // Si no hay colisión mueve el objeto
        objetoR.SetValue(Canvas.LeftProperty, obR.dPosX);
        objetoR.SetValue(Canvas.TopProperty, obR.dPosY);
    }
}
```

El método checarColision recibe la información de dos objetos (ob1 y ob2) y se encarga de verificar si existe o no una colisión entre ellos. Cuando detecta una colisión, regresa el valor verdadero (**true**), en caso contrario da como resultado el valor de falso (**false**).

Para determinar si hay una colisión se emplea la distancia que existe entre el centro del primer círculo y el centro del segundo. Si dicha distancia es mayor a la suma del radio de los círculos, entonces no hay una colisión entre ellos.

Las siguientes imágenes muestran paso a paso el proceso para determinar si hay o no una colisión:

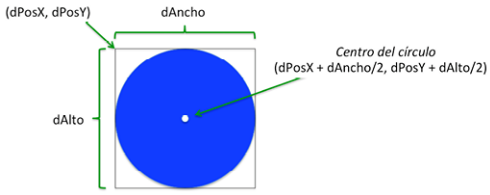
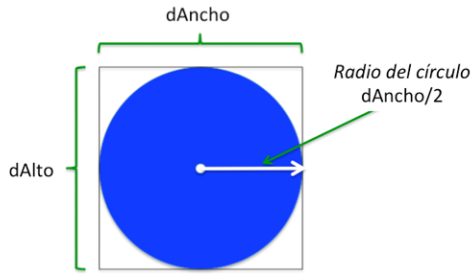
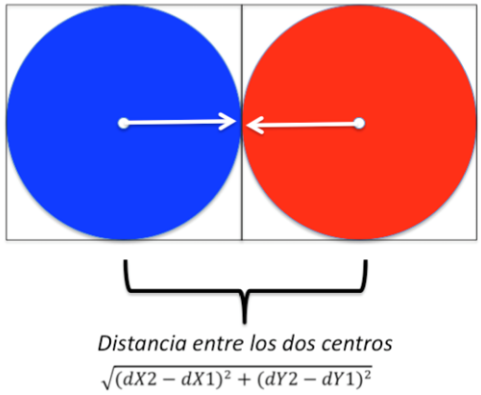
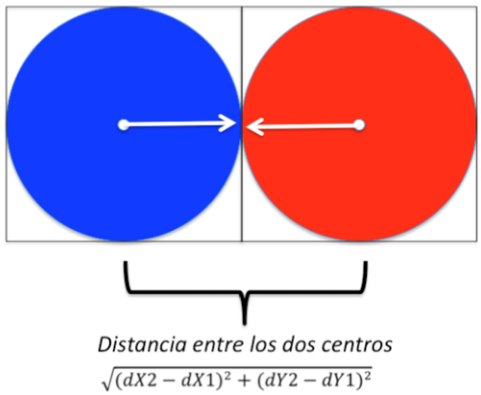
	Obtener el centro de cada uno de los círculos.
	Determinar el radio del círculo de cada círculo.
	Calcular la distancia entre el centro de los dos círculos.
	Verificar si hay o no una colisión. Si la distancia entre los dos centros es mayor a la suma de los radios entonces no hay colisión.

Tabla 3.2

private bool checarColision(Objeto ob1, Objeto ob2)

```
{
    //Coordenada del centro de cada objeto
    double dX1 = ob1.dPosX + (ob1.dAncho / 2);
    double dY1 = ob1.dPosY + (ob1.dAlto / 2);
    double dX2 = ob2.dPosX + (ob2.dAncho / 2);
    double dY2 = ob2.dPosY + (ob2.dAlto / 2);
    //Medida del radio de cada objeto
    double dRadio1 = ob1.dAncho / 2;
    double dRadio2 = ob2.dAncho / 2;
    //Distancia entre los dos objetos
    double dDistancia =
        Math.Sqrt(Math.Pow(dX2 - dX1, 2) + Math.Pow(dY2 - dY1, 2));
    //Verificar si hay colisión
    if (dDistancia > dRadio1 + dRadio2)
            return false;
    return true;
}
```

Ejecución

Para comprobar el funcionamiento de la aplicación se debe mover el círculo en todas direcciones alrededor del objeto que está estático para verificar si existe alguna colisión. Es interesante notar que ahora se puede detectar la colisión desde otros ángulos.

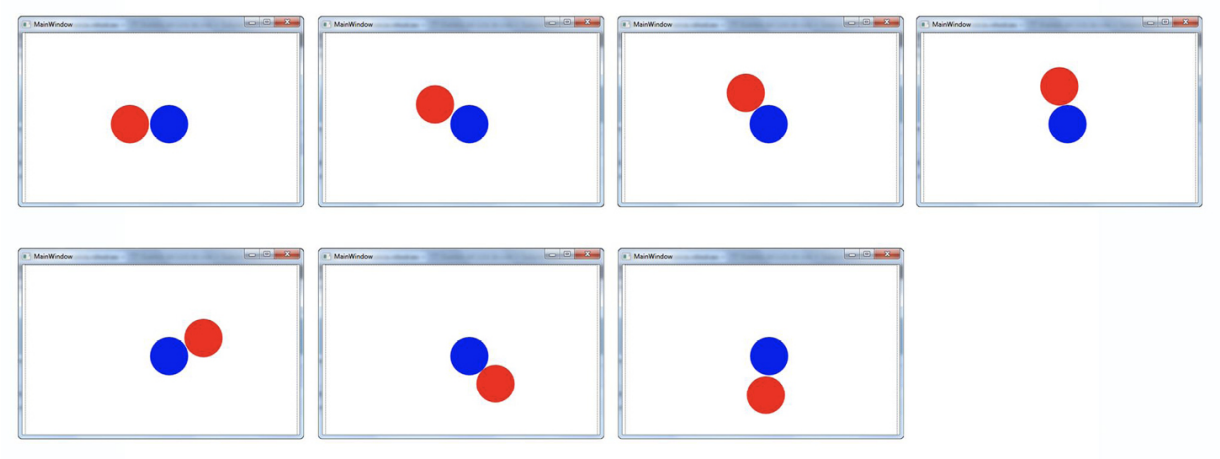


Figura 3.4

3.4 Ejemplo: detectar la colisión por secciones

Hay ocasiones en las que se desea detectar la colisión de una manera más precisa, es decir, descubrir si existe una cercanía mucho mayor a la que se puede tener si solamente se toma en cuenta el rectángulo invisible que rodea a los objetos. Para estos casos una posible solución es partir la figura, colocar un círculo de una manera imaginaria en cada una de las partes y, posteriormente, emplear la estrategia de la distancia entre dos puntos para determinar si hay o no colisión.

En esta aplicación se muestra el uso de esta estrategia al usar dos figuras, un círculo y un rectángulo que se dividen en varias partes.

Requerimientos

- Desarrollado en **WPF**
- Biblioteca: **System.Media**

Código en XAML

El archivo **MainWindow.xaml** define un **Canvas** con un rectángulo y una elipse. El rectángulo objetoA (color azul) permanecerá estático mientras que la elipse objetoR (color rojo) será desplazado por el usuario. El panel incluye el evento KeyDown para que el usuario pueda mover la elipse a través de las teclas flecha.

```
<Grid>
    <Canvas Name="MainCanvas" Height="320" Width="500"
```

```
        KeyDown="MainCanvas_KeyDown">
        <Rectangle x:Name="objetoA" Fill="#FF0D0DF0" Width="70"
        Height="70"
            Canvas.Left="234" Canvas.Top="116" />
        <Ellipse x:Name="objetoR" Fill="#FFF0220D" Height="70"
        Width="70"
            Canvas.Left="95" Canvas.Top="169" />
    </Canvas>
</Grid>
```

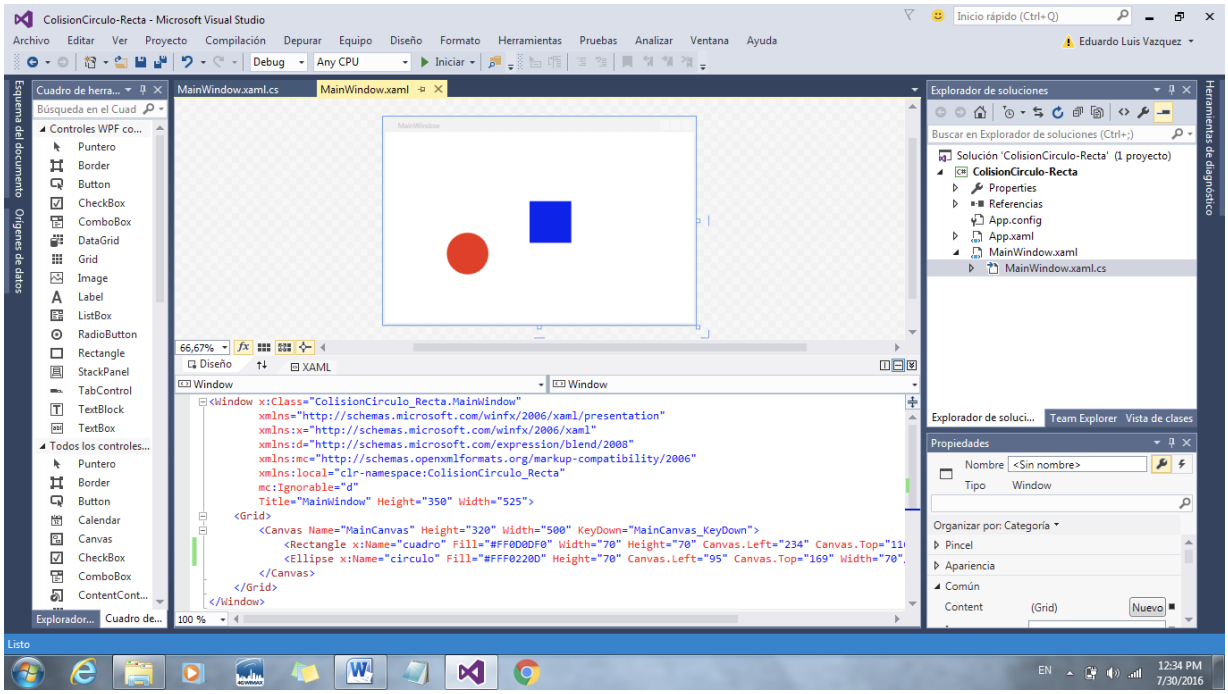


Figura 3.5

Código en C#

El programa **MainWindow.xaml.cs** es similar al del ejemplo anterior; lo único que se modifica es el método que verifica si existe o no una colisión.

```
using System.Media; //Sonido del Sistema

public partial class MainWindow : Window
```

```
{
    //Cantidad de pixeles que se mueve el elemento en cada ocasión
    int iPixeles = 3;
    //Variables que almacenarán las dimensiones del Canvas
    double iAnchoCanvas, iAltoCanvas;
    // Estructura para almacenar la información del objeto
    struct Objeto
    {
        public double dPosX;
        public double dPosY;
        public double dAncho;
        public double dAlto;
    }
    //Objetos que emplean la aplicación
    Objeto obCuadro, obCirculo;
public MainWindow()
 {
    InitializeComponent();
    //Enfocar el Canvas
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas
    iAnchoCanvas = MainCanvas.Width;
    iAltoCanvas = MainCanvas.Height;
    //Llenar la estructura para el objeto cuadro
```

```
    obCuadro.dPosX = (double)cuadro.GetValue(Canvas.LeftProperty);
    obCuadro.dPosY = (double)cuadro.GetValue(Canvas.TopProperty);
    obCuadro.dAncho = cuadro.Width;
    obCuadro.dAlto = cuadro.Height;
    //Llenar parte de la estructura para el objeto en movimiento
    obCirculo.dAncho = circulo.Width;
    obCirculo.dAlto = circulo.Height;}
private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
{
    //Coordenada de la esquina superior izquierda del objeto a mover
   círculo
    obCirculo.dPosX = (double)circulo.GetValue(Canvas.LeftProperty);
    obCirculo.dPosY = (double)circulo.GetValue(Canvas.TopProperty);
    switch (e.Key)
    {
    //Determina nueva coordenada hacia la izquierda
    case Key.Left:
        if (obCirculo.dPosX - iPixeles > 0)
            obCirculo.dPosX -= iPixeles;
            break;
    //Determina nueva coordenada hacia la derecha
    case Key.Right:
        if (obCirculo.dPosX + iPixeles + circulo.Width < iAnchoCanvas)
            obCirculo.dPosX += iPixeles;
            break;
    //Determina nueva coordenada hacia arriba
```

```
        case Key.Up:
            if (obCirculo.dPosY - iPixeles > 0)
                obCirculo.dPosY -= iPixeles;
                break;
        //Determina nueva coordenada hacia abajo
        case Key.Down:
            if (obCirculo.dPosY + iPixeles + circulo.Height < iAltoCanvas)
                obCirculo.dPosY += iPixeles;
                break;
        }
        //Si no hay colisión mover el objeto
            if (checarColision(obCirculo, obCuadro))
                SystemSounds.Hand.Play();
            else
        {
                circulo.SetValue(Canvas.LeftProperty, obCirculo.dPosX);
                circulo.SetValue(Canvas.TopProperty, obCirculo.dPosY);
        }
    }
```

Como en los ejemplos anteriores, el método checarColision recibe la información de dos objetos (ob1 y ob2) y se encarga de verificar si existe o no una colisión entre ellos. Cuando detecta una colisión, regresa el valor verdadero (**true**), en caso contrario da como resultado el valor de falso (**false**).

En este caso, para determinar si hay una colisión, el rectángulo se divide en cuatro secciones y se obtiene la distancia entre los centros de cada una de ellas con el centro del círculo. Es importante notar que la cantidad de partes en las que se divide el objeto y el lugar dentro de la figura en la que se encuentra cada una de ellas depende de la aplicación y del tipo de contenido del objeto.

Las siguientes imágenes muestran paso a paso el proceso para determinar si hay o no una colisión:

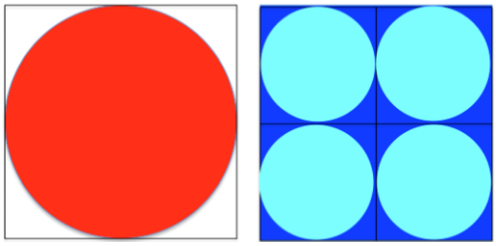
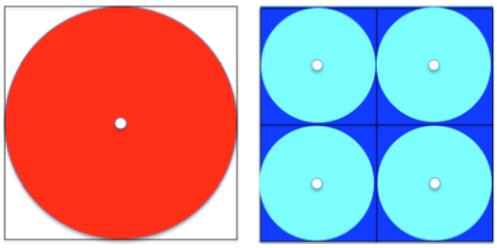
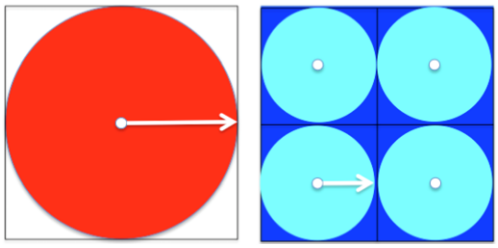
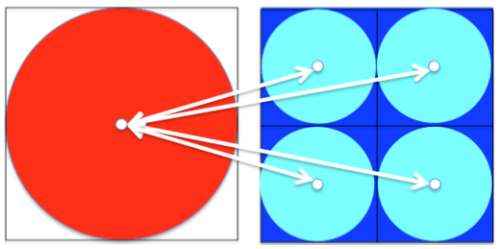
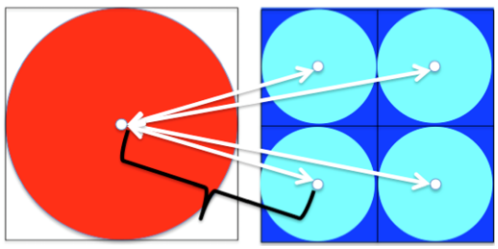
	Definir la cantidad de secciones en las que se dividirá cada uno de los objetos. En este caso, el cuadrado se dividió en cuatro círculos imaginarios y el objeto círculo se dejó tal y como está.
	Determinar la posición del centro de cada uno de los círculos.
	Calcular el radio de los círculos.
	Obtener la distancia entre el centro del objeto círculo con cada uno de los centros de los círculos imaginarios.
	Checar si hay colisión, la cual ocurre cuando la distancia es menor al valor resultante de sumar el radio del objeto círculo más el radio del círculo imaginario.

Tabla 3.3

```
private bool checarColision(Objeto obCirculo, Objeto obCuadro)
{
```

```
//Coordenada del centro del objeto círculo
double dX11 = obCirculo.dPosX + (obCirculo.dAncho / 2);
double dY11 = obCirculo.dPosY + (obCirculo.dAlto / 2);
//Coordenada del centro de cada círculo imaginario
double dX21 = obCuadro.dPosX + (obCuadro.dAncho / 4);
double dY21 = obCuadro.dPosY + (obCuadro.dAlto / 4);
double dX22 = obCuadro.dPosX + 3 * (obCuadro.dAncho / 4);
double dY22 = obCuadro.dPosY + (obCuadro.dAlto / 4);
double dX23 = obCuadro.dPosX + (obCuadro.dAncho / 4);
double dY23 = obCuadro.dPosY + 3 * (obCuadro.dAlto / 4);
double dX24 = obCuadro.dPosX + 3 * (obCuadro.dAncho / 4);
double dY24 = obCuadro.dPosY + 3 * (obCuadro.dAlto / 4);
//Medida del radio del objeto círculo y del círculo imaginario
double dRadioCirculo = obCirculo.dAncho / 2;
double dRadioCuadro = obCuadro.dAncho / 4;
//Distancia entre el centro del objeto círculo y el centro de cada
//círculo imaginario
double dDistancia11 =
    Math.Sqrt(Math.Pow(dX21 - dX11, 2) + Math.Pow(dY21 -
    dY11, 2));
double dDistancia12 =
    Math.Sqrt(Math.Pow(dX22 - dX11, 2) + Math.Pow(dY22 -
    dY11, 2));
double dDistancia13 =
    Math.Sqrt(Math.Pow(dX23 - dX11, 2) + Math.Pow(dY23 -
    dY11, 2));
```

```
        double dDistancia14 =
                Math.Sqrt(Math.Pow(dX24 - dX11, 2) + Math.Pow(dY24 -
                dY11, 2));
        //Verificar si hay colisión
        if (dDistancia11 < dRadioCirculo + dRadioCuadro ||
            dDistancia12 < dRadioCirculo + dRadioCuadro ||
            dDistancia13 < dRadioCirculo + dRadioCuadro ||
            dDistancia14 < dRadioCirculo + dRadioCuadro )
                return true;
        return false;
    }
```

Ejecución

Para comprobar el funcionamiento de la aplicación se debe mover el círculo en todas direcciones alrededor del objeto estático supervisando si se detecta una colisión en cualquiera de las direcciones. Ahora es posible detectar la colisión desde otros ángulos.

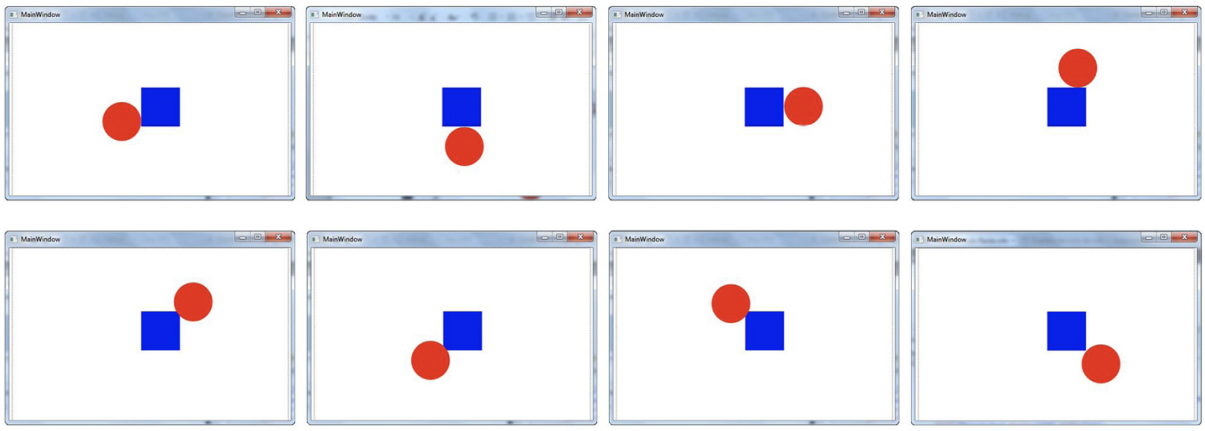


Figura 3.6

3.5 Ejemplo: uso de la clase *Rectangle* para detectar colisión

Otra forma de detectar la colisión cuando no se requiere de mucha precisión es emplear una clase predefinida de nombre *Rectangle* la cual posee un método que permite

determinar si hay o no colisión entre dos rectángulos.

Requerimientos

- Desarrollado en **WPF**.
- Referencia: **System.Drawing**
- Biblioteca: **System.Media**

Código en XAML

En el archivo **MainWindow.xaml** se definió un **Canvas** con dos rectángulos. El panel tiene asociado un evento de tipo KeyDown para que el usuario pueda mover uno de los rectángulos con las teclas flecha. El rectángulo que está en el centro de la ventana objetoA (color azul) permanecerá estático mientras que el otro, objetoR (color rojo), será desplazado por el usuario.

```
<Grid>
  <Canvas Name="MainCanvas" Height="320" Width="500"
      KeyDown="MainCanvas_KeyDown">
    <Rectangle x:Name="objetoA" Fill="#FF0D0DF0" Width="70"
    Height="70"
      Canvas.Left="234" Canvas.Top="116" />
    <Rectangle x:Name="objetoR" Fill="#FFF0220D" Width="70"
    Height="70"
      Canvas.Left="84" Canvas.Top="145" />
  </Canvas>
</Grid>
```

Añadir referencia

Para poder utilizar la clase *Rectangle* y los métodos que contiene, es necesario agregar una referencia al proyecto que se está construyendo. Se debe dar clic con el botón derecho del ratón en el nombre del proyecto dentro de la ventana del explorador de soluciones para ejecutar el comando **Referencia** del submenú **Agregar**.

Después de esto se debe seleccionar la referencia **System.Drawing** y presionar el botón aceptar.

Código en C#

El programa **MainWindow.xaml.cs**, además de incluir la biblioteca y las variables que se han estado inicializando anteriormente, debe incluir las variables que se emplearán para definir unos rectángulos imaginarios, alrededor de los objetos del proyecto, mismos que la clase **Rectangle** dispondrá para detectar la colisión.

```
using System.Media; //Sonido del Sistema
public partial class MainWindow : Window
{
  //Cantidad de pixeles que se mueve el elemento en cada ocasión
  int iPixeles = 3;
  //Variables que almacenarán las dimensiones del Canvas
  double iAnchoCanvas, iAltoCanvas;
  //Variables para almacenar las dimensiones del objeto en movimiento
  int iRAncho, iRAlto;
  //Variable que se empleará para dibujar los rectángulos imaginarios
  //para la verificación de la colisión
  System.Drawing.Rectangle Azul, Rojo;
```

Dentro del método **MainWindow**, además de enfocar y obtener la información del **Canvas**, se crea el rectángulo imaginario para el objeto estático (Azul), el cual será utilizado por otro método de la clase *Rectangle* para determinar si hay o no una colisión. Para construir dicho rectángulo, se requiere la posición dentro del **Canvas** de la esquina superior izquierda del objeto, así como su ancho y su alto. Todas estas cifras deben ser valores enteros por lo que se emplea int.Parse para realizar la conversión.

```
public MainWindow()
{
  InitializeComponent();
```

```
    //Establecer que el Canvas puede ser enfocado
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas
    iAnchoCanvas = MainCanvas.Width;
    iAltoCanvas = MainCanvas.Height;
    //Crear rectángulo imaginario para objeto estático (Azul)
    int iAPosX = int.Parse(objetoA.GetValue(Canvas.LeftProperty).ToString());
    int iAPosY = int.Parse(objetoA.GetValue(Canvas.TopProperty).ToString());
    int iAAncho = (int) objetoA.Width;
    int iAAlto = (int) objetoA.Height;
    Azul = new System.Drawing.Rectangle(iAPosX, iAPosY, iAAncho, iAAlto);
    //Obtener el ancho y alto del objeto en movimiento que después se emplearán
    //para crear el rectángulo imaginario del objeto en movimiento (Rojo)
    iRAncho = (int)objetoR.Width;
    iRAlto = (int)objetoR.Height;
}
```

El método MainCanvas_KeyDown, al igual que los ejemplos anteriores, obtiene la nueva posición del objeto en movimiento para después determinar si hay o no colisión. Si detecta que no hay colisión cambia el objeto de lugar, en caso contrario solamente genera un sonido.

Para detectar la colisión lo primero que hace es crear el rectángulo imaginario (Rojo) alrededor del objeto en movimiento utilizando la nueva posición que se acaba de calcular. Posteriormente emplea el método IntersectsWith de la clase *Rectangle* para determinar si hay colisión entre el rectángulo imaginario Azul y el rectángulo imaginario Rojo. Este

método regresa el valor **true** si hay colisión y **false** si no la hay.

```
private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
{
  //Coordenada de la esquina superior izquierda del objeto a mover
  int iRPosX = int.Parse(objetoR.GetValue(Canvas.LeftProperty).ToString());
  int iRPosY = int.Parse(objetoR.GetValue(Canvas.TopProperty).ToString());
  switch (e.Key)
  {
      //Determina nueva coordenada hacia la izquierda
      case Key.Left:
          if (iRPosX - iPixeles > 0)
            iRPosX -= iPixeles;
          break;
      //Determina nueva coordenada hacia la derecha
      case Key.Right:
          if (iRPosX + iPixeles + objetoR.Width < iAnchoCanvas)
            iRPosX += iPixeles;
          break;
      //Determina nueva coordenada hacia arriba
      case Key.Up:
          if (iRPosY - iPixeles > 0)
            iRPosY -= iPixeles;
          break;
      //Determina nueva coordenada hacia abajo
```

```
        case Key.Down:
            if (iRPosY + iPixeles + objetoR.Height < iAltoCanvas)
              iRPosY += iPixeles;
            break;
        }
        //Crear el rectángulo del objeto en movimiento
        Rojo = new System.Drawing.Rectangle(iRPosX, iRPosY, iRAncho,
        iRAlto);
        //Verificar si hay colisión entre los objetos
        if (Rojo.IntersectsWith(Azul))
            SystemSounds.Hand.Play();
        else
        {
            objetoR.SetValue(Canvas.LeftProperty, (double) iRPosX);
            objetoR.SetValue(Canvas.TopProperty, (double) iRPosY);
        }
    }
```

Ejecución

Para comprobar el funcionamiento de la aplicación se debe mover el rectángulo en todas direcciones alrededor del objeto fijo para ver si se detecta una colisión en cualquiera de las direcciones.

3.6 Ejemplo: encontrar frutas (laberinto)

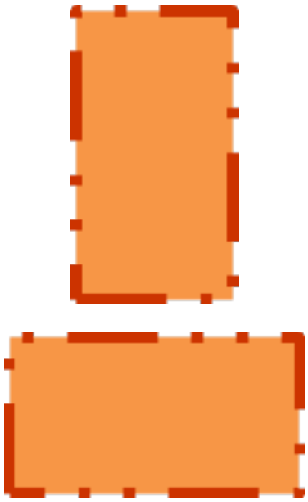
La aplicación que se describe a continuación muestra un laberinto en el que se han escondido frutas. El usuario moverá al personaje Malo a través de las teclas flecha por los caminos del laberinto para ir recolectando las frutas y con esto incrementar el contador.

Requerimientos

- Desarrollado en **WPF**.
- Biblioteca: **System.Media**
- Imágenes: fruta, dos paredes, dos dibujos de Malo

Recursos

Para formar las paredes del laberinto se utilizaron dos imágenes las cuales, además de emplearlas varias veces, fueron colocadas estratégicamente.



También se añadió la imagen de una fruta, la cual se coloca en varias partes del laberinto.

Finalmente, se colocaron dos imágenes para Malo; una para dar la impresión de que camina hacia la derecha y otra donde se le presenta caminando hacia la izquierda.

Código en XAML

En el archivo **MainWindow.xaml** se definió un panel tipo **Grid** que contiene un **Canvas**, la etiqueta aciertos para mostrar la cantidad de frutas recolectadas y la imagen de la fruta. Dentro del **Canvas** se colocó un rectángulo para marcar el área en la que el Malo puede caminar.

Dentro del rectángulo se colocaron diecinueve paredes, ocho frutas y la imagen del Malo. El panel **Canvas** tiene asociado un evento de tipo KeyDown para que el usuario pueda mover a Malo con las teclas flecha.

```
<Grid Background="#FF171515" >
  <Canvas Name="MainCanvas" Width="570" Height="400"
    KeyDown="MainCanvas_KeyDown" Margin="0,0,0,60">
   <Image x:Name="pared1" Height="70" Canvas.Left="64"
Canvas.Top="97"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
   <Image x:Name="pared2" Height="50" Canvas.Left="298"
Canvas.Top="47"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
   <Image x:Name="pared3" Height="70" Canvas.Left="64"
Canvas.Top="167"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
   <Image x:Name="pared4" Height="70" Canvas.Left="457"
Canvas.Top="147"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
   <Image x:Name="pared5" Height="70" Canvas.Left="160"
Canvas.Top="167"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
   <Image x:Name="pared6" Height="70" Canvas.Left="473"
Canvas.Top="287"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
```

```
 <Image x:Name="pared7" Height="70" Canvas.Left="160"
Canvas.Top="97"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
 <Image x:Name="pared8" Height="50" Canvas.Left="64"
Canvas.Top="237"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared9" Height="50" Canvas.Left="134"
Canvas.Top="237"
        Width="76" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared10" Height="50" Canvas.Left="228"
Canvas.Top="47"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared11" Height="50" Canvas.Left="318"
Canvas.Top="167"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared12" Height="50" Canvas.Left="160"
Canvas.Top="47"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared13" Height="70" Canvas.Left="268"
Canvas.Top="287"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
 <Image x:Name="pared14" Height="50" Canvas.Left="419"
Canvas.Top="307"
        Width="54" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared15" Height="70" Canvas.Left="369"
Canvas.Top="287"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
 <Image x:Name="pared16" Height="70" Canvas.Left="318"
```

```
Canvas.Top="97"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
 <Image x:Name="pared17" Height="70" Canvas.Left="369"
Canvas.Top="217"
        Width="50" Source="Imagenes/bloque1.png" Stretch="Fill"/>
 <Image x:Name="pared18" Height="50" Canvas.Left="387"
Canvas.Top="167"
        Width="70" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="pared19" Height="50" Canvas.Left="318"
Canvas.Top="307"
        Width="51" Source="Imagenes/bloque2.png" Stretch="Fill"/>
 <Image x:Name="fruta1" Height="34" Canvas.Left="39"
Canvas.Top="115"
        Width="20" Source="Imagenes/fruta.png"/>
 <Image x:Name="fruta2" Height="34" Canvas.Left="128"
Canvas.Top="188"
        Width="20" Source="Imagenes/fruta.png"/>
 <Image x:Name="fruta3" Height="34" Canvas.Left="228"
Canvas.Top="115"
        Width="20" Source="Imagenes/fruta.png"/>
 <Image x:Name="fruta4" Height="34" Canvas.Left="334"
Canvas.Top="268"
        Width="20" Source="Imagenes/fruta.png"/>
 <Image x:Name="fruta5" Height="34" Canvas.Left="437"
Canvas.Top="268"
        Width="20" Source="Imagenes/fruta.png"/>
 <Image x:Name="fruta6" Height="34" Canvas.Left="387"
Canvas.Top="115"
```

```
        Width="20" Source="Imagenes/fruta.png"/>
    <Image x:Name="fruta7" Height="34" Canvas.Left="128" Canvas.Top="287"
        Width="20" Source="Imagenes/fruta.png"/>
    <Image x:Name="fruta8" Height="34" Canvas.Left="528" Canvas.Top="307"
        Width="20" Source="Imagenes/fruta.png"/>
    <Image x:Name="malo" Height="38" Canvas.Left="21" Canvas.Top="19"
        Width="38" Source="Imagenes/maloD.png" Stretch="Fill"/>
    <Rectangle Height="400" Canvas.Left="2" Stroke="#FFF91E1E" Width="565"
        OpacityMask="#FFEC2929" Canvas.Top="2"/>
    </Canvas>
    <Label x:Name="aciertos" Content="0" Height="43" Width="88" FontSize="24"
        FontWeight="Bold" Foreground="#FFEADEDE" Margin="492,406,0,10"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
    <Image x:Name="fruta" Height="34" Width="20" Canvas.Left="237"
        Canvas.Top="253" Source="Imagenes/fruta.png" Margin="439,404,93,5"/>
</Grid>
```

Código en C#

El del programa **MainWindow.xaml.cs** se incluye la biblioteca que se emplea para generar efectos de sonido. Antes del método **MainWindow**, se deben definir todas las variables y estructuras para el resto de la aplicación. Aquí se crea el **BitmapImage** para cada una de las imágenes de Malo y se crea la estructura que almacenará las piezas de

información de los objetos que se mostrarán en la pantalla.

Dado que el comportamiento es diferente cuando Malo choca con una pared a cuando colisiona con una fruta, se añadió a la estructura una variable que indica el tipo de objeto del que se trata.

Posteriormente, se emplea la definición de la estructura para crear el repositorio de Malo y un arreglo de repositorios para cada uno de los objetos que están en el **Canvas**, es decir, un espacio para cada pared y/o fruta contra la que puede colisionar Malo. Finalmente se cuenta con una variable para almacenar la cantidad de objetos en el **Canvas** y la cantidad de aciertos (fruta recolectada).

```
using System.Media; //Sonido del Sistema
public partial class MainWindow : Window
{
  //Cantidad de pixeles que se mueve el malo en cada ocasión
  int iPixeles = 5;
  //Variables que almacenarán las dimensiones del Canvas
  double dAnchoCanvas, dAltoCanvas;
  //Imágenes del malo
  BitmapImage bmMaloI = new BitmapImage(new
Uri(@"Imagenes/maloI.png",
                    UriKind.RelativeOrAbsolute));
  BitmapImage bmMaloD = new BitmapImage(new
Uri(@"Imagenes/maloD.png",
                    UriKind.RelativeOrAbsolute));
  // Estructura que almacenará la información del objeto
  struct Objeto
  {
     public int iTipo; // 1 = pared , 2 = fruta
     public Image imImagen;
```

```
        public double dPosX;
        public double dPosY;
        public double dAncho;
        public double dAlto;
    }
    //Objetos que emplea la aplicación
    Objeto obMalo;
    Objeto [ ] obLista = new Objeto[28];
    int iCantidad; //Cantidad de objetos que están dentro del Canvas
    int iContAciertos = 0; //Cantidad de frutas recolectadas
```

El método **MainWindow**, además de otras acciones, da valor inicial a algunos de los campos de la estructura de Malo. También solicita la ejecución del método GeneraListaObjetos, el cual se encarga de llenar el arreglo de estructuras para cada uno de los objetos que están colocados dentro del **Canvas**. Para terminar, despliega en la ventana el valor inicial del contador de frutas recolectadas.

```
public MainWindow()
{
    InitializeComponent();
    //Establecer que el Canvas puede ser enfocado
    MainCanvas.Focusable = true;
    MainCanvas.Focus();
    //Obtener el ancho y la altura del Canvas
    dAnchoCanvas = MainCanvas.Width;
    dAltoCanvas = MainCanvas.Height;
    //Llenar la estructura para el malo
    obMalo.dAncho = malo.Width;
    obMalo.dAlto = malo.Height;
```

```
        //Genera la lista de objetos pared y fruta
        GeneraListaObjetos();
        //Despliega la cantidad de aciertos inicial
        aciertos.Content = iContAciertos.ToString();
    }
```

El método GeneraListaObjetos se encarga de llenar el arreglo obLista con los datos de cada pared, fruta y el personaje Malo, que están en el Canvas. Usa el método FindVisualChildren que fue elaborado por Bryce Kahle el cual crea una lista con todos los objetos del Canvas. Cada elemento de la lista es asignado a la variable imagen para obtener su información y llenar la estructura.

```
    private void GeneraListaObjetos()
    {
        String nombre;
        iCantidad = 0;
        foreach (Image imagen in FindVisualChildren<Image>(MainCanvas))
        {
            //Obtener la primera letra del nombre
            nombre = imagen.Name;
            nombre = nombre.Substring(0,1);
            //Determina el tipo de imagen
            if ( nombre == "f" ) //Es una fruta
            obLista[iCantidad].iTipo = 2;
            else if (nombre == "p") //Es una pared
            obLista[iCantidad].iTipo = 1;
            else obLista[iCantidad].iTipo = 3; //Cualquier otro tipo
            obLista[iCantidad].imImagen = imagen;
            obLista[iCantidad].dPosX                                    =
```

```
            (double)imagen.GetValue(Canvas.LeftProperty);
            obLista[iCantidad].dPosY                                    =
            (double)imagen.GetValue(Canvas.TopProperty);
            obLista[iCantidad].dAlto = imagen.Height;
            obLista[iCantidad].dAncho = imagen.Width;
            iCantidad++;
        }
    }
    /// <summary>
    /// Función que obtiene los controles que están dentro del objeto
especificado
    /// StackOverflow - Bryce Kahle
    ///    http://stackoverflow.com/questions/974598/find-all-controls-in-wpf-
window-by-type
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="depObj"></param>
    /// <returns></returns>
   public static IEnumerable<T> FindVisualChildren<T>
                (DependencyObject depObj) where T : DependencyObject
   {
    if (depObj != null)
    {
      for (int iK = 0; iK < VisualTreeHelper.GetChildrenCount(depObj);
     iK++)
      {
```

```
            DependencyObject child = VisualTreeHelper.GetChild(depObj,
            iK);
            if (child != null && child is T)
            {
                yield return (T)child;
            }
            foreach (T childOfChild in FindVisualChildren<T>(child))
            {
                yield return childOfChild;
            }
        }
    }
```

El método MainCanvas_KeyDown, al igual que en los ejemplos anteriores, calcula la nueva posición a la que se debe mover Malo y antes de realizar el movimiento verifica si hay o no colisión ya sea con una pared o con una fruta. Para verificar si hay una colisión se llama al método Colisiones.

```
    private void MainCanvas_KeyDown(object sender, KeyEventArgs e)
    {
        //Coordenada de la esquina superior izquierda del malo
        obMalo.dPosX = (double)malo.GetValue(Canvas.LeftProperty);
        obMalo.dPosY = (double)malo.GetValue(Canvas.TopProperty);
        switch (e.Key)
        {
            //Determina nueva coordenada hacia la izquierda
            case Key.Left:
                if (obMalo.dPosX - iPixeles > 0)
                    obMalo.dPosX -= iPixeles;
```

```
        malo.Source = bmMaloI;
      break;
//Determina nueva coordenada hacia la derecha
case Key.Right:
      if (obMalo.dPosX + iPixeles + malo.Width < dAnchoCanvas)
        obMalo.dPosX += iPixeles;
        malo.Source = bmMaloD;
break;
//Determina nueva coordenada hacia arriba
case Key.Up:
      if (obMalo.dPosY - iPixeles > 0)
      obMalo.dPosY -= iPixeles;
break;
//Determina nueva coordenada hacia abajo
case Key.Down:
      if (obMalo.dPosY + iPixeles + malo.Height < dAltoCanvas)
      obMalo.dPosY += iPixeles;
break;
}
//Si no hay colisión mover el objeto
if (Colisiones())
{
      SystemSounds.Asterisk.Play();
}
else
```

```
        {
            //Mueve al malo a la nueva coordenada
            malo.SetValue(Canvas.LeftProperty, obMalo.dPosX);
            malo.SetValue(Canvas.TopProperty, obMalo.dPosY);
        }
    }
```

La estrategia que se utilizó para el manejo de colisiones podrá no ser la más eficiente pero sí la más sencilla. Cada vez que Malo cambia de lugar, se verifica si colisionó o no con cada una de las paredes y frutas que están en el **Canvas**. El método Colisiones se encarga de seleccionar cada uno de los objetos con los cuales se debe verificar la colisión y utiliza el método checarColisión (cuyo contenido fue explicado anteriormente). En particular cuando Malo colisiona con una fruta, además de indicar lo anterior, colapsa la imagen para que desaparezca y cambia el tipo de objeto dentro de la estructura de tal manera que la próxima vez ya no se tome en cuenta para el chequeo de colisiones. La aplicación lleva un contador de frutas recolectadas por lo que al haber una colisión entre el malo y una fruta, este se incrementa y despliega en la ventana.

```
    private bool Colisiones()
    {
        for (int iK = 0; iK < iCantidad; iK++)
        {
            //Checar colisión de todos los objetos que sean frutas y paredes
            if (obLista[iK].iTipo != 3)
            {
                if (checarColision(obMalo, obLista[iK]))
                {
                    if (obLista[iK].iTipo == 2) //Chocó con una fruta
                    {
                        //Colapsa la fruta para que desaparezca de la vista
                        obLista[iK].imImagen.Visibility = Visibility.Collapsed;
```

```
                    //Cambia el tipo de la imagen
                    //para que no se tome en cuenta nuevamente
                    obLista[iK].iTipo = 3;
                    //Incrementa y despliega la cantidad de frutas comidas
                    iContAciertos++;
                    aciertos.Content = iContAciertos.ToString();
                }
                return true;
            }
        }
    }
    return false;
}
private bool checarColision(Objeto ob1, Objeto ob2)
{
    if (ob1.dPosX + ob1.dAncho < ob2.dPosX) //Colisión por izquierda de
 ob2
    return false;
    if (ob1.dPosY + ob1.dAlto < ob2.dPosY) //Colisión por arriba de ob2
    return false;
    if (ob1.dPosX > ob2.dPosX + ob2.dAncho) //Colisión por la derecha
 ob2
    return false;
    if (ob1.dPosY > ob2.dPosY + ob2.dAlto) //Colisión por abajo ob2
    return false;
    return true;
```

}

Ejecución

Las siguientes imágenes muestran distintos momentos de la ejecución de la aplicación.

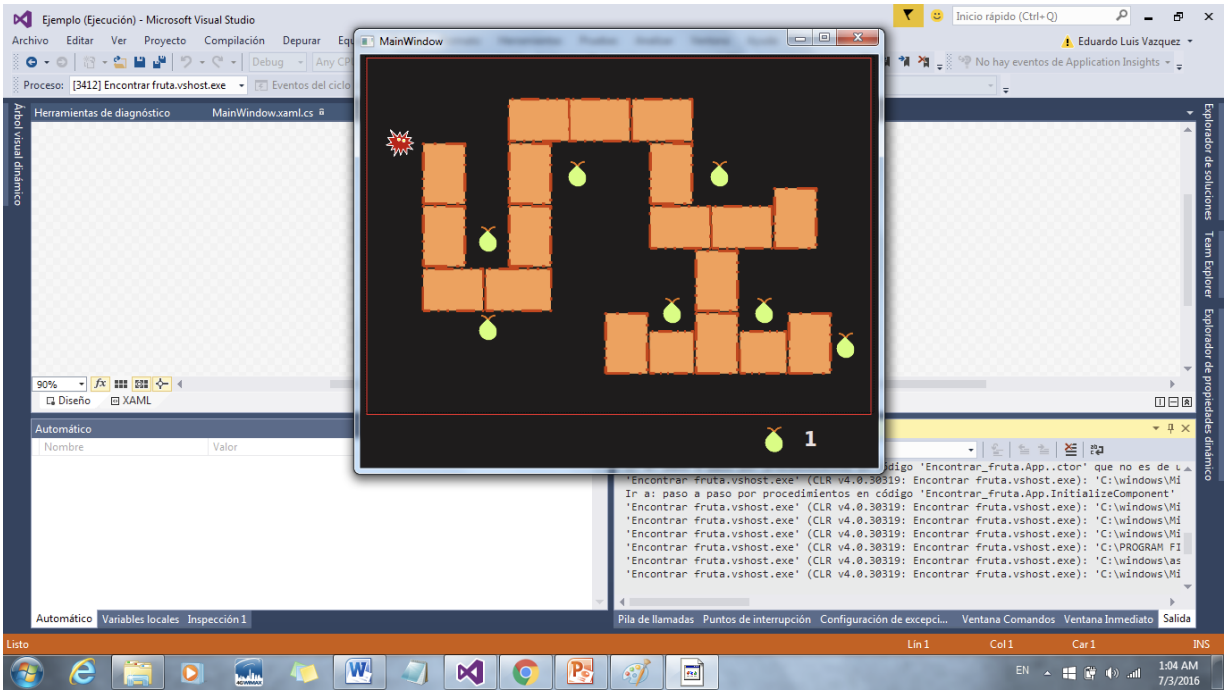


Figura 3.7

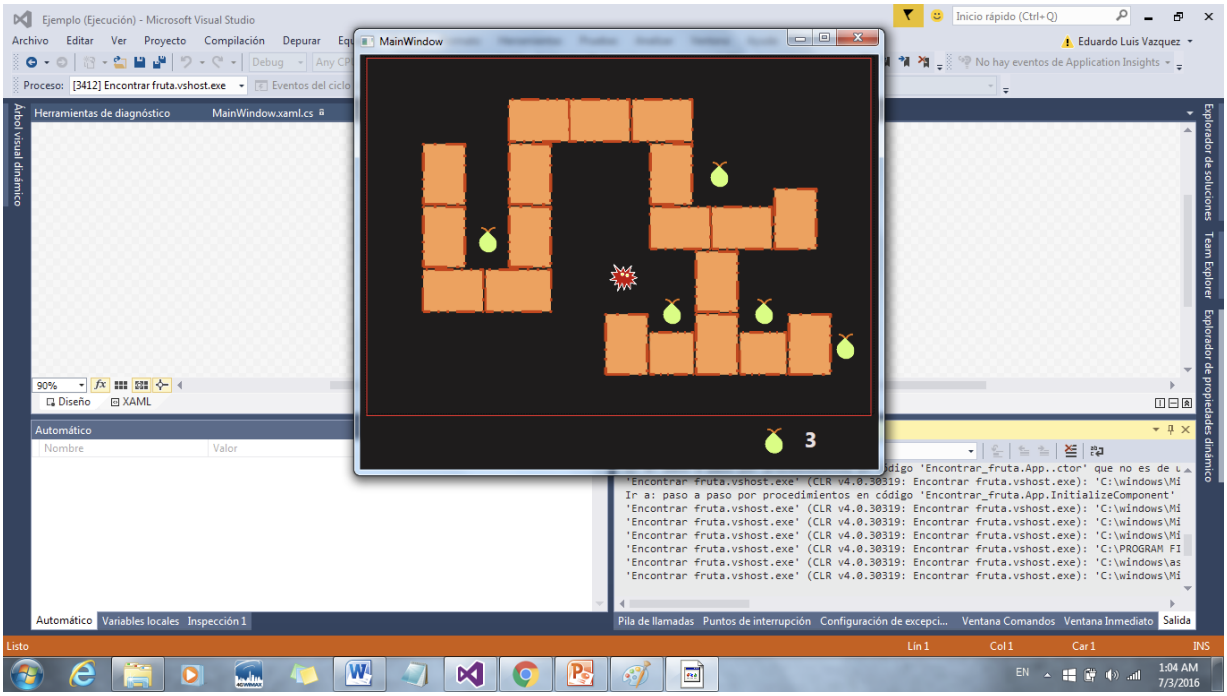


Figura 3.8

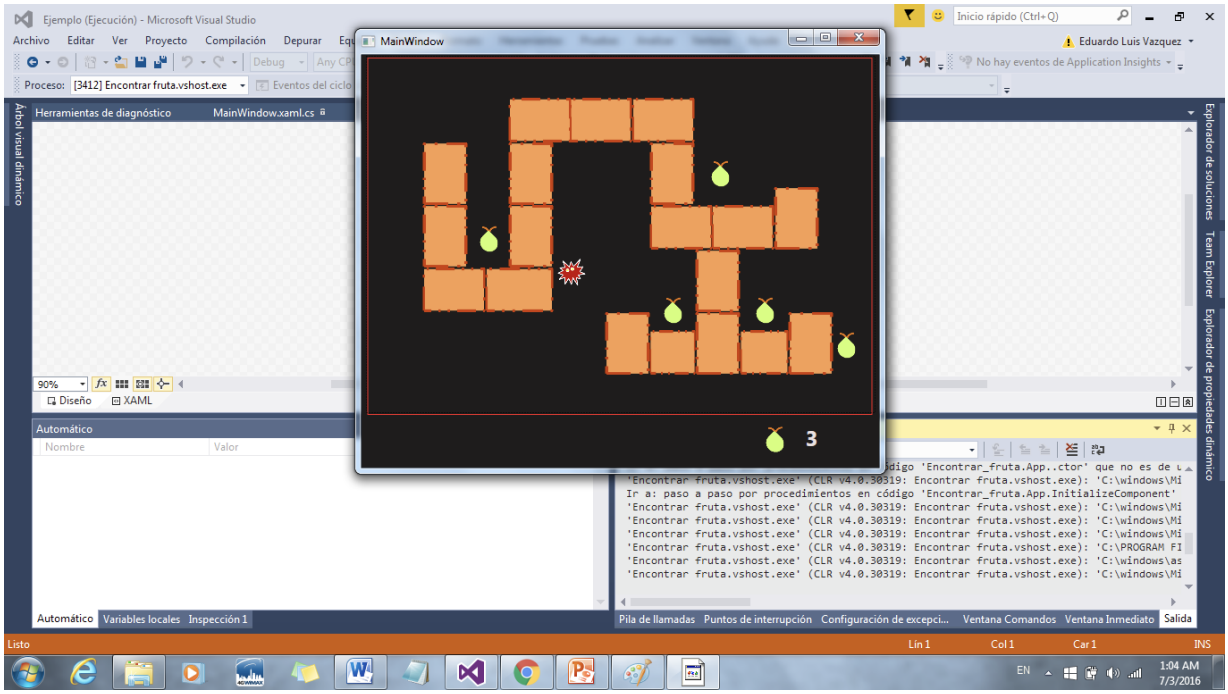


Figura 3.9

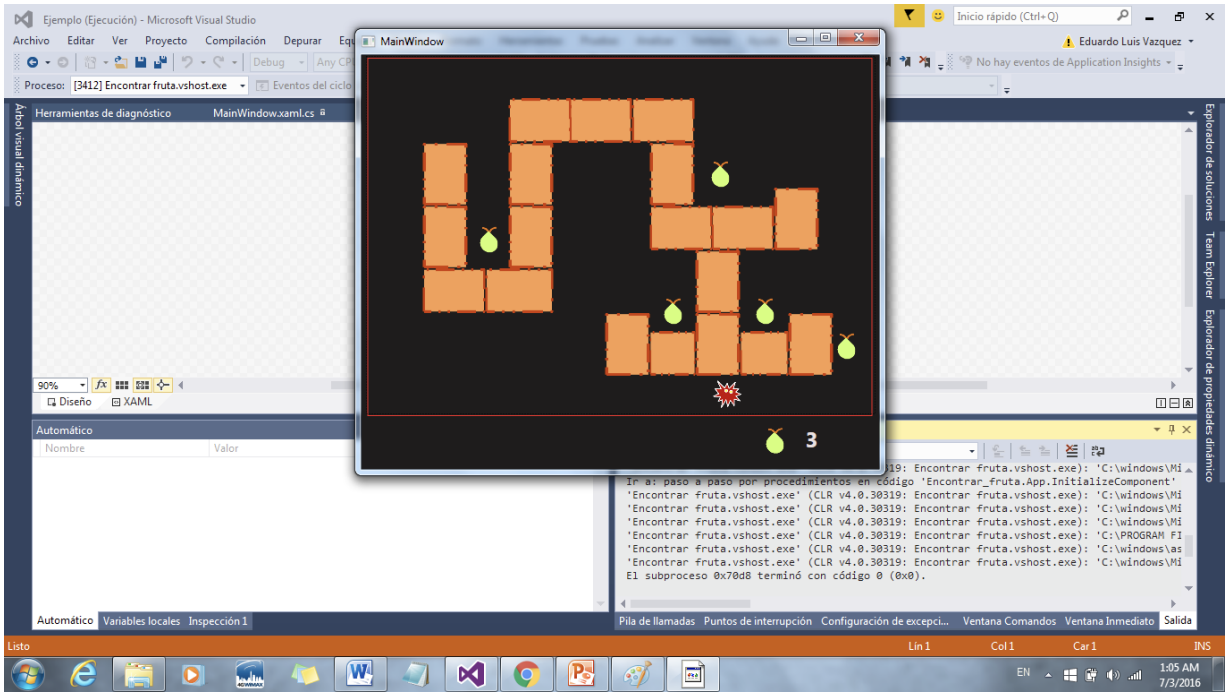


Figura 3.10

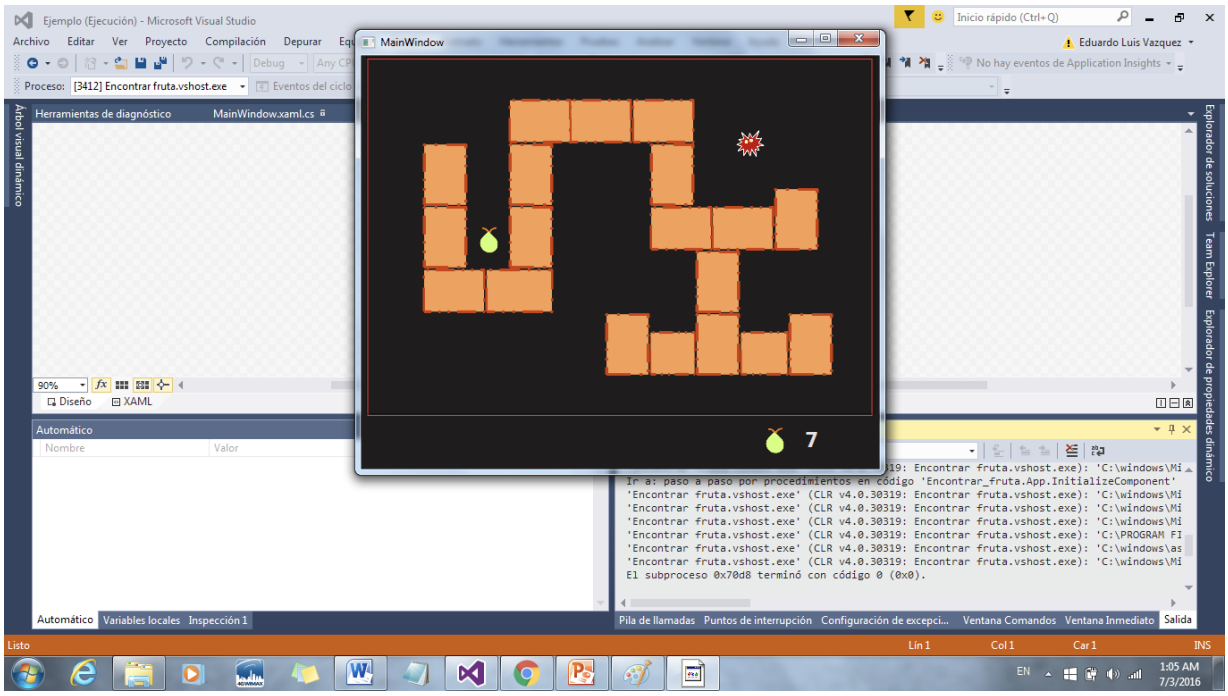


Figura 3.11

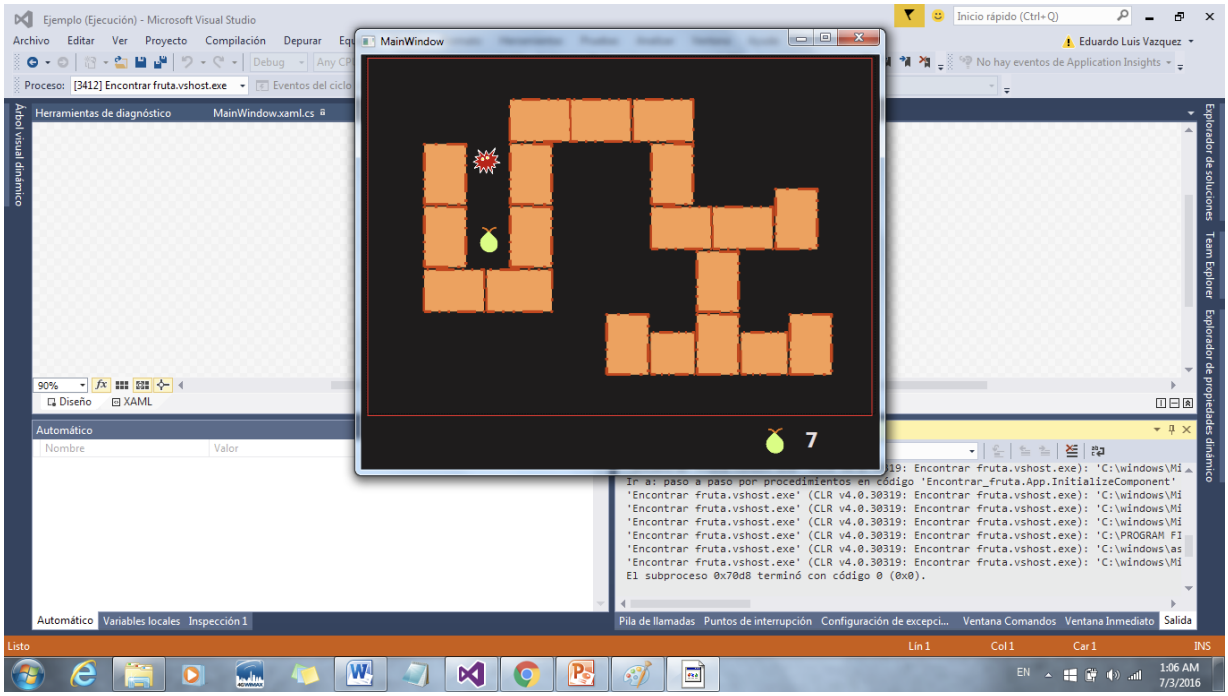


Figura 3.12

Capítulo 4. Audio

Los dispositivos que muestran el movimiento del cuerpo por lo general arrojan las coordenadas (x, y) en las que se encuentran cada uno de los puntos del cuerpo que este puede detectar. Algunas de estas aplicaciones, acostumbran a relacionar la posición de uno o más puntos del cuerpo con objetos mostrados en la pantalla; por ejemplo, cuando permite al usuario mover un círculo por toda la pantalla siguiendo el movimiento de su mano derecha.

4.1 Paquete de información de un objeto (*struct*)

Requerimientos

Las aplicaciones relacionadas con el movimiento del cuerpo comúnmente incluyen audio para lograr efectos, por ejemplo, la manifestación del movimiento o el sonido del rebote de una pelota. También utilizan sonidos como ambientación con el fin de animar al usuario a realizar alguna actividad.

4.1.1 Formas para generar sonido

El lenguaje C# proporciona diversas herramientas para la reproducción de audio y generación de sonido, algunas de ellas son:

- **SoundPlayer**
- **MediaElement**
- **Beep**

Su uso dependerá de lo que se quiera lograr en la aplicación.

4.2 Reproducir un archivo .wav

La clase **SoundPlayer** se emplea para reproducir audio digital en formato **.wav**. Este formato no es muy utilizado ya que, por lo general, son archivos de gran tamaño dado que no están comprimidos.

Para emplear la clase **SoundPlayer** se debe añadir al proyecto la biblioteca **Media**.

```
using System.Media;
```

Esto permite declaración de variables de tipo **SoundPlayer**.

```
SoundPlayer Nota;
```

A una variable de este tipo se le asigna un audio. La siguiente instrucción inicializa la variable Nota con el audio almacenado en el archivo NotaA.wav el cual debió ser agregado a los recursos del proyecto.

```
Nota = new SoundPlayer(Properties.Resources.NoteA);
```

Para reproducir el audio asignado a una variable de tipo **SoundPlayer** se emplea el método **Play**.

```
Nota.Play();
```

4.3 Ejemplo: secuencia musical (archivos .wav)

La aplicación se sirve de **SoundPlayer** para reproducir un conjunto de cinco sonidos en secuencia. Cuando se ejecuta, muestra en la ventana un botón que, al ser presionado, reproduce el siguiente sonido de la secuencia.

Requerimientos

- Desarrollado en **WPF**.
- Recursos: cinco notas musicales .wav que se denominarán: NotaA.wav, NotaB.wav, NotaC.wav, NotaD.wav, NotaE.wav
- Biblioteca: **System.Media**

Código en XAML

El archivo **MainWindow.xaml** del proyecto incluye un botón que contiene la palabra Play y que al darle clic dispara el evento play_Click:

```
<Grid>
  <Button x:Name="play" Content="Play" HorizontalAlignment="Left"
    Margin="44,108,0,0" VerticalAlignment="Top" Width="92"
    FontSize="29.333" Height="51" Click="play_Click"/>
```

```
</Grid>
```

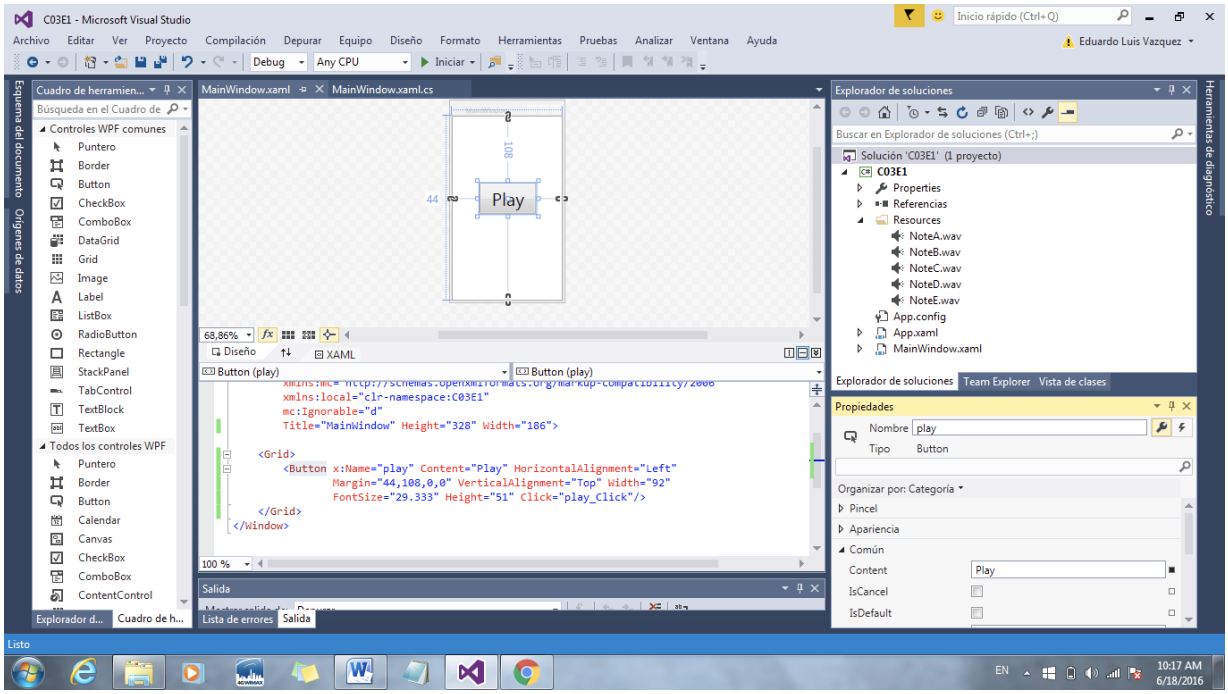


Figura 4.1

Código en C#

El primer cambio es agregar en el archivo **MainWindow.xaml.cs** la biblioteca **System.Media**:

```
using System.Media;
```

La variable **iContador** determina la nota que se debe reproducir (si la variable contiene el número 1 se reproduce la NotaA, si tiene el valor 2 se reproduce la NotaB, y así sucesivamente). Por otro lado, se inicializan las variables **Nota1**, **Nota2**, **Nota3**, **Nota4** y **Nota5** con los tonos musicales a reproducir. Para permitir que las variables puedan ser utilizadas en cualquiera de los métodos del proyecto, estas se declaran dentro de la clase antes del método **MainWindow**.

```
public partial class MainWindow : Window
{
    //Variable cuyo contenido indica la nota a reproducir
    int iContador = 0;
    //Asignar a cada Nota el sonido que le corresponde
```

```
        SoundPlayer Nota1 = new SoundPlayer(Properties.Resources.NoteA);
        SoundPlayer Nota2 = new SoundPlayer(Properties.Resources.NoteB);
        SoundPlayer Nota3 = new SoundPlayer(Properties.Resources.NoteC);
        SoundPlayer Nota4 = new SoundPlayer(Properties.Resources.NoteD);
        SoundPlayer Nota5 = new SoundPlayer(Properties.Resources.NoteE);
    public MainWindow()
```

Finalmente, se agrega al evento **play_Click** las instrucciones para realizar la secuencia de sonidos. La variable **iContador** se incrementa en uno; cuando alcanza el valor 6 se le asigna nuevamente el número 1 de tal manera que únicamente tome los valores del 1 al 5. Según el valor de **iContador** será el audio que se reproduzca.

```
    private void play_Click(object sender, RoutedEventArgs e)
    {
      //Definir la siguiente nota a reproducir
      iContador++;
      if (iContador == 6) iContador = 1;
      //Reproducir la nota musical que corresponde al valor del contador
      switch (iContador)
      {
          case 1: Nota1.Play(); break;
          case 2: Nota2.Play(); break;
          case 3: Nota3.Play(); break;
          case 4: Nota4.Play(); break;
          case 5: Nota5.Play(); break;
```

```
    }
}
```

Ejecución

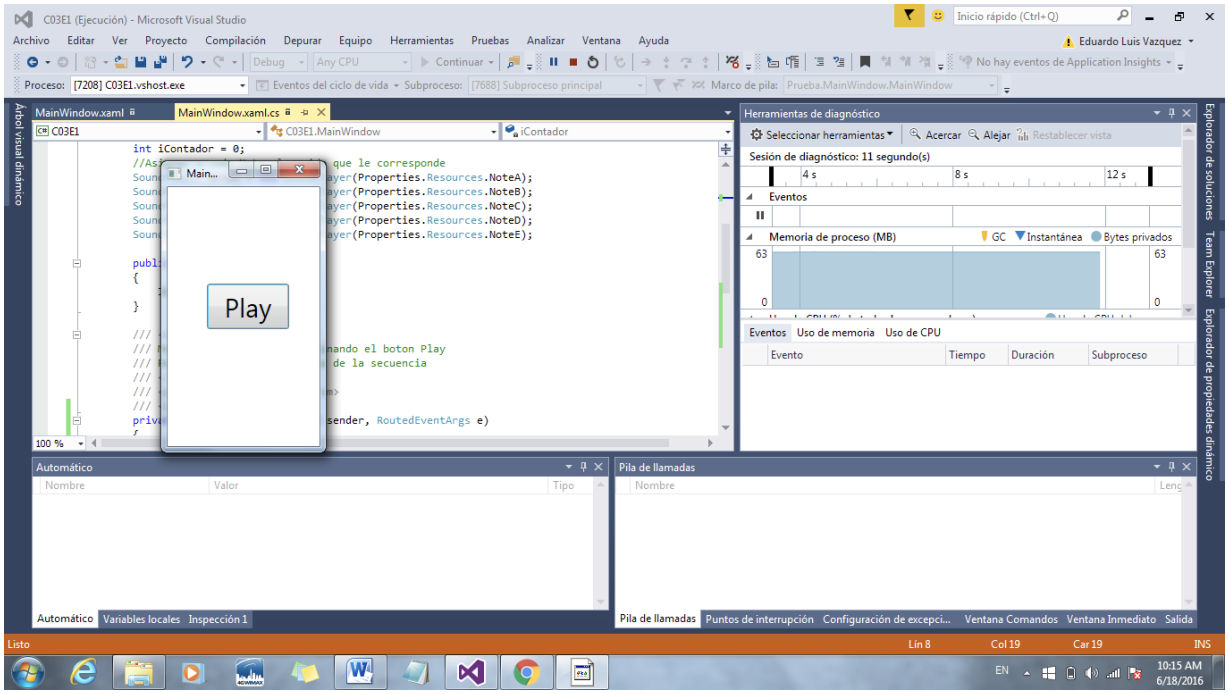


Figura 4.2

4.4 Reproducir cualquier tipo de archivo de audio

La clase **MediaElement** reproduce audio y video en una gran variedad de formatos.

La manera más sencilla de emplear la clase es agregar un elemento de tipo **mediaElement** dentro del código XAML. Este elemento debe tener un nombre (propiedad **x:Name**) para hacer referencia a él desde el programa de C#. A la propiedad **Source** del **mediaElement** se le asigna la trayectoria de acceso (ubicación dentro del disco de la computadora) en la que se encuentra el archivo de audio.

```
<MediaElement x:Name="cancion" HorizontalAlignment="Left"
              Height="100"                    Margin="-116,82,0,0"
              VerticalAlignment="Top"
              Width="100" Source="C:\miMusica.mp3"/>
```

Una de las ventajas que se tiene al utilizar esta herramienta es que la reproducción del

archivo se puede controlar desde C# ya que cuenta con distintos métodos para iniciar, pausar o detener la reproducción así como para control de volumen y la velocidad, por mencionar algunos.

4.5 Ejemplo: música de fondo (*mediaElement*)

Al ejecutar esta aplicación se reproduce una pieza musical completa.

Requerimientos

- Desarrollado en **WPF**.
- Recursos: archivo de audio.

Código en XAML

El archivo **MainWindow.xaml** del proyecto contiene un **mediaElement** de nombre canción el cual está relacionado con el archivo de audio de nombre miMusica.mp3 que se encuentra en el disco con la siguiente trayectoria: C:\miMusica.mp3

```
<Grid>
  <MediaElement x:Name="cancion" HorizontalAlignment="Left"
        Margin="-116,82,0,0" VerticalAlignment="Top"
        Height="100" Width="100"
        Source="C:\miMusica.mp3"/>
</Grid>
```

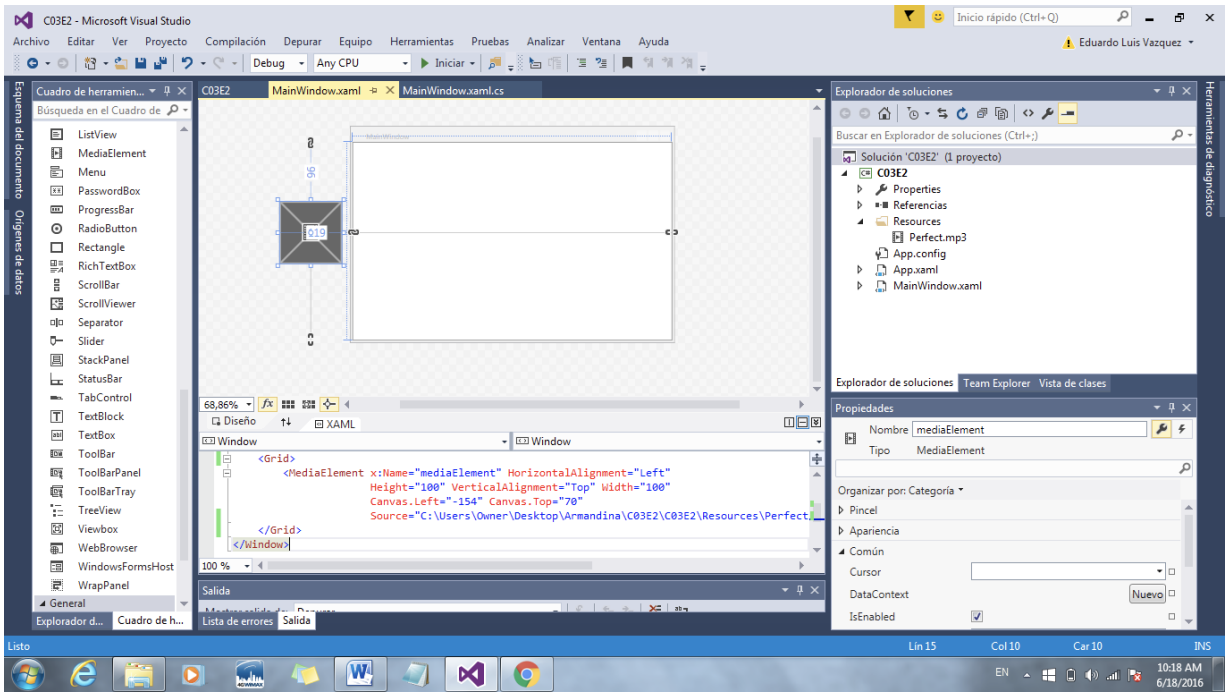


Figura 4.3

Código en C#

Para esta aplicación no es necesario realizar cambios al programa en C#.

Ejecución

Al ejecutar este programa se muestra una ventana vacía y se escucha la pieza musical.

4.6 Ejemplo: reproductor de audio (*load*, *play*, *pause*, *stop*, *volumen*, *velocidad*)

Esta aplicación es un reproductor de audio que permite al usuario buscar en el disco de su computadora y seleccionar el archivo que desea escuchar. También brinda controles para reproducir (*Play*), pausar (*Pause*), detener (*Stop*), controlar el volumen y velocidad de la reproducción.

Requerimientos

- Desarrollado en **WPF**.
- Recursos: archivo de audio.

- Biblioteca: **System.IO**

Código en XAML

El archivo **MainWindow.xaml** del proyecto está compuesto de nueve objetos:

- Un **mediaElement** de nombre canción al que se le ha modificado la propiedad **LoadedBehavior** a Manual para que su reproducción pueda ser controlada desde el programa en C#.
- Cuatro botones (*load*, *play*, *pause* y *stop*), cada uno de ellos con su correspondiente evento para controlar su funcionamiento.
- Una barra de desplazamiento (**slider**) de nombre volumen que permitirá aumentar o bajar el volumen en el que se está escuchando la melodía. La barra puede tomar valores de 0 a 1. Inicialmente estará en 0.5 (que es el valor que por defecto toma el **mediaElement** al ser reproducido). La barra tiene asociado el evento **ValueChanged** el cual se activa cuando se modifica el valor de la misma.
- Una barra de desplazamiento de nombre velocidad que permite variar la velocidad de reproducción. El valor inicial es 1 (por defecto lo toma el **mediaElement**). Esta barra también tiene asociado el evento **ValueChanged** el cual se activa cuando se modifica el valor de la misma.
- Dos etiquetas para identificar cada una de las barras de desplazamiento.

```
<Grid>
  <MediaElement x:Name="cancion" HorizontalAlignment="Left"
        Height="100" Margin="-137,110,0,0"
        VerticalAlignment="Top"  Width="100"
        LoadedBehavior="Manual"/>
  <Button  x:Name="load"  Content="Load"
HorizontalAlignment="Left"
      Margin="86,31,0,0" VerticalAlignment="Top" Width="75"
      Click="load_Click"/>
  <Button x:Name="play" Content="Play" HorizontalAlignment="Left"
      Margin="86,72,0,0" VerticalAlignment="Top" Width="75"
```

```
        Click="play_Click"/>
    <Button x:Name="pausa" Content="Pausa" HorizontalAlignment="Left"
        Margin="86,117,0,0" VerticalAlignment="Top" Width="75"
        Click="pausa_Click"/>
    <Button x:Name="stop" Content="Stop" HorizontalAlignment="Left"
        Margin="86,159,0,0" VerticalAlignment="Top" Width="75"
        Click="stop_Click" />
    <Slider x:Name="volumen" HorizontalAlignment="Left" Margin="110,210,0,0"
        VerticalAlignment="Top" Height="24" Width="100"
        Minimum="0" Maximum="1" Value="0.5"
     ValueChanged="volumen_ValueChanged"/>
    <Slider x:Name="velocidad" HorizontalAlignment="Left" Margin="110,255,0,0"
        VerticalAlignment="Top" Height="27" Width="100"
        Value="1" ValueChanged="velocidad_ValueChanged" />
    <Label x:Name="etiqueta1" Content="Volumen" HorizontalAlignment="Left"
        Height="24" Margin="29,210,0,0" VerticalAlignment="Top"
        Width="76"/>
    <Label x:Name="etiqueta2" Content="Velocidad" HorizontalAlignment="Left"
        Margin="29,255,0,0" VerticalAlignment="Top" Width="76"/>
</Grid>
```

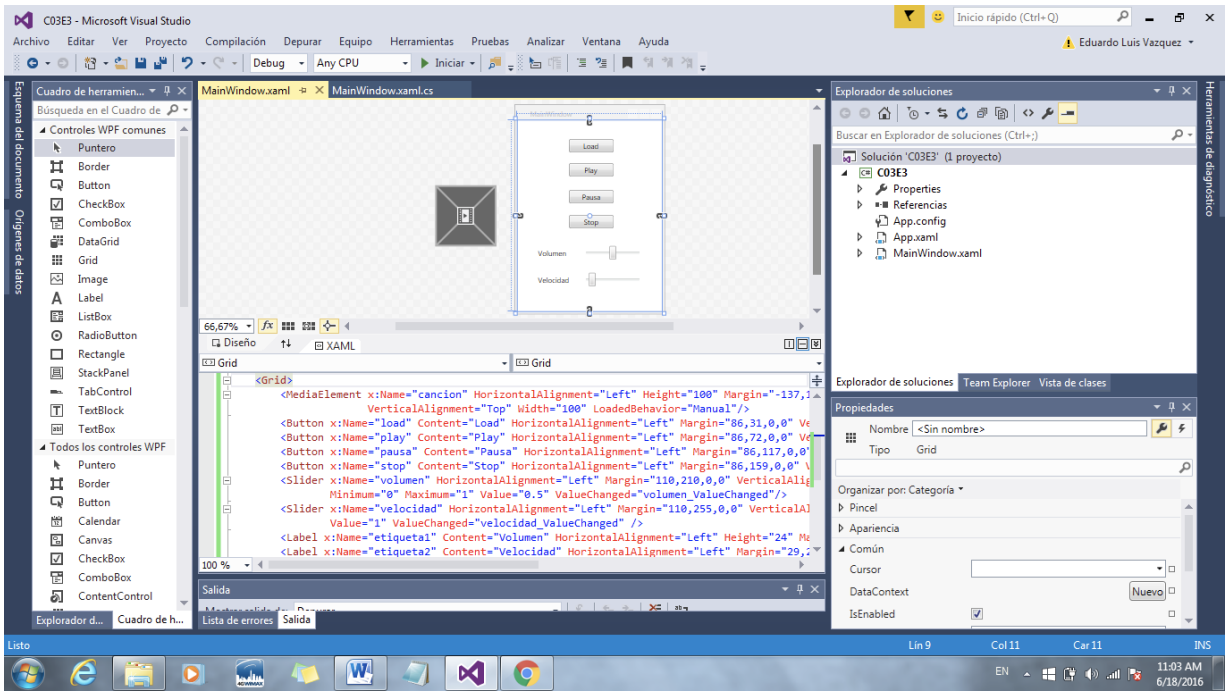


Figura 4.4

Código en C#

Para lograr que la aplicación en C# pueda visualizar el contenido del disco de la computadora para seleccionar y abrir un archivo de audio, es necesario agregar al archivo **MainWindow.xaml.cs** la biblioteca **System.IO**:

```
using System.IO;
```

El resto del programa debe estar ya conformado por los métodos de cada uno de los botones y las barras de desplazamiento.

El método **load_Click** asociado al botón Load contiene las instrucciones para abrir una ventana de Windows y mostrar los archivos del disco de los formatos establecidos en el filtro (*.mp3,*.wav,*.wmv,*.mpeg,*.avi). El archivo de audio seleccionado por el usuario se asigna al **mediaElement** canción. En caso de detectar que se cerró la ventana sin seleccionar el archivo, se genera un mensaje de error.

```
private void load_Click(object sender, RoutedEventArgs e)
{
  //Abre ventana de Windows para seleccionar el audio a reproducir
  Stream checaStream = null;
```

```
 Microsoft.Win32.OpenFileDialog dlg = new Microsoft.Win32.OpenFileDialog();
 //Realiza filtro para que la ventana solo muestre archivos del tipo indicado
 dlg.Filter = "All supported File
    Types(*.mp3,*.wav,*.wmv,*.mpeg,*.avi)|*.mp3;*.wav;*.mpeg;*.mpeg;*.avi";
 if ((bool)dlg.ShowDialog()) //Selección del archivo
 {
    try
    {
        //Se asigna al mediaElement el audio seleccionado
        if ((checaStream = dlg.OpenFile()) != null)
        {
         cancion.Source = new Uri(dlg.FileName);
        }
    }
    catch (Exception ex)
    { //Si no se seleccionó el archivo genera un error
      MessageBox.Show("Error: Falta el archivo" + ex.Message);
    }
 }
}
```

En el método **play_Click** se ejecuta el método Play para el **mediaElement** canción. Reproduce el audio a partir de la posición actual (ya sea desde el inicio o desde la posición en la que quedó cuando al pausarla). Solicita la ejecución de **InicializaPropiedades** que se encarga de asignar al **mediaElement** el volumen y la velocidad establecida en las barras de desplazamiento.

```
    private void play_Click(object sender, RoutedEventArgs e)
    {
        cancion.Play();
        InicializaPropiedades();
    }
    private void InicializaPropiedades()
    {
            cancion.Volume = volumen.Value;
            cancion.SpeedRatio = velocidad.Value;
  }
```

El método **pausa_Click** detiene la reproducción del audio con *Pause*.

```
  private void pausa_Click(object sender, RoutedEventArgs e)
  {
    cancion.Pause();
}
```

Al ejecutar **stop_Click** se detiene la reproducción y se coloca al inicio de tal manera que cuando se presiona el botón Play la melodía empiece nuevamente.

```
  private void stop_Click(object sender, RoutedEventArgs e)
  {
    cancion.Stop();
}
```

Cuando se mueve la barra de desplazamiento **Volumen** se ejecuta el método **volumen_ValueChanged** el cual establece el nuevo nivel de volumen al que debe ser escuchada la melodía. En el encabezado del método se debe especificar el tipo de dato del valor que será recibido, en este caso es double. El valor de la barra es el volumen que se le asigna al **mediaElement**.

```
  private void volumen_ValueChanged(object sender,
```

```
        RoutedPropertyChangedEventArgs<double> e)
    {
      cancion.Volume = (double)volumen.Value;
    }
```

Al igual que el volumen, cuando se mueve la barra de desplazamiento de velocidad se ejecuta el método velocidad_ValueChanged pero en este caso para definir la rapidez con la que se hará la reproducción. En el encabezado del método se debe especificar el tipo de dato del valor que será recibido; también en este caso es double. El valor de la barra es la velocidad que se le asigna al **mediaElement**.

```
    private void velocidad_ValueChanged(object sender,
        RoutedPropertyChangedEventArgs<double> e)
    {
      cancion.SpeedRatio = (double)velocidad.Value;
    }
```

Ejecución

Para reproducir la pieza musical es necesario presionar el botón **Load** para seleccionar el archivo y posteriormente utilizar el botón **Play** para escucharlo. En la ventana de Windows se observa el filtro para el tipo de archivos que se pueden seleccionar.

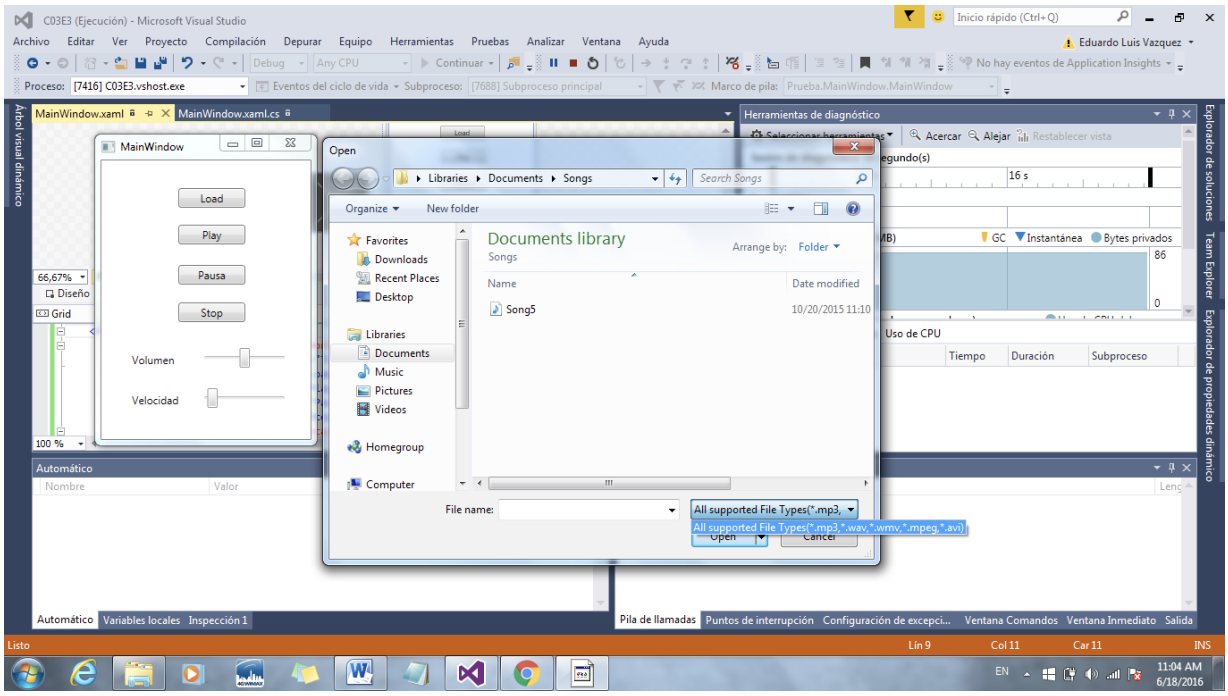


Figura 4.5

4.7 Ejemplo: repetir por siempre

Algunas veces se requiere que el audio se reproduzca una y otra vez mientras el programa está en ejecución. La clase **Storyboard** posee métodos y propiedades para crear animaciones; con ellas es posible crear una línea de tiempo y especificar que el audio debe reproducirse ilimitadamente. Esta aplicación hace uso de esta herramienta para lograr ese efecto con el audio.

Requerimientos

- Desarrollado en **WPF**.
- Recursos: archivo de audio.

Código en XAML

El archivo **MainWindow.xaml** del proyecto incluye un **mediaElement** de nombre canción al cual se le ha agregado un **Storyboard** con el que se puede controlar el comportamiento de la repetición a Forever:

```
<Grid>
```

```
<MediaElement Name="cancion" Margin="-242,74,562,101" >
    <MediaElement.Triggers>
        <EventTrigger RoutedEvent="MediaElement.Loaded">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <MediaTimeline Source="C:\NoteA.wav"
                            Storyboard.TargetName="cancion"
                            RepeatBehavior="Forever" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </MediaElement.Triggers>
</MediaElement>
</Grid>
```

Código en C#

Para esta aplicación no es necesario realizar cambios al programa en C#.

Ejecución

Al ejecutar este programa se muestra una ventana vacía y se escucha la melodía.

4.8 Ejemplo: reproducción simultanea de dos audios

Una forma de crear efectos musicales es con la reproducción simultánea de varios audios al controlar el volumen, o bien, adelantarla y atrasarla. También es posible

elaborar este tipo de efectos con un **mediaElement**. En esta aplicación, el usuario puede iniciar la reproducción de los dos audios para escucharlos al mismo tiempo. Además, tiene la posibilidad de silenciar (*mute*) o continuar la reproducción subiendo y bajando el volumen. Además puede atrasar o adelantar la reproducción de cada uno de los audios.

Requerimientos

- Desarrollado en **WPF**.
- Recursos: dos archivos de audio.

Código en XAML

Para esta aplicación, el archivo **MainWindow.xaml** contiene dos **mediaElement** de nombre audio1 y audio2. Para cada uno de ellos se cuenta con el botón de *Play* (play1 y play2), silenciar/continuar (mute1 y mute2) y adelantar/atrasar (posicion1 y posicion2). Además, se ejemplifica una manera de sincronizar (sincroniza) los audios, es decir, hacer que inicien al mismo tiempo.

Cada **mediaElement** cuenta con un evento de tipo **MediaOpened** que se activa cuando el elemento es abierto y se usa para configurar los audios desde el programa en C#. Para manipular el **mediaElement** se asigna a la propiedad **LoadedBehavior** el valor Manual.

```
<Grid>
  <MediaElement x:Name="audio1" HorizontalAlignment="Left"
Height="100"
                Margin="-124,61,0,0" VerticalAlignment="Top"
Width="100"
                Source="C:\audio1.mp3"
                LoadedBehavior="Manual"
MediaOpened="audio1_MediaOpened"/>
  <MediaElement x:Name="audio2" HorizontalAlignment="Left"
Height="100"
                Margin="-124,199,0,0" VerticalAlignment="Top"
Width="100"
                Source="C:\audio2.mp3"
```

```
                LoadedBehavior="Manual"
                MediaOpened="audio2_MediaOpened"/>
    <Button x:Name="play1" Content="Play"
HorizontalAlignment="Left"
        Margin="50,75,0,0" Height="45" VerticalAlignment="Top"
        Width="175" FontSize="26.667" Click="play1_Click"/>
    <Button x:Name="mute1" Content="Silenciar"
HorizontalAlignment="Left"
        Margin="50,152,0,0" Height="45" VerticalAlignment="Top"
        Width="175" FontSize="26.667" Click="mute1_Click"/>
    <Slider x:Name="posicion1" HorizontalAlignment="Left"
Margin="50,277,0,0"
        VerticalAlignment="Top" Width="175"
        ValueChanged="posicion1_ValueChanged"/>
    <Button x:Name="play2" Content="Play"
HorizontalAlignment="Left"
        Margin="304,75,0,0" Height="45" VerticalAlignment="Top"
        Width="175" FontSize="26.667" Click="play2_Click"/>
    <Button x:Name="mute2" Content="Silenciar"
HorizontalAlignment="Left"
        Margin="304,152,0,0" Height="45" VerticalAlignment="Top"
        Width="175" FontSize="26.667" Click="mute2_Click"/>
    <Slider x:Name="posicion2" HorizontalAlignment="Left"
Margin="304,277,0,0"
        VerticalAlignment="Top" Width="175"
        ValueChanged="posicion2_ValueChanged"/>
    <Button x:Name="sincroniza" Content="Sincronizar"
```

HorizontalAlignment="Left" Margin="526,152,0,0" Height="45"

VerticalAlignment="Top" Width="175" FontSize="26.667"

Click="sincroniza_Click"/>

<Rectangle HorizontalAlignment="Left" Height="252" Margin="269,60,0,0"

VerticalAlignment="Top" Width="231" Stroke="Black" Opacity="0.2"/>

<Rectangle HorizontalAlignment="Left" Height="251" Margin="24,61,0,0"

VerticalAlignment="Top" Width="231" Stroke="Black" Opacity="0.2"/>

<Label x:Name="label1" Content="Audio #1" HorizontalAlignment="Left"

Height="43" Margin="24,25,0,0" VerticalAlignment="Top" Width="231"

FontSize="20" FontStyle="Oblique" FontWeight="Bold"/>

<Label x:Name="label2" Content="Audio #2" HorizontalAlignment="Left"

Height="43" Margin="269,25,0,0" VerticalAlignment="Top" Width="231"

FontSize="20" FontStyle="Oblique" FontWeight="Bold"/>

<Label x:Name="labelPos1" Content="Posición" HorizontalAlignment="Left"

Margin="95,237,0,0" VerticalAlignment="Top" FontSize="18.667"/>

<Label x:Name="labelPos2" Content="Posición" HorizontalAlignment="Left"

Margin="345,237,0,0" VerticalAlignment="Top"

```
            FontSize="18.667"/>
    </Grid>
```

Código en C#

Al ejecutar la aplicación se reproducen automáticamente los dos audios; con el fin de lograr un mejor control, se añaden a la función MainWindow las instrucciones para detener ambos audios y así esperar a que el usuario los inicie cuando él lo desee.

```
public MainWindow()
{
        InitializeComponent();
        //Detener la reproducción de los audios
        audio1.Stop();
        audio2.Stop();
}
```

Cada **mediaElement** dispara un evento de tipo **MediaOpened** el cual se emplea para configurar la barra de desplazamiento utilizada para avanzar o retroceder el audio. La barra debe tener como valor mínimo 0 para representar el inicio del audio y como valor máximo la duración del audio en segundos. La duración se obtiene con la propiedad NaturalDuration la cual es convertida al formato TimeSpan (utilizado para representar un intervalo de tiempo). Para convertir un intervalo de tiempo a segundos se emplea la propiedad TotalSeconds.

```
private void audio1_MediaOpened(object sender, RoutedEventArgs e)
{
   //Obtiene la duración del audio 1
   TimeSpan pos1 = audio1.NaturalDuration.TimeSpan;
   //Configura la barra de desplazamiento para el audio 1
   posicion1.Minimum = 0;
   posicion1.Maximum = pos1.TotalSeconds;
}
```

El siguiente método se ejecuta al presionar el botón **Silenciar**. Modificar el valor del volumen de tal forma que si se está escuchando el audio le resta volumen y, por el contrario, si no se escucha lo sube. Por otro lado, para representar el cambio, se emplea la propiedad Content de la etiqueta para modificar la apariencia del botón; cuando se quita el volumen muestra la palabra Continuar pero cuando se activa el volumen cambia a Silenciar.

```
private void mute1_Click(object sender, RoutedEventArgs e)
{
  // Si el audio se esta escuchando se baja el volumen a 0
  // y el botón cambia al título Continuar
  if ( audio1.Volume == 0.5 )
  {
     audio1.Volume = 0;
     mute1.Content = "Continuar";
  }
  else
  {
     // Si el audio no se esta escuchando se sube el volumen a 0.5
     // y el botón cambia al título Silenciar
     audio1.Volume = 0.5;
     mute1.Content = "Silenciar";
  }
}
```

El método **posicion_ValueChanged** se ejecuta cuando el usuario de la aplicación hace un cambio en la barra de desplazamiento. Primero obtiene el valor de la barra, este debe corresponder con el audio ya que se hizo el ajuste a la duración en segundos. Dicho número se convierte al formato TimeSpan para asignarlo a la propiedad Position del **mediaElement**. **TimeSpan** recibe cinco valores que corresponden a días, horas, minutos, segundos y milisegundos, por esta razón, el valor de la barra se coloca en la cuarta posición.

```
private void posicion1_ValueChanged(object sender,
                RoutedPropertyChangedEventArgs<double> e)
{
    //Obtiene la posición de la barra de desplazamiento
    int pos = Convert.ToInt32(posicion1.Value);
    //Modifica la posición de reproducción
    audio1.Position = new TimeSpan(0, 0, 0, pos, 0);
}
```

Finalmente, para sincronizar los audios, se modifican las posiciones de cada uno de ellos al tiempo 0. Para esto se emplea nuevamente el formato **TimeSpan** al crear un tiempo igual a cero (new TimeSpan(0, 0, 0, 0, 0)), o bien, emplear la propiedad Zero para generarlo.

```
private void sincroniza_Click(object sender, RoutedEventArgs e)
{
    //Establece la posición del audio1 al inicio
    TimeSpan ts = new TimeSpan(0, 0, 0, 0, 0);
    audio1.Position = ts;
    //Establece la posición del audio2 al inicio
    audio2.Position = TimeSpan.Zero;
}
```

4.9 Generación de audio basado en frecuencias

La clase **Console**, que es la que se emplea para los procesos de lectura y desplegado, también genera sonidos a través del método **Beep**.

Beep tiene dos presentaciones, la primera produce un sonido especifico cuando se ejecuta la siguiente instrucción:

```
Console.Beep();
```

Otra forma de producir audio con el método **Beep** es especificar la frecuencia de la nota

y la duración de la misma. En Internet es fácil conseguir piezas musicales representadas de esta manera.

```
Console.Beep(frecuencia, duración);
```

El valor para la frecuencia debe ser un número entero en el rango de 37 a 32767. La duración también es un entero medido en milisegundos.

4.10 Ejemplo: generar audio (*beep*)

Esta aplicación genera audio reproduciendo frecuencias a distintas duraciones.

Requerimientos

- Desarrollado en **WPF**.

Código en XAML

El archivo **MainWindow.xaml** del proyecto únicamente contiene el botón Play que al dar clic dispara el evento play_Click:

```
<Grid>
  <Button x:Name="play" Content="Play" HorizontalAlignment="Left"
      Margin="107,67,0,0" VerticalAlignment="Top" Width="108"
      FontSize="24" Click="play_Click" Height="57"/>
</Grid>
```

Código en C#

El programa en C# es muy simple; solo requiere incluir en el método del botón las secuencias de sonidos que se desean reproducir especificando para cada uno de ellos la frecuencia y duración.

```
private void play_Click(object sender, RoutedEventArgs e)
{
   Console.Beep(300, 600);
   Console.Beep(400, 600);
   Console.Beep(500, 600);
```

```
        Console.Beep(600, 600);
        Console.Beep(700, 600);
        Console.Beep(400, 1000);
        Console.Beep(300, 1000);
    }
```

Ejecución

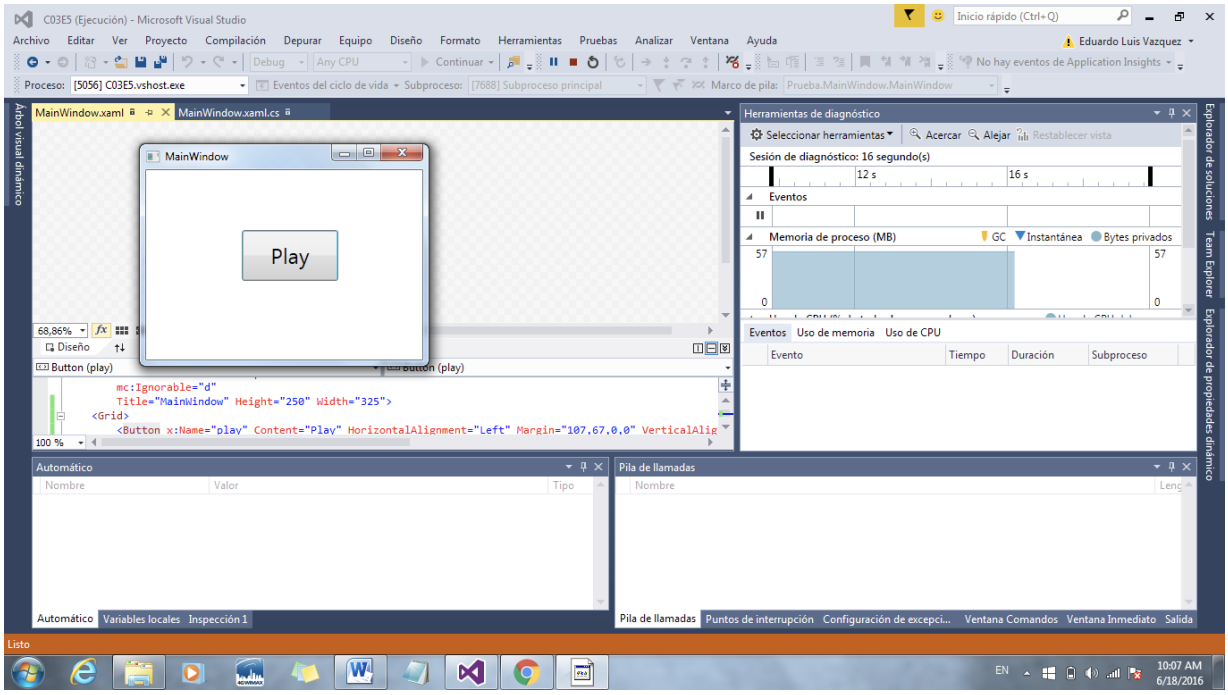


Figura 4.6

Capítulo 5. Reflejar el movimiento de la persona

Una de las funcionalidades del Kinect consiste en detectar la o las personas enfrente a él y obtener información de la posición en la que se encuentran algunos puntos de su cuerpo.

En este capítulo se analizará la forma de adquirir esta información y cómo puede ser utilizada en aplicaciones relacionadas con el movimiento del cuerpo.

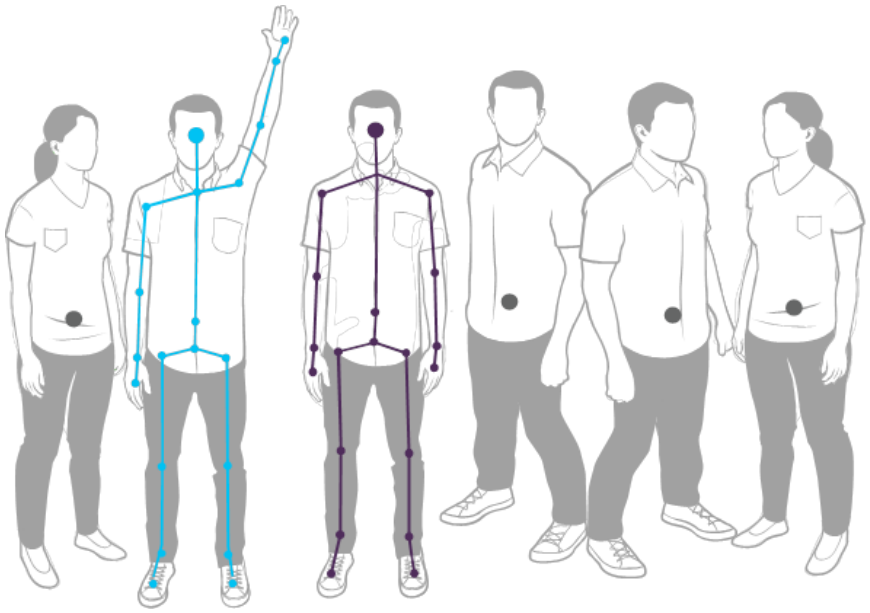


Figura 5.1

5.1 Puntos del cuerpo (*joints*)

Lo primero que debe hacer un programa con el Kinect es establecer comunicación con él y configurarlo para indicar, entre otras cosas, cuál será el método a ejecutar cada vez que el dispositivo tenga un paquete de información que desee enviar al programa.

Una vez que el Kinect se encuentra conectado, inicia un ciclo ilimitado, que consiste en registrar lo que capta de la persona frente a él (**Tracking** o mapeo) y formar paquetes de información (**Frames**). El Kinect puede generar distintos tipos de paquetes; el que se verá en esta ocasión es el relacionado con el esqueleto o cuerpo de la persona.

Cada vez que el dispositivo tiene un **Frame**, genera un evento que inicia la ejecución de la función del programa, toma la información que necesita del **Frame** y la utiliza para proveer alguna funcionalidad.

Aunque en términos generales el proceso realizado por las dos versiones del Kinect (1 y 2) es similar, se pueden encontrar algunas diferencias, sobre todo porque el modelo más reciente posee mayores capacidades. En lo que respecta al nombre que se le da a los elementos relacionados con los paquetes, el Kinect V1 emplea la palabra **Skeleton** mientras que el Kinect V2 la cambia por **Body** como se puede ver en la siguiente tabla:

Kinect V1	Kinect V2	Kinect V3
Skeleton Tracking	**Body Tracking**	Nombre que se le da al mapeo del esqueleto que realiza el dispositivo.
Skeleton Frame	**Body Frame**	Paquete de información generado.
SkeletonStream	**BodyFrameReader**	Proceso realizado para indicar al Kinect que debe iniciar la lectura de esqueleto.
SkeletonFrameReady	**FrameArrived**	Evento que se activa cada vez que se tiene listo un Frame.
Skeleton	**Body**	Objeto almacenado en el Frame que contiene la información del esqueleto.

Tabla 5.1

El Kinect está constantemente generando paquetes de información (más rápido que lo que una persona se mueve); por esto no conviene realizar actividades cada vez que se tenga un paquete. Lo más común es que el programa incluya funciones de **Timer** que lean lo que está haciendo la persona cada cierto tiempo, acorde a la capacidad real de movimiento de un individuo.

La versión 2 del Kinect tiene la capacidad de obtener la información completa de seis personas detectadas, sin embargo, la versión 1 solo alcanza a obtener datos del esqueleto de dos.

Cada punto del cuerpo detectado por el Kinect es un **Joint**. Cada Joint es un punto representado por tres coordenadas (X, Y y Z), de las cuales X y Y corresponden a la posición del **Joint** mientras que Z es la distancia del **Joint** al sensor.

Los **Joints** son identificados por el nombre en inglés de la parte del cuerpo a la que representan (*head*, *shoulders*, *elbows*, *wrists*, *arms*, *spine*, *hips*, *knees*, *ankles*, *etc*). La versión 1 del Kinect detecta 20 Joints (imagen izquierda), mientras que la versión 2 maneja un total de 25 (imagen derecha).

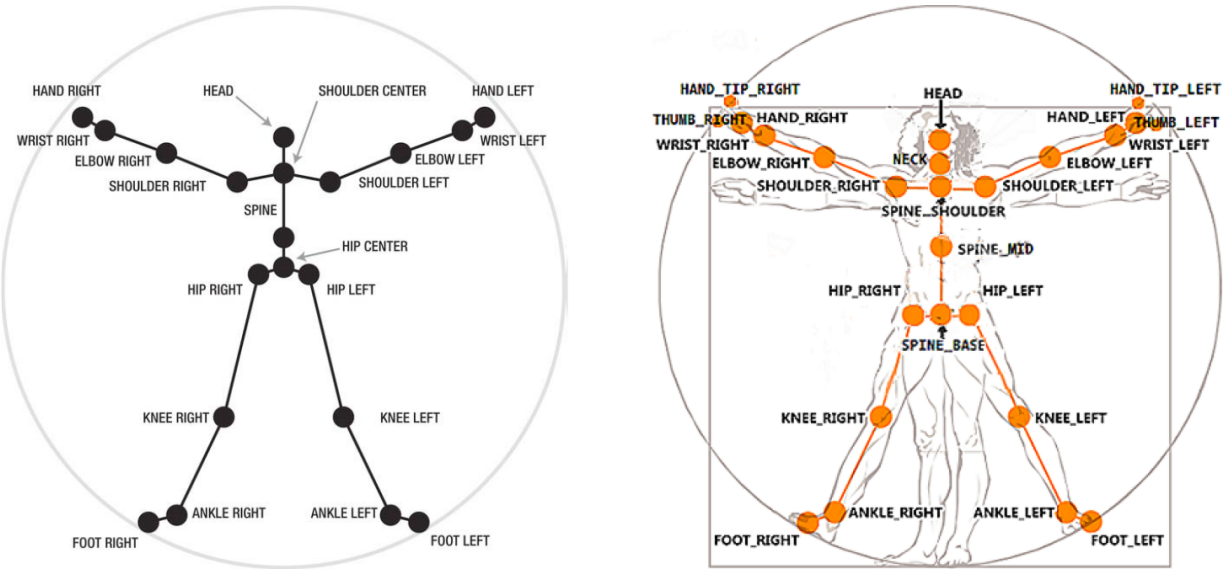


Figura 5.2

5.2 Utilizar los diferentes Joints

Se puede reflejar el movimiento de la persona que está frente al Kinect a través de la coordenada de los Joints para mover figuras geométricas o imágenes. Además, al aplicar cálculos matemáticos con esas mismas coordenadas es posible determinar la estatura de la persona o el ángulo que forma su antebrazo con el codo.

Como ya se mencionó, el dispositivo está constantemente enviando Frames; cada uno de ellos contiene objetos de tipo Skeleton o Body, según la versión utilizada. La cantidad de Skeletons (o Bodies) varía según las personas presentes frente al Kinect. Los pasos para obtener los Joints son los siguientes:

1. Abrir el Frame.

2. Copiar los datos del Frame a un arreglo de Skeletons (Bodies).

3. Acceder a cada elemento del arreglo deseado e indicar el Joint a utilizar.

Lo pasos anteriores se logran mediante las siguientes líneas de código, las cuales también se pueden ver en las plantillas (ver anexo) tanto para versión 1 como para la versión 2.

Kinect versión 1

```
//Este método se ejecuta cada vez que el Kinect tiene un Frame.

//Se encarga de tomar la información de los esqueletos y pasarla a un arreglo.
```

```
    //de donde se extrae los datos del primer esqueleto y lo
    //envía al método usarSkeleton para realizar algún proceso.
    private void Kinect_FrameReady(object sender,
SkeletonFrameReadyEventArgs e)
    {
      // Arreglo para almacenar la información de los esqueletos
      Skeleton[] skeletons = new Skeleton[0];
      Skeleton skeleton;
      // Copia al arreglo la información de los esqueletos
      using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
      {
      if (skeletonFrame != null)
      {
      skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
      skeletonFrame.CopySkeletonDataTo(skeletons);
      }
    }
    // Extrae la información del primer esqueleto
    skeleton = (from trackSkeleton in skeletons where
trackSkeleton.TrackingState ==
            SkeletonTrackingState.Tracked select
            trackSkeleton).FirstOrDefault();
    if (skeleton == null)
    {
            return;
    }
```

```
// Enviar la información del primer esqueleto para su uso
this.usarSkeleton(skeleton);
}
```

```
//Recibe la información de un esqueleto y la utiliza para hacer que una elipse
//denominada Puntero siga el movimiento de la mano derecha
private void usarSkeleton(Skeleton skeleton)
{ //Extrae la información del Joint de la mano derecha
  Joint joint1 = skeleton.Joints[JointType.HandRight];
  // Si el Joint está listo obtener las coordenadas
  if (joint1.TrackingState == JointTrackingState.Tracked)
  {
     // Obtiene las coordenadas (x, y) del Joint
     joint_Point = this.SkeletonPointToScreen(joint1.Position);
     dMano_X = joint_Point.X;
     dMano_Y = joint_Point.Y;
     //Emplea las coordenadas del Joint para mover la elipse
     Puntero.SetValue(Canvas.TopProperty, dMano_Y);
     Puntero.SetValue(Canvas.LeftProperty, dMano_X);
     // Obtiene el Id de la persona mapeada
     LID.Content = skeleton.TrackingId;
  }
}
```

Kinect versión 2

```
    //Este método se ejecuta cada vez que el Kinect tiene un Frame.
    //Se encarga de tomar la información de los cuerpos y pasarla a un arreglo.
    //Posteriormente cada cuerpo es enviado al método usarBody para realizar
    //algún proceso.
    private void Kinect_FrameReady(object sender, BodyFrameArrivedEventArgs e)
    {
      // Adquirir el BodyFrame
      using (BodyFrame mibodyFrame = e.FrameReference.AcquireFrame())
      {
        // Verificar contenido de datos
        if (mibodyFrame != null)
        {
          // Crea un arreglo y almacena en él la información de los cuerpos
          this.bodies = new Body[mibodyFrame.BodyCount];
          mibodyFrame.GetAndRefreshBodyData(this.bodies);
          // Ejecuta el método usarBody para cada uno de los cuerpos
          foreach (Body body in bodies)
          {
            if (body != null)
            {
              this.usarBody(body);
            }
          }
        }
```

```
    }
}
```

```
//Recibe la información de un cuerpo y la utiliza para hacer que una elipse
//denominada Puntero siga el movimiento de la mano derecha
private void usarBody(Body miBody)
{
    //Extrae la información del Joint de la mano derecha
    Joint miJoint = miBody.Joints[JointType.HandRight];
    // Verifica estado del Joint
    if (miJoint.TrackingState == TrackingState.Tracked)
    {
        // Obtiene la coordenada del Joint
        joint_Point = this.BodyPointToScreen(miJoint);
        // Emplea las coordenadas del Joint para mover la elipse
        Puntero.SetValue(Canvas.LeftProperty, (double)joint_Point.X);
        Puntero.SetValue(Canvas.TopProperty, (double)joint_Point.Y);
        // Obtiene el Id de la persona mapeada
        LID.Content = miBody.TrackingId;
    }
}
```

Como se puede observar en los métodos anteriores, **usarSkeleton** y **usarBody**, la forma de indicar el Joint que se desea utilizar es mediante la enumeración JointType y el nombre de la parte del cuerpo al que está haciendo referencia:

Kinect versión 1

```
Joint joint1 = skeleton.Joints[JointType.HandRight]; //Mano derecha
Joint joint1 = skeleton.Joints[JointType.Head]; //Cabeza
Joint joint1 = skeleton.Joints[JointType.FoodRight]; //Pie derecha
```

Kinect versión 2

```
Joint joint1 = miBody.Joints[JointType.HandRight]; //Mano derecha
Joint joint1 = miBody.Joints[JointType.Head]; //Cabeza
Joint joint1 = miBody.Joints[JointType.FoodRight]; //Pie derecha
```

La coordenada del Joint no corresponde con las coordenadas de la pantalla. Las coordenadas de un Joint (X, Y, Z) representan un punto en el espacio tridimensional donde el valor de X o de Y puede ser negativo o positivo. En cambio, un punto de la ventana dentro del **Canvas** es definido por valores positivos que va de uno en adelante. Por lo tanto, para poder ubicar un objeto en la ventana cuyo movimiento coincida con uno de los Joint, es necesario convertir (**mapear**) los puntos (X, Y) del Joint a un punto (X, Y) de la ventana.

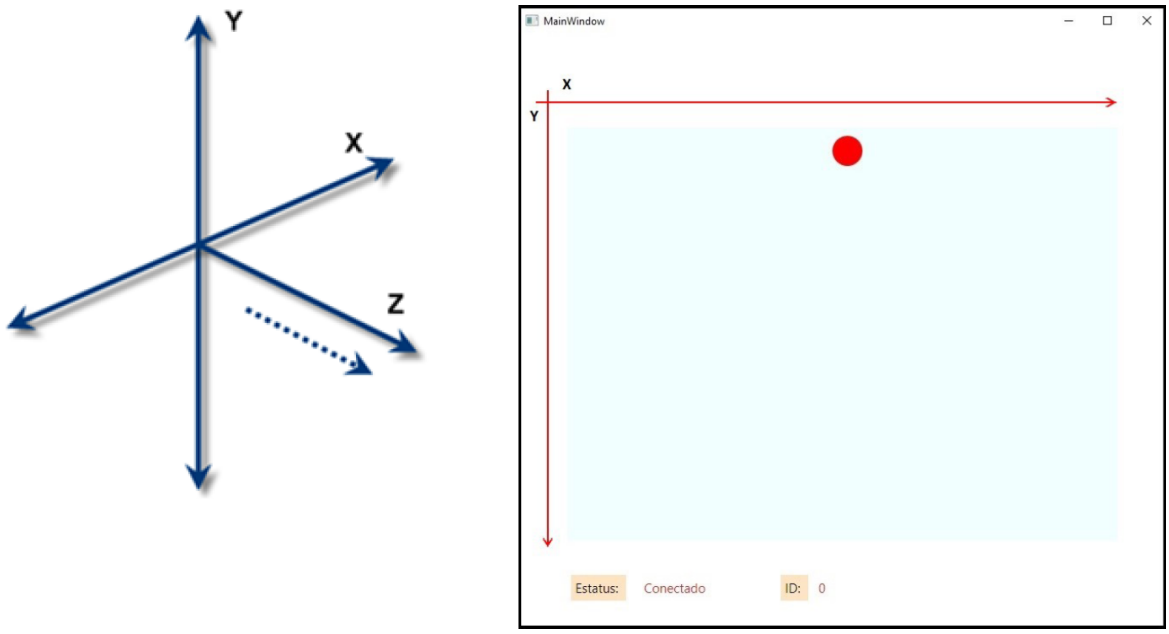


Figura 5.3

Para realizar este mapeo se emplea el método SkeletonPointToScreen, en Kinect versión 1 y BodyPointToScreen en Kinect versión 2:

Kinect versión 1

```
joint_Point = this.SkeletonPointToScreen(joint1.Position);
```

Kinect versión 2

```
joint_Point = this.BodyPointToScreen(miJoint);
```

Estos métodos emplean códigos de las librerías del SDK de Kinect para realizar la conversión como se puede ver a continuación:

Kinect versión 1

```
//Este método se emplea para convertir la coordenada de un Joint a un punto
//de la ventana del programa.
private Point SkeletonPointToScreen(SkeletonPoint skelpoint)
{
  // Convertir un punto a "Depth Space" en una resolución de 640x480
  DepthImagePoint depthPoint = this.miKinect.CoordinateMapper.
                               MapSkeletonPointToDepthPoint(skelpoint,
                               DepthImageFormat.Resolution640x480Fps30);
  return new Point(depthPoint.X, depthPoint.Y);
}
```

Kinect versión 2

```
//Este método se emplea para convertir la coordenada de un Joint a un punto
//de la ventana del programa.
private Point BodyPointToScreen(Joint miJoint)
{
  // Convertir un punto a "Depth Space"
```

```
    DepthSpacePoint depthSpacePoint = miKinect.CoordinateMapper.
                                      MapCameraPointToDepthSpace(miJoint.Position
                                      );
    return new Point(depthSpacePoint.X, depthSpacePoint.Y);
}
```

5.3 Ejemplo: mantener la trayectoria (Kinect V1)

Este sencillo ejemplo muestra una forma de interactuar con elementos en pantalla mediante los datos que proporciona el Kinect. Esta aplicación es un juego en el que se pide a la persona que mueva con la mano derecha un objeto rojo siguiendo un trayectoria circular

Requerimientos

- Proyecto **WPF**.
- Plantilla V1 (configuración Skeleton).

Nota: como parte del anexo se incluye el proyecto WPF para el manejo del esqueleto. Este proyecto ya tiene las referencias a Kinect. Se puede sacar una copia del mismo y modificarlo para realizar la aplicación. Si se decide iniciar desde cero se deben añadir al proyecto las referencias al Kinect. En caso de copiar el código y pegarlo en un nuevo proyecto es importante verificar que el nombre del proyecto creado sea el empleado tanto en el programa en XAML como en el programa en C#.

Código en XAML

Dentro del programa en XAML se tienen tres elipses. Dos de ellas (Circulo1 y Circulo2) se emplearán para definir la trayectoria en la que se debe mover la mano y el Puntero utilizado para ubicar la posición de la mano derecha con respecto a la pantalla. El resto del contenido del Grid son etiquetas que se emplean para mostrar el estatus del Kinect.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
 Grid.Row="1"
      Background="Azure" HorizontalAlignment="Center">
```

```
    <Ellipse x:Name="Circulo1" Width="300" Height="300"
        Canvas.Left="170" Canvas.Top="90" Fill="Black" />
    <Ellipse x:Name="Circulo2" Width="200" Height="200"
        Canvas.Left="220" Canvas.Top="140" Fill="Azure" />
    <Ellipse x:Name="Puntero" Width="30" Height="30"
        Canvas.Left="576" Canvas.Top="355" Fill="Red" />
    </Canvas>
    <Label x:Name="L1" Content="Estatus: " FontSize="15"
FontWeight="Bold"
        Background="Bisque" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="61,628,0,0"/>
    <Label x:Name="LEstatus" Content="Desconectado" FontSize="15"
        Foreground="#FFDA2828" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="141,628,0,0"/>
    <Label x:Name="L3" Content="ID: " FontSize="15"
FontWeight="Bold"
        Background="Bisque" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="305,628,0,0"/>
    <Label x:Name="LID" Content="0" FontSize="15"
Foreground="#FFDA2828"
        HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="344,628,0,0"/>
</Grid>
```

Las figuras **Circulo1** y **Circulo2** se acomodan, una encima de la otra, para formar un aro suficientemente ancho que simule ser el camino por el que pasará el **Puntero**.

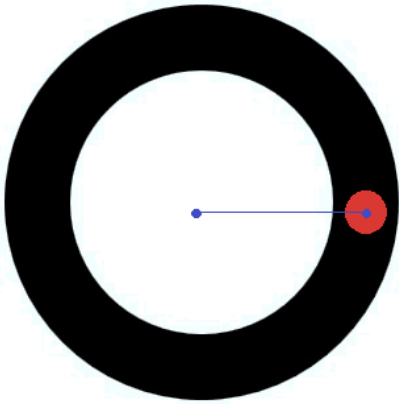


Figura 5.4

Código en C#

Esta aplicación toma como base la plantilla editada para **Skeleton** a la cual se le agregan instrucciones para verificar si el movimiento de la mano derecha hace que el círculo rojo siga la trayectoria especificada.

Una forma simple de verificarlo es calcular la distancia que existe entre el centro del aro y el centro del círculo rojo y compararla con los radios de ambos círculos. En la parte superior del programa se deben agregar las variables que se emplean para almacenar las coordenadas del centro del aro y el radio de cada uno de los círculos y dentro del **MainWindow** se realizan los cálculos para dar valor a dichas variables.

```
    //Variables que se emplearán para almacenar el centro del aro
    double dXC, dCY;
    //Variables que almacenan el radio de cada uno de los círculos.
    double dRadioC1, dRadioC2;
  public MainWindow()
  {
    InitializeComponent();
    //Calcula la coordenada del centro del aro
    dXC = (double)Circulo2.GetValue(Canvas.LeftProperty) +
  (Circulo2.Width / 2);
    dYC = (double)Circulo2.GetValue(Canvas.TopProperty) +
  (Circulo2.Height / 2);
```

```
    //Calcular el radio de cada uno de los círculos
    dRadioC1 = Circulo1.Width / 2;
    dRadioC2 = Circulo2.Width / 2;
    // Realizar configuraciones e iniciar el Kinect
    Kinect_Config();
}
```

Además se debe añadir al programa una función que calcule la distancia entre el centro del aro y el centro del círculo rojo. Determine si este último se encuentra dentro del trayecto requerido.

```
private bool checarDistancia()
{
    //Obtiene la coordenada del centro del círculo que mueve la persona
    double dX1 = (double)Puntero.GetValue(Canvas.LeftProperty) +
(Puntero.Width / 2);
    double dY1 = (double)Puntero.GetValue(Canvas.TopProperty) +
(Puntero.Height / 2);
    //Calcula la distancia entre el centro del Puntero (círculo rojo) y
    //el centro del aro
    double dDistancia = Math.Sqrt(Math.Pow(dXC - dX1, 2) +
Math.Pow(dYC - dY1, 2));
    //Compara la distancia calculada con los radios de los dos círculos que
forman
    //el aro en el entendido de que si la distancia es mayor al círculo más
grande
    //o menor al círculo más pequeño, entonces el círculo rojo
    //se ha salido del trayecto.
    if (dDistancia < dRadioC1 || dDistancia > dRadioC2)
        return true;
```

```
        else
            return false;
    }
```

Para finalizar, se añade al método **usarSkeleton**, la instrucción para solicitar la ejecución del método **checarDistancia** con el fin de determinar el color del aro dependiendo de si el círculo rojo continúa o no dentro del aro.

```
    //Recibe la información de un esqueleto y la utiliza para hacer que una elipse
    //denominada Puntero siga el movimiento de la mano derecha
    private void usarSkeleton(Skeleton skeleton)
    { //Extrae la información del Joint de la mano derecha
      Joint joint1 = skeleton.Joints[JointType.HandRight];
      // Si el Joint está listo obtener las coordenadas
      if (joint1.TrackingState == JointTrackingState.Tracked)
      {
          // Obtiene las coordenadas (x, y) del Joint
          joint_Point = this.SkeletonPointToScreen(joint1.Position);
          dMano_X = joint_Point.X;
          dMano_Y = joint_Point.Y;
          //Emplea las coordenadas del Joint para mover la elipse
          Puntero.SetValue(Canvas.TopProperty, dMano_Y);
          Puntero.SetValue(Canvas.LeftProperty, dMano_X);
          // Obtiene el Id de la persona mapeada
          LID.Content = skeleton.TrackingId;
          // Verificar si el círculo rojo se encuentra dentro de la trayectoria
          if (checarDistancia())
          {
```

```
            Circulo1.Fill = Brushes.Yellow; //No se encuentra
        }
        else
        {
            Circulo1.Fill = Brushes.Black; //Sí se encuentra
        }
    }
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución. Cuando se mueve la mano fuera de la trayectoria el color es amarillo, si se mueve dentro es negro. Al agregar un poco más de código, este tipo de aplicaciones se pueden orientar hacia la realización de ejercicios de coordinación o corrección de movimientos y posturas.

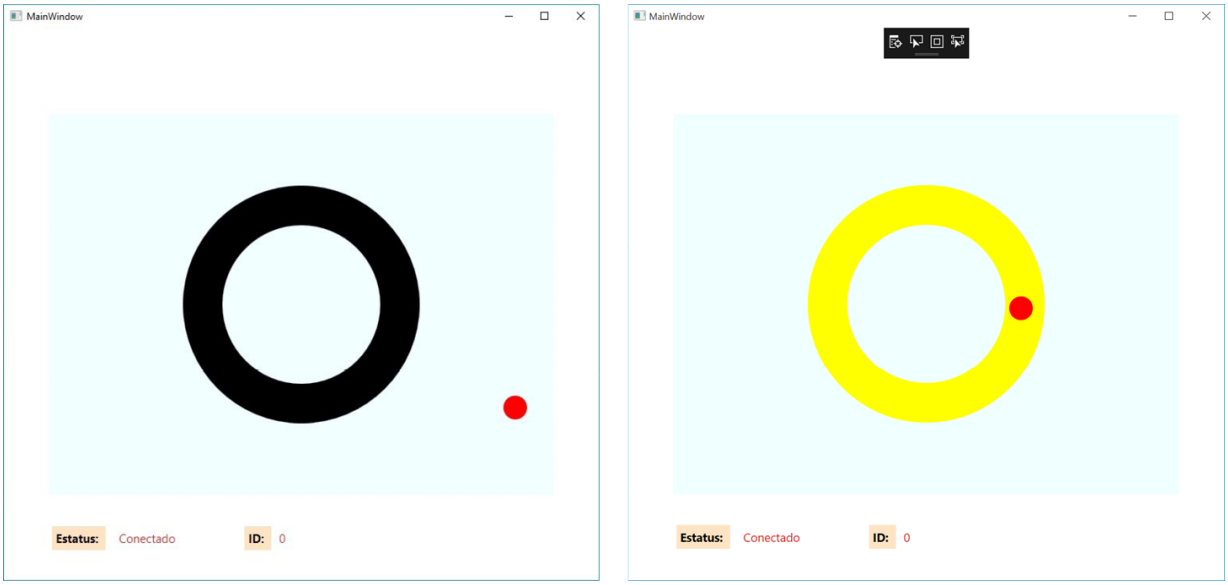


Figura 5.5

5.4 Ejemplo: mantener la trayectoria (Kinect V2)

Este sencillo ejemplo muestra una forma de interactuar con elementos en pantalla mediante los datos que proporciona el Kinect. Esta aplicación es un juego en el que se pide al usuario mover con la mano derecha un objeto rojo siguiendo una trayectoria circular.

La trayectoria cambia de color cada vez que la persona se sale del camino.

Requerimientos

- Proyecto **WPF**.
- Plantilla V2 (configuración Skeleton).

Nota: como parte del anexo se incluye el proyecto WPF para el manejo del cuerpo. Este ya tiene las referencias a Kinect. Se puede sacar una copia del mismo y modificarlo para realizar la aplicación. Si se decide iniciar desde cero, no hay que olvidar añadir al proyecto las referencias al Kinect. En caso de copiar el código y pegarlo en un nuevo proyecto es importante verificar que el nombre del proyecto creado sea el mismo en el programa en XAML como en el programa en C#.

Código en XAML

Dentro del programa en XAML se tienen tres elipses. Dos de ellas (Circulo1 y Circulo2) se emplearán para definir la trayectoria en la que se debe mover la mano y el Puntero se utilizará para que la persona ubique la posición de su mano derecha con respecto a la pantalla. El resto del contenido del Grid son etiquetas que muestran el estatus del Kinect.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480" Grid.Row="1"
          Background="Azure" HorizontalAlignment="Center">
    <Ellipse x:Name="Circulo1" Width="300" Height="300"
             Canvas.Left="170" Canvas.Top="90" Fill="Black" />
    <Ellipse x:Name="Circulo2" Width="200" Height="200"
             Canvas.Left="220" Canvas.Top="140" Fill="Azure" />
    <Ellipse x:Name="Puntero" Width="30" Height="30"
             Canvas.Left="576" Canvas.Top="355" Fill="Red" />
  </Canvas>
  <Label x:Name="L1" Content="Estatus: " FontSize="15" FontWeight="Bold"
```

```
        Background="Bisque"  HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="61,628,0,0"/>
    <Label x:Name="LEstatus" Content="Desconectado" FontSize="15"
        Foreground="#FFDA2828"  HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="141,628,0,0"/>
    <Label  x:Name="L3"  Content="ID:  "  FontSize="15"
FontWeight="Bold"
        Background="Bisque"  HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="305,628,0,0"/>
    <Label  x:Name="LID"  Content="0"  FontSize="15"
Foreground="#FFDA2828"
        HorizontalAlignment="Left"  VerticalAlignment="Top"
        Margin="344,628,0,0"/>
</Grid>
```

Las figuras **Circulo1** y **Circulo2** se acomodan una encima de la otra para formar un aro suficientemente ancho que simule ser el camino por el que pasará el **Puntero**.

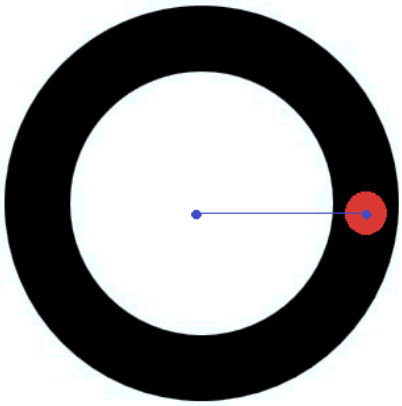


Figura 5.6

Código en C#

Esta aplicación toma como base la plantilla editada para Skeleton versión 2 a la cual se agregan instrucciones para verificar si el movimiento de la mano derecha hace que el círculo rojo siga el trayecto especificado.

Una forma simple de verificarlo es calcular la distancia que existe entre el centro del aro y el centro del círculo rojo y compararla con los radios de ambos círculos. En la parte superior del programa se deben agregar las variables que se emplean para almacenar las coordenadas del centro del aro y el radio de cada uno de los círculos y dentro del **MainWindow** se calculan los valores de dichas variables.

```
    //Variables que se emplearán para almacenar el centro del aro
    double dXC, dCY;
    //Variables que almacenan el radio de cada uno de los círculos.
    double dRadioC1, dRadioC2;
  public MainWindow()
  {
    InitializeComponent();
    //Calcula la coordenada del centro del aro
    dXC = (double)Circulo2.GetValue(Canvas.LeftProperty) +
   (Circulo2.Width / 2);
    dYC = (double)Circulo2.GetValue(Canvas.TopProperty) +
   (Circulo2.Height / 2);
    //Calcular el radio de cada uno de los círculos
    dRadioC1 = Circulo1.Width / 2;
    dRadioC2 = Circulo2.Width / 2;
    // Realizar configuraciones e iniciar el Kinect
    Kinect_Config();
  }
```

Además se debe añadir al programa una función que calcule la distancia entre el centro del aro y el centro del círculo rojo y determine si este último se encuentra en el trayecto requerido.

```
private bool checarDistancia()
{
  //Obtiene la coordenada del centro del círculo que mueve la persona
  double dX1 = (double)Puntero.GetValue(Canvas.LeftProperty) + (Puntero.Width / 2);
  double dY1 = (double)Puntero.GetValue(Canvas.TopProperty) + (Puntero.Height / 2);
  //Calcula la distancia entre el centro del Puntero (círculo rojo) y
  //el centro del aro
  double dDistancia = Math.Sqrt(Math.Pow(dXC - dX1, 2) + Math.Pow(dYC - dY1, 2));
  //Compara la distancia calculada con los radios de los dos círculos que forman
  //el aro en el entendido de que si la distancia es mayor al círculo más grande
  //o menor al círculo más pequeño, entonces el círculo rojo
  //se ha salido del trayecto.
  if (dDistancia < dRadioC1 || dDistancia > dRadioC2)
      return true;
  else
      return false;
}
```

Para finalizar, se añade al método **usarBody**, la instrucción solicitar la ejecución del método **checarDistancia** con el fin de determinar el color del aro dependiendo de si el círculo rojo continua o no dentro del aro.

```
//Recibe la información de un cuerpo y la utiliza para hacer que una elipse
//denominada Puntero siga el movimiento de la mano derecha
private void usarBody(Body miBody)
```

```
    {
      //Extrae la información del Joint de la mano derecha
      Joint miJoint = miBody.Joints[JointType.HandRight];
      // Verifica estado del Joint
      if (miJoint.TrackingState == TrackingState.Tracked)
      {
      // Obtiene la coordenada del Joint
      joint_Point = this.BodyPointToScreen(miJoint);
      // Emplea las coordenadas del Joint para mover la elipse
      Puntero.SetValue(Canvas.LeftProperty, (double)joint_Point.X);
      Puntero.SetValue(Canvas.TopProperty, (double)joint_Point.Y);
      // Obtiene el Id de la persona mapeada
      LID.Content = miBody.TrackingId;
      // Verificar si se encuentra dentro de la trayectoria
      if (checarDistancia())
      {
          Circulo1.Fill = Brushes.Yellow;
      }
      else
      {
          Circulo1.Fill = Brushes.Black;
      }
    }
  }
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución de la aplicación. Cuando se mueve la mano fuera de la trayectoria su color es amarillo, si se mueve dentro es negro. De agregar un poco más de código este tipo de aplicaciones se podrían orientar a la realización de ejercicios de coordinación o corrección de movimientos y posturas.

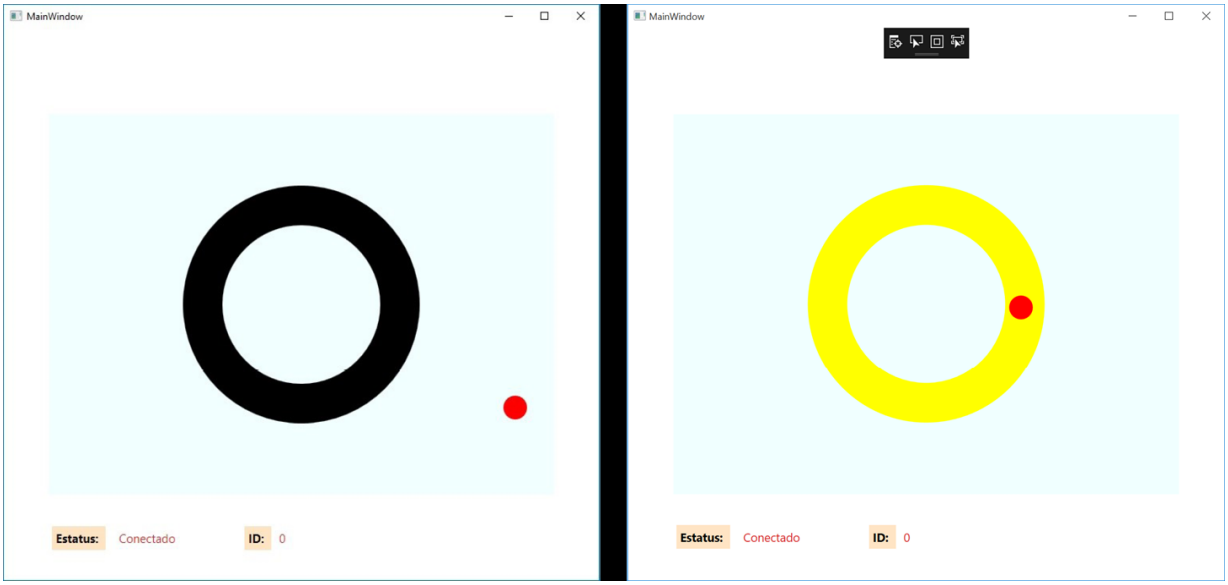


Figura 5.7

5.5 Ejemplo: tocar batería (Kinect V1)

Este ejemplo muestra la forma de combinar el manejo de colisiones, el uso de un timer y el empleo de la información de los Joints que proporciona el Kinect para desarrollar una aplicación que permita la simulación de tocar una batería usando las manos para controlar el movimiento de las baquetas. La aplicación muestra la imagen de una batería en la que se han resaltado con círculos color cian, las secciones que pueden producir un sonido.

Requerimientos

- Proyecto **WPF**.
- Plantilla V1 (configuración Skeleton).
- Archivos de audio en formato MP3.
- Imagen de batería y baquetas en formato PNG.

Código en XAML

En el archivo XAML se incluye una imagen de fondo que muestra una batería, cuatro

elipses para indicar los elementos de la batería que cuentan con un sonido: efecto1, efecto2, efecto3, efecto4 y efecto5 y las baquetas que son representadas por las imágenes manoIzq y manoDer. Para crear los sonidos se añade un **MediaElement** para cada uno de los efectos en los que se indica la trayectoria dentro del disco de la computadora donde se localiza el audio; estas trayectorias deben ser ajustadas a la computadora en la que se ejecutará la aplicación. Los elementos restantes son usados para mostrar el estatus del Kinect.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
 Grid.Row="1"
      Background="Azure" HorizontalAlignment="Center">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480"
      Width="640" Source="bateria.png" Stretch="Fill"/>
    <Ellipse x:Name="efecto1" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="102" Canvas.Top="270"/>
    <Ellipse x:Name="efecto2" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="223" Canvas.Top="256"/>
    <Ellipse x:Name="efecto3" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="281" Canvas.Top="369"/>
    <Ellipse x:Name="efecto4" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="345" Canvas.Top="146"/>
    <Ellipse x:Name="efecto5" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="445" Canvas.Top="86"/>
    <Image Name="manoIzq" Canvas.Left="10" Canvas.Top="430"
    Height="40"
      Width="140" Source="baqueta1.png" Stretch="Fill"/>
    <Image Name="manoDer" Canvas.Left="500" Canvas.Top="430"
    Height="40"
```

```
        Width="140" Source="baqueta2.png" Stretch="Fill"/>
    </Canvas>
    <MediaElement x:Name="miPlayer" HorizontalAlignment="Left"
Height="50"
            Margin="381,597,0,0"            VerticalAlignment="Top"
            Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\CH.mp3"/>
    <MediaElement x:Name="miPlayer2" HorizontalAlignment="Left"
Height="50"
            Margin="445,597,0,0"            VerticalAlignment="Top"
            Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\SD0010.mp3"/>
    <MediaElement x:Name="miPlayer3" HorizontalAlignment="Left"
Height="50"
            Margin="509,597,0,0"            VerticalAlignment="Top"
            Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\SD0000.mp3"/>
    <MediaElement x:Name="miPlayer4" HorizontalAlignment="Left"
Height="50"
            Margin="573,597,0,0"            VerticalAlignment="Top"
            Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\BD0000.mp3"/>
    <MediaElement x:Name="miPlayer5" HorizontalAlignment="Left"
Height="50"
            Margin="636,597,0,0"            VerticalAlignment="Top"
            Width="50"
```

```
                LoadedBehavior="Manual"
                Source="C:\Sounds\CY0000.mp3"/>
        <Label x:Name="L1" Content="Estatus: " FontSize="25"
        FontWeight="Bold"
            Background="Bisque" HorizontalAlignment="Left"
            VerticalAlignment="Top"
            Margin="61,628,0,0"/>
        <Label x:Name="LEstatus" Content="Desconectado"
        FontSize="25"
            Foreground="#FFDA2828" HorizontalAlignment="Left"
            VerticalAlignment="Top"
            Margin="165,628,0,0"/>
</Grid>
```

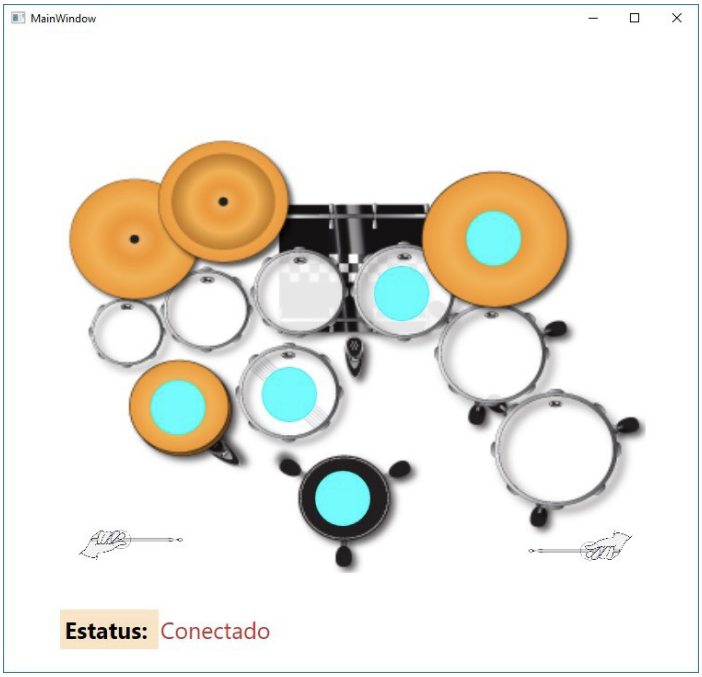


Figura 5.8

Código en C#

A la plantilla editada para Skeleton debe añadirse la biblioteca para el uso de DispatchTimer.

```
using System.Windows.Threading;
```

Para el desarrollo de esta aplicación se emplean dos estrategias. La primera de ellas es el manejo de colisiones. Cada vez que las baquetas (manoIzq, manoDer) colisionan con los círculos cian (**efecto1**, **efecto2**, **efecto3**, **efecto4**, **efecto5**) se activa el audio correspondiente. La segunda estrategia, está relacionada con la duración del efecto de sonido para lo cual se empleará el timer que indicará cuándo apagar el sonido activado en la colisión.

En la parte superior del programa antes del MainWindow se incluye la declaración de las variables que se emplean en varios métodos del programa. Se declaran cinco timers para cada uno de los efectos de sonido que se producirán. También se definen las estructuras **Imágenes** que almacenan la información requerida para el manejo de colisiones.

```
Point joint_Point = new Point(); //Permite obtener los datos del Joint
int iSkeletonID = 0; //Guarda el ID del Skeleton
DispatcherTimer timer1; //Timer para el sonido del efecto 1
DispatcherTimer timer2; //Timer para el sonido del efecto 2
DispatcherTimer timer3; //Timer para el sonido del efecto 3
DispatcherTimer timer4; //Timer para el sonido del efecto 4
DispatcherTimer timer5; //Timer para el sonido del efecto 5
// Estructura que almacenará la información del objeto
struct Imagenes
{
  public double dPosX;
  public double dPosY;
  public double dAncho;
  public double dAlto;
}
// Objetos a utilizar
Imagenes mano_Izq, mano_Der, efecto_1, efecto_2, efecto_3, efecto_4, efecto_5;
```

En el método **mainWindow** se realizarán las inicializaciones de los objetos y la configuración de los **timers**. Se toman las coordenadas de los elementos en el **Canvas** y sus dimensiones para asignarlas a los objetos del tipo **Imágenes**. El tiempo que se fija a los timers es un estimado de la duración del efecto de sonido. El método **mainWindow** queda de la siguiente forma:

```
public MainWindow()
{
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
        // Llenar estructura de efecto 1 para el manejo de colisiones
        efecto_1.dPosX =
(double)efecto1.GetValue(Canvas.LeftProperty);
        efecto_1.dPosY =
(double)efecto1.GetValue(Canvas.TopProperty);
        efecto_1.dAlto = efecto1.Height;
        efecto_1.dAncho = efecto1.Width;
        // Llenar estructura de efecto 2 para el manejo de colisiones
        efecto_2.dPosX =
(double)efecto2.GetValue(Canvas.LeftProperty);
        efecto_2.dPosY =
(double)efecto2.GetValue(Canvas.TopProperty);
        efecto_2.dAlto = efecto2.Height;
        efecto_2.dAncho = efecto2.Width;
        // Llenar estructura de efecto 3 para el manejo de colisiones
        efecto_3.dPosX =
(double)efecto3.GetValue(Canvas.LeftProperty);
        efecto_3.dPosY =
(double)efecto3.GetValue(Canvas.TopProperty);
```

```
efecto_3.dAlto = efecto1.Height;
efecto_3.dAncho = efecto1.Width;
// Llenar estructura de efecto 4 para el manejo de colisiones
efecto_4.dPosX =
(double)efecto4.GetValue(Canvas.LeftProperty);
efecto_4.dPosY =
(double)efecto4.GetValue(Canvas.TopProperty);
efecto_4.dAlto = efecto1.Height;
efecto_4.dAncho = efecto1.Width;
// Llenar estructura de efecto 5 para el manejo de colisiones
efecto_5.dPosX =
(double)efecto5.GetValue(Canvas.LeftProperty);
efecto_5.dPosY =
(double)efecto5.GetValue(Canvas.TopProperty);
efecto_5.dAlto = efecto1.Height;
efecto_5.dAncho = efecto1.Width;
// Llenar estructura de mano derecha para el manejo de colisiones
mano_Der.dPosX =
(double)manoDer.GetValue(Canvas.LeftProperty);
mano_Der.dPosY =
(double)manoDer.GetValue(Canvas.TopProperty);
mano_Der.dAlto = manoDer.Height;
mano_Der.dAncho = manoDer.Width;
// Llenar estructura de mano izquierda para el manejo de
colisiones
mano_Izq.dPosX =
(double)manoIzq.GetValue(Canvas.LeftProperty);
mano_Izq.dPosY =
```

```
            (double)manoIzq.GetValue(Canvas.TopProperty);
            mano_Izq.dAlto = manoIzq.Height;
            mano_Izq.dAncho = manoIzq.Width;
            // Configurar e iniciar el timer para cada efecto
            timer1 = new DispatcherTimer();
            timer1.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer1.Tick += new EventHandler(timer_Tick1);
            timer2 = new DispatcherTimer();
            timer2.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer2.Tick += new EventHandler(timer_Tick2);
            timer3 = new DispatcherTimer();
            timer3.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer3.Tick += new EventHandler(timer_Tick3);
            timer4 = new DispatcherTimer();
            timer4.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer4.Tick += new EventHandler(timer_Tick4);
            timer5 = new DispatcherTimer();
            timer5.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer5.Tick += new EventHandler(timer_Tick5);
    }
```

Dado que se modificó mainWindow se alterará el método usarSkeleton para que obtenga el Joint de ambas manos.

```
Joint joint1 = skeleton.Joints[JointType.HandRight];
        Joint joint2 = skeleton.Joints[JointType.HandLeft];
```

Posteriormente se realizan las instrucciones tanto para posicionar las baquetas de acuerdo a las manos y actualizar la información de la estructura de la imagen para poder checar la colisión.

Finalmente se ejecuta el método **checarColision** para verificar si hay colisión entre la mano y cada uno de los efectos. Cuando sucede se inicia el audio correspondiente y se habilita el timer para que después de cierto tiempo este se detenga y poder utilizarlo nuevamente.

```
private void usarSkeleton(Skeleton skeleton)
{ //Obtener coordenadas de la mano derecha y la mano izquierda
   Joint joint1 = skeleton.Joints[JointType.HandRight];
   Joint joint2 = skeleton.Joints[JointType.HandLeft];
   // Si el Joint está listo obtener las coordenadas
   if (joint1.TrackingState == JointTrackingState.Tracked)
   {
      //Coloca la baqueta de la mano derecha en
      //en la posición del Joint correspondiente. También ajusta
      //el contenido de la estructura para el manejo de la colisión
      joint_Point = this.SkeletonPointToScreen(joint1.Position);
      manoDer.SetValue(Canvas.LeftProperty, joint_Point.X);
      manoDer.SetValue(Canvas.TopProperty, joint_Point.Y);
      mano_Der.dPosX = joint_Point.X;
      mano_Der.dPosY = joint_Point.Y;
      //Coloca la baqueta de la mano izquierda en
      //en la posición del Joint correspondiente. También ajusta
      //el contenido de la estructura para el manejo de la colisión
      joint_Point = this.SkeletonPointToScreen(joint2.Position);
      manoIzq.SetValue(Canvas.LeftProperty, joint_Point.X);
      manoIzq.SetValue(Canvas.TopProperty, joint_Point.Y);
      mano_Izq.dPosX = joint_Point.X;
      mano_Izq.dPosY = joint_Point.Y;
```

```
//Verifica si hay colisión entre la baqueta de la mano derecha
//y cada uno de los efectos. En caso de haber colisión se
//activa el sonido correspondiente y habilita su timer.
if (checarColision(mano_Der, efecto_1))
{
  miPlayer.Play();
  timer1.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_2))
{
  miPlayer2.Play();
  timer2.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_3))
{
  miPlayer3.Play();
  timer3.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_4))
{
  miPlayer4.Play();
  timer4.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_5))
{
```

```
  miPlayer5.Play();
  timer5.IsEnabled = true;
}
//Verifica si hay colisión entre la baqueta de la mano izquierda
//y cada uno de los efectos. En caso de haber colisión se
//activa el sonido correspondiente y habilita su timer.
if (checarColision(mano_Izq, efecto_1))
{
  miPlayer.Play();
  timer1.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_2))
{
  miPlayer2.Play();
  timer2.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_3))
{
  miPlayer3.Play();
  timer3.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_4))
{
  miPlayer4.Play();
  timer4.IsEnabled = true;
```

```
        }
        if (checarColision(mano_Izq, efecto_5))
        {
          miPlayer5.Play();
          timer5.IsEnabled = true;
        }
      }
    }
```

Para el manejo de colisiones se agrega el método **checarColision** el cual recibe la información para determinar si la imagen img1 está colisionando con la imagen img2.

```
  private bool checarColision(Imagenes img1, Imagenes img2)
  {
    if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de img2
      return false;
    if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de img2
      return false;
    if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha img2
      return false;
    if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
      return false;
    return true;
  }
```

Cada método timer_Tick realiza un **Stop** al sonido correspondiente y deshabilita el timer relacionado para que dicho audio pueda activarse nuevamente si sucede una colisión con el mismo objeto.

```
// Reproducción del efecto de sonido 1
void timer_Tick1(object sender, EventArgs e)
{
   miPlayer.Stop();
   timer1.IsEnabled = false;
}
// Reproducción del efecto de sonido 2
void timer_Tick2(object sender, EventArgs e)
{
   miPlayer2.Stop();
   timer2.IsEnabled = false;
}
// Reproducción del efecto de sonido 3
void timer_Tick3(object sender, EventArgs e)
{
   miPlayer3.Stop();
   timer3.IsEnabled = false;
}
// Reproducción del efecto de sonido 4
void timer_Tick4(object sender, EventArgs e)
{
   miPlayer4.Stop();
   timer4.IsEnabled = false;
}
// Reproducción del efecto de sonido 5
```

```
void timer_Tick5(object sender, EventArgs e)
{
   miPlayer5.Stop();
   timer5.IsEnabled = false;
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando se mueven las baquetas y se hace contacto con los círculos color cian se reproduce un sonido específico, así se da el efecto de tocar la batería.

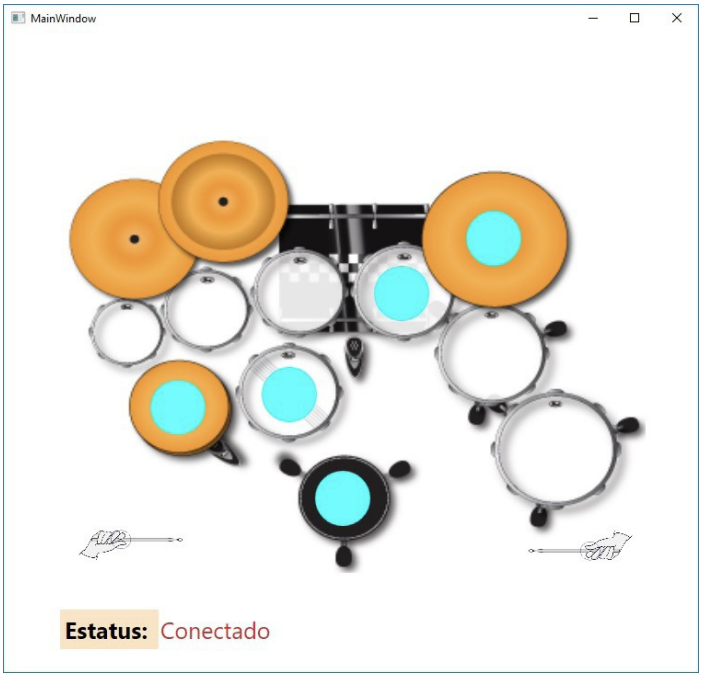


Figura 5.9

5.6 Ejemplo: tocar batería (Kinect V2)

Este ejemplo muestra la forma de combinar el manejo de colisiones, el uso de un timer y el empleo de la información de los Joints que proporciona el Kinect para desarrollar una aplicación que permite simular tocar una batería. La aplicación muestra la imagen de este instrumento en la que se han resaltado con círculos color cian, las secciones que pueden producir un sonido.

Requerimientos

- Proyecto **WPF**.
- Plantilla V2 (configuración Skeleton).
- Archivos de audio en formato MP3.
- Imagen de batería y baquetas en formato PNG.

Código en XAML

En el archivo XAML se incluye una imagen de fondo que muestra una batería, cuatro elipses para indicar los elementos de la batería que cuentan con un sonido: efecto1, efecto2, efecto3, efecto4 y efecto5 y las baquetas que son representadas por las imágenes manoIzq y manoDer. Para crear los efectos de sonido se añade un **MediaElement** para cada uno de los efectos en los que se indica la trayectoria dentro del disco de la computadora donde se localiza el audio; estas trayectorias deben ser ajustadas a la computadora en la que se ejecutará la aplicación. Los elementos restantes son usados para mostrar el estatus del Kinect.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
 Grid.Row="1" Background="Azure"
  HorizontalAlignment="Center">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480"
        Width="640" Source="bateria.png" Stretch="Fill"/>
  <Ellipse x:Name="efecto1" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="102" Canvas.Top="270"/>
  <Ellipse x:Name="efecto2" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="223" Canvas.Top="256"/>
  <Ellipse x:Name="efecto3" Fill="Aqua" Height="60" Width="60"
      Canvas.Left="281" Canvas.Top="369"/>
  <Ellipse x:Name="efecto4" Fill="Aqua" Height="60" Width="60"
```

```
        Canvas.Left="345" Canvas.Top="146"/>
 <Ellipse x:Name="efecto5" Fill="Aqua" Height="60" Width="60"
        Canvas.Left="445" Canvas.Top="86"/>
 <Image Name="manoIzq" Canvas.Left="10" Canvas.Top="430"
Height="40"
       Width="140" Source="baqueta1.png" Stretch="Fill"/>
 <Image Name="manoDer" Canvas.Left="500" Canvas.Top="430"
Height="40"
       Width="140" Source="baqueta2.png" Stretch="Fill"/>
 </Canvas>
 <MediaElement x:Name="miPlayer" HorizontalAlignment="Left"
Height="50"
            Margin="381,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual" Source="C:\Sounds\CH.mp3"/>
 <MediaElement x:Name="miPlayer2" HorizontalAlignment="Left"
Height="50"
            Margin="445,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\SD0010.mp3"/>
 <MediaElement x:Name="miPlayer3" HorizontalAlignment="Left"
Height="50"
            Margin="509,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\SD0000.mp3"/>
 <MediaElement x:Name="miPlayer4" HorizontalAlignment="Left"
Height="50"
            Margin="573,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual"
```

```
            Source="C:\Sounds\BD0000.mp3"/>
    <MediaElement x:Name="miPlayer5" HorizontalAlignment="Left"
Height="50"
            Margin="636,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual"
            Source="C:\Sounds\CY0000.mp3"/>
    <Label x:Name="L1" Content="Estatus: " FontSize="25"
FontWeight="Bold"
        Background="Bisque" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="61,628,0,0"/>
    <Label x:Name="LEstatus" Content="Desconectado" FontSize="25"
        Foreground="#FFDA2828" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="165,628,0,0"/>
</Grid>
```

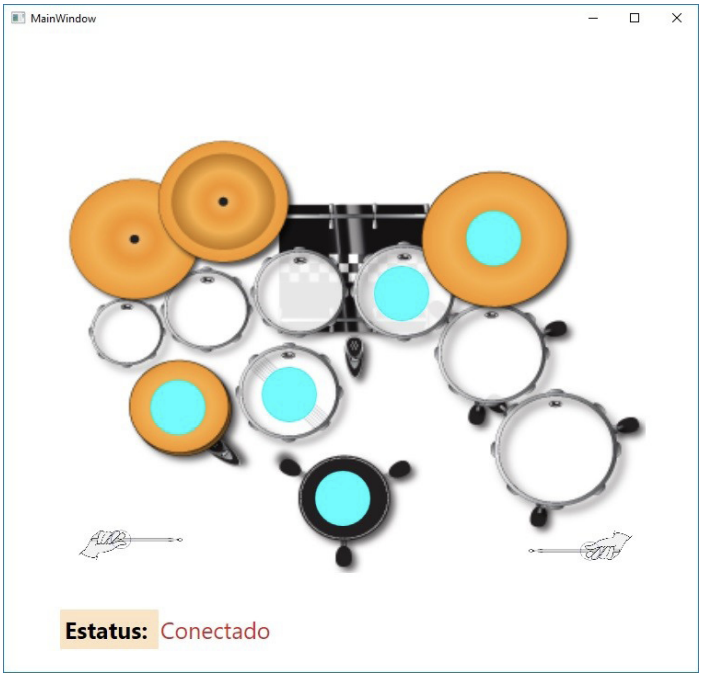


Figura 5.10

Código en C#

A la plantilla editada para Skeleton se debe añadir la biblioteca para el uso de **DispatchTimer**.

```
using System.Windows.Threading;
```

Para el desarrollo de esta aplicación se emplean dos estrategias. La primera de ellas es el manejo de colisiones. Cada vez que las baquetas (**manoIzq**, **manoDer**) colisionan con los círculos cian (**efecto1**, **efecto2**, **efecto3**, **efecto4**, **efecto5**) se activa el audio correspondiente. La segunda estrategia, está relacionada con la duración del efecto de sonido para lo cual se empleará el timer que indicará cuando apagar el sonido activado en la colisión.

En la parte superior del programa antes del MainWindow se incluye la declaración de las variables que se emplean en varios métodos del programa. Se declaran cinco timers para cada uno de los efectos de sonido que se producirán. También se define las estructuras **Imágenes** que almacena la información requerida para el manejo de colisiones.

```
private BodyFrameReader miBodyFrameReader; // FrameReader para recibir datos
private Body[] bodies = null; // Arreglo para recibir datos de Body
Point joint_Point = new Point(); // Permite obtener los datos de un Joint
DispatcherTimer timer1; //Timer para el sonido del efecto 1
DispatcherTimer timer2; //Timer para el sonido del efecto 2
DispatcherTimer timer3; //Timer para el sonido del efecto 3
DispatcherTimer timer4; //Timer para el sonido del efecto 4
DispatcherTimer timer5; //Timer para el sonido del efecto 5
// Estructura que almacenará la información de los objetos
struct Imagenes
{
  public double dPosX;
  public double dPosY;
  public double dAncho;
```

```
    public double dAlto;
}
// Objetos que pueden colisionar
Imagenes mano_Izq, mano_Der, efecto_1, efecto_2, efecto_3, efecto_4, efecto_5;
```

En el método **mainWindow** se realizarán las inicializaciones de los objetos y la configuración de los **timers**. Se toman las coordenadas de los elementos en el **Canvas** y sus dimensiones para asignarlas a los objetos del tipo **Imágenes**. El tiempo que se asigna a los timers es un estimado de la duración del efecto de sonido. El método **mainWindow** queda de la siguiente forma:

```
public MainWindow()
{
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
        // Llenar estructura de efecto 1 para el manejo de colisiones
        efecto_1.dPosX = (double)efecto1.GetValue(Canvas.LeftProperty);
        efecto_1.dPosY = (double)efecto1.GetValue(Canvas.TopProperty);
        efecto_1.dAlto = efecto1.Height;
        efecto_1.dAncho = efecto1.Width;
        // Llenar estructura de efecto 2 para el manejo de colisiones
        efecto_2.dPosX = (double)efecto2.GetValue(Canvas.LeftProperty);
        efecto_2.dPosY = (double)efecto2.GetValue(Canvas.TopProperty);
        efecto_2.dAlto = efecto2.Height;
        efecto_2.dAncho = efecto2.Width;
        // Llenar estructura de efecto 3 para el manejo de colisiones
```

```
efecto_3.dPosX =
(double)efecto3.GetValue(Canvas.LeftProperty);
efecto_3.dPosY = (double)efecto3.GetValue(Canvas.TopProperty);
efecto_3.dAlto = efecto1.Height;
efecto_3.dAncho = efecto1.Width;
// Llenar estructura de efecto 4 para el manejo de colisiones
efecto_4.dPosX =
(double)efecto4.GetValue(Canvas.LeftProperty);
efecto_4.dPosY = (double)efecto4.GetValue(Canvas.TopProperty);
efecto_4.dAlto = efecto1.Height;
efecto_4.dAncho = efecto1.Width;
// Llenar estructura de efecto 5 para el manejo de colisiones
efecto_5.dPosX =
(double)efecto5.GetValue(Canvas.LeftProperty);
efecto_5.dPosY = (double)efecto5.GetValue(Canvas.TopProperty);
efecto_5.dAlto = efecto1.Height;
efecto_5.dAncho = efecto1.Width;
// Llenar estructura de mano derecha para el manejo de colisiones
mano_Der.dPosX =
(double)manoDer.GetValue(Canvas.LeftProperty);
mano_Der.dPosY =
(double)manoDer.GetValue(Canvas.TopProperty);
mano_Der.dAlto = manoDer.Height;
mano_Der.dAncho = manoDer.Width;
// Llenar estructura de mano izquierda para el manejo de colisiones
mano_Izq.dPosX =
(double)manoIzq.GetValue(Canvas.LeftProperty);
```

```
            mano_Izq.dPosY =
            (double)manoIzq.GetValue(Canvas.TopProperty);
            mano_Izq.dAlto = manoIzq.Height;
            mano_Izq.dAncho = manoIzq.Width;
            // Configurar e iniciar el timer para cada efecto
            timer1 = new DispatcherTimer();
            timer1.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer1.Tick += new EventHandler(timer_Tick1);
            timer2 = new DispatcherTimer();
            timer2.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer2.Tick += new EventHandler(timer_Tick2);
            timer3 = new DispatcherTimer();
            timer3.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer3.Tick += new EventHandler(timer_Tick3);
            timer4 = new DispatcherTimer();
            timer4.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer4.Tick += new EventHandler(timer_Tick4);
            timer5 = new DispatcherTimer();
            timer5.Interval = new TimeSpan(0, 0, 0, 0, 300);
            timer5.Tick += new EventHandler(timer_Tick5);
    }
```

Ya que se modificó **mainWindow** se cambiará el método usarBody para que obtenga el Joint de ambas manos.

```
Joint joint1 = miBody.Joints[JointType.HandRight];
Joint joint2 = miBody.Joints[JointType.HandLeft];
```

Posteriormente se realizan las instrucciones para acomodar las baquetas de acuerdo a la posición de las manos y para actualizar la información de la estructura de la imagen para

poder checar la colisión.

Finalmente se ejecuta el método **checarColision** para verificar si hay colisión entre la mano y cada uno de los efectos. Cuando colisiona se inicia el audio correspondiente y se habilita el timer para que después de cierto tiempo se detenga el audio con el fin de volver a utilizarlo.

```
private void usarBody(Body miBody)
{
  //Obtener coordenadas de la mano derecha y la mano izquierda
  Joint joint1 = miBody.Joints[JointType.HandRight];
  Joint joint2 = miBody.Joints[JointType.HandLeft];
  // Si el Joint está listo obtener las coordenadas
  if (joint1.TrackingState == TrackingState.Tracked)
  {
    //Coloca la baqueta de la mano derecha en
    //en la posición del Joint correspondiente. También ajusta
    //el contenido de la estructura para el manejo de la colisión
    joint_Point = this.BodyPointToScreen(joint1);
    manoDer.SetValue(Canvas.LeftProperty, joint_Point.X);
    manoDer.SetValue(Canvas.TopProperty, joint_Point.Y);
    mano_Der.dPosX = joint_Point.X;
    mano_Der.dPosY = joint_Point.Y;
    //Coloca la baqueta de la mano izquierda en
    //en la posición del Joint correspondiente. También ajusta
    //el contenido de la estructura para el manejo de la colisión
    joint_Point = this.BodyPointToScreen(joint2);
    manoIzq.SetValue(Canvas.LeftProperty, joint_Point.X);
    manoIzq.SetValue(Canvas.TopProperty, joint_Point.Y);
```

```
mano_Izq.dPosX = joint_Point.X;
mano_Izq.dPosY = joint_Point.Y;
//Verifica si hay colisión entre la baqueta de la mano derecha
//y cada uno de los efectos. En caso de haber colisión se
//activa el sonido correspondiente y habilita su timer.
if (checarColision(mano_Der, efecto_1))
{
  miPlayer.Play();
  timer1.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_2))
{
  miPlayer2.Play();
  timer2.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_3))
{
  miPlayer3.Play();
  timer3.IsEnabled = true;
}
if (checarColision(mano_Der, efecto_4))
{
  miPlayer4.Play();
  timer4.IsEnabled = true;
}
```

```
if (checarColision(mano_Der, efecto_5))
{
  miPlayer5.Play();
  timer5.IsEnabled = true;
}
//Verifica si hay colisión entre la baqueta de la mano izquierda
//y cada uno de los efectos. En caso de haber colisión se
//activa el sonido correspondiente y habilita su timer.
if (checarColision(mano_Izq, efecto_1))
{
  miPlayer.Play();
  timer1.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_2))
{
  miPlayer2.Play();
  timer2.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_3))
{
  miPlayer3.Play();
  timer3.IsEnabled = true;
}
if (checarColision(mano_Izq, efecto_4))
{
```

```
            miPlayer4.Play();
            timer4.IsEnabled = true;
        }
        if (checarColision(mano_Izq, efecto_5))
        {
            miPlayer5.Play();
            timer5.IsEnabled = true;
        }
    }
}
```

Para el manejo de colisiones se agrega el método **checarColision** el cual recibe la información para determinar si la imagen img1 está colisionando con la imagen img2.

```
private bool checarColision(Imagenes img1, Imagenes img2)
{
  if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de img2
    return false;
  if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de img2
    return false;
  if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha img2
    return false;
  if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
    return false;
  return true;
}
```

Cada método timer_Tick realiza un **Stop** al sonido correspondiente y deshabilita el timer relacionado para que dicho audio pueda activarse nuevamente si sucede una colisión con el mismo objeto.

```
// Reproducción del efecto de sonido 1
void timer_Tick1(object sender, EventArgs e)
{
   miPlayer.Stop();
   timer1.IsEnabled = false;
}
// Reproducción del efecto de sonido 2
void timer_Tick2(object sender, EventArgs e)
{
   miPlayer2.Stop();
   timer2.IsEnabled = false;
}
// Reproducción del efecto de sonido 3
void timer_Tick3(object sender, EventArgs e)
{
   miPlayer3.Stop();
   timer3.IsEnabled = false;
}
// Reproducción del efecto de sonido 4
void timer_Tick4(object sender, EventArgs e)
{
   miPlayer4.Stop();
   timer4.IsEnabled = false;
```

```
}
// Reproducción del efecto de sonido 5
void timer_Tick5(object sender, EventArgs e)
{
   miPlayer5.Stop();
   timer5.IsEnabled = false;
}
```

Ejecución

La siguiente imagen muestra cómo se vería la ejecución del proyecto. Cuando se mueven las baquetas y se hace contacto con los círculos color **cian** se reproduce un sonido específico, así se da el efecto como si se tocara la batería.

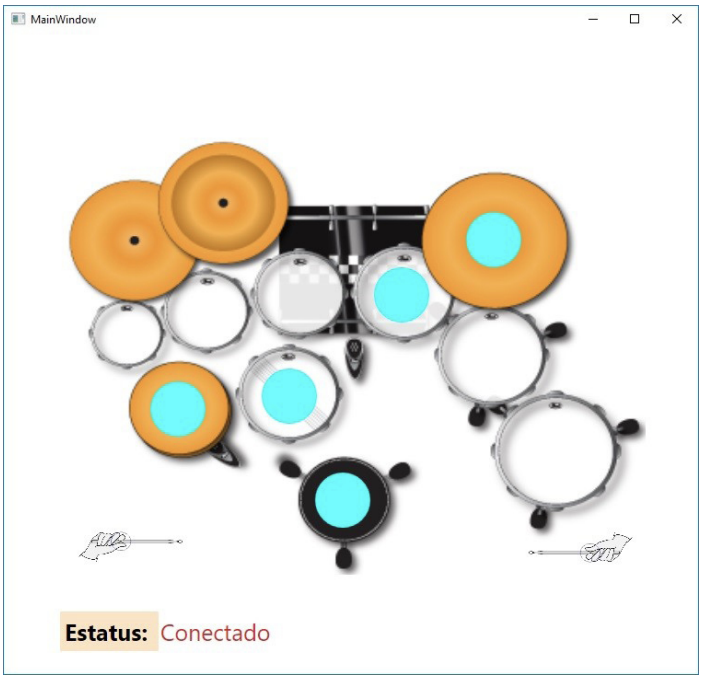


Figura 5.11

Capítulo 6. Crear animaciones que responden al movimiento de la persona

Hasta ahora se ha repasado el proceso de la información que brindan los **Joints**, con el fin de interactuar con las aplicaciones, sin embargo, en la mayoría de los ejemplos se han realizado interacciones con elementos estáticos, aquellos que se encuentran en la ventana de las aplicaciones como **Ellipses**, **Imágenes**, etc., estos no presentan animación alguna.

En este capítulo se harán proyectos que permitan una interacción más dinámica con sus elementos. Se crearán animaciones sencillas que permitan lograr lo antes mencionado, por ejemplo, el movimiento del fondo de la ventana y respuestas a interacciones con el movimiento de diferentes partes del cuerpo. Estas acciones se implementarán mediante el uso de **DispatchTimer**, los cuales ya se han trabajado anteriormente. Estos permiten ejecutar rutinas de forma periódica mediante códigos sencillos, los cuales pueden manipular las propiedades de los elementos en ventana para crear la sensación de movimiento y aprovechar para generar ese movimiento como respuesta a interacciones con diferentes partes del cuerpo.

6.1 Ejemplo: nave espacial (Kinect V1 y V2)

Se presenta una aplicación sencilla en la cual se busca tener el ángulo correcto para realizar un disparo, se puede asemejar a un tiro con arco. Durante la ejecución, la aplicación provee información sobre el ángulo del brazo derecho respecto a la horizontal formada por el hombro derecho, tal como se muestra en la siguiente figura:

Requerimientos

- Proyecto **WPF.**
- Plantilla V1 o V2.
- Imágenes: fondo y nave.

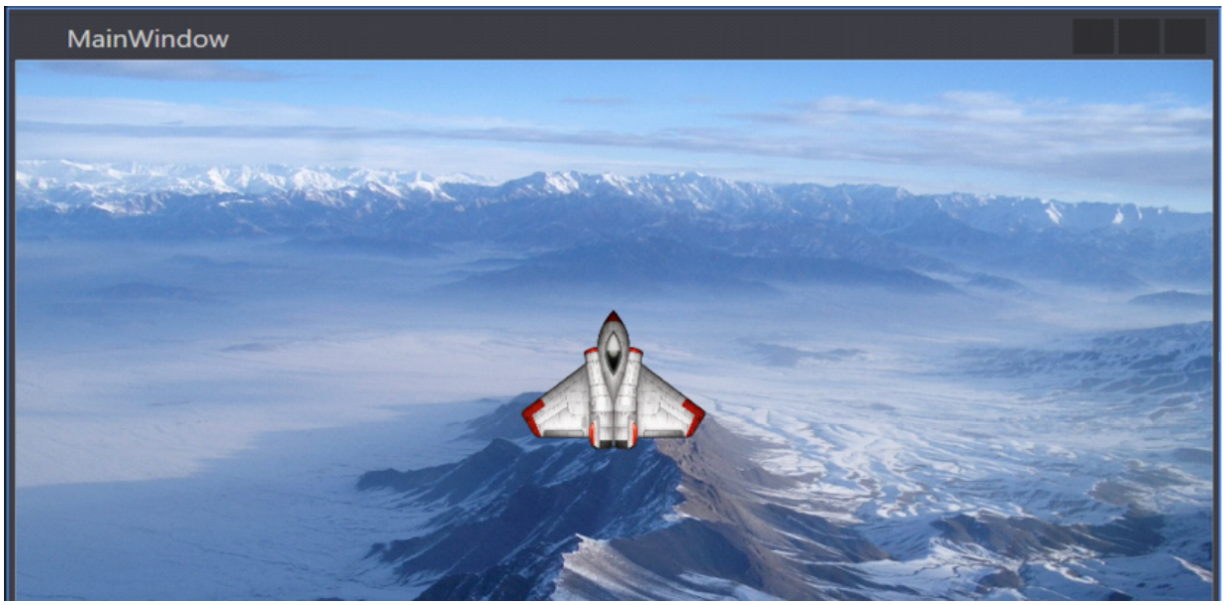


Figura 6.1

Código en XAML

Los elementos que se usan en el archivo XAML son: dos **Imágenes**, el fondo y la nave *spaceship*, y la propiedad de rotación o ángulo. Ambos localizados en el mismo **Canvas**. Los elementos restantes son **Labels** para generar fondos e indicadores. Las dimensiones de la ventana son **Height="350"** y **Width="525"**. El código es el siguiente:

```
<Grid>
  <Image x:Name="background" Source="Resources/sky.jpg"
 Width="525" Height="425"/>
  <Image x:Name="spaceship" Source="Resources/spaceship.png"
 Width="80"
          Height="80">
    <Image.RenderTransform>
          <TransformGroup>
            <RotateTransform Angle="0"
            x:Name="naveRotation"/>
          </TransformGroup>
    </Image.RenderTransform>
```

```
    </Image>
    <Label x:Name="L1" Content="Estatus: " Background="Bisque" FontSize="15"
        HorizontalAlignment="Left" Margin="17,276,0,0" VerticalAlignment="Top"/>
    <Label x:Name="LEstatus" Content="Desconectado" FontSize="15" Background="Gray"
        Foreground="#FFDA2828" HorizontalAlignment="Left" Margin="94,276,0,0"
        VerticalAlignment="Top"/>
    </Grid>
</Window>
```

Código en C#

Se crea un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla para profundidad, esta será editada para poder implementar la aplicación antes descrita. Una vez creado el proyecto, se agregan algunas bibliotecas.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

Las variables que se utilizan son las siguientes.

```
double degrees = 90;
MultiSourceFrameReader FrameReader;
IList<Body> BodyCn;
DispatcherTimer timer;
double spaceX, spaceY;
bool up;
```

Posteriormente, se implementa el Timer_tick encargado de revisar coordenadas de

imágenes y actualización de datos cada cierto tiempo definido.

```
public MainWindow() {
        InitializeComponent();
        // Realizar configuraciónes
        Kinect_Config();
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 1);
        timer.Tick += new EventHandler(Timer_Tick);
        timer.IsEnabled = true;
}
```

En el apartado de la identificación de los joints se declaran las siguientes instrucciones. Estas indican que hay que guardar como "handRight" a la mano derecha detectada por el Kinect, además de escalar la posición de la mano, gracias a la instrucción del "DepthSpacePoint". El DepthSpace acomoda a las coordenadas.

```
Joint handRight = body.Joints[JointType.HandRight];
                        Joint handLeft = body.Joints[JointType.HandLeft];
DepthSpacePoint dspRight =
miKinect.CoordinateMapper.MapCameraPointToDepthSpace(handRight.
Position);
```

Por consiguiente, se realizarán modificaciones en la función del Timer_tick la cual se encargara de actualizar y recopilar datos cada cierto tiempo. En este caso, serían los datos de la nave *spaceship*, en donde se usaron variables llamadas "SpaceX", para la posición respecto del eje X en el canvas, y "SpaceY", para la posición respecto del eje Y, que servirán para guardar las coordenadas respecto al canvas de la imagen.

Además se incluirá una condición, si la nave se encuentra en una posición mayor o menor de 500 y 15 esta debe regresar a las posiciones (150,150) del canvas, siendo (X,Y) las posiciones y también si se encuentran en un lugar mayor o menor en el eje Y de 300 o 15.

```
private void Timer_Tick(object sender, EventArgs e) {
   spaceX = (double)spaceship.GetValue(Canvas.LeftProperty);
```

```
        spaceY = (double)spaceship.GetValue(Canvas.TopProperty);
        if(spaceX > 510 || spaceX < 15 || spaceY < 15 || spaceY>300)
        {
            Canvas.SetLeft(spaceship, 150);
            Canvas.SetTop(spaceship, 150);
        }
    }
```

Ahora se declaran las instrucciones para el control de la nave, así como su ángulo de rotación para generar el efecto de estar dando vuelta.

Ángulo de la imagen

Se indicara que si la mano derecha se encuentra en una posición mayor a 375, la imagen *spaceship* rotara 90° los cuales están almacenados en la variable *degrees*, en caso de que sea menor a 150, rotara hacia la izquierda en donde se restaran 90° a su actual rotación. Cabe recalcar que "naveRotacion.Angle" almacena el dato del ángulo en que se encuentra la imagen, la cual se modifica sumando o restando ángulo a la imagen, por medio de grados y no radianes.

Si se requiere retroceder o avanzar se debe de checar la propiedad de la nave, la cual es naveRotation.Angle, para verificar e indicar la forma en que debe de moverse para retroceder o avanzar, por lo cual se indicaron cuatro condiciones para cada movimiento, los cuales son Avanzar o Retroceder.

Avanzar y/o Retroceder

- Si se requiere retroceder o avanzar pero la nave se encuentra en un ángulo de 0° este debe de avanzar 10 pixeles o retroceder 10 pixeles, dependiendo del caso, en el eje X(LeftProperty).
- Si se requiere avanzar o retroceder pero la nave se encuentra en un ángulo de 90° este debe de avanzar o retrocedes 10 pixeles, dependiendo del caso, en el eje Y(TopProperty).
- Si se requiere avanzar o retroceder pero la nave se encuentra en un ángulo de 180° este debe de avanzar o retroceder 10 pixeles, dependiendo del caso, en el eje X(LeftProperty).

- Si se requiere avanzar o retroceder pero la nave se encuentra en un ángulo de 180° este debe de avanzar o retrocedes 10 pixeles, dependiendo del caso, en el eje Y(TopProperty).

```
if(dspRight.X >375 )
        {
            naveRotation.Angle = -degrees;
        }
        else if ( dspRight.X < 150)
        {
            naveRotation.Angle = +degrees;
        }
        //Este es para avanzar
        if ( dspRight.Y<50 && (naveRotation.Angle ==0))
        {
            Canvas.SetLeft(spaceship, spaceX + 10);
        }
        else if(dspRight.Y < 50 && (naveRotation.Angle == 90))
        {
            Canvas.SetTop(spaceship, spaceY - 10);
        }
        else if(dspRight.Y<50 &&(naveRotation.Angle ==180))
        {
            Canvas.SetLeft(spaceship, spaceX - 10);
        }
        else if(dspRight.Y<50 &&(naveRotation.Angle==270))
        {
```

```
        Canvas.SetTop(spaceship, spaceY - 10);
    }
    //Este es para retroceder
    if (dspRight.Y > 300 && (naveRotation.Angle == 0))
    {
        Canvas.SetLeft(spaceship, spaceX - 10);
    }
    else if (dspRight.Y > 300 && (naveRotation.Angle == 90))
    {
        Canvas.SetTop(spaceship, spaceY + 10);
    }
    else if (dspRight.Y > 300 && (naveRotation.Angle == 180))
    {
        Canvas.SetLeft(spaceship, spaceX + 10);
    }
    else if (dspRight.Y > 300 && (naveRotation.Angle == 270))
    {
        Canvas.SetTop(spaceship, spaceY + 10);
    }
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando la mano derecha se encuentre en la parte derecha del canvas, esta rotara 90º hacia la derecha y cuando se encuentre en la izquierda, esta rotara 90° hacia la izquierda. Si la mano se encuentra hacia arriba, la nave empezara a moverse hacia delante pero si se encuentra abajo esta empezara a retroceder.

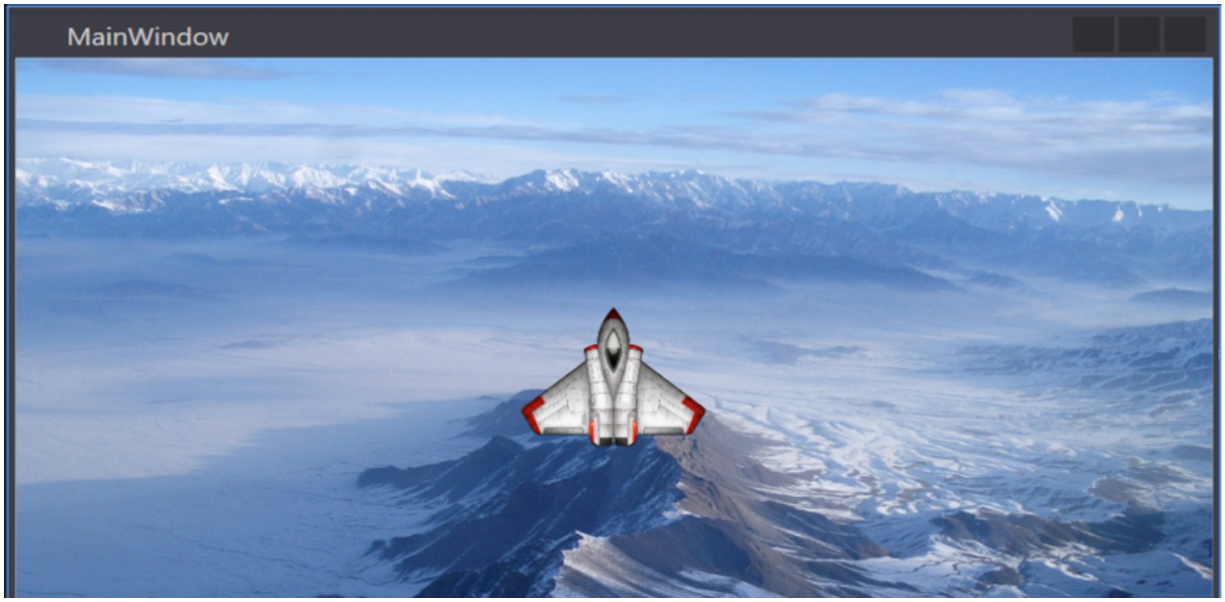


Figura 6.2

6.2 Ejemplo: golpear un saco de box (Kinect V1 y V2)

Es una aplicación sencilla en la cual se busca realizar una interacción, a través de movimientos del cuerpo, para generar movimiento en el saco de box, y así ,simular el boxeo.

Requerimientos

- Proyecto **WPF**.
- Plantilla V1 o V2.
- Imágenes: fondo del programa y saco de box.

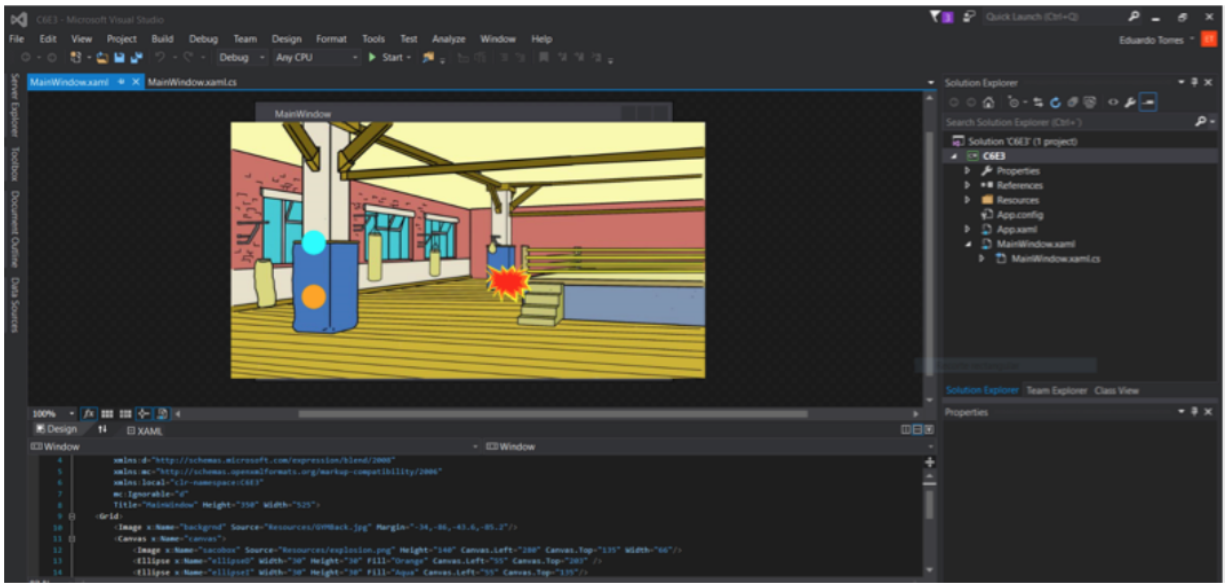


Figura 6.3

Código en XAML

Los elementos que se usan en el archivo XAML son dos **Imágenes**: el fondo y el saco de box (sacobox). Además, se incluyen dos elipses: "ellipseD" y "ellipseI", se pusieron de diferente color para diferenciarlos, sus medidas son Widht="30" y Heigth="30". Las dimensiones de la ventana son **Height="350"** y **Width="525"**. Así como dos Labels, los cuales indican el estado del Kinect.

El código es el siguiente:

```
Title="MainWindow" Height="350" Width="525">
<Grid>
  <Image x:Name="backgrnd" Source="Resources/GYMBack.jpg"
 Width="600"
        Height="320"/>
  <Canvas x:Name="canvas">
    <Image x:Name="sacobox" Source="Resources/explosion.png"
    Height="140"
      Canvas.Left="338" Canvas.Top="81" Width="66"/>
    <Ellipse x:Name="ellipseD" Width="30" Height="30"
    Fill="Orange"
```

```
            Canvas.Left="55" Canvas.Top="203" />
        <Ellipse x:Name="ellipseI" Width="30" Height="30" Fill="Aqua"
            Canvas.Left="55" Canvas.Top="135"/>
    </Canvas>
</Grid>
```

Código en C#

Se crea un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla para profundidad y se usa la plantilla para Kinect V1 o V2, esta será editada para poder implementar la aplicación antes descrita. Una vez creado el proyecto se agregan unas bibliotecas.

```
using Microsoft.Kinect;
using System.IO;
using System.Threading;
using System.Windows.Threading;
using System.Media;
```

Las variables que se utilizan son las siguientes.

```
DispatcherTimer timer;
double ellipseDx, ellipseDy , ellipseIx, ellipseIy, sacoBx,sacoBy;
SoundPlayer notaA = new SoundPlayer(Properties.Resources.Punch);
BitmapImage bmGolpe = new BitmapImage(new Uri(@"Resources/explosion.png",
                    UriKind.RelativeOrAbsolute));
BitmapImage bmSaco = new BitmapImage(new Uri(@"Resources/saco.png",
                    UriKind.RelativeOrAbsolute));
```

Posteriormente, se implementa el Timer_tick, el cual será el encargado de revisar coordenadas de imágenes y actualización de datos cada cierto tiempo definido.

```
public MainWindow() {
        InitializeComponent();
        // Realizar configuraciónes
        Kinect_Config();
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 1);
        timer.Tick += new EventHandler(Timer_Tick);
        timer.IsEnabled = true;
}
```

Una vez declarado lo anterior, en el apartado de la identificación de los joints, se declaran las instrucciones que indican que se guarde como "handRight" a la mano derecha y como "handLeft" a la mano izquierda, las cuales son detectadas del body por el Kinect. Además de escalar la posición de la mano, la instrucción del "SkeletonPointToScreen" o “BodyPointToScreen” (dependiendo de la versión de Kinect) acomodará a las coordenadas del joint detectado.

También se incluyen el posicionamiento de dos elipses las cuales se declaran variables para guardar sus coordenadas . Posteriormente se incluirán las elipses en el XAML.

```
Joint handRight = body.Joints[JointType.HandRight];
Joint handLeft = body.Joints[JointType.HandLeft];
---------- Kinect V1-------
        Point dspRight, dspLeft;
        dspRight = this.SkeletonPointToScreen(joint1.Position);
        dspLeft = this.SkeletonPointToScreen(joint2.Position);
------------Kinect V2 -------
        Point dspRight, dspLeft;
        dspRight = this.BodyPointToScreen(joint1);
        dspLeft = this.BodyPointToScreen(joint2);
------------------------------
```

```
Canvas.SetTop(ellipseD, dspRight.Y);
Canvas.SetLeft(ellipseD, dspRight.X);
Canvas.SetLeft(ellipseI, dspLeft.X );
Canvas.SetTop(ellipseI, dspLeft.Y);
```

Se realizarán modificaciones en la función del Timer_tick la cual se encargará de actualizar y recopilar datos cada cierto tiempo. En este caso, serán recolectados los datos de las coordenadas de las dos elipses y del saco de box, los cuales están almacenados en las variables ellipseDx, ellipseDy, ellipseIx, ellipseIy, sacoBx y sacoBy. Se identifica con "D" la que va en la mano derecha y "I" la que va en la mano izquierda, además del sufijo "x" o "y" para verificar si guardan propiedades de localización TopProperty o LeftProperty. Asimismo, las coordenadas del saco de box serán guardadas en las variables sacoBx sacoBy respetando la posición en el eje "x" y "y".

Se incluirán condiciones las cuales revisaran choques. La primera de ellas corroborará si hay un choque con la ellipseD, posicionada en la mano derecha. En caso de que la condición sea correcta, la imagen se cambiara por una llamada bmGolpe, la cual indicará y creará el efecto de choque, además de reproducir un sonido.

La segunda condición realiza lo mismo pero ahora respecto a la ellipseI ,la cual está posicionada en la mano izquierda, cuando esta choque con el saco de box, cambiará su imagen y se reproducirá un sonido.

En dado caso que ninguna de estas condiciones se ejecute, la imagen regresara a la anterior.

```
ellipseDx = (double)ellipseD.GetValue(Canvas.LeftProperty);
ellipseDy = (double)ellipseD.GetValue(Canvas.TopProperty);
ellipseIx = (double)ellipseI.GetValue(Canvas.LeftProperty);
ellipseIy = (double)ellipseI.GetValue(Canvas.TopProperty);
sacoBx = (double)sacobox.GetValue(Canvas.LeftProperty);
sacoBy = (double)sacobox.GetValue(Canvas.TopProperty);
//Choque por los lados.
if ((ellipseDx > sacoBx - 50 && ellipseDx < sacoBx + 10 ) ||
   (ellipseDy > sacoBy + 10 && ellipseDy < sacoBy - 10)) {
            sacobox.Source = bmGolpe;
```

```
            notaA.Play();
}
   else if((ellipseIx > sacoBx - 50 && ellipseIx < sacoBx + 10) &&
       (ellipseIy < sacoBy + 10 && ellipseDy > sacoBy - 140))
{
            sacobox.Source = bmGolpe;
            notaA.Play();
} else {
            sacobox.Source = bmSaco;
}
```

Aquí se puede observar la diferencia cuando chocan y cuando no.

Figura 6.4

Ahora que se han declarado las instrucciones para el control del programa, hay que realizar las instrucciones en código XAML para la parte gráfica y que quede como las imágenes anteriores.

Ejecución

La siguiente imagen muestra cómo se ve el producto final de la aplicación.

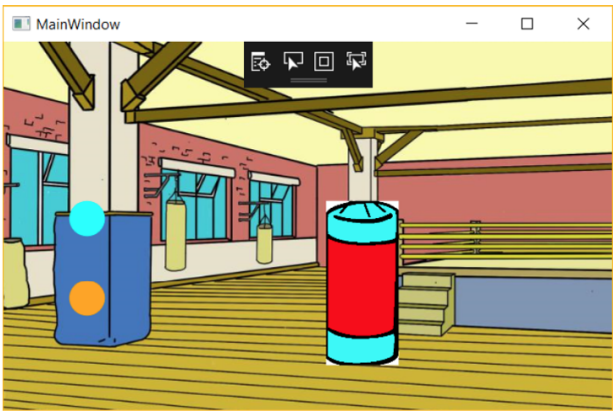


Figura 6.5

6.3 Ejemplo: tiro con arco (Kinect V1)

Este ejemplo es una aplicación sencilla en la cual se busca tener el ángulo correcto para realizar un disparo, se puede asemejar a un tiro con arco. Durante la ejecución, la aplicación provee información sobre el ángulo del brazo derecho respecto a la horizontal formada por el hombro derecho, tal como se muestra en la figura:

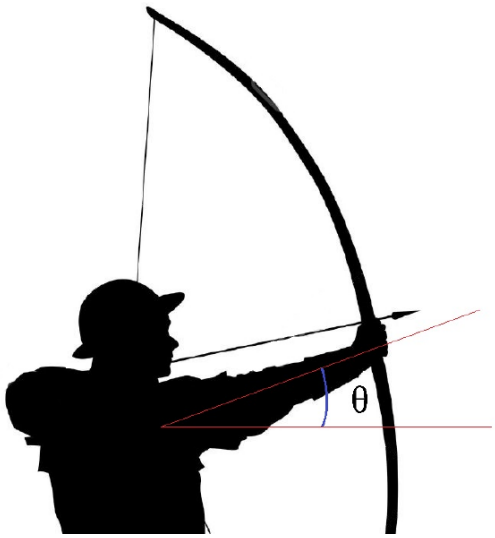


Figura 6.6

Una vez que se tenga el ángulo deseado se puede accionar un disparo, para lo cual se verá un pequeño círculo verde simulando el disparo. Este círculo seguirá una trayectoria parabólica al tomar como base el ángulo del brazo.

La ventana de la aplicación está dividida en dos partes, la primera muestra un video en tiempo real de la persona y un círculo, el cual, al posicionar la mano izquierda en el mismo, acciona el tiro. La segunda parte muestra una imagen de una manzana, objetivo que se busca derribar, una vez que se acciona el tiro se puede ver el círculo verde en un

movimiento parabólico, el cual al golpear la manzana genera un sonido beep del sistema.

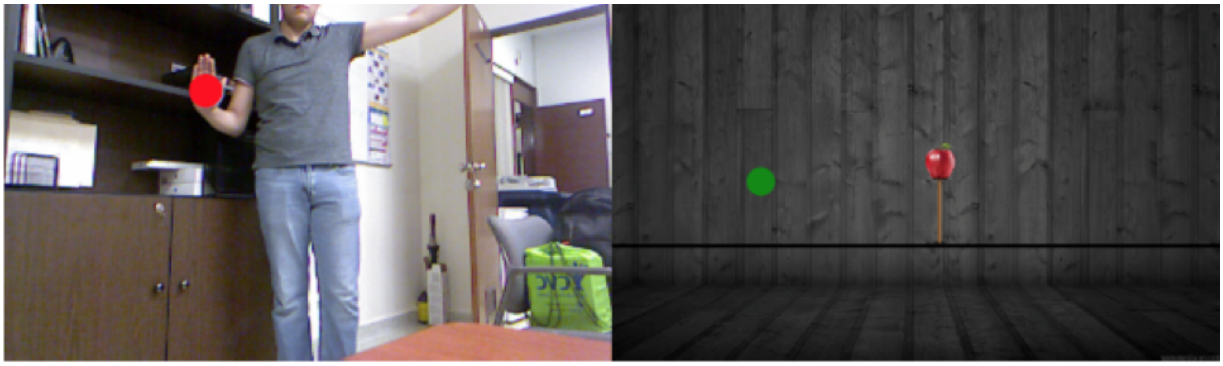


Figura 6.7

Para modelar el tiro parabólico se recurre a las fórmulas de un objeto estacionario con posiciones iniciales igual a cero, es decir $\mathbf{X}_0 = 0$ y $\mathbf{Y}_0=0$. La velocidad en X ($V0_x$) se toma constante, no se contempla aceleración. Lo que se busca es conocer la altura y el alcance según transcurre el tiempo, y así, estos valores pueden ser asignados a un elemento en la aplicación para reproducir el tiro parabólico.

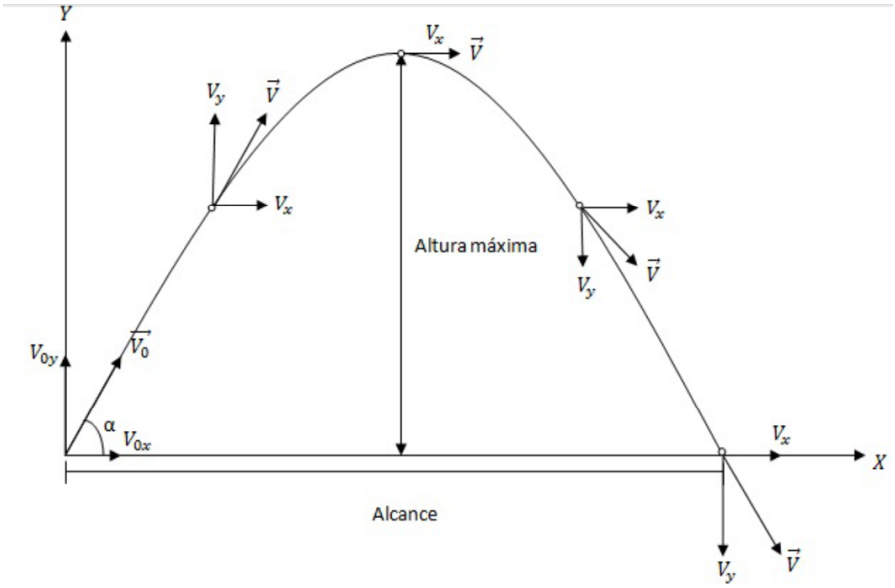


Figura 6.8

Las ecuaciones de movimiento para el tiro parabólico con aceleración constante de la gravedad son:

$$a_x = 0; \quad V_{0x} = V_0 * \cos(\alpha) \quad V_x = V_{0x} \quad X = V_{0x} * t$$
$$a_y = -g \quad V_{0y} = V_0 * \text{sen}(\alpha) \quad V_y = V_{0y} - (g*t) \quad Y = Y_0 + (V_{0y} * t) - (g * t^2)/2$$

Donde **ax** y **ay** representan la aceleración en **x** y **y**, **Vx** y **Vy** son las velocidades en **x** y **y**, **t** representa el tiempo, g es la gravedad, **V0x** y **V0y** son velocidades iniciales en x y u. α representa el ángulo con la horizontal y finalmente **X** y **Y** representan la altura y alcance respecto al tiempo.

Ahora que se ha visto cómo se implementa la animación del tiro parabólico y cómo funciona la aplicación se procederá a su realización.

Requerimientos

- Proyecto **WPF**.
- Plantilla modificada para cámara RGB.
- Imágenes de fondo en formato PNG.

Código en XAML

Los elementos que se usan en el archivo XAML son dos **Ellipse**: azul para accionar el disparo (Puntero3) y verde para seguir el movimiento parabólico (Puntero). Ambos localizados en distintas **Canvas**. Los elementos restantes son **Images** y **Labels** para generar fondos e indicadores. Las dimensiones de la ventana son **Height="735"** y **Width="770"**. Estas son diferentes a las del Canvas. El código es el siguiente:

```
<Grid>
  <Canvas Name="MainCanvas" Background="Azure"
 HorizontalAlignment="Center"
  Width="640" Height="480" Margin="646,112,6,112">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480" Width="640"
    Source="back.jpg"/>
    <Ellipse x:Name="Puntero" Width="30" Height="30"
    Canvas.Left="0"
    Canvas.Top="328" Fill="Green" ></Ellipse>
    <Image Name="manzana" Canvas.Left="460" Canvas.Top="252"
    Height="40"
    Width="40" Source="manzana.png"/>
    <Image Name="baseM" Canvas.Left="451" Canvas.Top="273"
    Height="100"
    Width="60" Source="poste.png"/>
```

```
        <Line X1="0" Y1="360" X2="640" Y2="360" Stroke="Black"
        StrokeThickness="4"/>
    </Canvas>
    <Canvas Name="MainCanvas2" Width="640" Height="480"
  HorizontalAlignment="Center"
    Margin="3,112,649,112">
        <Image Name="Image" Width="640" Height="480"
        Canvas.Left="4"/>
        <Ellipse x:Name="Puntero3" Width="35" Height="35"
        Canvas.Left="200"
        Canvas.Top="180" Fill="Blue" ></Ellipse>
    </Canvas>
</Grid>
```

Código en C#

Se crea un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla editada para la cámara RGB, esta será modificada para poder implementar la aplicación antes descrita. Una vez creado el proyecto, se agregan las bibliotecas Kinect, IO, Threading y Media. La biblioteca Threading se utiliza para implementar los timers y posteriormente las animaciones mientras que Media, en este ejemplo, se usa para recurrir a sonidos del sistema.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
using System.Media;
```

Las variables que se utilizan son las siguientes, así como de unas estructuras que se han utilizado antes para detectar colisiones.

```
private WriteableBitmap imagen; //Generar la imagen a partir del arreglo de bytes
                                //recibidos
```

```
    private byte[] cantidadPixeles; //Arreglo para recibir los bytes que envía el Kinect
    double dMano_X; //Representa la coordenada X de la mano derecha
    double dMano_Y; //Representa la coordenada Y de la mano derecha
    Point joint_Point = new Point(); //Permite obtener los datos del Joint
    Random numero = new Random(); //Se utiliza para generar numero aleatoriamente
    DispatcherTimer timer; //Timer a utilizar
    //Variables utilizadas para calcular posición X y Y en el tiro parabólico
    bool bSetAngulo = false; //Para no modificar el ángulo si se ha ejecutado un tiro
    double dAngulo = 38.0; //Guarda el ángulo de tiro
    double dTiempo = 0.0; //Guarda el tiempo de tiro transcurrido
    double dStepTime = 0.500; //Incremento de tiempo para simular tiempo transcurrido
    double dVelocidad = 80.0; //Velocidad constante en X
    // Estructura que almacenara la información del objeto
    struct Imagenes
    {
      public double dPosX;
      public double dPosY;
      public double dAncho;
      public double dAlto;
    }
    //Objetos que emplea la aplicación
    Imagenes bala, manzana_1, mano_Izq, circulo_Disparar;
```

Uno de los primeros métodos a modificar es el método **mainWindow**, aquí se obtienen

las propiedades de las imágenes para las colisiones y se asignan a sus respectivas estructuras. También se realizan las configuraciones iniciales del **timer**. El método queda de la siguiente forma:

```
public MainWindow() {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
        // Asignar propiedades iniciales Manzana
        manzana_1.dPosX = numero.Next(300, 500);
        manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
        manzana_1.dPosY = 252.0;
        manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
        manzana_1.dAncho = manzana.Width;
        manzana_1.dAlto = manzana.Height;
        // Asignar propiedades iniciales base de Manzana y elementos restantes
        baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
        baseM.SetValue(Canvas.TopProperty, manzana_1.dPosY + 21.0);
        bala.dAncho = Puntero.Width;
        bala.dAlto = Puntero.Height;
        mano_Izq.dAncho = Puntero3.Width;
        mano_Izq.dAlto = Puntero3.Height;
        circulo_Disparar.dPosX =
(double)Puntero3.GetValue(Canvas.LeftProperty);
        circulo_Disparar.dPosY =
(double)Puntero3.GetValue(Canvas.TopProperty);
        circulo_Disparar.dAncho = Puntero3.Width;
```

```
        circulo_Disparar.dAlto = Puntero3.Height;
        // Configurar e iniciar el timer
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
        timer.Tick += new EventHandler(timer_Tick);
    }
```

Como se ha visto, se utilizará la **cámara RGB** y **Skeleton**, esto quiere decir que se va a trabajar con dos tecnologías del Kinect simultáneamente. En el área que se ha reservado para los métodos que utilizan los datos del Kinect se necesita agregar el método **usarSkeleton**, esto es lo único de esta área que se modifica, el método **usarCamara** queda sin cambio alguno.

Los cambios que se realizan en el método usarSkeleton son la obtención de los Joints a utilizar, en este caso son el de la mano izquierda, el de la mano derecha y el del hombro derecho.

```
Joint joint1 = skeleton.Joints[JointType.HandRight];
Joint joint2 = skeleton.Joints[JointType.HandLeft];
Joint joint3 = skeleton.Joints[JointType.ShoulderRight];
```

Después se evalúa que los joints estén trazados y una vez corroborado lo anterior se analiza una condición en la cual la mano derecha debe estar posicionada por arriba del hombro derecho, esto con el fin de solo calcular ángulos positivos.

Si las condiciones se dan, se verifica que no exista un tiro en ejecución para poder calcular un nuevo ángulo y así no afectar la trayectoria, si es que hay un tiro en ejecución.

Para finalizar solo se adquieren las coordenadas de la mano izquierda y se verifica si se ha accionado el disparo. Hay que recordar que el disparo se inicia si hay interacción entre la mano izquierda y el **Ellipse** de color azul. En resumen, el objetivo de este método es solo verificar ciertas condiciones en la postura para poder calcular el ángulo que forma el brazo con el hombro y confirmar si se acciona el disparo. El método **usarSkeleton** queda de la siguiente forma:

```
private void usarSkeleton(Skeleton skeleton) {
  // Declarar joints a utlizar
  Joint joint1 = skeleton.Joints[JointType.HandRight];
```

```
Joint joint2 = skeleton.Joints[JointType.HandLeft];
Joint joint3 = skeleton.Joints[JointType.ShoulderRight];
// Si los Joints están listos obtener las coordenadas
if (joint1.TrackingState == JointTrackingState.Tracked &&
                    joint2.TrackingState ==
                    JointTrackingState.Tracked) {
    joint_Point = this.SkeletonPointToScreen(joint1.Position);
    dMano_X = joint_Point.X;
    dMano_Y = joint_Point.Y;
    // Verificar posición correcta de mano y hombro
    if (dMano_Y < SkeletonPointToScreen(joint3.Position).Y) {
            // Verificar que no hay tiro en ejecución
        if (!bSetAngulo) {
            dAngulo = calculaAngulo(joint1, joint3);
            LAngulo.Content = dAngulo;
        }
        // Obtener coordenadas mano izquierda
        joint_Point = this.SkeletonPointToScreen(joint2.Position);
        dMano_X = joint_Point.X;
        dMano_Y = joint_Point.Y;
        mano_Izq.dPosX = dMano_X;
        mano_Izq.dPosY = dMano_Y;
        // Verificar si se acciona el tiro
        if (checarColision(mano_Izq, circulo_Disparar)) {
            timer.IsEnabled = true;
            bSetAngulo = true;
```

```
                    Puntero3.Fill = Brushes.Red;
                }
            }
        }
    }
```

Ya que se van a usar dos tecnologías del Kinect se requiere agregar el método que recibe los datos del Kinect referentes a Skeleton. Como se recordará, estos métodos se activan por eventos y se necesitan enlistar sus rutinas al respectivo evento. El método que se va a agregar es **Kinect_FrameReady2(object sender, SkeletonFrameReadyEventArgs e)**, como se puede ver el segundo parámetro hace referencia a Skeleton y el contenido del método es el mismo que se utiliza en la plantilla Skeleton para recibir los datos del Kinect. El método queda de la siguiente forma:

```
    private void Kinect_FrameReady2(object sender, SkeletonFrameReadyEventArgs e) {
        // Arreglo que recibe los datos
        Skeleton[] skeletons = new Skeleton[0];
        Skeleton skeleton;
        // Abrir el frame recibido y copiarlo al arreglo skeletons
        using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
        {
            if (skeletonFrame != null)
            {
                skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
                                skeletonFrame.CopySkeletonDataTo(skeletons);
            }
        }
        // Seleccionar el primer Skeleton trazado
        skeleton = (from trackSkeleton in skeletons where
```

```
            trackSkeleton.TrackingState == SkeletonTrackingState.Tracked
            select
            trackSkeleton).FirstOrDefault();
        if (skeleton == null)
        {
            LID.Content = "0";
            return;
        }
        LID.Content = skeleton.TrackingId;
        // Enviar el Skelton a usar
        this.usarSkeleton(skeleton);
    }
```

Las configuraciones necesarias para utilizar estas dos tecnologías están en el método **Kinect_Config**, referente a las variables que reciben los datos y enlistar las rutinas con los eventos. Las configuraciones del método son las siguientes:

```
    // Habilitar ColorStream con una resolución de 640x480 a una razón de 30 frames / seg
    this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
    // Enlistar la función que se llama cada vez que se tiene listo un frame de datos
    this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
    // Crear el arreglo que recibe los datos de los pixeles
    this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];
    // Crear el WriteableBitmap que tendrá la imagen
    this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
                                      this.miKinect.ColorStream.FrameHeight,
                                      96.0, 96.0, PixelFormats.Bgr32, null);
```

```
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
// Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
this.miKinect.SkeletonStream.Enable();
// Enlistar al evento que se ejecuta cada vez que el Kinect tiene datos listos
this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady2;
```

A continuación se presentan las configuraciones necesarias en el método **Kinect_StatusChanged** cuando el estado del Kinect es **Connected**:

```
// Habilitar ColorStream con una resolución de 640x480 a una razón de 30 frames / seg
this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
// Enlistar la función que se llama cada vez que se tiene listo un frame de datos
this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
// Crear el arreglo que recibe los datos de los pixeles
this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];
// Crear el WriteableBitmap que tendrá la imagen
this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
                                  this.miKinect.ColorStream.FrameHeight,
                                  96.0, 96.0, PixelFormats.Bgr32, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
// Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
this.miKinect.SkeletonStream.Enable();
```

```
// Enlistar al evento que se ejecuta cada vez que el Kinect tiene datos listos
this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady2;
```

Y cuando el estado de Kinect es **Disconnected** son las siguientes:

```
this.miKinect.ColorFrameReady -°©-= this.Kinect_FrameReady;
this.miKinect.SkeletonFrameReady -°©-= this.Kinect_FrameReady2;
```

Finalmente se pasará a los métodos nuevos, aquellos que se designan para el área que lleva el mismo nombre. El primer método es **timer_Tick**. Su principal función es calcular la altura y alcance del tiro parabólico para generar la animación, esto a partir del ángulo del brazo, así como la detección de colisiones entre el **Ellipse** verde y la manzana.

La primera condición que se evalúa es que el **Ellipse** esté dentro de las dimensiones del tiro. Se comparan sus propiedades con constantes que las representan; los valores **610** representan a la distancia máxima en X para el movimiento y **330** representa la distancia mínima en Y que se puede tener, si esta condición no se cumple quiere decir que el tiro ha terminado y todo se reinicia. Si no hay colisión se siguen calculando las posiciones X y Y del tiro parabólico y se actualiza el tiempo, si existe colisión todo se reinicia. El código es el siguiente:

```
void timer_Tick(object sender, EventArgs e) {
  // Asignar coordenadas iniciales Manzana
  bala.dPosX = (double)Puntero.GetValue(Canvas.LeftProperty);
  bala.dPosY = (double)Puntero.GetValue(Canvas.TopProperty);
  if ((double)Puntero.GetValue(Canvas.LeftProperty) < 610.0 &&
     (double)Puntero.GetValue(Canvas.TopProperty) <= 330.0)
  {
     if (checarColision(bala, manzana_1))
     {
          SystemSounds.Beep.Play();
          // Asignar coordenadas iniciales Manzana
          manzana_1.dPosX = numero.Next(300, 500);
          manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
```

```
            manzana_1.dPosY = 252.0;
            manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
            // Asignar coordenadas iniciales base Manzana
            baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
            baseM.SetValue(Canvas.TopProperty, manzana_1.dPosY +
            21.0);
            // Actualizar Ellipse en tiempo = 0
            dTiempo = 0.0;
            Puntero.SetValue(Canvas.LeftProperty, alcance());
            Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
            // Detener timer
            timer.IsEnabled = false;
            bSetAngulo = false;
            Puntero3.Fill = Brushes.Blue;
        }
        else
        {
            Puntero.SetValue(Canvas.LeftProperty, alcance());
            Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
            dTiempo += dStepTime;
        }
    }
    else
    {
        // Asignar coordenadas iniciales Manzana
        manzana_1.dPosX = numero.Next(300, 500);
```

```
            manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
            manzana_1.dPosY = 252.0;
            manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
            // Asignar coordenadas iniciales base Manzana
            baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
            baseM.SetValue(Canvas.TopProperty, manzana_1.dPosY + 21.0);
            // Actualizar Ellipse en tiempo = 0
            dTiempo = 0.0;
            Puntero.SetValue(Canvas.LeftProperty, alcance());
            Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
            // Detener timer
            timer.IsEnabled = false;
            bSetAngulo = false;
            Puntero3.Fill = Brushes.Blue;
        }
    }
```

Los siguientes tres métodos son necesarios para la generación del movimiento parabólico, se usan para calcular el ángulo del brazo, así como el **alcance** y la **altura** del movimiento. Los métodos alcance y altura son la implementación de las fórmulas que se dieron al inicio. El ángulo se calcula con trigonometría, específicamente las leyes de senos y cosenos, así como del cálculo de distancia entre dos puntos. La siguiente imagen ilustra el porqué de esas leyes, ya que el brazo y el hombro forman un triángulo rectángulo entre sí.

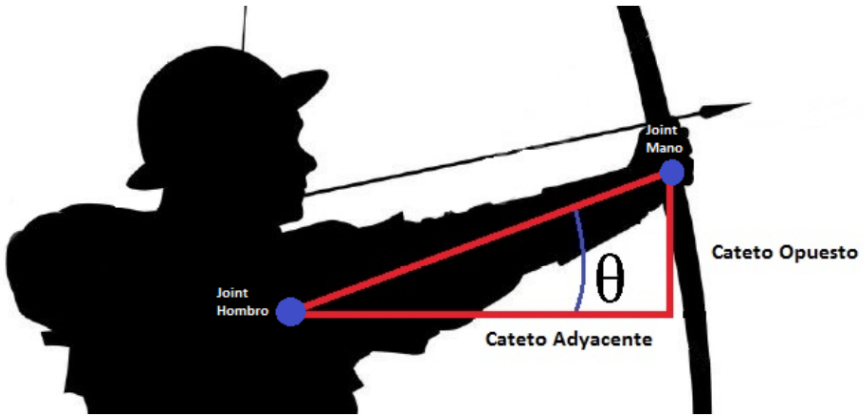


Figura 6.9

El código para los métodos es el siguiente:

```
private double calculaAngulo(Joint jointA, Joint jointB) {
   double angulo;
   double catetoOpuesto;
   double catetoAdyacente;
   double hipotenusa;
   // Puntos para las coordenadas del Joint de la mano y hombro derechos
   Point joint_PointA = new Point();
   Point joint_PointB = new Point();
   // Se obtienen los puntos de la mano y hombro derechos
   joint_PointA = this.SkeletonPointToScreen(jointA.Position);
   joint_PointB = this.SkeletonPointToScreen(jointB.Position);
   // Se calculan los lados del triángulo rectángulo formado por la mano y
  la
   // horizontal que se forma con el hombro
   catetoOpuesto = joint_PointB.Y - joint_PointA.Y;
   catetoAdyacente = joint_PointA.X - joint_PointB.X;
   // La hipotenusa es la distancia entre los Joints
   hipotenusa = Math.Sqrt(Math.Pow(catetoOpuesto, 2) +
```

```
            Math.Pow(catetoAdyacente, 2));
    // Se calcula el ángulo
    angulo = (Math.Asin(catetoOpuesto / hipotenusa))* 180.0 / Math.PI;
    return angulo;
}
private double altura()
{
    double alturaY;
    double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en
 radianes
                                    // Math.Sin asi lo recibe
    alturaY = (dVelocidad * Math.Sin(radianes) * dTiempo) –
            (9.81 * Math.Pow(dTiempo, 2) / 2);
    return alturaY;
}
private double alcance()
{
    double alcanceX;
    double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en
 radianes
                                    // Math.Sin asi lo recibe
    alcanceX = dVelocidad * Math.Cos(radianes) * dTiempo;
    return alcanceX;
}
```

Para finalizar solo se agregan los métodos restantes, estos se trabajaron anteriormente:

```
private bool checarColision(Imagenes img1, Imagenes img2) {
```

```
    if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de ob2
            return false;
    if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de ob2
            return false;
    if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha ob2
            return false;
    if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo ob2
            return false;
    return true;
  }
  private Point SkeletonPointToScreen(SkeletonPoint skelpoint)
  {
    // Convertir un punto a "Depth Space" en una resolución de 640x480
    DepthImagePoint depthPoint = this.miKinect.CoordinateMapper.
                        MapSkeletonPointToDepthPoint(skelpoint,
                        DepthImageFormat.Resolution640x480Fps30);
    return new Point(depthPoint.X, depthPoint.Y);
  }
```

Así también, se modifica el método **Window_Closing** para liberar recursos, este método queda de la siguiente forma:

```
  private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e) {
    if (this.miKinect != null && this.miKinect.IsRunning)
    {
```

```
        /* ------------------- Configuración del Kinect ------------------- */
            this.miKinect.ColorFrameReady -= this.Kinect_FrameReady;
            this.miKinect.SkeletonFrameReady                                -=
            this.Kinect_FrameReady2;
        /* ------------------------------------------------------------------ */
            this.miKinect.Stop();
    }
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando el brazo derecho toma la posición de sostener un arco se empieza a calcular el ángulo que forma, una vez que se obtiene el ángulo que se desea se puede mover la mano izquierda para tocar el **Ellipse** azul y así se accionará el tiro. Mientras el movimiento parabólico está en proceso no se afectará el ángulo hasta que el tiro termine.

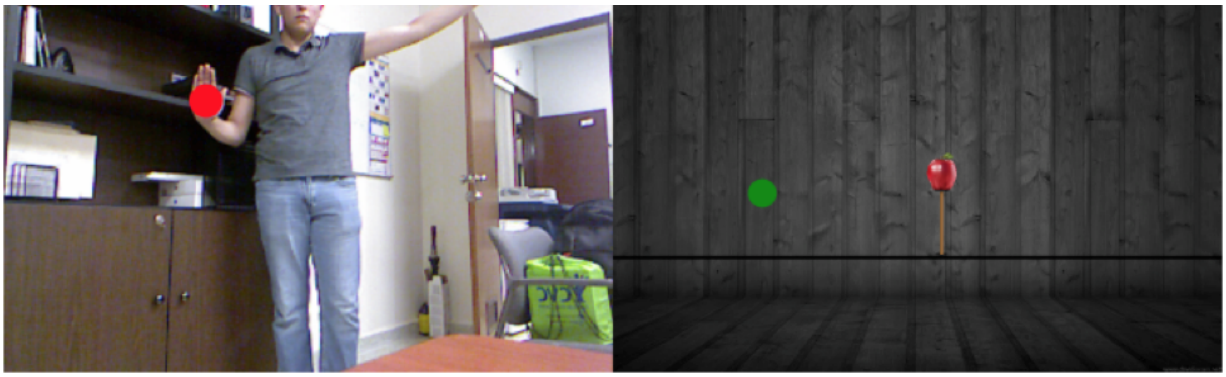


Figura 6.10

6.4 Ejemplo: tiro con arco (Kinect V2)

Este ejemplo es una aplicación sencilla en la cual se busca tener el ángulo correcto para realizar un disparo, se puede asemejar a un tiro con arco. Durante la ejecución, la aplicación provee información sobre el ángulo al cual se encuentra el brazo derecho respecto a la horizontal formada por el hombro derecho, tal como se muestra en la figura:

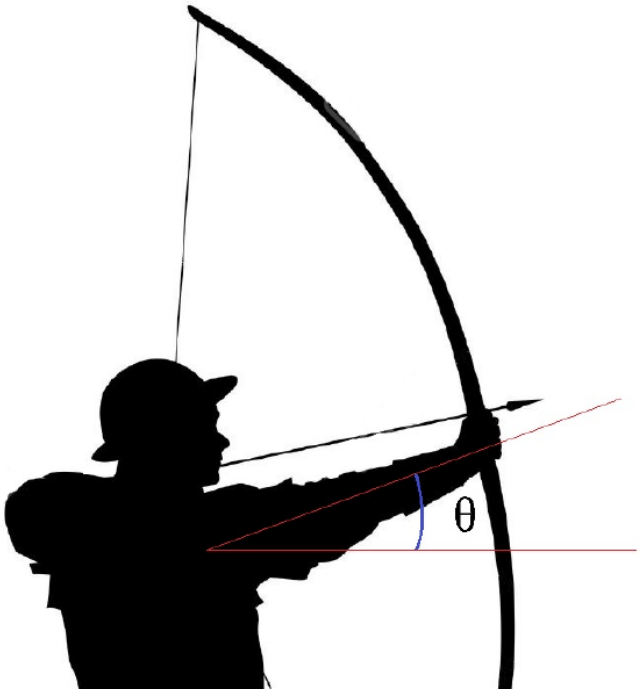


Figura 6.11

Una vez que se tiene el ángulo deseado se puede accionar un disparo, un círculo verde simulará el disparo. Este círculo seguirá una trayectoria parabólica al tomar como base para el disparo el ángulo del brazo.

La ventana de la aplicación está dividida en dos partes, la primera muestra un video en tiempo real de la persona y un círculo, el cual, al posicionar la mano izquierda en el mismo acciona el tiro. La segunda parte muestra una imagen de una manzana, este es el objetivo que se busca derribar, una vez que se acciona el tiro se puede ver el círculo verde en un movimiento parabólico, al golpear la manzana se escuchará un *beep* del sistema.

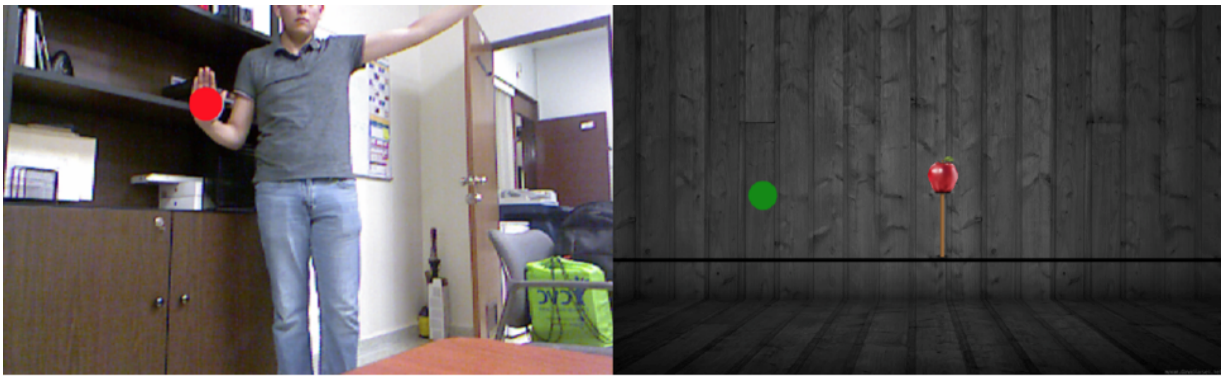


Figura 6.12

Para modelar el tiro parabólico se recurre a las fórmulas de un objeto estacionario con posiciones iniciales iguales a cero, es decir $\mathbf{X}_0 = 0$ y $\mathbf{Y}_0=0$. La velocidad en **X** ($\mathbf{V}_{0x}$) se toma constante, no se contempla aceleración. Se busca conocer la altura y el alcance según

transcurra el tiempo y así, estos valores, pueden ser asignados a un elemento en la aplicación para reproducir el tiro parabólico.

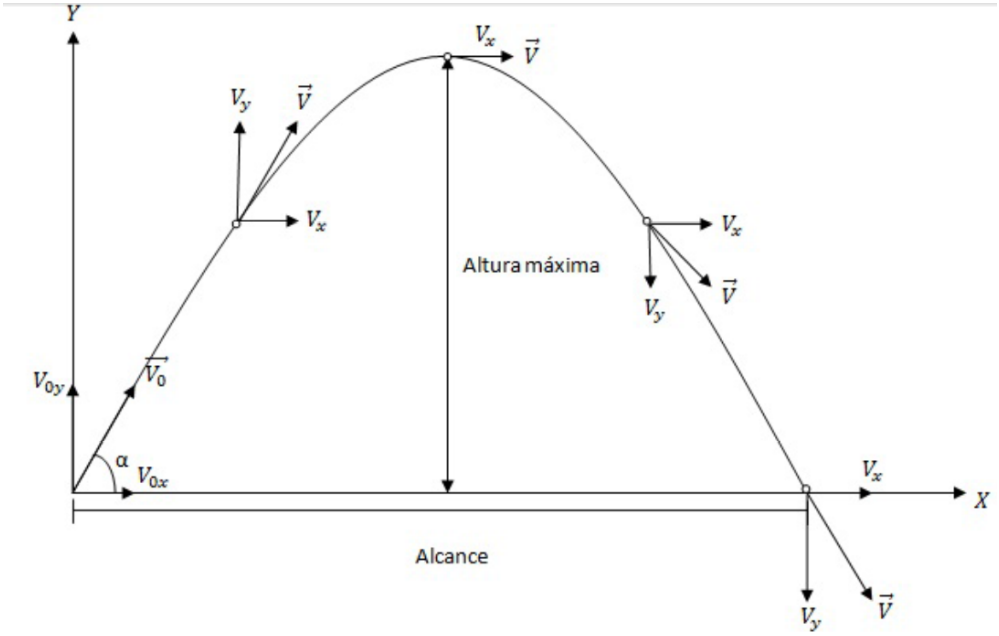


Figura 6.13

Las ecuaciones de movimiento para el tiro parabólico con aceleración constante de la gravedad son:

$$a_x = 0; \quad V_{0x} = V_0 * \cos(\alpha) \quad V_x = V_{0x} \quad X = V_{0x} * t$$
$$a_y = -g \quad V_{0y} = V_0 * \text{sen}(\alpha) \quad V_y = V_{0y} - (g*t) \quad Y = Y_0 + (V_{0y} * t) - (g * t^2)/2$$

Donde $\mathbf{a}_x$ y $\mathbf{a}_y$ representan la aceleración en **x** y **y**, $\mathbf{V}_x$ y $\mathbf{V}_y$ son las velocidades en **x** y **y**, **t** representa el tiempo, **g** es la gravedad, $\mathbf{V}_{0x}$ y $\mathbf{V}_{0y}$ son velocidades iniciales en **x** y **u**, **α** representa el ángulo con la horizontal y finalmente **X** y **Y** representan la altura y alcance respecto al tiempo.

Ahora que se ha visto cómo se implementa la animación del tiro parabólico y el funcionamiento de la aplicación, se realizará el ejemplo.

Requerimientos

- Proyecto **WPF**.
- Plantilla modificada para **cámara RGB Kinect V2.**
- Imágenes de fondo en formato PNG.

Código en XAML

Los elementos que se usan en el archivo XAML son dos **Ellipse**: azul para accionar el disparo (Puntero3) y verde para seguir el movimiento parabólico (Puntero). Ambos localizados en distintas **Canvas**. Los elementos restantes son **Images** y **Labels** para generar fondos e indicadores. Las dimensiones de la ventana son **Height="735"** y **Width="770"**, estas son diferentes a las del Canvas. El código es el siguiente:

```
<Grid>
  <Canvas Name="MainCanvas" Background="Azure"
 HorizontalAlignment="Center"
  Width="640" Height="480" Margin="646,112,6,112">
     <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
     Height="480" Width="640"
     Source="back.jpg"/>
     <Ellipse x:Name="Puntero" Width="30" Height="30"
     Canvas.Left="0"
     Canvas.Top="328" Fill="Green" ></Ellipse>
     <Image Name="manzana" Canvas.Left="460" Canvas.Top="252"
     Height="40"
     Width="40" Source="manzana.png"/>
     <Image Name="baseM" Canvas.Left="451" Canvas.Top="273"
     Height="100"
     Width="60" Source="poste.png"/>
     <Line X1="0" Y1="360" X2="640" Y2="360" Stroke="Black"
     StrokeThickness="4"/>
  </Canvas>
  <Canvas Name="MainCanvas2" Width="640" Height="480"
 HorizontalAlignment="Center"
  Margin="3,112,649,112">
     <Image Name="Image" Width="640" Height="480"
     Canvas.Left="4"/>
     <Ellipse x:Name="Puntero3" Width="35" Height="35"
```

```
        Canvas.Left="200"
        Canvas.Top="180" Fill="Blue" ></Ellipse>
    </Canvas>
</Grid>
```

Código en C#

Se crea un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla para la cámara RGB, esta será editada para poder implementar la aplicación antes descrita. Una vez creado el proyecto, se agregan las bibliotecas: **Kinect**, **IO**, **Threading** y **Media**. La biblioteca Threading se utiliza para implementar los timers y posteriormente las animaciones mientras que Media, en este ejemplo, se usa para recurrir a sonidos del sistema.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
using System.Media;
```

Las variables que se utilizan son las siguientes (también se han utilizado para detectar colisiones). Algo importante a resaltar es la diferencia con el código del Kinect V1, ya que en este caso, en lugar de utilizar el Stream para la cámara y el Skeleton, en Kinect V2 se cuenta un **FrameReader** con el cual se pueden leer datos de las distintas fuentes y como en este ejemplo se requiere usar la cámara y Body se implementa el uso del **MultiSourceFrameReader**.

```
// FrameReader para recibir datos de distintos tipos
MultiSourceFrameReader multiReader;
private Body[] bodies = null; // Arreglo para recibir datos de Body
Body miBody; // Body a utilizar
Point joint_Point = new Point(); // Permite obtener los datos de un Joint
double dMano_X; //Representa la coordenada X de la mano derecha
double dMano_Y; //Representa la coordenada Y de la mano derecha
private WriteableBitmap imagen = null; //Se utiliza para generar la
```

```
imagen
    int iWidth = 0; // Ancho de la imagen
    int iHeight = 0; // Alto de la imagen
    byte[] cantidadPixeles; // Recibir los datos de la cámara (pixeles)
    Random numero = new Random(); //Se utiliza para generar numero aleatoriamente
    DispatcherTimer timer; //Timer a utilizar
    //Variables utilizadas para calcular posición X y Y en el tiro parabólico
    bool bSetAngulo = false; //Para no modificar el ángulo si se ha ejecutado un tiro
    double dAngulo = 38.0; //Guarda el ángulo de tiro
    double dTiempo = 0.0; //Guarda el tiempo de tiro transcurrido
    double dStepTime = 0.500; //Incremento de tiempo para simular tiempo transcurrido
    double dVelocidad = 80.0; //Velocidad constante en X
    // Estructura que almacenara la información del objeto
    struct Imagenes {
       public double dPosX;
       public double dPosY;
       public double dAncho;
       public double dAlto;
    }
    //Objetos que emplea la aplicación
    Imagenes bala, manzana_1, mano_Izq, circulo_Disparar;
```

Uno de los primeros métodos a modificar es **mainWindow**, aquí se obtienen las propiedades de las imágenes para las colisiones y se asignan a sus respectivas estructuras, también se realizan las configuraciones iniciales del **timer**. El método queda de la siguiente forma:

```
public MainWindow() {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
        // Asignar coordenadas iniciales Manzana
        manzana_1.dPosX = numero.Next(300, 500);
        manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
        manzana_1.dPosY = 252.0;
        manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
        manzana_1.dAncho = manzana.Width;
        manzana_1.dAlto = manzana.Height;
        // Asignar coordenadas iniciales base Manzana
        baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
        baseM.SetValue(Canvas.TopProperty, manzana_1.dPosY +
21.0);
        // Iniciar los valores del resto de los objetos
        bala.dAncho = Puntero.Width;
        bala.dAlto = Puntero.Height;
        mano_Izq.dAncho = Puntero3.Width;
        mano_Izq.dAlto = Puntero3.Height;
        circulo_Disparar.dPosX =
(double)Puntero3.GetValue(Canvas.LeftProperty);
        circulo_Disparar.dPosY =
(double)Puntero3.GetValue(Canvas.TopProperty);
        circulo_Disparar.dAncho = Puntero3.Width;
        circulo_Disparar.dAlto = Puntero3.Height;
```

```
            // Configurar el timer
            timer = new DispatcherTimer();
            timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
            timer.Tick += new EventHandler(timer_Tick);
    }
```

Como se ha visto, se pretende utilizar la **Cámara RGB** y además se utilizará **Body**, esto quiere decir que se va a trabajar con dos tecnologías del Kinect simultáneamente. En el área que se ha reservado para los métodos que utilizan los datos del Kinect es necesario agregar el método **usarBody**. Este es el único de esta área que se modifica, el método **usarCamara** queda sin cambio alguno.

Los cambios que se realizan en el método **usarBody** son la obtención de los Joints a utilizar, en este caso se usaran tres: mano izquierda, derecha y hombro derecho.

```
Joint joint1 = miBody.Joints[JointType.HandRight];
Joint joint2 = miBody.Joints[JointType.HandLeft];
Joint joint3 = miBody.Joints[JointType.ShoulderRight];
```

Después se evalúa que los joints estén trazados y una vez corroborado lo anterior se analiza una condición en la cual la mano derecha debe estar posicionada por arriba del hombro derecho, esto con el fin de solo calcular ángulos positivos. Si las condiciones se dan se verifica que no exista un tiro en ejecución para poder calcular un nuevo ángulo y así no afectar la trayectoria.

Para finalizar solo se adquieren las coordenadas de la mano izquierda y se verifica si se ha accionado el disparo, como se recordará, el disparo se inicia si hay interacción entre la mano izquierda y el **Ellipse** de color azul. En resumen, lo que se realiza en este método es solo verificar ciertas condiciones en la postura para poder calcular el ángulo que forma el brazo con el hombro y comprobar si se acciona el disparo. El método **usarBody** queda de la siguiente forma:

```
private void usarBody(Body miBody) {
  // Crear Joints para obtener las propiedade deseadas (Coordenadas)
      Joint joint1 = miBody.Joints[JointType.HandRight];
      Joint joint2 = miBody.Joints[JointType.HandLeft];
      Joint joint3 = miBody.Joints[JointType.ShoulderRight];
```

```
// Verificar estado del Joint
    if (joint1.TrackingState == TrackingState.Tracked
    && joint2.TrackingState == TrackingState.Tracked) {
    // Obtener coordenadas
    joint_Point = this.BodyPointToScreen(joint1);
    dMano_X = joint_Point.X;
    dMano_Y = joint_Point.Y;
    // Calcular ángulo solo cuando la mano esta por arriba del hombro
        if (dMano_Y < BodyPointToScreen(joint3).Y) {
            if (!bSetAngulo) {
                dAngulo = calculaAngulo(joint1, joint3);
                LAngulo.Content = dAngulo;
    }
    // Obtener coordenadas
    joint_Point = this.BodyPointToScreen(joint2);
    dMano_X = joint_Point.X;
    dMano_Y = joint_Point.Y;
    mano_Izq.dPosX = dMano_X;
    mano_Izq.dPosY = dMano_Y;
    // Verificar si se acciona el tiro
    if (checarColision(mano_Izq, circulo_Disparar)) {
        timer.IsEnabled = true;
        bSetAngulo = true;
        Puntero3.Fill = Brushes.Red;
        }
    }
```

```
    }
}
```

Ya que se van a usar dos tecnologías del Kinect, en este ejemplo se utiliza **MultiSiurceFrameReader**, el cual permite leer datos de diferentes fuentes del Kinect. El método **Kinect_FrameReady** será un poco diferente a los vistos anteriormente, esto debido al uso de **MultiSourceFrameReader**. Uno de los primeros cambios al método es el parámetro que recibe, este es **MultiSourceFrameArrivedEventArgs e**. El método queda de la siguiente forma:

```
private void Kinect_FrameReady(object sender, MultiSourceFrameArrivedEventArgs e)
{
    …
}
```

Ahora que se tienen los parámetros requeridos se verá el código necesario para esta rutina, el cual es una combinación de los códigos de la cámara y Body. Lo primero que se hace es abrir el *frame* que se recibe al asignar los datos a una variable tipo var.

Mediante la variable **var** se pueden obtener los datos referentes a cada *frame* ya sea de la cámara o de Body. Entonces se abre el *frame* solicitando datos referentes a la cámara y si el resultado es **null** quiere decir que no había datos de la cámara. Si es diferente de **null** se procede de la misma forma que antes se han procesado los datos, tal y como se explicó en la plantilla de la cámara para Kinect V2.

Ahora que se han solicitado los datos de la cámara se procede a solicitar los datos de Body y, de igual forma, si el resultado es **null** quiere decir que no había datos de Body. Si es diferente a **null** se procede como se explicó en la plantilla de Body. El método queda de la siguiente forma:

```
private void Kinect_FrameReady(object sender, MultiSourceFrameArrivedEventArgs e) {
    // Adquirir el Frame
    var reference = e.FrameReference.AcquireFrame();
    // Adquirir el ColorFrame
    using (var colorFrame = reference.ColorFrameReference.AcquireFrame())
```

```
{
    // Verificar contenido de datos
    if (colorFrame != null)
    {
    if (colorFrame.RawColorImageFormat == ColorImageFormat.Bgra)
    {
        colorFrame.CopyRawFrameDataToArray(cantidadPixeles);
    }
    else
    {
        colorFrame.CopyConvertedFrameDataToArray(cantidadPixeles,
                ColorImageFormat.Bgra);
        }
        // Llamar método que manipula los datos
        this.usarCamara();
    }
}
 // Adquirir el BodyFrame
 using (var mibodyFrame = reference.BodyFrameReference.AcquireFrame()) {
    // Verificar contenido de datos
    if (mibodyFrame != null) {
        // Crear arreglo de Bodies y copiar datos recibidos en el mismo
        this.bodies = new Body[mibodyFrame.BodyCount];
        mibodyFrame.GetAndRefreshBodyData(this.bodies);
        // Seleccionar el primer Body o default
```

```
                miBody = (from trackBody in bodies where
                trackBody.LeanTrackingState ==
                    TrackingState.Tracked select trackBody).FirstOrDefault();
                if (miBody == null)
                {
                    LID.Content = "0";
                    return;
                }
                // Indicar ID de la persona trazada
                LID.Content = miBody.TrackingId;
                // Llamar método que manipula los datos
                this.usarBody(miBody);
            }
        }
    }
```

Las configuraciones necesarias que se realizan para utilizar estas dos tecnologías son en el método **Kinect_Config**, el cual es referente a las variables que reciben los datos y enlistar las rutinas con los eventos. Las configuraciones del método son las siguientes:

```
    // Abrir el FrameReader para poder recibir los datos de las distintas
fuentes
    // expresadas como parámetros
    multiReader =
miKinect.OpenMultiSourceFrameReader(FrameSourceTypes.Color |
                                        FrameSourceTypes.Body);
    // Enlistar la rutina que se llama cuando hay datos disponibles
    multiReader.MultiSourceFrameArrived += Kinect_FrameReady;
    // Crear FrameDescription desde ColorFrameSource usando un formato
Bgra
```

```
FrameDescription colorFrameDescription =
this.miKinect.ColorFrameSource.
                                    CreateFrameDescription(ColorImageForm
                                    at.Bgra);
iWidth = colorFrameDescription.Width;
iHeight = colorFrameDescription.Height;
cantidadPixeles = new byte[iWidth * iHeight *
((PixelFormats.Bgr32.BitsPerPixel + 7)
                                    / 8)];
// Crear el bitmap a mostrar (Imagen que se visualiza)
this.imagen = new WriteableBitmap(colorFrameDescription.Width,
                                    colorFrameDescription.Height, 96.0, 96.0,
                                    PixelFormats.Bgr32, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
```

Ahora se pasará a los métodos nuevos, aquellos que se designan para el área que lleva el mismo nombre. El primer método es **timer_Tick**. Su principal función es calcular la altura y alcance del tiro parabólico para generar la animación, esto a partir del ángulo del brazo, así como la detección de colisiones entre la elipse verde y la manzana.

La primera condición que se evalúa es que el **Ellipse** esté dentro de las dimensiones del tiro. Se comparan sus propiedades con constantes que las representan; los valores **610** representan la distancia máxima en X para el movimiento y **330** representa la distancia mínima en Y que se puede tener, si esta condición no se cumple quiere decir que el tiro ha terminado y todo se reinicia. Si no hay colisión se siguen calculando las posiciones X y Y del tiro parabólico y se actualiza el tiempo. El código es el siguiente:

```
void timer_Tick(object sender, EventArgs e) {
  // Asignar coordenadas iniciales bala
  bala.dPosX = (double)Puntero.GetValue(Canvas.LeftProperty);
  bala.dPosY = (double)Puntero.GetValue(Canvas.TopProperty);
  if ((double)Puntero.GetValue(Canvas.LeftProperty) < 610.0 &&
```

```
(double)Puntero.GetValue(Canvas.TopProperty) <= 330.0) {
if (checarColision(bala, manzana_1)) {
        SystemSounds.Beep.Play();
        // Asignar coordenadas iniciales Manzana
        manzana_1.dPosX = numero.Next(300, 500);
        manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
        manzana_1.dPosY = 252.0;
        manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
        // Asignar coordenadas iniciales base Manzana
        baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
        baseM.SetValue(Canvas.TopProperty,    manzana_1.dPosY    +
        21.0);
        // Asignar valores iniciales (t = 0.0)
        dTiempo = 0.0;
        Puntero.SetValue(Canvas.LeftProperty, alcance());
        Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
        // Reiniciar valores
        timer.IsEnabled = false;
        bSetAngulo = false;
        Puntero3.Fill = Brushes.Blue;
}
else {
        // Calcular posiciones respecto al tiempo e incrementar tiempo
        Puntero.SetValue(Canvas.LeftProperty, alcance());
        Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
        dTiempo += dStepTime;
```

```
        }
    }
    else {
        // Asignar coordenadas iniciales Manzana
        manzana_1.dPosX = numero.Next(300, 500);
        manzana.SetValue(Canvas.LeftProperty, manzana_1.dPosX);
        manzana_1.dPosY = 252.0;
        manzana.SetValue(Canvas.TopProperty, manzana_1.dPosY);
        // Asignar coordenadas iniciales base Manzana
        baseM.SetValue(Canvas.LeftProperty, manzana_1.dPosX - 9.0);
        baseM.SetValue(Canvas.TopProperty, manzana_1.dPosY + 21.0);
        // Asignar valores iniciales (t = 0.0)
        dTiempo = 0.0;
        Puntero.SetValue(Canvas.LeftProperty, alcance());
        Puntero.SetValue(Canvas.TopProperty, 330.0 - altura());
        // Reiniciar valores
        timer.IsEnabled = false;
        bSetAngulo = false;
        Puntero3.Fill = Brushes.Blue;
    }
}
```

Los siguientes tres métodos son necesarios para la generación del movimiento parabólico, ya que se usan para calcular el ángulo del brazo, el alcance y la altura del movimiento. Los métodos **alcance** y **altura** son la implementación de las fórmulas que se dieron al inicio. El ángulo se calcula con trigonometría, específicamente las leyes de senos y cosenos, así como del cálculo de distancia entre dos puntos. La siguiente imagen ilustra el porqué de esas leyes, ya que el brazo y el hombro forman un triángulo rectángulo entre sí.

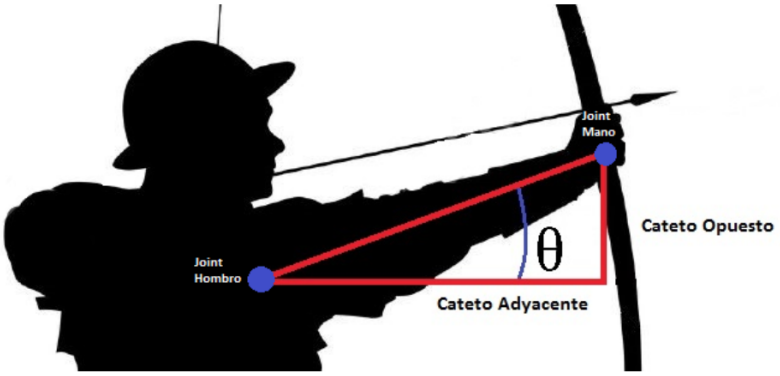


Figura 6.14

El código para los métodos es el siguiente:

```
private double calculaAngulo(Joint jointA, Joint jointB) {
        double angulo;
        double catetoOpuesto;
        double catetoAdyacente;
        double hipotenusa;
        Point joint_PointA = new Point();
        Point joint_PointB = new Point();
        // Obtener coordenadas de la mano y hombro
        joint_PointA = this.BodyPointToScreen(jointA);
        joint_PointB = this.BodyPointToScreen(jointB);
        // Calcular catetos e hipotenusa
        catetoOpuesto = joint_PointB.Y - joint_PointA.Y;
        catetoAdyacente = joint_PointA.X - joint_PointB.X;
        hipotenusa = Math.Sqrt(Math.Pow(catetoOpuesto, 2)
        + Math.Pow(catetoAdyacente, 2));
        // Calcular ángulo
        ángulo = (Math.Asin(catetoOpuesto / hipotenusa)) * 180.0 / Math.PI;
        return angulo;
```

```
}
private double altura()
{
        double alturaY;
        double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en
        radianes
                                        // Math.Sin así lo recibe
        alturaY = (dVelocidad * Math.Sin(radianes) * dTiempo) –
                                (9.81 * Math.Pow(dTiempo, 2) / 2);
        return alturaY;
}
private double alcance()
{
   double alcanceX;
   double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en
  radianes
                                        // Math.Sin así lo recibe
   alcanceX = dVelocidad * Math.Cos(radianes) * dTiempo;
   return alcanceX;
}
```

También se agregan los siguientes métodos, los cuales ya ha sido trabajados:

```
private bool checarColision(Imagenes img1, Imagenes img2) {
   if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la
  izquierda de ob2
            return false;
   if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de
  ob2
```

```
            return false;
        if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha ob2
            return false;
        if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo ob2
            return false;
        return true;
    }
    private Point BodyPointToScreen(Joint miJoint)
    {
        // Convertir un punto a "Depth Space"
        DepthSpacePoint depthSpacePoint = miKinect.CoordinateMapper.
                                          MapCameraPointToDepthSpace(miJoint.Position);
        return new Point(depthSpacePoint.X, depthSpacePoint.Y);
    }
```

Con lo anterior se tiene listo el procesado de los datos. Ahora para finalizar se editará el método **Window_Closing**, el cual se encarga de tomar las acciones necesarias al finalizar la aplicación. La única modificación aquí es sobre el **FrameReader**, lo que se hace es "liberarlo" para que pueda ser utilizado por otros programas, el método es **Dispose**. El método **Window_Closing** queda de la siguiente manera:

```
    private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
    {
        /* ------------------- Configuración del Kinect ------------------- */
        if (this.multiReader != null)
        {
            // Disponer BodyFrameReader
```

```
            this.multiReader.Dispose();
            this.multiReader = null;
        }
        /* ------------------------------------------------------------ */
        if (this.miKinect != null)
        {
            this.miKinect.Close();
            this.miKinect = null;
        }
    }
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando el brazo derecho toma una posición semejante a la de sostener un arco se empieza a calcular el ángulo que forma, una vez que se obtiene se puede mover la mano izquierda para tocar el **Ellipse** azul y así accionar al tiro. Mientras el movimiento parabólico está en proceso, no se afectará el ángulo hasta que el tiro termine.

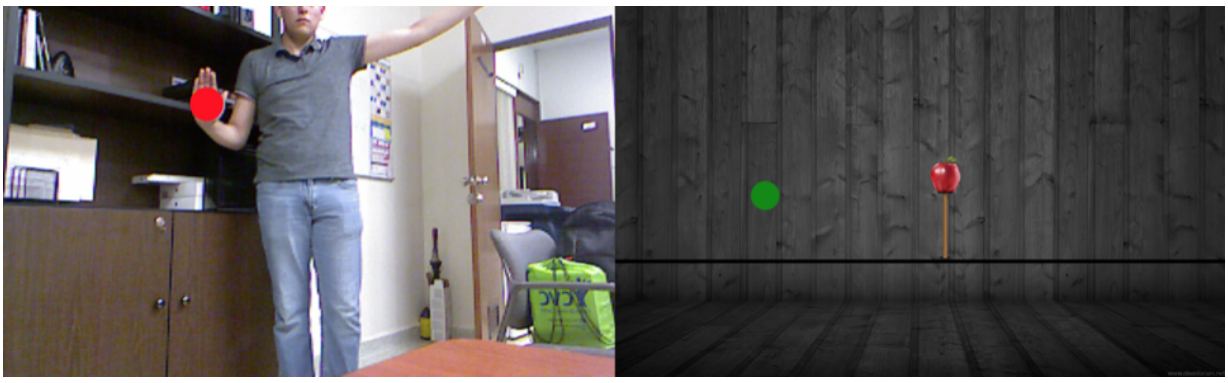


Figura 6.15

6.5 Ejemplo: dominar la pelota (Kinect V1)

Se trabajará con la cámara y Skeleton para interactuar con las rodillas y el programa, el objetivo es simular dominar una pelota. En la ventana de la aplicación aparece un balón que siempre tiene un movimiento descendente el cual, cuando se tiene alguna interacción con las rodillas, se produce un efecto de golpe con un movimiento parabólico. Para lograr

implementarlo se modela siguiendo las características en la siguiente gráfica:

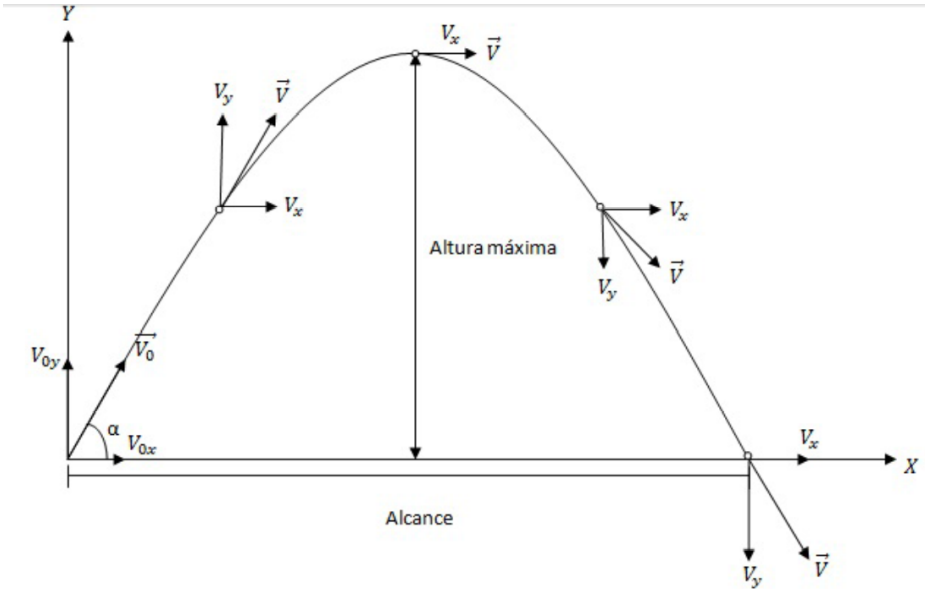


Figura 6.16

Al igual que el ejemplo “Corrección de posturas”, se utilizarán las matemáticas de la ejecución del tiro parabólico. Para el cálculo de altura y alcance según transcurre el tiempo, se implementaron las siguientes fórmulas:

$$a_x = 0; \quad V_{0x} = V_0 * \cos(\alpha) \quad V_x = V_{0x} \quad X = V_{0x} * t$$
$$a_y = -g \quad V_{0y} = V_0 * \text{sen}(\alpha) \quad V_y = V_{0y} - (g*t) \quad Y = Y_0 + (V_{0y} * t) - (g * t^2)/2$$

Donde $\mathbf{a}_x$ y $\mathbf{a}_y$ representan la aceleración en **x** y **y**, $\mathbf{V}_x$ y $\mathbf{V}_y$ son las velocidades en **x** y **y**, **t** representa el tiempo, g es la gravedad, $\mathbf{V}_{0x}$ y $\mathbf{V}_{0y}$ son velocidades iniciales en **x** y **u**, **α** representa el ángulo con la horizontal y finalmente **X** y **Y** representan la altura y alcance respecto al tiempo.

Requerimientos

- Proyecto **WPF**.
- Plantilla modificada para cámara RGB.
- Imagen de una pelota.

Figura 6.17

Código en XAML

Los elementos que se usan en el archivo XAML son una imagen que muestra la pelota en movimiento, los elementos restantes son **Images** y **Labels** para mostrar las imágenes de la cámara y otros indicadores. Las dimensiones de la ventana son **Height="770"** y **Width="770"**. Estas son diferentes a las del Canvas.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
 HorizontalAlignment="Center">
    <Image Name="Image" Width="640" Height="480"/>
    <Image Name="Pelota" Width="60" Height="60"
    Source="pelota.png"
    Canvas.Left="570"/>
  </Canvas>
</Grid>
```

Código en C#

Se elabora un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla para la cámara RGB, esta será

editada para poder implementar la aplicación antes descrita. Después de crear el proyecto se agregan las bibliotecas: **Kinect**, **IO** y **Threading**. La biblioteca **Threading** se utiliza para implementar los **timers**.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

Las variables que se utilizan son las siguientes, también se han utilizado para detectar colisiones. Una variable que se tendría que describir es **iEstadoGlobal**, esta se ha creado para detectar la colisión entre la rodilla izquierda (estado 2) y derecha (estado 3) dado que el movimiento que resulta de dicha colisión va en direcciones diferentes. Entonces se tendrían dos casos para las rodillas más otros dos, de los cuales, uno es que no hay colisiones y se sigue el movimiento inicial descendente (estado 1) y el otro es para inicializar todos los datos cuando cualquiera de los movimientos ha terminado (estado 0). Entonces, el flujo que se sigue en los estados de la variable **iEstadoGlobal** es el siguiente:

Figura 6.18

Las variables para este ejemplo son las siguientes:

```
private WriteableBitmap imagen; // Generar imagen a partir de los datos recibidos
private byte[] cantidadPixeles; // Arreglo para recibir los bytes que envía el Kinect
Point joint_Point = new Point();// Permite obtener los datos del Joint
Random numero = new Random();
DispatcherTimer timer;
```

```
int iEstadoGlobal = 0; //Determinar estado de movimiento
int iStepPixeles = 30; //Incremento en pixeles del movimiento
//Variables utilizadas para calcular posición X y Y en el tiro parabólico
double dXinicial = 0.0; //Posición X inicial
double dYinicial = 0.0; //Posición Y inicial
double dAngulo = 87.0; //Guarda el ángulo de tiro
double dTiempo = 0.0; //Guarda el tiempo de tiro transcurrido
double dStepTime = 0.8; //Incremento de tiempo para simular tiempo transcurrido
double dVelocidad = 90.0; //Velocidad constante en X
// Estructura que almacenará la información del objeto
struct Imagenes
{
  public double dPosX;
  public double dPosY;
  public double dAncho;
  public double dAlto;
}
Imagenes pelota_1, rodilla_Der, rodilla_Izq;
```

Uno de los primeros métodos a modificar es el método **mainWindow**, aquí se obtienen las propiedades de las imágenes para las colisiones y se asignan a sus respectivas estructuras, también se realizan las configuraciones iniciales del **timer**. El método queda de la siguiente forma:

```
public MainWindow() {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
```

```
            // Asignar valores iniciales a las estructuras
            pelota_1.dPosX =
(double)Pelota.GetValue(Canvas.LeftProperty); ;
            pelota_1.dPosY =
(double)Pelota.GetValue(Canvas.TopProperty);
            pelota_1.dAlto = Pelota.Height;
            pelota_1.dAncho = Pelota.Width;
            rodilla_Der.dPosX = 0.0;
            rodilla_Der.dPosY = 0.0;
            rodilla_Der.dAlto = 5;
            rodilla_Der.dAncho = 5;
            rodilla_Izq.dPosX = 0.0;
            rodilla_Izq.dPosY = 0.0;
            rodilla_Izq.dAlto = 5;
            rodilla_Izq.dAncho = 5;
            // Configurar e iniciar el timer
            timer = new DispatcherTimer();
            timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
            timer.Tick += new EventHandler(timer_Tick);
            timer.IsEnabled = true;
    }
```

Como se ha visto, se pretende utilizar la **cámara RGB** y el **Skeleton**, esto quiere decir que se trabajará con dos tecnologías del Kinect. En el área que se ha reservado para los métodos que utilizan los datos del Kinect se necesita agregar el método **usarSkeleton**, este será el único que se modifica, el método **usarCamara** quedará sin alteraciones.

Los cambios que se realizan en el método **usarSkeleton** son la obtención de los Joints a utilizar: rodilla izquierda y derecha.

```
Joint joint1 = skeleton.Joints[JointType.KneeLeft];
```

```
Joint joint2 = skeleton.Joints[JointType.KneeRight];
```

Después se evalúa que los joints estén trazados. Una vez corroborado lo anterior se obtienen las coordenadas de cada uno mediante el método **SkeletonPointToScreen** y se asignan a las estructuras correspondientes a ambas rodillas. El método **usarSkeleton** queda de la siguiente forma:

```
private void usarSkeleton(Skeleton skeleton) {
  // Se declaran los Joints a utilizar
  Joint joint1 = skeleton.Joints[JointType.KneeLeft];
  Joint joint2 = skeleton.Joints[JointType.KneeRight];
  // Si el Joint está listo obtener las coordenadas
  if (joint1.TrackingState == JointTrackingState.Tracked) {
    // Obtener coordenadas del primer Joint y asignarlas a la estructura
    joint_Point = this.SkeletonPointToScreen(joint1.Position);
    rodilla_Izq.dPosX = joint_Point.X;
    rodilla_Izq.dPosY = joint_Point.Y;
    // Obtener coordenadas del segundo Joint y asignarlas a la estructura
    joint_Point = this.SkeletonPointToScreen(joint2.Position);
    rodilla_Der.dPosX = joint_Point.X;
    rodilla_Der.dPosY = joint_Point.Y;
  }
}
```

Ya que se van a usar dos tecnologías del Kinect, se requiere agregar el método que recibe los datos del Kinect referentes a Skeleton. Como se recordará, estos métodos se activan por eventos y se necesitan enlistar sus rutinas al mismo. El método que se va a agregar es **Kinect_FrameReady2(object sender, SkeletonFrameReadyEventArgs e)**, como se puede ver el segundo parámetro hace referencia a Skeleton y el contenido del método es el mismo que se utiliza en la plantilla Skeleton para recibir los datos del Kinect. El método queda de la siguiente forma:

```
private void Kinect_FrameReady2(object sender, SkeletonFrameReadyEventArgs e) {
```

```
    // Arreglo que recibe los datos
    Skeleton[] skeletons = new Skeleton[0];
    Skeleton skeleton;
    // Abrir el frame recibido y copiarlo al arreglo skeletons
    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame()) {
        if (skeletonFrame != null) {
            skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
            skeletonFrame.CopySkeletonDataTo(skeletons);
        }
    }
    // Seleccionar el primer Skeleton trazado
    skeleton = (from trackSkeleton in skeletons where
            trackSkeleton.TrackingState == SkeletonTrackingState.Tracked
            select
            trackSkeleton).FirstOrDefault();
    if (skeleton == null) {
        LID.Content = "0";
        return;
    }
        LID.Content = skeleton.TrackingId;
        // Enviar el Skelton a usar
        this.usarSkeleton(skeleton);
}
```

Las configuraciones necesarias que se realizan para utilizar estas dos tecnologías están en el método **Kinect_Config** y estas son referente a las variables que reciben los datos y enlistan las rutinas con los eventos. Las configuraciones del método son las siguientes:

```
// Habilitar ColorStream con una resolución de 640x480 a una razón de 30
```

```
frames / seg
    this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640
x480Fps30);
    // Enlistar la función que se llama cada vez que se tiene listo un frame de
datos
    this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
    // Crear el arreglo que recibe los datos de los pixeles
    this.cantidadPixeles                =                new
byte[this.miKinect.ColorStream.FramePixelDataLength];
    // Crear el WriteableBitmap que tendrá la imagen
    this.imagen                =                new
WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
                        this.miKinect.ColorStream.FrameHeight,
                        96.0, 96.0, PixelFormats.Bgr32, null);
    // Asignar la imagen como fuente para ser mostrada en la ventana
    this.Image.Source = this.imagen;
    // Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
    this.miKinect.SkeletonStream.Enable();
    // Enlistar al evento que se ejecuta cada vez que el Kinect tiene datos
listos
    this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady2;
```

En el método **Kinect_StatusChanged** las configuraciones necesarias cuando el estado del Kinect es **Connected** son las siguientes:

```
    // Habilitar ColorStream con una resolución de 640x480 a una razón de 30
frames / seg
    this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640
x480Fps30);
    // Enlistar la función que se llama cada vez que se tiene listo un frame de
```

```
datos
    this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
    // Crear el arreglo que recibe los datos de los pixeles
    this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];
    // Crear el WriteableBitmap que tendrá la imagen
    this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
                                      this.miKinect.ColorStream.FrameHeight,
                                      96.0, 96.0, PixelFormats.Bgr32, null);
    // Asignar la imagen como fuente para ser mostrada en la ventana
    this.Image.Source = this.imagen;
    // Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
    this.miKinect.SkeletonStream.Enable();
    // Enlistar al evento que se ejecuta cada vez que el Kinect tiene datos listos
    this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady2;
```

Y cuando el estado de Kinect es **Disconnected** son las siguientes:

```
this.miKinect.ColorFrameReady -°©-= this.Kinect_FrameReady;
this.miKinect.SkeletonFrameReady -°©-= this.Kinect_FrameReady2;
```

Ahora se trabajarán los métodos nuevos, aquellos que se designan para el área que lleva el mismo nombre. El primer método es **timer_Tick**, su principal función es implementar los estados que se describieron al inicio. Los valores de cada estado y la descripción de las acciones en cada uno son las siguientes:

1. **Estado 0**: se asignan valores iniciales a la posición de la pelota, el tiempo utilizado en el tiro parabólico y se indica el siguiente estado (Estado 1).
2. **Estado 1**: se evalúa si ha existido alguna colisión, si existe colisión entre la pelota y la rodilla izquierda se toma la posición inicial de la pelota y se indica que el siguiente estado es el 2. Si existe colisión entre la pelota y la rodilla derecha se toma la posición

inicial de la pelota y se indica que el siguiente estado es el 3. Si no existe colisión se sigue el movimiento descendente de la pelota, pare esto se actualiza solo **TopProperty** de la imagen de la pelota, y cuando finaliza el movimiento se indica que el siguiente estado es el 0.

3. **Estado 2**: se calculan las posiciones en **X** y **Y** del tiro parabólico para asignarse a las propiedades *Top* y *Left* de la imagen de la pelota y se actualiza el tiempo de cálculo. Cuando finaliza el movimiento se indica que el siguiente estado es el 0. El movimiento parabólico va hacia la derecha.

4. **Estado 3**: se calculan las posiciones en **X** y **Y** del tiro parabólico para asignarse a las propiedades *Top* y *Left* de la imagen de la pelota y se actualiza el tiempo de cálculo. Cuando termina el movimiento se indica que el siguiente estado es el 0. El movimiento parabólico es hacia la izquierda.

El valor (**420**) contra el que se compara **pelota_1.dPosY** es un valor designado para indicar cuando la pelota aún se encuentra dentro de las dimensiones del canvas. Cuando **pelota_1.dPosY** es mayor que este valor la imagen queda fuera del canvas, lo cual sirve como limitante para indicar el fin del movimiento. El código de **timer_Tick** es el siguiente:

```
void timer_Tick(object sender, EventArgs e) {
  switch (iEstadoGlobal) {
    case 0: // Asignar valores iniciales
        pelota_1.dPosX = numero.Next(200, 440);
        Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
        pelota_1.dPosY = 0.0;
        Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
        dTiempo = 0.0;
        iEstadoGlobal = 1;
        break;
    case 1: // Evaluar colisiones
        if (checarColision(pelota_1, rodilla_Izq)) {
            dXinicial = pelota_1.dPosX;
            dYinicial = pelota_1.dPosY;
```

```
            iEstadoGlobal = 2;
        }
        if (checarColision(pelota_1, rodilla_Der)) {
            dXinicial = pelota_1.dPosX;
            dYinicial = pelota_1.dPosY;
            iEstadoGlobal = 3;
        }
        if (pelota_1.dPosY < 420) {
            Pelota.SetValue(Canvas.TopProperty,  pelota_1.dPosY  +=
            iStepPixeles);
        }
        else { iEstadoGlobal = 0; }
        break;
    case 2: // Movimiento debido a la colisión con la rodilla izquierda
        if (pelota_1.dPosY < 420) {
            pelota_1.dPosY = dYinicial - altura();
            pelota_1.dPosX = dXinicial + alcance();
            Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
            Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
            dTiempo += dStepTime;
        }
        else { iEstadoGlobal = 0; }
        break;
    case 3: // Movimiento debido a la colisión con la rodilla derecha
        if (pelota_1.dPosY < 420) {
        pelota_1.dPosY = dYinicial - altura();
```

```
            pelota_1.dPosX = dXinicial - alcance();
            Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
            Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
            dTiempo += dStepTime;
        }
        else{ iEstadoGlobal = 0; }
        break;
    }
}
```

Los siguientes métodos son necesarios para la generación del movimiento parabólico, estos son **altura** y **alcance**, son también la implementación de las fórmulas que se dieron al inicio. El código para los métodos es el siguiente:

```
private double altura() {
    double alturaY;
    double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en radianes
                                                 // Math.Sin así lo recibe
    alturaY = (dVelocidad * Math.Sin(radianes) * dTiempo) –
            (9.81 * Math.Pow(dTiempo, 2) / 2);
    return alturaY;
}
private double alcance()
{
    double alcanceX;
    double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en radianes
                                                 // Math.Sin así lo recibe
```

```
    alcanceX = dVelocidad * Math.Cos(radianes) * dTiempo;
    return alcanceX;
}
```

Los métodos restantes a agregar son los siguientes, estos son necesarios para la detección de colisiones y obtención de las coordenadas de los joints, estos ya se han trabajado anteriormente:

```
private bool checarColision(Imagenes img1, Imagenes img2) {
    if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de ob2
            return false;
    if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de ob2
            return false;
    if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha ob2
            return false;
    if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo ob2
            return false;
    return true;
}
private Point SkeletonPointToScreen(SkeletonPoint skelpoint)
{
    // Convertir un punto a "Depth Space" en una resolución de 640x480
    DepthImagePoint depthPoint = this.miKinect.CoordinateMapper.
                        MapSkeletonPointToDepthPoint(skelpoint,
                        DepthImageFormat.Resolution640x480Fps30);
    return new Point(depthPoint.X, depthPoint.Y);
```

```
}
```

Para finalizar, el método que se modifica es **Window_Closing**, en el cual solo se desligan las rutinas que se llaman cuando hay *frames* listos tanto de la cámara como del Skeleton, el método queda de la siguiente forma:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
  if (this.miKinect != null && this.miKinect.IsRunning)
  {
    /* ------------------- Configuración del Kinect ------------------- */
        this.miKinect.ColorFrameReady -= this.Kinect_FrameReady;
        this.miKinect.SkeletonFrameReady -= this.Kinect_FrameReady2;
    /* ---------------------------------------------------------------- */
        this.miKinect.Stop();
  }
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando se ejecuta la pelota se mueve en forma descendente, cuando realiza una interacción con las rodillas se puede ver que el movimiento de la pelota cambia a un movimiento parabólico semejando un golpe.

Figura 6.19

6.6 Ejemplo: dominar la pelota (Kinect V2)

En este sencillo ejemplo se trabajará con la cámara y Body para interactuar con las rodillas y el programa, el objetivo es simular dominar una pelota. En la ventana de la aplicación aparece una esfera que siempre tiene un movimiento descendente, cuando tiene alguna interacción con las rodillas, se produce un efecto de golpe con las rodillas y se presenta un movimiento parabólico. Para su implementación, se modelará siguiendo las características en la siguiente imagen:

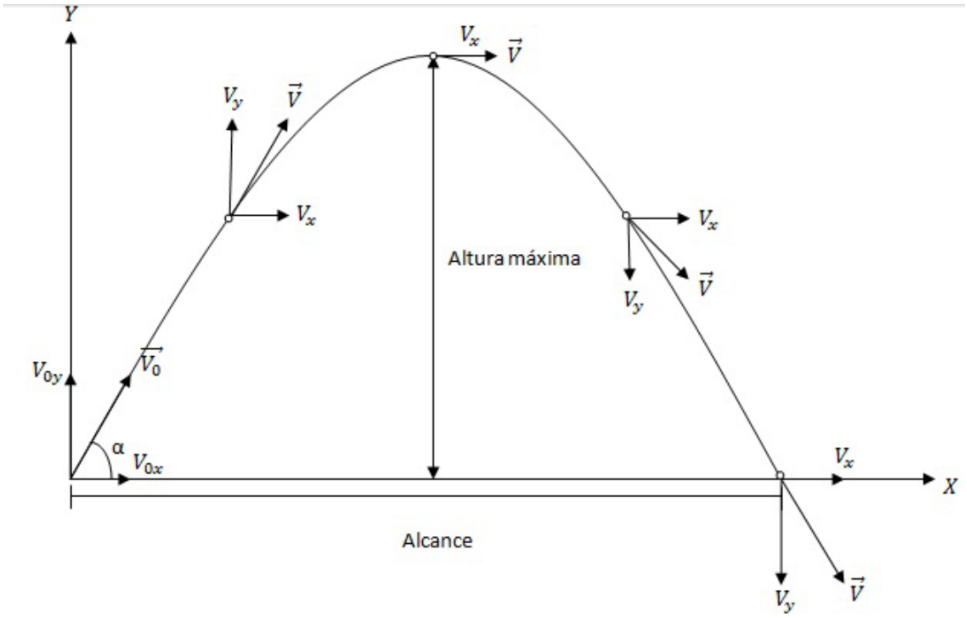


Figura 6.20

Se utilizarán las mismas matemáticas que en el ejemplo de “Corrección de posturas”. También para el cálculo de altura y alcance según transcurra el tiempo, las fórmulas que se implementaron son las siguientes:

$$a_x = 0; \quad V_{0x} = V_0 * \cos(\alpha) \quad V_x = V_{0x} \quad X = V_{0x} * t$$
$$a_y = -g \quad V_{0y} = V_0 * \text{sen}(\alpha) \quad V_y = V_{0y} - (g*t) \quad Y = Y_0 + (V_{0y} * t) - (g * t^2)/2$$

Donde $\mathbf{a}_x$ y $\mathbf{a}_y$ representan la aceleración en **x** y **y**, $\mathbf{V}_x$ y $\mathbf{V}_y$ son las velocidades en **x** y **y**, **t** representa el tiempo, g es la gravedad, $\mathbf{V}_{0x}$ y $\mathbf{V}_{0y}$ son velocidades iniciales en **x** y **u**, **α** representa el ángulo con la horizontal y finalmente **X** y **Y** representan la altura y alcance respecto al tiempo.

Ahora que se ha visto cómo se implementa la animación del tiro parabólico y cómo funciona la aplicación se procederá a su realización.

Requerimientos

- Proyecto **WPF**.
- Plantilla modificada para cámara RGB Kinect V2.
- Imagen de una pelota.

Figura 6.21

Código en XAML

Los elementos que se usan en el archivo XAML son una imagen que muestra la pelota en movimiento, los elementos restantes son **Images** y **Labels** para mostrar las imágenes de la cámara y otros indicadores. Las dimensiones de la ventana son **Height="770"** y **Width="770"**. Estas son diferentes a las del Canvas.

```
<Grid>
```

```
    <Canvas Name="MainCanvas" Width="640" Height="480"
  HorizontalAlignment="Center">
        <Image Name="Image" Width="640" Height="480"/>
        <Image Name="Pelota" Width="60" Height="60"
        Source="pelota.png"
        Canvas.Left="570"/>
    </Canvas>
</Grid>
```

Código en C#

Se abre un proyecto WPF nuevo y se agregan las referencias necesarias de Kinect. En el código de este proyecto se toma como base la plantilla para la cámara RGB, esta será editada para poder implementar la aplicación antes descrita. Después de crear el proyecto se agregan las bibliotecas: **Kinect**, **IO** y **Threading**. La biblioteca **Threading** se utiliza para implementar los **timers**.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

Las variables que se utilizan son las siguientes, así como las de estructuras que se han utilizado para detectar colisiones. Una de las variables que se tendría que describir es **iEstadoGlobal**, esta se ha creado para detectar la colisión entre la rodilla izquierda (estado 2) y derecha (estado 3) dado que el movimiento resultante de dicha colisión sería en direcciones diferentes. Se tendrían dos casos para las rodillas más otros dos, de los cuales, uno es en el que no hay colisiones y se sigue el movimiento inicial descendente (estado 1) y el otro es para inicializar todos los datos cuando cualquiera de los movimientos ha terminado (estado 0). Entonces, el flujo que se sigue en los estados de la variable **iEstadoGlobal** es el siguiente:

Figura 6.22

Algo importante a resaltar es la diferencia con el código del Kinect V1, ya que, en este caso, en lugar de utilizar el Stream para la cámara y el Skeleton, se cuenta un **FrameReader** con el cual se pueden leer datos de las distintas fuentes. Como en este ejemplo se requiere usar la cámara y Body se implementa el uso del **MultiSourceFrameReader**. Las variables para este ejemplo son las siguientes:

```
// FrameReader para recibir datos de distintos tipos
MultiSourceFrameReader multiReader;
private Body[] bodies = null; // Arreglo para recibir datos de Body
Body miBody; // Body a utilizar
Point joint_Point = new Point(); // Permite obtener los datos de un Joint
private WriteableBitmap imagen = null; //Se utiliza para generar la imagen
int iWidth = 0; // Ancho de la imagen
int iHeight = 0; // Alto de la imagen
byte[] cantidadPixeles; // Arreglo para recibir los datos de la cámara (pixeles)
Random numero = new Random(); // Generar numero random
DispatcherTimer timer; // Timer para producir la animación
int iEstadoGlobal = 0; //Determinar estado de movimiento
int iStepPixeles = 30; //Incremento en pixeles del movimiento
//Variables utilizadas para calcular posición X y Y en el tiro parabólico
```

```
double dXinicial = 0.0; //Posición X inicial
double dYinicial = 0.0; //Posición Y inicial
double dAngulo = 87.0; //Guarda el ángulo de tiro
double dTiempo = 0.0; //Guarda el tiempo de tiro transcurrido
double dStepTime = 0.8; //Incremento de tiempo para simular tiempo transcurrido
double dVelocidad = 90.0; //Velocidad constante en X
// Estructura que almacenará la información del objeto
struct Imagenes {
  public double dPosX;
  public double dPosY;
  public double dAncho;
  public double dAlto;
}
Imagenes pelota_1, rodilla_Der, rodilla_Izq;
```

Uno de los primeros métodos a modificar es el método **mainWindow**, aquí se obtienen las propiedades de las imágenes para las colisiones y se asignan a sus respectivas estructuras, también se elaboran las configuraciones iniciales del **timer**. El método queda de la siguiente forma:

```
public MainWindow() {
    InitializeComponent();
    // Realizar configuraciones e iniciar el Kinect
    Kinect_Config();
    // Asignar valores iniciales a las estructuras
    pelota_1.dPosX = (double)Pelota.GetValue(Canvas.LeftProperty); ;
    pelota_1.dPosY = (double)Pelota.GetValue(Canvas.TopProperty);
    pelota_1.dAlto = Pelota.Height;
```

```
        pelota_1.dAncho = Pelota.Width;
        rodilla_Der.dPosX = 0.0;
        rodilla_Der.dPosY = 0.0;
        rodilla_Der.dAlto = 5;
        rodilla_Der.dAncho = 5;
        rodilla_Izq.dPosX = 0.0;
        rodilla_Izq.dPosY = 0.0;
        rodilla_Izq.dAlto = 5;
        rodilla_Izq.dAncho = 5;
        // Configurar e iniciar el timer
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
        timer.Tick += new EventHandler(timer_Tick);
        timer.IsEnabled = true;
    }
```

Se pretende utilizar la **cámara RGB** y **Body**, esto quiere decir que se va a trabajar con dos tecnologías del Kinect. En el área que se ha reservado para los métodos que utilizan los datos del Kinect se necesita agregar el método **usarBody**. Este método es el único de esta área que se modifica, el método **usarCamara** queda sin cambios.

Las modificaciones que se realizan en el método **usarBody** son la obtención de los Joints a utilizar: la rodilla izquierda y la derecha.

```
Joint joint1 = miBody.Joints[JointType.KneeLeft];
Joint joint2 = miBody.Joints[JointType.KneeRight];
```

Después se evalúa que los joints estén trazados. Una vez corroborado lo anterior se obtienen las coordenadas de cada uno mediante el método **BodyPointToScreen** y se asignan las estructuras correspondientes a ambas rodillas. El método **usarBody** queda de la siguiente forma:

```
private void usarBody(Body miBody) {
  // Crear Joints para obtener las propiedade deseadas (Coordenadas)
```

```
        Joint joint1 = miBody.Joints[JointType.KneeLeft];
        Joint joint2 = miBody.Joints[JointType.KneeRight];
        // Verificar estado del Joint
        if (joint1.TrackingState == TrackingState.Tracked) {
            // Obtener coordenadas
            joint_Point = this.BodyPointToScreen(joint1);
            rodilla_Izq.dPosX = joint_Point.X;
            rodilla_Izq.dPosY = joint_Point.Y;
            // Obtener coordenadas
            joint_Point = this.BodyPointToScreen(joint2);
            rodilla_Der.dPosX = joint_Point.X;
            rodilla_Der.dPosY = joint_Point.Y;
            // Indicar Id de la persona que es trazada
            LID.Content = miBody.TrackingId;
        }
    }
```

Ya que se van a usar dos tecnologías del Kinect, en este ejemplo se utiliza **MultiSiurceFrameReader**, el cual permite leer datos de diferentes fuentes del Kinect. El método **Kinect_FrameReady** será un poco diferente a los vistos con anterioridad debido al uso de MultiSourceFrameReader. Uno de los primeros cambios al método es el parámetro que recibe, este es **MultiSourceFrameArrivedEventArgs e**. El método queda de la siguiente forma:

```
    private void Kinect_FrameReady(object sender, MultiSourceFrameArrivedEventArgs e)
    {
        …
    }
```

Ahora que se tienen los parámetros requeridos se verá el código necesario para esta

rutina el cual es una combinación de los códigos de la cámara y Body. Lo primero que se hace es abrir el *frame* que se recibe asignando los datos a una variable tipo **var**.

Mediante la variable **var** se pueden obtener los datos referentes a cada *frame* ya sea de la cámara o de Body. Entonces se abre el *frame* solicitando datos referentes a la cámara y si el resultado es **null** quiere decir que no había datos de la cámara. Si es diferente de **null** se procede de la misma forma que antes se han procesado los datos, tal y como se explicó en la plantilla de la cámara para Kinect V2.

Ahora que se han solicitado los datos de la cámara se procede a solicitar los datos de Body y, de igual forma, si el resultado es **null** quiere decir que no había datos. Si es diferente de **null** se procede de la misma forma que antes se han procesado los datos, tal y como se explicó en la plantilla de Body. El método queda de la siguiente forma:

```
private void Kinect_FrameReady(object sender, MultiSourceFrameArrivedEventArgs e){
    // Adquirir el Frame
    var reference = e.FrameReference.AcquireFrame();
    // Adquirir el ColorFrame
    using (var colorFrame = reference.ColorFrameReference.AcquireFrame()){
        // Verificar contenido de datos
        if (colorFrame != null) {
            if (colorFrame.RawColorImageFormat == ColorImageFormat.Bgra) {
                colorFrame.CopyRawFrameDataToArray(cantidadPixeles);
            }
            else {
                colorFrame.CopyConvertedFrameDataToArray(cantidadPixeles,
                    ColorImageFormat.Bgra);
            }
            // Llamar método que manipula los datos
```

```
            this.usarCamara();
        }
    }
    // Adquirir el BodyFrame
    using (var mibodyFrame = reference.BodyFrameReference.AcquireFrame()) {
        // Verificar contenido de datos
        if (mibodyFrame != null) {
            // Crear arreglo de Bodies y copiar datos recibidos en el mismo
            this.bodies = new Body[mibodyFrame.BodyCount];
            mibodyFrame.GetAndRefreshBodyData(this.bodies);
            // Seleccionar el primer Body o default
            miBody = (from trackBody in bodies where trackBody.LeanTrackingState ==
                    TrackingState.Tracked select trackBody).FirstOrDefault();
            if (miBody == null) {
                LID.Content = "0";
                return;
            }
            // Indicar ID de la persona trazada
            LID.Content = miBody.TrackingId;
            // Llamar método que manipula los datos
            this.usarBody(miBody);
        }
    }
```

```
}
```

Las configuraciones necesarias que se realizan para utilizar estas dos tecnologías están en el método **Kinect_Config** y es referente a las variables que reciben los datos y enlistan las rutinas con los eventos. Las configuraciones del método son las siguientes:

```
// Abrir el FrameReader para poder recibir los datos de las distintas fuentes
// expresadas como parámetros
multiReader = miKinect.OpenMultiSourceFrameReader(FrameSourceTypes.Color |
                                                  FrameSourceTypes.Body);
// Enlistar la rutina que se llama cuando hay datos disponibles
multiReader.MultiSourceFrameArrived += Kinect_FrameReady;
// Crear FrameDescription desde ColorFrameSource usando un formato Bgra
FrameDescription colorFrameDescription = this.miKinect.ColorFrameSource.
                                         CreateFrameDescription(ColorImageFormat.Bgra);
iWidth = colorFrameDescription.Width;
iHeight = colorFrameDescription.Height;
cantidadPixeles = new byte[iWidth * iHeight * ((PixelFormats.Bgr32.BitsPerPixel + 7)
                                               / 8)];
// Crear el bitmap a mostrar (Imagen que se visualiza)
this.imagen = new WriteableBitmap(colorFrameDescription.Width,
                                  colorFrameDescription.Height, 96.0, 96.0,
                                  PixelFormats.Bgr32, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
```

```
this.Image.Source = this.imagen;
```

Ahora se trabajarán los métodos nuevos, aquellos que se designan para el área que lleva el mismo nombre. El primer método es **timer_Tick**, cuya función principal es la implementación de los estados que se describieron al inicio. Los valores de cada estado y la descripción de las acciones en cada uno son las siguientes:

1. **Estado 0**: se asignan valores iniciales a la posición de la pelota, el tiempo utilizado en el tiro parabólico y se indica el siguiente estado (Estado 1).

2. **Estado 1**: se evalúa si ha existido alguna colisión. De haber colisión entre la pelota y la rodilla izquierda se toma la posición inicial de esta y se indica que el siguiente estado es el 2, si existe colisión entre la pelota y la rodilla derecha se toma la posición inicial de la pelota y se indica que el siguiente estado es el 3. Si no existe colisión se sigue el movimiento descendente de la pelota, pare esto se actualiza solo **TopProperty** de la imagen, y cuando finaliza el movimiento se indica que el siguiente estado es el 0.

3. **Estado 2**: se calculan las posiciones en **X** y **Y** del tiro parabólico para asignarse a las propiedades *Top* y *Left* de la imagen de la pelota y se actualiza el tiempo de cálculo. Cuando el movimiento finaliza se indica que el siguiente estado es el 0. El movimiento parabólico es hacia el lado derecho.

4. **Estado 3**: se calculan las posiciones en **X** y **Y** del tiro parabólico para asignarse a las propiedades *Top* y *Left* de la imagen de la pelota y se actualiza el tiempo de cálculo. Cuando el movimiento finaliza se indica que el siguiente estado es el 0. El movimiento parabólico es hacia el lado izquierdo.

El valor (**420**) contra el que se compara **pelota_1.dPosY** es un valor designado para indicar cuando la pelota aún se encuentra dentro de las dimensiones del canvas, si **pelota_1.dPosY** es mayor que este valor la imagen queda fuera, lo cual sirve como limitante para indicar el fin del movimiento. El código de **timer_Tick** es el siguiente:

```
void timer_Tick(object sender, EventArgs e) {
   switch (iEstadoGlobal) {
      case 0: // Asignar valores iniciales
            pelota_1.dPosX = numero.Next(200, 440);
            Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
            pelota_1.dPosY = 0.0;
            Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
            dTiempo = 0.0;
```

```
        iEstadoGlobal = 1;
        break;
    case 1: // Evaluar colisiones
        if (checarColision(pelota_1, rodilla_Izq)) {
            dXinicial = pelota_1.dPosX;
            dYinicial = pelota_1.dPosY;
            iEstadoGlobal = 2;
        }
        if (checarColision(pelota_1, rodilla_Der)) {
            dXinicial = pelota_1.dPosX;
            dYinicial = pelota_1.dPosY;
            iEstadoGlobal = 3;
        }
        if (pelota_1.dPosY < 420) {
            Pelota.SetValue(Canvas.TopProperty,  pelota_1.dPosY  +=
            iStepPixeles);
        }
        else { iEstadoGlobal = 0; }
        break;
    case 2: // Movimiento debido a la colisión con la rodilla izquierda
        if (pelota_1.dPosY < 420) {
            pelota_1.dPosY = dYinicial - altura();
            pelota_1.dPosX = dXinicial + alcance();
            Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
            Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
            dTiempo += dStepTime;
```

```
            }
            else { iEstadoGlobal = 0; }
            break;
        case 3: // Movimiento debido a la colisión con la rodilla derecha
            if (pelota_1.dPosY < 420) {
                pelota_1.dPosY = dYinicial - altura();
                pelota_1.dPosX = dXinicial - alcance();
                Pelota.SetValue(Canvas.TopProperty, pelota_1.dPosY);
                Pelota.SetValue(Canvas.LeftProperty, pelota_1.dPosX);
                dTiempo += dStepTime;
            }
            else{ iEstadoGlobal = 0; }
            break;
    }
}
```

Los métodos son **altura** y **alcance** son necesarios para la generación del movimiento parabólico y son la implementación de las fórmulas que se dieron al inicio. El código para estos métodos es el siguiente:

```
private double altura() {
  double alturaY;
  double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en radianes
                                               // Math.Sin así lo recibe
  alturaY = (dVelocidad * Math.Sin(radianes) * dTiempo) –
            (9.81 * Math.Pow(dTiempo, 2) / 2);
  return alturaY;
}
```

```
private double alcance()
{
  double alcanceX;
  double radianes = Math.PI * dAngulo / 180.0; // Expresar ángulo en radianes
                                               // Math.Sin así lo recibe
  alcanceX = dVelocidad * Math.Cos(radianes) * dTiempo;
  return alcanceX;
}
```

Los métodos restantes a agregar vienen a continuación, estos son necesarios para la detección de colisiones y la obtención de las coordenadas de los joints, ya se ha trabajado anteriormente con ellos:

```
private bool checarColision(Imagenes img1, Imagenes img2) {
  if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de ob2
      return false;
  if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de ob2
      return false;
  if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha ob2
      return false;
  if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo ob2
      return false;
  return true;
}
private Point BodyPointToScreen(Joint miJoint) {
  // Convertir un punto a "Depth Space"
```

```
        DepthSpacePoint depthSpacePoint = miKinect.CoordinateMapper.
                                    MapCameraPointToDepthSpace(miJoint.Position);
        return new Point(depthSpacePoint.X, depthSpacePoint.Y);
    }
```

Con lo anterior se tiene listo el procesado de los datos. Ahora para finalizar se editará el método **Window_Closing**, el cual se encarga de tomar las acciones necesarias cuando se finaliza la aplicación. La única modificación es sobre el **FrameReader**, lo que se hace es "liberarlo" para que pueda ser utilizado por otros programas, el método es **Dispose**. El método **Window_Closing** queda de la siguiente manera:

```
    private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e) {
        /* ------------------- Configuración del Kinect ------------------- */
        if (this.multiReader != null)
        {
            // Disponer BodyFrameReader
            this.multiReader.Dispose();
            this.multiReader = null;
        }
        /* ------------------------------------------------------------ */
        if (this.miKinect != null)
        {
            this.miKinect.Close();
            this.miKinect = null;
        }
    }
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. La pelota se mueve

en forma descendente, cuando interactúa con las rodillas se puede ver que el movimiento del balón cambia a un movimiento parabólico semejando un golpe.

Figura 6.23

Capítulo 7. Captar el movimiento de varias personas

Las aplicaciones desarrolladas hasta el momento suponen que solo una persona se encontrará frente al Kinect por lo que es probable que el programa muestre resultados extraños, en caso de detectar a más de una. Por ejemplo, ¿qué sucedería si dos personas se colocan frente al Kinect cuando se está ejecutando un programa que desplaza una esfera a través de la mano derecha? Es probable que se mueva la esfera cuando cualquiera de las dos mueva e incluso tenga un comportamiento errático si lo llegasen a hacer al mismo tiempo.

Este capítulo está dedicado al desarrollo de aplicaciones que permitan seleccionar y seguir el movimiento de una o más personas.

7.1 Seleccionar a la persona que se va a mapear

Las plantillas para ambas versiones del Kinect que se han empleado, incluyen el método **Kinect_FrameReady** el cual se encarga de adquirir los datos que este envía, tomar el primer **Skeleton (Body)** que se encuentre y enviarlo al método **usarSkeleton (usarBody)** para que se procese el contenido. En otras palabras, estos métodos no realizan algún proceso para elegir a la persona y mucho menos para continuar detectándola aunque más gente se encuentre junto a ella. Si se desea que la aplicación elija la persona que se va a mapear, se deben realizar algunas modificaciones tanto al método **Kinect_FrameReady** como a **usarSkeleton (usarBody)** tal y como se explica a continuación.

Cada **Skeleton (Body)** cuenta con un **TrackingId** que lo identifica de forma única. Este número se le asigna a la persona detectada y existe mientras ella se encuentre frente al Kinect. Para almacenar este número, se agrega en la parte superior del programa una variable la cual se inicializa con cero para indicar que aún no se ha realizado la selección.

Kinect versión 1

```
public partial class MainWindow : Window
{
  int Skeleton_ID = 0; //Almancena el ID del Skeleton recibido
```

Kinect versión 2

```
public partial class MainWindow : Window
{
  ulong iBodyID = 0; //Almancena el ID del Body recibido
```

El nuevo contenido del método **Kinect_FrameReady** inicia de la misma manera que el anterior al leer el *frame* y almacenar la información de cada una de las personas detectadas. La modificación comienza determinando si ya se ha seleccionado o no el sujeto a seguir. Si la variable para almacenar el número de identificación contiene un cero, entonces aún no se ha realizado la elección por lo que procede a seleccionar la primera persona detectada, toma su **TrackingId** y lo guarda en la variable. Por el contrario, si la variable ya contiene el identificador de una persona, solo procederá a buscar su información. En caso de ya no detectarla, es decir que la persona ya no esté frente al Kinect, se vuelven a iniciar los valores para seleccionar nuevamente.

El método **usarSkeleton** para el Kinect V1 no requiere cambios ya que solamente recibe la información del esqueleto seleccionado. Sin embargo, el método **usarBody** sufre algunos ajustes para que solamente procese el **Body** seleccionado.

Kinect versión 1

```
//Este método se ejecuta cada vez que el Kinect tiene un Frame.
//Se encarga de tomar la información de los esqueletos y pasarla a un arreglo.
//Posteriormente cada esqueleto es enviado al método usarSkeleton para realizar
//algún proceso.
private void Kinect_FrameReady(object sender, SkeletonFrameReadyEventArgs e)
    {
      //Variable para almacenar la información de los esqueletos
      Skeleton[] skeletons = new Skeleton[0];
      Skeleton skeleton;
      //Abre el frame recibido y copia su contenido en la variable skeletons
```

```
using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
{
    if (skeletonFrame != null)
    {
        skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];
        skeletonFrame.CopySkeletonDataTo(skeletons);
    }
}
//Selección del esqueleto
switch (Skeleton_ID)
{
    //Dado que no hay esqueleto seleccionado se realiza la elección
    case 0:
        //Obtiene el primer Skeleton que se haya mapeado
        skeleton = (from trackSkeleton in skeletons where
            trackSkeleton.TrackingState == SkeletonTrackingState.Tracked
            select trackSkeleton).FirstOrDefault();
        if (skeleton == null)
        {
          LID.Content = "0";
          return;
        }
        //Obtiene el ID del Skeleton
        Skeleton_ID = skeleton.TrackingId;
```

```
                //Modifica la propiedad AppChoosesSkeletons para que la aplicación
                //pueda seleccionar el esqueleto
                this.miKinect.SkeletonStream.AppChoosesSkeletons = true;
                //Indica al Kinect cuál será el esqueleto a seguir
                this.miKinect.SkeletonStream.ChooseSkeletons(Skeleton_ID);
                this.usarSkeleton(skeleton);
                LID.Content = Skeleton_ID;
                break;
        //Obtiene la información del esqueleto seleccionado
        default:
                //Obtiene la información del esqueleto con el Skeleton_ID seleccionado
                skeleton = (from trackSkeleton in skeletons where
                        trackSkeleton.TrackingState == SkeletonTrackingState.Tracked
                        && trackSkeleton.TrackingId == Skeleton_ID select
                        trackSkeleton).FirstOrDefault();
                if (skeleton == null)
                {
                    //Cuando la persona ya no se encuentra frente al Kinect,
                    //se regresa todo al estado inicial para que se vuelva a realizer
                    //la selección
                    Skeleton_ID = 0;
```

```
                    this.miKinect.SkeletonStream.AppChoosesSkeletons =
                    false;
                    return;
                }
                //Envia el Skelton a usar
                this.usarSkeleton(skeleton);
                break;
            }
        }
```

```
    //Recibe la información de un esqueleto y la utiliza para hacer que una
elipse
    //denominada Puntero siga el movimiento de la mano derecha
    private void usarSkeleton(Skeleton skeleton)
    { //Extrae la información del Joint de la mano derecha
        Joint joint1 = skeleton.Joints[JointType.HandRight];
        // Si el Joint está listo obtener las coordenadas
        if (joint1.TrackingState == JointTrackingState.Tracked)
        {
            // Obtiene las coordenadas (x, y) del Joint
            joint_Point = this.SkeletonPointToScreen(joint1.Position);
            dMano_X = joint_Point.X;
            dMano_Y = joint_Point.Y;
            //Emplea las coordenadas del Joint para mover la elipse
            Puntero.SetValue(Canvas.TopProperty, dMano_Y);
            Puntero.SetValue(Canvas.LeftProperty, dMano_X);
```

```
            // Obtiene el Id de la persona mapeada
            LID.Content = skeleton.TrackingId;
        }
    }
```

Kinect versión 2

```
    //Este método se ejecuta cada vez que el Kinect tiene un Frame.
    //Se encarga de tomar la información de los cuerpos y pasarla a un arreglo.
    //Posteriormente cada cuerpo es enviado al método usarBody para realizar
    //algún proceso.
    private void Kinect_FrameReady(object sender, BodyFrameArrivedEventArgs e)
    {
        Body miBody;
        //Adquiere un BodyFrame
        using (BodyFrame mibodyFrame = e.FrameReference.AcquireFrame())
        {
            //Las siguientes instrucciones se realizan si hay personas frente al Kinect
            if (mibodyFrame != null) {
                //Obtiene la información de los Body
                this.bodies = new Body[mibodyFrame.BodyCount];
                mibodyFrame.GetAndRefreshBodyData(this.bodies);
                //Selección del Body
                switch (iBodyID) {
                    //Dado que no hay un Body seleccionado se realiza la
```

```
elección
case 0:
    //Obtiene el primer Body trazado
    miBody = (from trackBody in bodies where
        trackBody.LeanTrackingState == TrackingState.Tracked
        select trackBody).FirstOrDefault();
    if (miBody == null) { return; }
    //Obtiene el número de identificación del Body seleccionado
    iBodyID = miBody.TrackingId;
    //Envia a usarBody todos Boodies detectados por el Kinect
    //esto será útil en aplicaciones que requieren la selección
    //de más de un Body
    foreach (Body mibody in bodies) {
        if (mibody.LeanTrackingState == TrackingState.Tracked) {
        //Envía el Body a usarBody
        this.usarBody(mibody);
        }
    }
    break;
//Envia a usarBody todos Boodies detectados por el Kinect
default:
```

```
                    if (bodies.Length != 0) {
                        foreach (Body mibody in bodies) {
                            if (mibody.LeanTrackingState == TrackingState.Tracked) {
                            //Enviar Body a usar
                            this.usarBody(mibody);
                            }
                        }
                    }
                break;
            }
        }
    }
}
```

```
//Recibe la información de un cuerpo y la utiliza para hacer que una elipse
//denominada Puntero siga el movimiento de la mano derecha de la persona cuyo
//ID está almacenado en la variable iBodyID
private void usarBody(Body miBody)
{
  // Obtiene el Joint de la parte del cuerpo deseada
  Joint miJoint = miBody.Joints[JointType.HandRight];
  // Verifica el estado del Joint
  if (miJoint.TrackingState == TrackingState.Tracked)
  {
```

```
            // Obtiene las coordenadas del Joint
            joint_Point = this.BodyPointToScreen(miJoint);
            dMano_X = joint_Point.X;
            // Verificar si el ID del Body es igual al del Body selecionado.
            if (miBody.TrackingId == iBodyID)
            { // Realiza el movimiento del puntero
            Puntero.SetValue(Canvas.LeftProperty, dMano_X);
            }
            else
            {
                  // Colocar aquí las instrucciones para el caso de otro Body
            }
      }
}
```

Si se ejecuta el programa de la plantilla con las modificaciones antes mencionadas se puede observar que, sin importar el número de personas presentes, una vez que se ha seleccionado una no habrá interferencia de las demás.

También es importante mencionar que existen diversas formas de realizar la selección de la persona. Se decidió utilizar el **TrackingId** de la primera persona detectada. Otra forma es ejemplificada en MSDN (**Microsoft Developer Network**) donde la elección se toma con base a la distancia de las personas con el Kinect.

7.2 Seleccionar a dos personas

En ocasiones se requiere diseñar aplicaciones en las que es necesario realizar el mapeo de dos personas. A continuación se muestra una manera de hacerlo con el **TackingId** correspondiente a cada **Skeleton (Body**) seleccionado.

Para elegir a dos personas nuevamente será necesario modificar el método **Kinect_FrameReady** y **usarSkeleton (usarBody)** para que ahora realicen el filtrado de la información empleando el **TackingId** de los dos seleccionados. El proceso a seguir es

similar al de la selección de una persona solo que en esta ocasión se deberá almacenar información de dos números de identificación y la cantidad de casos a seguir para determinar el estatus de dichos números aumenta a tres, tal y como se explica en los comentarios de los siguientes códigos según la versión del Kinect.

Kinect versión 1

```
int iSkeleton_ID1 = 0; //ID para la persona 1

int iSkeleton_ID2 = 0; //ID para la persona 2

int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
```

```
//Este método se ejecuta cada vez que el Kinect tiene un Frame.

//Se encarga de tomar la información de los esqueletos y pasarla a un arreglo.

//Posteriormente cada esqueleto es enviado al método usarSkeleton para realizar

//algún proceso.

private void Kinect_FrameReady(object sender, SkeletonFrameReadyEventArgs e) {

    //Variable para almacenar la información de los esqueletos

    Skeleton[] skeletons = new Skeleton[0];

    Skeleton skeleton;

    //Abre el frame recibido y copia su contenido en la variable skeletons

    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame()) {

        if (skeletonFrame != null) {

            skeletons = new Skeleton[skeletonFrame.SkeletonArrayLength];

            skeletonFrame.CopySkeletonDataTo(skeletons);

        }

    }
```

```
//Selección de los esqueletos a seguir
switch (iCasos_ID){
    case 0:
        //Caso inicial. Se ejecuta cuando se selecciona por primera vez
        //los esqueletos a seguir
        //Obtiene el ID del primer esqueleto
        skeleton = (from trackSkeleton in skeletons where
            trackSkeleton.TrackingState ==
            SkeletonTrackingState.Tracked
            select trackSkeleton).FirstOrDefault();
        if (skeleton == null){
            // No se encuentran Skeleton's
            return;
        }
        iSkeleton_ID1 = skeleton.TrackingId;
        //Obtiene el ID del segundo Skeleton que sea diferente del
        primero
        skeleton = (from trackSkeleton in skeletons where
            trackSkeleton.TrackingState ==
            SkeletonTrackingState.Tracked
            && trackSkeleton.TrackingId != iSkeleton_ID1 select
            trackSkeleton).FirstOrDefault();
        if (skeleton == null) {
            // No se encuentran Skeleton's
            return;
        }
```

```
            iSkeleton_ID2 = skeleton.TrackingId;
            //Enviar la información de los dos esqueletos para su procesamiento
            foreach (Skeleton skel in skeletons) {
                if (skel.TrackingState == SkeletonTrackingState.Tracked){
                    //Enviar el Skelton a usar
                    this.usarSkeleton(skel);
                }
            }
            //Pasar al siguiente caso
            iCasos_ID = 1;
            break;
        case 1:
            //Enviar la información de los dos esqueletos para su procesamiento
            if (skeletons.Length != 0) {
            foreach (Skeleton skel in skeletons) {
                if (skel.TrackingState == SkeletonTrackingState.Tracked)
                {
                    //Enviar el Skelton a usar
                    this.usarSkeleton(skel);
                }
            }
            }
        break;
    default:
```

```
//Este caso se ejecuta cuando se detecta que una de las personas
//seleccionadas ya no está frente al Kinect.
if (skeletons.Length != 0) {
    //Busca la información del primer esqueleto seleccionado. Si no
    lo
    //encuentra busca otro esqueleto que sea diferente al de la
    persona 2
    //y se envía la información para su procesamiento.
    skeleton = (from trackSkeleton in skeletons where
            trackSkeleton.TrackingState ==
            SkeletonTrackingState.Tracked
            && trackSkeleton.TrackingId == iSkeleton_ID1 select
            trackSkeleton).FirstOrDefault();
    if (skeleton == null) {
        skeleton = (from trackSkeleton in skeletons where
                trackSkeleton.TrackingState ==
                SkeletonTrackingState.Tracked &&
                trackSkeleton.TrackingId != iSkeleton_ID2 select
                trackSkeleton).FirstOrDefault();
        if (skeleton != null) {
        iSkeleton_ID1 = skeleton.TrackingId;
        this.usarSkeleton(skeleton);
        }
        else { return; }
    }
    else {
```

```
        this.usarSkeleton(skeleton);
    }
    //Busca la información del segundo esqueleto seleccionado. Si no lo
    //encuentra busca otro esqueleto que sea diferente al de la persona 1
    //y se envía la información para su procesamiento.
    skeleton = (from trackSkeleton in skeletons where
                trackSkeleton.TrackingState ==
                SkeletonTrackingState.Tracked
                && trackSkeleton.TrackingId == iSkeleton_ID2 select
                trackSkeleton).FirstOrDefault();
    if (skeleton == null) {
        skeleton = (from trackSkeleton in skeletons where
                    trackSkeleton.TrackingState ==
                    SkeletonTrackingState.Tracked &&
                    trackSkeleton.TrackingId != iSkeleton_ID1 select
                    trackSkeleton).FirstOrDefault();
    if (skeleton != null) {
                iSkeleton_ID2 = skeleton.TrackingId;
                this.usarSkeleton(skeleton);
        }
        else { return; }
    }
    else {
        this.usarSkeleton(skeleton);
```

```
                }
            }
                //Restablecer el caso 1 para continuar con el procesamiento.
                iCasos_ID = 1;
                break;
            }
        }
        //Recibe la información de un esqueleto y la utiliza.
        //Este método se ha dejado incompleto. En los comentarios se indica los lugares
        //En lo que se debe poner el código requerido para procesar cada uno de ellos
        private void usarSkeleton(Skeleton skeleton)
        {
            Joint joint1 = skeleton.Joints[JointType.HandRight];
            if (joint1.TrackingState == JointTrackingState.Tracked)
            {
                //PERSONA 1
                if (skeleton.TrackingId == iSkeleton_ID1)
                {
                    //Colocar aquí las instrucciones para procesar el
                    //esqueleto de la Persona 1
                    return;
                }
                //PERSONA 2
                if (skeleton.TrackingId == iSkeleton_ID2)
```

```
            {
                //Colocar aquí las instrucciones para procesar el
                //esqueleto de la Persona 2
                return;
            }
            //Se establece el Caso 2 para que se realice nuevamente la selección
            //de otra persona ya que no se encontró información de ninguna de las dos
            //personas seleccionadas (una de las personas se salió)
            iCasos_ID = 2;
        }
    }
```

Kinect versión 2

```
ulong uBody_ID1 = 0; //ID para la persona 1
ulong uBody_ID2 = 0; //ID para la persona 2
int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
private Body[] bodies = null; //Arreglo que recibe los Body's
```

```
//Este método se ejecuta cada vez que el Kinect tiene un Frame.
//Se encarga de tomar la información de los cuerpos y pasarla a un arreglo.
//Posteriormente cada cuerpo es enviado al método usarBody para realizar
//algún proceso.
private void Kinect_FrameReady(object sender, BodyFrameArrivedEventArgs e)
```

```
{
    //Adquiere un BodyFrame
    Body miBody;
    using (BodyFrame mibodyFrame = e.FrameReference.AcquireFrame())
    {
        //Las siguientes instrucciones se realizan si hay personas frente al Kinect
        if (mibodyFrame != null)
        {
            //Obtiene la información de los Body
            this.bodies = new Body[mibodyFrame.BodyCount];
            mibodyFrame.GetAndRefreshBodyData(this.bodies);
        }
    }
//Selección de los Bodys a seguir
switch (iCasos_ID)
{
    case 0:
        //Caso inicial. Se ejecuta cuando se selecciona por primera vez
        //los Bodys a seguir
        //Obtiene el ID del primer body
        miBody = (from trackBody in bodies where
            trackBody.LeanTrackingState == TrackingState.Tracked
            select trackBody).FirstOrDefault();
        if (miBody == null)
```

```
{
        // No se encuentran Body's
        return;
}
iBody_ID1 = miBody.TrackingId;
//Obtiene el ID del segundo body que sea diferente del primero
miBody = (from trackBody in bodies where
        trackBody.LeanTrackingState == TrackingState.Tracked
        && trackBody.TrackingId != iBody_ID1 select
        trackBody).FirstOrDefault();
if (miBody == null)
{
        // No se encuentran Body's
        return;
}
iBody_ID2 = miBody.TrackingId;
//Enviar la información de los dos bodys para su procesamiento
foreach (Body mibody in bodies)
{
        if (mibody.LeanTrackingState == TrackingState.Tracked)
        {
            this.usarBody(mibody);
        }
}
//Pasar al siguiente caso
```

```
            iCasos_ID = 1;
            break;
            case 1:
                //Enviar la información de los dos esqueletos para su
                procesamiento
                if (bodies.Length != 0)
                {
                    foreach (Body mibody in bodies)
                    {
                        if (mibody.LeanTrackingState ==
                        TrackingState.Tracked)
                        {
                                this.usarBody(mibody);
                        }
                    }
                }
                break;
        default:
            //Este caso se ejecuta cuando se detecta que una de las personas
            //seleccionadas ya no está frente al Kinect.
            if (bodies.Length != 0)
            {
            //Busca la información del primer body seleccionado. Si no lo
            //encuentra busca otro que sea diferente al de la persona 2
            //y se envía la información para su procesamiento.
            miBody = (from trackBody in bodies where
```

```
            trackBody.LeanTrackingState == TrackingState.Tracked
            && trackBody.TrackingId == iBody_ID1 select
            trackBody).FirstOrDefault();
if (miBody == null)
{
        miBody = (from trackBody in bodies where
                trackBody.LeanTrackingState                    ==
                TrackingState.Tracked
                && trackBody.TrackingId != iBody_ID2 select
                trackBody).FirstOrDefault();
        if (miBody != null)
        {
            iBody_ID1 = miBody.TrackingId;
            this.usarBody(miBody);
        }
        else { return; }
}
else
{
        this.usarBody(miBody);
}
//Busca la información del segundo body seleccionado. Si no lo
//encuentra busca otro que sea diferente al de la persona 1
//y se envía la información para su procesamiento.
miBody = (from trackBody in bodies where
            trackBody.LeanTrackingState == TrackingState.Tracked
```

```
                    && trackBody.TrackingId == iBody_ID2 select
                    trackBody).FirstOrDefault();
            if (miBody == null)
            {
                miBody = (from trackBody in bodies where
                        trackBody.LeanTrackingState ==
                        TrackingState.Tracked
                        && trackBody.TrackingId != iBody_ID1 select
                        trackBody).FirstOrDefault();
                if (miBody != null)
                {
                    iBody_ID2 = miBody.TrackingId;
                    this.usarBody(miBody);
                }
                else { return; }
            }
            else
            {
                this.usarBody(miBody);
            }
        }
        //Restablecer el caso 1 para continuar con el procesamiento.
        iCasos_ID = 1;
        break;
    }
}
```

```
    //Recibe la información de un body y la utiliza.
    //Este método se ha dejado incompleto. En los comentarios se indica los lugares
    //En lo que se debe poner el código requerido para procesar cada uno de ellos
    private void usarBody(Body miBody)
    {
        Joint joint1 = miBody.Joints[JointType.HandRight];
        if (joint1.TrackingState == TrackingState.Tracked)
        {
            //PERSONA 1
            if (miBody.TrackingId == iBody_ID1)
            {
                //Colocar aquí las instrucciones para procesar datos de la persona 1
                return;
            }
            //PERSONA 2
            if (miBody.TrackingId == iBody_ID2)
            {
                //Colocar aquí las instrucciones para procesar datos de la persona 2
                return;
            }
                //Se establece el Caso 2 para que se realice nuevamente la selección
```

```
            //de otra persona ya que no se encontró información de ninguna
            de las 2 dos
            //personas seleccionadas (una de las personas se salió)
            iCasos_ID = 2;
        }
    }
```

7.3 Ejemplo: atrapar monedas (Kinect V1)

Esta aplicación es un juego en el que se registra la cantidad de monedas que atrapa cada uno de los dos participantes. Para esto se muestran en la pantalla dos calderos (uno para cada jugador) para atrapar las monedas. Para ser justos en la competencia, los calderos no se moverán a menos que las dos personas se encuentren frente al Kinect. Los jugadores mueven los calderos empleando su mano derecha.

Requerimientos

- Proyecto **WPF**.
- Plantilla V1 (configuración Skeleton).
- Imágenes: fondo, monedas y calderos.

Recursos

Para esta aplicación se requiere la imagen de fondo para dar ambientación al juego. También se necesita una imagen que represente el recipiente para atrapar un objeto (caldero) y otra para el objeto a atrapar (moneda). Se sugiere que sean tipo png para que sus fondos sean transparentes.

Código en XAML

La ventana de la aplicación tiene un Grid el cual contiene el Canvas con las imágenes a emplear y las etiquetas que se utilizan para mostrar la cantidad de monedas atrapadas por cada jugador. Dentro del Canvas, se incluye la imagen del fondo, una moneda y un caldero para cada uno de los jugadores. Las etiquetas que se muestran en la parte superior de la ventana indicarán los puntos acumulados para cada jugador.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
Grid.Row="1"
        Background="Azure" HorizontalAlignment="Center">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480"
        Width="640" Source="castillo.png"/>
    <Image Name="moneda1" Canvas.Left="10" Canvas.Top="10"
    Height="40"
        Width="40" Source="Moneda.png"/>
    <Image Name="moneda2" Canvas.Left="580" Canvas.Top="10"
    Height="40"
        Width="40" Source="Moneda.png"/>
    <Image Name="Puntero1" Canvas.Left="143" Canvas.Top="250"
    Height="60"
        Width="60" Source="caldero.png"/>
    <Image Name="Puntero2" Canvas.Left="463" Canvas.Top="250"
    Height="60"
        Width="60" Source="caldero.png"/>
  </Canvas>
  <Label x:Name="label" Content="Puntos: " FontSize="15"
FontWeight="Bold"
        Background="Bisque" HorizontalAlignment="Left"
        VerticalAlignment="Top"
        Margin="61,72,0,0"/>
  <Label x:Name="Jugador1" Content="0" FontSize="15"
HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="137,72,0,0"/>
```

```
<Label x:Name="label2" Content="Puntos: " FontSize="15" FontWeight="Bold"
        Background="Bisque" HorizontalAlignment="Left" VerticalAlignment="Top"
        Margin="390,72,0,0"/>
<Label x:Name="Jugador2" Content="0" FontSize="15" HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="466,72,0,0"/>
</Grid>
```

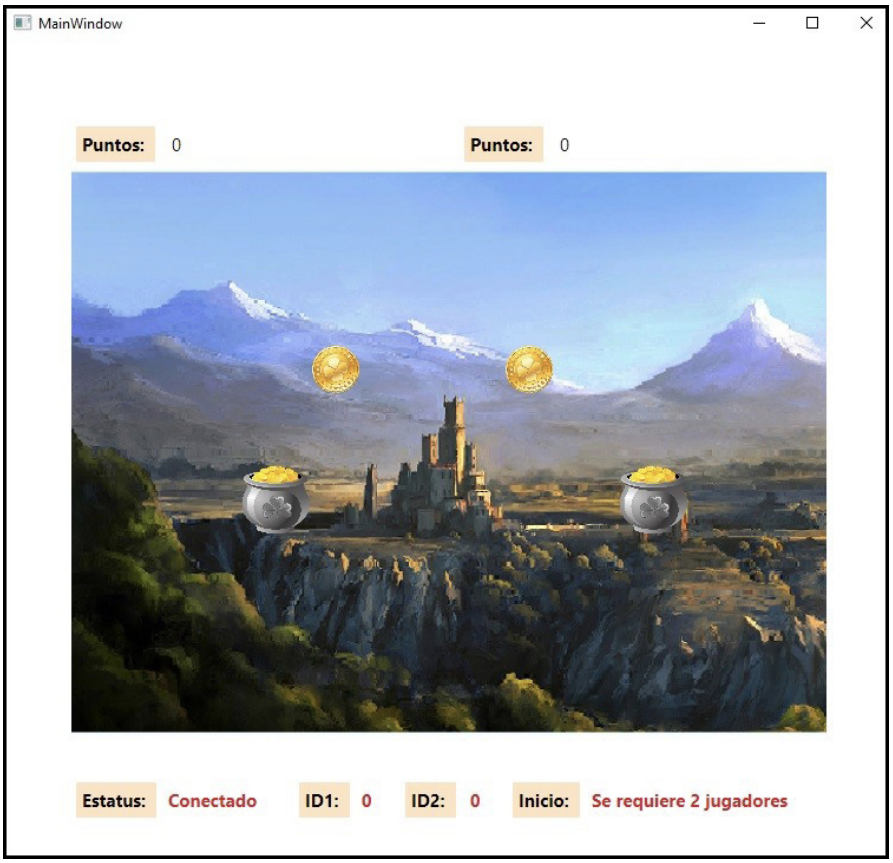


Figura 7.1. Muestra de cómo se podría ver el ejemplo.

Código en C#

Esta aplicación toma como base la plantilla para **Skeleton** a la cual se le debieron realizar las modificaciones para la selección de dos personas (métodos **Kinect_FrameReady y usarSkeleton**).

El programa emplea las siguientes bibliotecas.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

En la parte superior del programa, se deben incluir las variables tanto para la selección de las personas, como las que se requieren para el manejo de colisiones y el conteo de puntos acumulados por jugador.

```
double dMano_X; //Representa la coordenada X de la mano
Point joint_Point = new Point(); //Permite obtener los datos del Joint
int iSkeleton_ID1 = 0; //ID primera persona
int iSkeleton_ID2 = 0; //ID segunda persona
int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
Random numero = new Random(); //Generador de números al azar
DispatcherTimer timer;
int iStep = 25; //Píxeles a avanzar para simular moviento en monedas
int iPuntos1 = 0; //Contador puntos primera persona
int iPuntos2 = 0; //Contador puntos segunda persona
// Estructura que almacenará la información de objetos que colisionan
struct Imagenes
{
   public double dPosX;
   public double dPosY;
   public double dAncho;
   public double dAlto;
}
//Objetos que colisionan
```

```
Imagenes moneda_1, moneda_2, caldero_1, caldero_2;
```

Las modificaciones al método **mainWindow** solo son segmentos de código en el cual se dan valores iniciales a los objetos creados y se configura el **timer**, el método queda de la siguiente forma:

```
public MainWindow()
{
  InitializeComponent();
  //Realiza configuraciones e iniciar el Kinect
  Kinect_Config();
  //Colocar la moneda 1 de lado izquierdo de la parte superior de la ventana
  //y almacena la información requerida para el manejo de la colisión
  moneda_1.dPosX = numero.Next(15, 260);
  moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
  moneda_1.dPosY = 0.0;
  moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
  moneda_1.dAncho = moneda1.Width;
  moneda_1.dAlto = moneda1.Height;
  //Actualizar la información que no se modifica del caldero 1
  caldero_1.dAncho = Puntero1.Width;
  caldero_1.dAlto = Puntero1.Height;
  //Colocar la moneda 2 de lado derecho de la parte superior de la ventana
  //y almacena la información requerida para el manejo de la colisión
  moneda_2.dPosX = numero.Next(330, 580);
  moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
  moneda_2.dPosY = 0.0;
```

```
        moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
        moneda_2.dAncho = moneda2.Width;
        moneda_2.dAlto = moneda2.Height;
        //Actualizar la información que no se modifica del caldero 2
        caldero_2.dAncho = Puntero2.Width;
        caldero_2.dAlto = Puntero2.Height;
        //Configura e inicia el timer
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
        timer.Tick += new EventHandler(timer_Tick);
        timer.IsEnabled = true;
    }
```

Las instrucciones que se deben añadir al método **usarSkeleton** son las que se requieren para colocar el caldero en la posición de la mano derecha del jugador correspondiente. El jugador solo puede mover caldero horizontalmente. Para evitar que un jugador invada el área de su contrincante, se realiza una división imaginaria de la ventana de tal manera que el jugador 1 solo puede mover su caldero si la posición de su mano en el eje x es menor a 261. De la misma forma, el jugador 2 puede mover su caldero cuando la posición de su mano en el mismo eje x es mayor a 321.

```
    //Recibe la información de un esqueleto y la utiliza para colocar la imagen
    //de los calderos en la mismo posición que la mano derecha del jugador.
    private void usarSkeleton(Skeleton skeleton)
    {
        Joint joint1 = skeleton.Joints[JointType.HandRight];
        if (joint1.TrackingState == JointTrackingState.Tracked)
        {
            joint_Point = this.SkeletonPointToScreen(joint1.Position);
```

```
        dMano_X = joint_Point.X;
        //PERSONA 1
        if (skeleton.TrackingId == iSkeleton_ID1)
        { //Coloca la imagen en la posición de la mano derecha siempre y cuando
            //esta se encuentre a la izquierda de la ventana
            if (dMano_X < 261)
                Puntero1.SetValue(Canvas.LeftProperty, dMano_X);
            return;
            }
            //PERSONA 2
            if (skeleton.TrackingId == iSkeleton_ID2)
            { //Coloca la imagen en la posición de la mano derecha siempre y cuando
                //esta se encuentre a la derecha de la ventana
                if (dMano_X > 321)
                    Puntero2.SetValue(Canvas.LeftProperty, dMano_X);
                return;
            }
            //Se establece el Caso 2 para que se realice nuevamente la selección
            //de otra persona ya que no se encontró información de ninguna de las dos
            //personas seleccionadas (una de las personas se salió)
            iCasos_ID = 2;
        }
}
```

Otro método que se debe añadir al proyecto es el **timer_Tick**, que se usa para crear la animación de las monedas y contar de monedas atrapadas por cada jugador. Las monedas saldrán de la parte superior de la ventana y bajarán a cierta velocidad hasta llegar al fondo de la ventana (**dPosY = 420**) donde desaparecerán para mostrarse nuevamente en la parte superior en otra posición con respecto al eje **x**. Cuando el jugador haga que su caldero choque con la moneda, además de acreditarle un punto, hará que eseta realice el mismo proceso que cuando llega al final.

```
//Método que realiza la animación de la aplicación, moviendo las monedas y
//llevando el control de los puntos ganados por cada uno de los jugadores
void timer_Tick(object sender, EventArgs e) {
  //Obtener coordenadas de la Moneda y Puntero (caldero) del Jugador 1
  moneda_1.dPosX = (double)moneda1.GetValue(Canvas.LeftProperty);
  moneda_1.dPosY = (double)moneda1.GetValue(Canvas.TopProperty);
  caldero_1.dPosX = (double)Puntero1.GetValue(Canvas.LeftProperty);
  caldero_1.dPosY = (double)Puntero1.GetValue(Canvas.TopProperty);
  //Obtener coordenadas de la Moneda y Puntero (caldero) del Jugador 2
  moneda_2.dPosX = (double)moneda2.GetValue(Canvas.LeftProperty);
  moneda_2.dPosY = (double)moneda2.GetValue(Canvas.TopProperty);
  caldero_2.dPosX = (double)Puntero2.GetValue(Canvas.LeftProperty);
  caldero_2.dPosY = (double)Puntero2.GetValue(Canvas.TopProperty);
  //JUGADOR 1: El proceso se realiza solamente si la moneda del jugador
  //está aún dentro de la ventana
  if (moneda_1.dPosY < 420) {
     if (checarColision(moneda_1, caldero_1)) {
        //Cuando hay colisión se coloca la moneda en la parte superior de la
        //ventana en una posición al azar en el eje x
```

```
            moneda_1.dPosX = numero.Next(15, 260);
            moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
            moneda_1.dPosY = 0.0;
            moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
            //Acredita un punto al jugador y despliega el resultado en la
            ventana
            iPuntos1++;
            Jugador1.Content = iPuntos1.ToString();
        }
        else { //Como no hay colisión solamente mueve la moneda
            moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY +=
            iStep);
        }
    }
    else { //Coloca la moneda en la parte superior de la ventana
        moneda_1.dPosX = numero.Next(15, 260);
        moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
        moneda_1.dPosY = 0.0;
        moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
    }
    //JUGADOR 2: El proceso se realiza solamente si la moneda del
jugador
    //está aún dentro de la ventana
    if (moneda_2.dPosY < 420) {
        //Cuando hay colisión se coloca la moneda en la parte superior de la
        //ventana en una posición al azar en el eje x
```

```
            if (checarColision(moneda_2, caldero_2)) {
                moneda_2.dPosX = numero.Next(330, 580);
                moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
                moneda_2.dPosY = 0.0;
                moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
                //Acredita un punto al jugador y despliega el resultado en la
                ventana
                iPuntos2++;
                Jugador2.Content = iPuntos2.ToString();
            }
            else { //Como no hay colisión solamente mueve la moneda
                moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY +=
                iStep);
            }
        }
        else { //Coloca la moneda en la parte superior de la ventana
            moneda_2.dPosX = numero.Next(330, 580);
            moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
            moneda_2.dPosY = 0.0;
            moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
        }
    }
```

Solo quedará agregar el método para checar colisiones que se ha empleado en otras ocasiones.

```
    private bool checarColision(Imagenes img1, Imagenes img2)
    {
        if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la
```

```
izquierda de img2
        return false;
    if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de
img2
        return false;
    if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la
derecha img2
        return false;
    if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
        return false;
    return true;
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando se detectan a las dos personas los calderos podrán moverse siguiendo el movimiento de las manos y así poder atrapar la mayor cantidad de monedas.

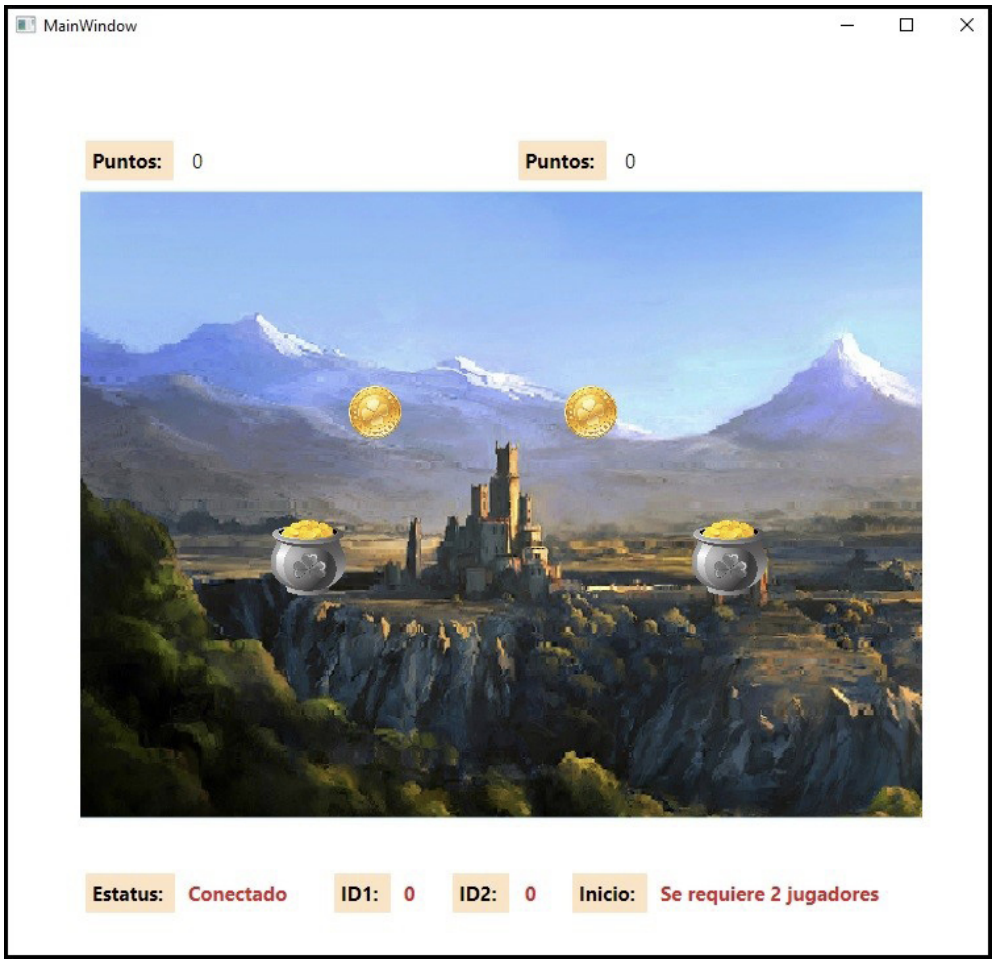


Figura 7.2. Muestra de cómo se podría ver el ejemplo.

7.4 Ejemplo: atrapar monedas (Kinect V2)

Esta aplicación es un juego en el que se registra la cantidad de monedas que atrapa cada uno de los dos jugadores. Se muestran en la pantalla dos calderos (uno para cada jugador) que son movidos para atrapar las monedas. Para ser justos en la competencia, los calderos no se mueven a menos que haya dos personas frente al Kinect. Los jugadores mueven los calderos con su mano derecha.

Requerimientos

- Proyecto **WPF**.
- Plantilla V2 (configuración Skeleton).
- Imágenes: fondo, monedas y calderos.

Recursos

Para esta aplicación se requiere un fondo para ambientar al juego. También se necesitará una imagen para el recipiente para atrapar un objeto (caldero) y otra para el objeto a atrapar (moneda). Se sugiere que sean tipo png para que sus fondos sean transparentes.

Código en XAML

La ventana de la aplicación tiene un **Grid** el cual contiene el **Canvas** con las imágenes y las etiquetas que se utilizan para mostrar la cantidad de monedas atrapadas por cada jugador. Dentro del Canvas, se incluye la imagen de fondo, una moneda y un caldero para cada uno de los jugadores. Las etiquetas que se muestran en la parte superior de la ventana indicarán los puntos acumulados para cada jugador.

```
<Grid>
  <Canvas Name="MainCanvas" Width="640" Height="480"
 Grid.Row="1"
      Background="Azure" HorizontalAlignment="Center">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480"
      Width="640" Source="castillo.png"/>
    <Image Name="moneda1" Canvas.Left="10" Canvas.Top="10"
    Height="40"
      Width="40" Source="Moneda.png"/>
    <Image Name="moneda2" Canvas.Left="580" Canvas.Top="10"
    Height="40"
      Width="40" Source="Moneda.png"/>
    <Image Name="Puntero1" Canvas.Left="143" Canvas.Top="250"
    Height="60"
      Width="60" Source="caldero.png"/>
    <Image Name="Puntero2" Canvas.Left="463" Canvas.Top="250"
    Height="60"
```

```
            Width="60" Source="caldero.png"/>
</Canvas>
 <Label x:Name="label" Content="Puntos: " FontSize="15" FontWeight="Bold"
      Background="Bisque" HorizontalAlignment="Left" VerticalAlignment="Top"
      Margin="61,72,0,0"/>
 <Label x:Name="Jugador1" Content="0" FontSize="15" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="137,72,0,0"/>
 <Label x:Name="label2" Content="Puntos: " FontSize="15" FontWeight="Bold"
      Background="Bisque" HorizontalAlignment="Left" VerticalAlignment="Top"
      Margin="390,72,0,0"/>
 <Label x:Name="Jugador2" Content="0" FontSize="15" HorizontalAlignment="Left"
      VerticalAlignment="Top" Margin="466,72,0,0"/>
</Grid>
```

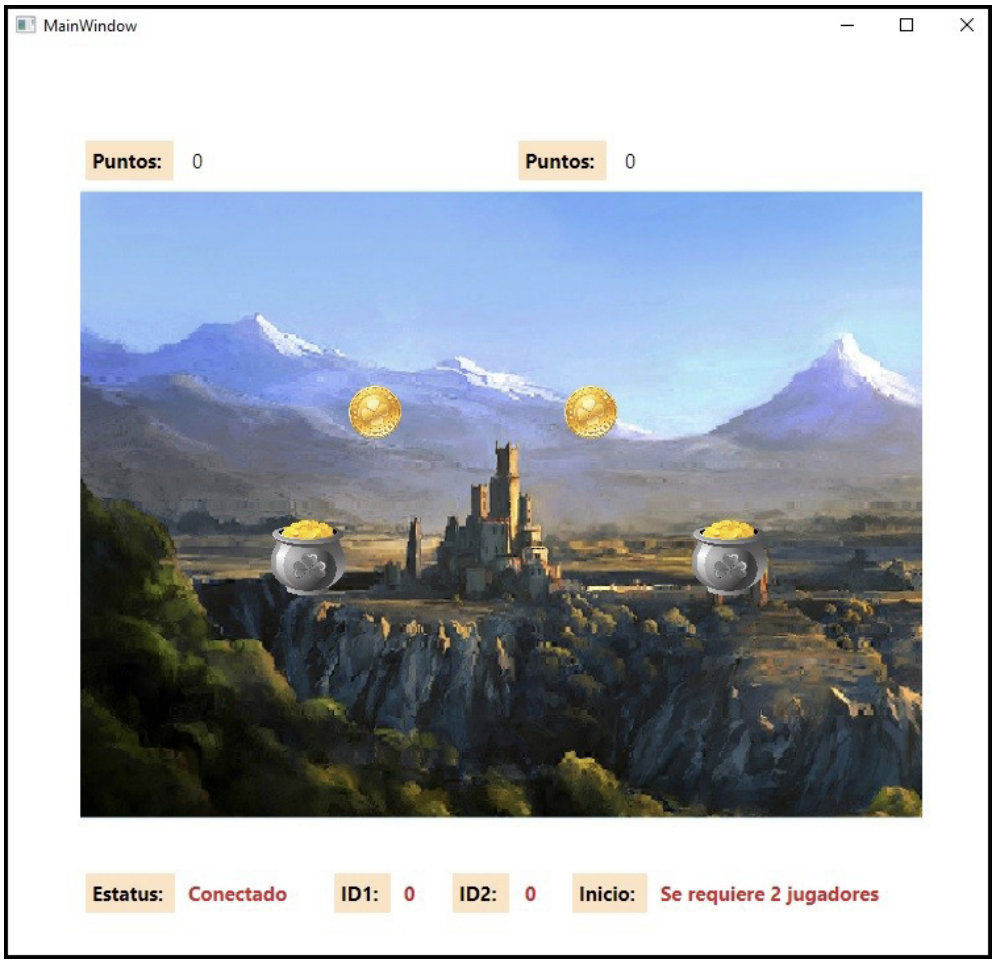


Figura 7.3. Muestra de cómo se podría ver el ejemplo.

Código en C#

Esta aplicación toma como base la plantilla para **Body** a la cual se le debieron realizar las modificaciones para la selección de dos personas (métodos Kinect_FrameReady y usarBody).

El programa emplea las siguientes bibliotecas.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

En la parte superior del programa, se deben incluir las variables tanto para la selección de las personas, como las que se requieren para el manejo de colisiones y el conteo de puntos acumulados por jugador.

```
    private BodyFrameReader miBodyFrameReader; //FrameReader para recibir datos
        private Body[] bodies = null; //Variable para almacenar datos de Body
        Point joint_Point = new Point(); //Permite obtener los datos del Joint
        double dMano_X; //Representa la coordenada X de la mano
        ulong uBody_ID1 = 0; //ID primera persona
        ulong uBody_ID2 = 0; //ID segunda persona
        int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
        Random numero = new Random(); //Generador de números al azar
        DispatcherTimer timer;
        int iStep = 25; //Píxeles a avanzar para simular moviento en monedas
        int iPuntos1 = 0; //Contador puntos primera persona
        int iPuntos2 = 0; //Contador puntos segunda persona
        // Estructura que almacenará la información de objetos que colisionan
        struct Imagenes
        {
            public double dPosX;
            public double dPosY;
            public double dAncho;
            public double dAlto;
        }
        //Objetos que colisionan
        Imagenes moneda_1, moneda_2, caldero_1, caldero_2;
```

Las modificaciones al método **mainWindow** solo son segmentos de código en el cual se da valores iniciales a los objetos creados y se configura el **timer**, el método queda de la siguiente forma:

```
public MainWindow()
{
  InitializeComponent();
  //Realiza configuraciones e iniciar el Kinect
  Kinect_Config();
  //Colocar la moneda 1 de lado izquierdo de la parte superior de la ventana
  //y almacena la información requerida para el manejo de la colisión
  moneda_1.dPosX = numero.Next(15, 260);
  moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
  moneda_1.dPosY = 0.0;
  moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
  moneda_1.dAncho = moneda1.Width;
  moneda_1.dAlto = moneda1.Height;
  //Actualizar la información que no se modifica del caldero 1
  caldero_1.dAncho = Puntero1.Width;
  caldero_1.dAlto = Puntero1.Height;
  //Colocar la moneda 2 de lado derecho de la parte superior de la ventana
  //y almacena la información requerida para el manejo de la colisión
  // Asignar coordenadas iniciales Moneda 2
  moneda_2.dPosX = numero.Next(330, 580);
  moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
  moneda_2.dPosY = 0.0;
  moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
  moneda_2.dAncho = moneda2.Width;
```

```
    moneda_2.dAlto = moneda2.Height;
    //Actualizar la información que no se modifica del caldero 2
    caldero_2.dAncho = Puntero2.Width;
    caldero_2.dAlto = Puntero2.Height;
    //Configura e inicia el timer
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
    timer.Tick += new EventHandler(timer_Tick);
    timer.IsEnabled = true;
  }
```

Las instrucciones que se deben añadir al método **usarBody** son las necesarias para colocar el caldero en la posición de la mano derecha del jugador correspondiente. El jugador solo puede mover el caldero de manera horizontal. Para evitar que un jugador invada el área de su contrincante, se realiza una división imaginaria de la ventana de tal manera que el jugador 1 solo puede mover su caldero si la posición de su mano en el eje **x** es menor a 261. De la misma forma, el jugador 2 mueve su caldero cuando la posición de su mano en el mismo eje **x** es mayor a 321.

```
  //Recibe la información de un body y la utiliza.
  private void usarBody(Body miBody)
  {
    Joint joint1 = miBody.Joints[JointType.HandRight];
    if (joint1.TrackingState == TrackingState.Tracked)
    {
      //Obtiene la posición de la mano
      joint_Point = this.BodyPointToScreen(joint1);
      dMano_X = joint_Point.X;
      //PERSONA 1
      if (miBody.TrackingId == iBody_ID1)
```

```
        { //Coloca la imagen en la posición de la mano derecha siempre y
        cuando
            //ésta se encuentre a la izquierda de la ventana
                if (dMano_X < 261.0)
                    Puntero1.SetValue(Canvas.LeftProperty, dMano_X);
                return;
            }
            //PERSONA 2
            if (miBody.TrackingId == iBody_ID2)
            { //Coloca la imagen en la posición de la mano derecha siempre
            y cuando
                //esta se encuentre a la derecha de la ventana
                if (dMano_X < 321.0)
                    Puntero2.SetValue(Canvas.LeftProperty, dMano_X);
                return;
            }
            //Se establece el Caso 2 para que se realice nuevamente la
            selección
            //de otra persona ya que no se encontró información de ninguna
            de las dos
            //personas seleccionadas (una de las personas se salió)
            iCasos_ID = 2;
        }
    }
```

Otro método que se debe añadir al proyecto es el **timer_Tick**, que se emplea para crear la animación de las monedas y contar las monedas atrapadas por cada jugador. En términos generales, las monedas salen de la parte superior de la ventana y bajan a cierta velocidad hasta llegar al fondo de la ventana (**dPosY =420**) donde desaparecen para

mostrarse nuevamente en algún punto de la parte superior con respecto al eje x. Cuando el jugador hace que su caldero choque con la moneda, además de acreditarle un punto, hace que la moneda realice el mismo proceso que cuando llega al final.

```
//Método que realiza la animación de la aplicación, moviendo las monedas y
//llevando el control de los puntos ganados por cada uno de los jugadores
void timer_Tick(object sender, EventArgs e) {
  //Obtener coordenadas de la Moneda y Puntero (caldero) del Jugador 1
  moneda_1.dPosX = (double)moneda1.GetValue(Canvas.LeftProperty);
  moneda_1.dPosY = (double)moneda1.GetValue(Canvas.TopProperty);
  caldero_1.dPosX = (double)Puntero1.GetValue(Canvas.LeftProperty);
  caldero_1.dPosY = (double)Puntero1.GetValue(Canvas.TopProperty);
  //Obtener coordenadas de la Moneda y Puntero (caldero) del Jugador 2
  moneda_2.dPosX = (double)moneda2.GetValue(Canvas.LeftProperty);
  moneda_2.dPosY = (double)moneda2.GetValue(Canvas.TopProperty);
  caldero_2.dPosX = (double)Puntero2.GetValue(Canvas.LeftProperty);
  caldero_2.dPosY = (double)Puntero2.GetValue(Canvas.TopProperty);
  //JUGADOR 1: El proceso se realiza solamente si la moneda del jugador
  //está aún dentro de la ventana
  if (moneda_1.dPosY < 420) {
      if (checarColision(moneda_1, caldero_1)) {
          //Cuando hay colisión se coloca la moneda en la parte superior de la
          //ventana en una posición al azar en el eje x
          moneda_1.dPosX = numero.Next(15, 260);
          moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
```

```
            moneda_1.dPosY = 0.0;
            moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
            //Acredita un punto al jugador y despliega el resultado en la
            ventana
            iPuntos1++;
            Jugador1.Content = iPuntos1.ToString();
        }
        else { //Como no hay colisión solamente mueve la moneda
            moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY +=
            iStep);
        }
    }
    else { //Coloca la moneda en la parte superior de la ventana
        moneda_1.dPosX = numero.Next(15, 260);
        moneda1.SetValue(Canvas.LeftProperty, moneda_1.dPosX);
        moneda_1.dPosY = 0.0;
        moneda1.SetValue(Canvas.TopProperty, moneda_1.dPosY);
    }
    //JUGADOR 2: El proceso se realiza solamente si la moneda del
jugador
    //está aún dentro de la ventana
    if (moneda_2.dPosY < 420) {
        //Cuando hay colisión se coloca la moneda en la parte superior de la
        //ventana en una posición al azar en el eje x
        if (checarColision(moneda_2, caldero_2)) {
            moneda_2.dPosX = numero.Next(330, 580);
```

```
                moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
                moneda_2.dPosY = 0.0;
                moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
                //Acredita un punto al jugador y despliega el resultado en la ventana
                iPuntos2++;
                Jugador2.Content = iPuntos2.ToString();
            }
            else { //Como no hay colisión solamente mueve la moneda
                moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY += iStep);
            }
        }
        else { //Coloca la moneda en la parte superior de la ventana
            moneda_2.dPosX = numero.Next(330, 580);
            moneda2.SetValue(Canvas.LeftProperty, moneda_2.dPosX);
            moneda_2.dPosY = 0.0;
            moneda2.SetValue(Canvas.TopProperty, moneda_2.dPosY);
        }
    }
```

Solo hace fatla agregar el método para checar colisiones que se ha empleado otras veces.

```
    private bool checarColision(Imagenes img1, Imagenes img2)
    {
        if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de img2
            return false;
```

```
  if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de
img2
      return false;
  if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la
derecha img2
      return false;
  if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
      return false;
  return true;
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando se detectan las dos personas, los calderos pueden moverse siguiendo a las manos y así poder atrapar la mayor cantidad de monedas.

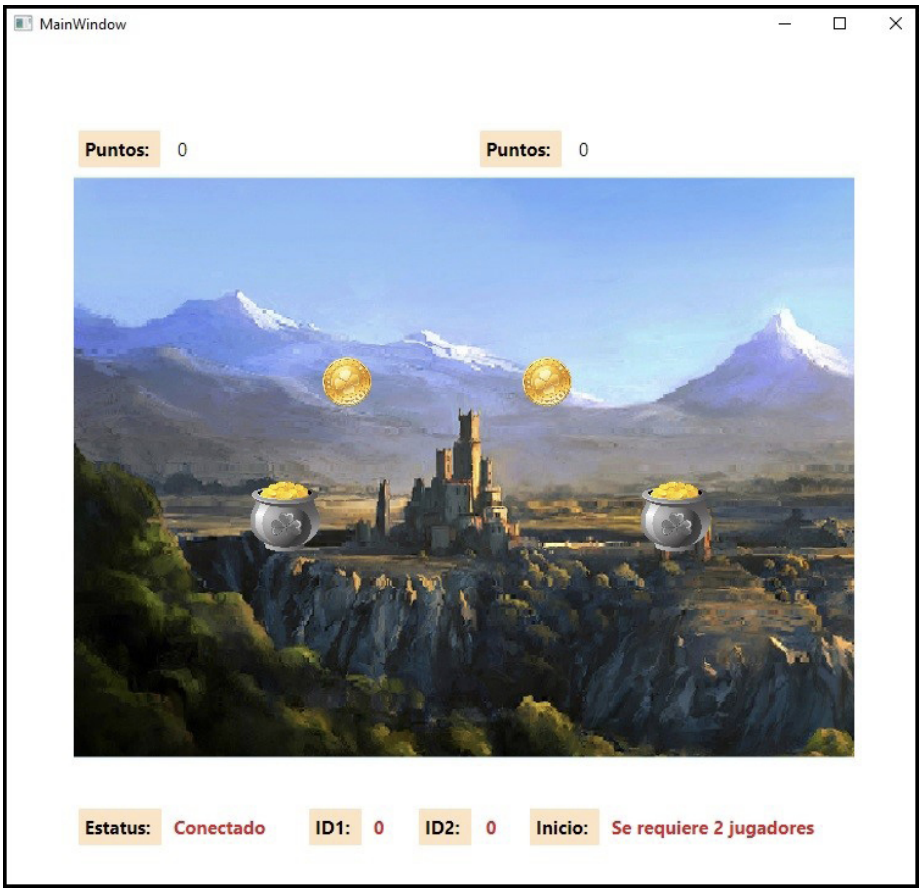


Figura 7.4. Muestra de cómo se podría ver el ejemplo.

7.5 Ejemplo: naves espaciales (Kinect V1)

La aplicación a desarrollar consiste en una competencia entre dos personas donde ganará aquel que logre destruir más asteroides. Cada jugador tiene una nave que puede mover horizontalmente para apuntar y disparar un misil que destruye al asteroide.

Requerimientos

- Proyecto **WPF**.
- Plantilla V1 (configuración Skeleton).
- Imágenes: fondo, nave, misil, asteroide y explosión.

Recursos

Para esta aplicación se requiere un fondo para dar ambientación al juego. También se necesitan las siguientes imágenes: una nave, un misil, un asteroide y una explosión que

sustituye al asteroide cuando el misil lo golpea. Se sugiere que sean tipo png para que los fondos de las mismas sean transparentes.

Código en XAML

El programa en XAML cuenta con un Grid que incluye un Canvas para el área de juego, un conjunto de etiquetas para mostrar los puntos ganados por jugador y elementos multimedia para crear efectos de sonido. No hay que olvidar que los elementos de sonido deben incluir la propiedad LoadedBehavior en manual y la propiedad Source con la trayectoria completa, dentro de la computadora, donde se encuentra el audio.

```
<Grid>
  <Canvas Name="MainCanvas" Grid.Row="1"
 HorizontalAlignment="Center"
      Height="480" Width="640">
    <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
    Height="480"
      Width="640" Source="espacio.jpg"/>
    <Image Name="Nave1" Canvas.Left="143" Canvas.Top="350"
    Height="60"
      Width="50" Source="F22.png"/>
    <Image Name="Nave2" Canvas.Left="463" Canvas.Top="350"
    Height="60"
      Width="50" Source="F22.png"/>
    <Image Name="Asteroide1" Canvas.Left="143" Canvas.Top="200"
    Height="50"
      Width="50" Source="asteroide.png"/>
    <Image Name="Asteroide2" Canvas.Left="463" Canvas.Top="200"
    Height="50"
      Width="50" Source="asteroide.png"/>
    <Image Name="Misil1" Canvas.Left="103" Canvas.Top="350"
    Height="30"
```

```
        Width="30" Source="rocket1.png"/>
    <Image Name="Misil2" Canvas.Left="423" Canvas.Top="350"
    Height="30"
        Width="30" Source="rocket1.png"/>
 </Canvas>
 <Label x:Name="L5" Content="Puntos:" FontSize="15"
FontWeight="Bold"
    Background="Aqua" HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="131,56,0,0"/>
 <Label x:Name="Puntaje1" Content="0" FontSize="15"
HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="207,56,0,0"/>
 <Label x:Name="L6" Content="Puntos:" FontSize="15"
FontWeight="Bold"
    Background="Aqua" HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="460,56,0,0"/>
 <Label x:Name="Puntaje2" Content="0" FontSize="15"
HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="538,56,0,0"/>
 <MediaElement x:Name="ExplosionS1" HorizontalAlignment="Left"
Height="50"
        Margin="381,597,0,0" VerticalAlignment="Top" Width="50"
        LoadedBehavior="Manual" Source="C:\BigExplosion1.mp3"/>
 <MediaElement x:Name="ExplosionS2" HorizontalAlignment="Left"
Height="50"
        Margin="445,597,0,0" VerticalAlignment="Top" Width="50"
```

```
        LoadedBehavior="Manual" Source="C:\BigExplosion2.mp3"/>
</Grid>
```

Figura 7.5. Muestra de cómo se podría ver el ejemplo.

Código en C#

Esta aplicación toma como base la plantilla para Skeleton a la cual se le debieron realizar las modificaciones para la selección de dos usuarios (métodos Kinect_FrameReady y usarSkeleton).

El programa emplea las siguientes bibliotecas.

```
using Microsoft.Kinect;
using System.IO;
using System.Windows.Threading;
```

En la parte superior de la pantalla se agregan las variables necesarias para el manejo de dos jugadores, la contabilización de puntos y el manejo de la colisión entre el misil y el

asteroide. Además se incluye un DispatcherTimer para el movimiento automático del asteroide y otros dos más para el control de los efectos de sonido. Finalmente, se agregan dos variables tipo BitmapImage que hacen el reemplazo del asteroide por la explosión.

```
double dMano_X; //Representa la coordenada X de la mano
Point joint_Point = new Point(); //Permite obtener los datos del Joint
int iSkeleton_ID1 = 0; //ID primera persona
int iSkeleton_ID2 = 0; //ID segunda persona
int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
Random numero = new Random(); //Generador de números al azar
DispatcherTimer timer;
DispatcherTimer timerSonido1; //Control sonido 1
DispatcherTimer timerSonido2; //Control sonido 2
int iStep = 25; //Píxeles a avanzar para simular moviento en monedas
int iPuntos1 = 0; //Contador puntos primera persona
int iPuntos2 = 0; //Contador puntos segunda persona
// Estructura que almacenará la información de objetos que colisionan
struct Imagenes
{
   public double dPosX;
   public double dPosY;
   public double dAncho;
   public double dAlto;
}
//Objetos que colisionan
Imagenes asteroide_1, asteroide_2, misil_1, misil_2;
// Imágenes para simular explosión del asteroide
```

```
BitmapImage explosionBitmap = new BitmapImage(new Uri(@"explosion.png",
                                        UriKind.RelativeOrAbsolute));
BitmapImage asteroideBitmap = new BitmapImage(new Uri(@"asteroide.png",
                                        UriKind.RelativeOrAbsolute));
```

En el método **mainWindow** se dan valores iniciales a los objetos y se configuran los **timers**. El método queda de la siguiente forma:

```
public MainWindow() {
    InitializeComponent();
    //Realiza configuraciones e iniciar el Kinect
    Kinect_Config();
    // Asignar coordenadas iniciales Asteroide 1
    //Colocar el asteroide 1 de lado izquierdo de la parte superior de la ventana
    //y almacena la información requerida para el manejo de la colisión
    asteroide_1.dPosX = numero.Next(15, 260);
    Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
    asteroide_1.dPosY = 0.0;
    Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
    asteroide_1.dAncho = Asteroide1.Width;
    asteroide_1.dAlto = Asteroide1.Height;
    //Colocar el asteroide 2 de lado derecho de la parte superior de la ventana
    //y almacena la información requerida para el manejo de la colisión
    asteroide_2.dPosX = numero.Next(330, 580);
    Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
```

```
asteroide_2.dPosY = 0.0;
Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
asteroide_2.dAncho = Asteroide2.Width;
asteroide_2.dAlto = Asteroide2.Height;
//Coloca al misil 1 a una distancia apropiada a la nave 1 y
//almacena información requerida para el manejo de la colisión
Misil1.SetValue(Canvas.LeftProperty,
        (double)Nave1.GetValue(Canvas.LeftProperty) + 10);
misil_1.dPosY = (double)Nave1.GetValue(Canvas.TopProperty) - 40;
Misil1.SetValue(Canvas.TopProperty, misil_1.dPosY);
misil_1.dAncho = Misil1.Width;
misil_1.dAlto = Misil1.Height;
//Coloca al misil 2 a una distancia apropiada a la nave 2 y
//almacena información requerida para el manejo de la colisión
Misil2.SetValue(Canvas.LeftProperty,
                (double)Nave2.GetValue(Canvas.LeftProperty) + 10);
misil_2.dPosY = (double)Nave2.GetValue(Canvas.TopProperty) – 40;
Misil2.SetValue(Canvas.TopProperty, misil_2.dPosY);
misil_2.dAncho = Misil2.Width;
misil_2.dAlto = Misil2.Height;
// Configurar e iniciar timer que se encarga del juego
timer = new DispatcherTimer();
timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
timer.Tick += new EventHandler(timer_Tick);
timer.IsEnabled = true;
```

```
        // Configurar timer que detiene el efecto de sonido para Jugador 1
        timerSonido1 = new DispatcherTimer();
        timerSonido1.Interval = new TimeSpan(0, 0, 0, 0, 550);
        timerSonido1.Tick += new EventHandler(timer_Tick2);
        // Configurar timer que detiene el efecto de sonido para Jugador 2
        timerSonido2 = new DispatcherTimer();
        timerSonido2.Interval = new TimeSpan(0, 0, 0, 0, 550);
        timerSonido2.Tick += new EventHandler(timer_Tick3);
    }
```

Al método **usarSkeleton** solo se le agregan las instrucciones para colocar las naves en la posición de la mano derecha de cada uno de los jugadores.

```
    //Recibe la información de un esqueleto y la utiliza para colocar la imagen
    //de las naves en la mismo posición que la mano derecha del jugador.
    private void usarSkeleton(Skeleton skeleton) {
        Joint joint1 = skeleton.Joints[JointType.HandRight];
        if (joint1.TrackingState == JointTrackingState.Tracked) {
            joint_Point = this.SkeletonPointToScreen(joint1.Position);
            dMano_X = joint_Point.X;
            //PERSONA 1
            if (skeleton.TrackingId == iSkeleton_ID1) {
                //Coloca la imagen en la posición de la mano derecha siempre y cuando
                //esta se encuentre a la izquierda de la ventana
                if (dMano_X < 261.0)
                    Nave1.SetValue(Canvas.LeftProperty, dMano_X);
                return;
```

```
        }
        //PERSONA 2
        if (skeleton.TrackingId == iSkeleton_ID2) {
            //Coloca la imagen en la posición de la mano derecha siempre y
            cuando
            //esta se encuentre a la derecha de la ventana
            if (dMano_X > 321)
                Nave2.SetValue(Canvas.LeftProperty, dMano_X);
            return;
        }
        //Se establece el Caso 2 para que se realice nuevamente la selección
        //de otra persona ya que no se encontró información de ninguna de
        las dos
        //personas seleccionadas (una de las personas se salió)
        iCasos_ID = 2;
    }
}
```

El método **timer_Tick** se encarga de producir la animación en los elementos del juego. En él se encuentran las instrucciones para mover verticalmente los asteroides y determinar cuándo deben aparecer nuevamente en la parte superior en otra posición dentro del eje x. También incluye el movimiento de los misiles, estos se trasladan verticalmente, inician cerca de su nave y terminan cuando han llegado al límite superior de la ventana o han chocado con un asteroide. Los misiles están en un movimiento constante, es decir, el jugador no necesita hacer una acción para dispararlo. Cuando ocurre una colisión entre el misil y el asteroide, se habilita el **timer** que controlará el efecto de sonido de la explosión.

```
//Método que realiza la animación de la aplicación, moviendo naves,
asteroides y
//misiles además del control de los puntos ganados por cada uno de los
jugadores
```

```
void timer_Tick(object sender, EventArgs e) {
  //Obtener coordenadas Asteroide y Misil para el Jugador 1
  asteroide_1.dPosX = (double)Asteroide1.GetValue(Canvas.LeftProperty);
  asteroide_1.dPosY = (double)Asteroide1.GetValue(Canvas.TopProperty);
  misil_1.dPosX = (double)Misil1.GetValue(Canvas.LeftProperty);
  misil_1.dPosY = (double)Misil1.GetValue(Canvas.TopProperty);
  //Obtener coordenadas Asteroide y Misil para el Jugador 2
  asteroide_2.dPosX = (double)Asteroide2.GetValue(Canvas.LeftProperty);
  asteroide_2.dPosY = (double)Asteroide2.GetValue(Canvas.TopProperty);
  misil_2.dPosX = (double)Misil2.GetValue(Canvas.LeftProperty);
  misil_2.dPosY = (double)Misil2.GetValue(Canvas.TopProperty);
  //JUGADOR 1
  if (asteroide_1.dPosY < 420) {
     //Colisión entre asteriode y misil
     if (checarColision(asteroide_1, misil_1)) {
        ExplosionS1.Play(); //Crear efecto de explosión (sonido e imagen)
        Asteroide1.Source = explosionBitmap;
        asteroide_1.dPosX = numero.Next(15, 260); //Colocar asteroide al inicio
        Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
        asteroide_1.dPosY = 0.0;
        Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
```

```
            iPuntos1++; //Contabilizar puntos del jugador
            Puntaje1.Content = iPuntos1.ToString();
            timerSonido1.IsEnabled = true; //Habilitar el timer para efecto
            de sonido
        }
        else { //Continuar movimiento normal del asteroide
            Asteroide1.Source = asteroideBitmap;
            Asteroide1.SetValue(Canvas.TopProperty,     asteroide_1.dPosY
            += iStep);
        }
    }
    else { //Colocar asteroide al inicio en una posición x al azar
        asteroide_1.dPosX = numero.Next(15, 260);
        Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
        asteroide_1.dPosY = 0.0;
        Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
    }
    // Actualizar posición Misil
    if (misil_1.dPosY > 0) {
        Misil1.SetValue(Canvas.TopProperty, misil_1.dPosY -= 40);
    }
    else { //Coloca misil en posición original
        Misil1.SetValue(Canvas.LeftProperty,
                (double)Nave1.GetValue(Canvas.LeftProperty) + 10);
        Misil1.SetValue(Canvas.TopProperty,
                (double)Nave1.GetValue(Canvas.TopProperty) - 40);
```

```
    misil_1.dPosY = (double)Nave1.GetValue(Canvas.TopProperty) -
    40;
}
//JUGADOR 2
if (asteroide_2.dPosY < 420) {
    //Colisión entre asteriode y misil
    if (checarColision(asteroide_2, misil_2)) {
        ExplosionS2.Play(); //Crear efecto de explosión (sonido e
        imagen)
        Asteroide2.Source = explosionBitmap;
        asteroide_2.dPosX = numero.Next(330, 580); //Colocar
        asteroide al inicio
        Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
        asteroide_2.dPosY = 0.0;
        Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
        iPuntos2++; //Contabilizar puntos del jugador
        Puntaje2.Content = iPuntos2.ToString();
        timerSonido2.IsEnabled = true; //Habilitar el timer para efecto
        de sonido
    }
    else { //Continuar movimiento normal del asteroide
    Asteroide2.Source = asteroideBitmap;
    Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY +=
    iStep);
    }
}
else { //Colocar asteroide al inicio en una posición x al azar
```

```
            asteroide_2.dPosX = numero.Next(330, 580);
            Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
            asteroide_2.dPosY = 0.0;
            Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
        }
        // Actualizar posicion Misil
        if (misil_2.dPosY > 0) {
            Misil2.SetValue(Canvas.TopProperty, misil_2.dPosY -= 40);
        }
        else { //Coloca misil en posición original
            Misil2.SetValue(Canvas.LeftProperty,
                    (double)Nave2.GetValue(Canvas.LeftProperty) + 10);
            Misil2.SetValue(Canvas.TopProperty,
                    (double)Nave2.GetValue(Canvas.TopProperty) - 40);
            misil_2.dPosY = (double)Nave2.GetValue(Canvas.TopProperty) -
            40;
        }
    }
```

Otros métodos nuevos son los **timer** que se encargan de controlar la reproducción de los sonidos. Cuando el tiempo especificado ha transcurrido aplican el método **Stop** a su respectivo audio y deshabilitan su **timer**, para volver a reproducir el sonido.

```
    void timer_Tick2(object sender, EventArgs e)
    {
            ExplosionS1.Stop();
            timerSonido1.IsEnabled = false;
    }
    void timer_Tick3(object sender, EventArgs e)
```

```
{
        ExplosionS2.Stop();
        timerSonido2.IsEnabled = false;
}
```

Finalmente, solo hay agregar el método para checar colisiones, se ha recurrido a esto en otras ocasiones.

```
private bool checarColision(Imagenes img1, Imagenes img2)
{
  if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de img2
    return false;
  if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de img2
    return false;
  if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha img2
    return false;
  if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
    return false;
  return true;
}
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Cuando se detectan las dos personas las naves pueden manipularse al seguir las manos y así poder direccionar los misiles para destruir los asteroides.

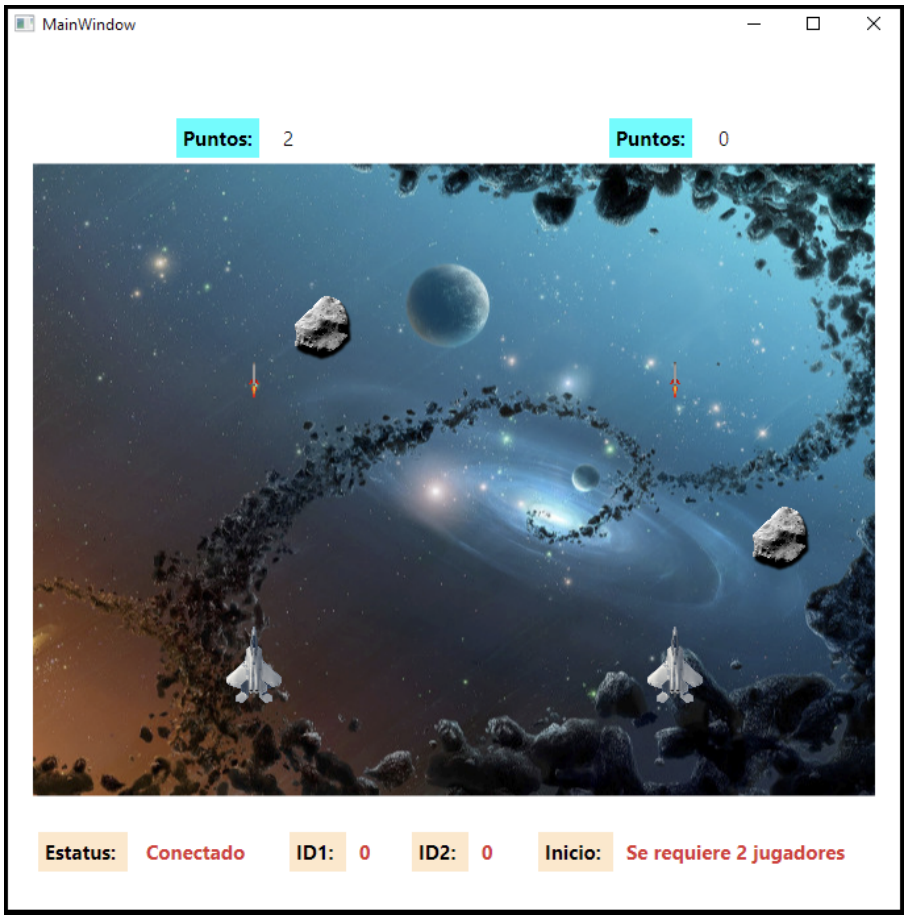


Figura 7.6. Muestra de cómo se podría ver el ejemplo.

7.6 Ejemplo: naves espaciales (Kinect V2)

La aplicación a desarrollar consiste en una competencia entre dos personas en la que ganará quien destruya más asteroides. Cada jugador tiene una nave que puede mover horizontalmente para apuntar y disparar un misil para destruir el asteroides.

Requerimientos

- Proyecto **WPF**.
- Plantilla (configuración para Skeleton).
- Imágenes: fondo, nave, misil, asteroide y explosión.

Recursos

Para esta aplicación se requiere la imagen que será colocada como fondo para

ambientar el juego. También se necesita la imagen de una nave, un misil, un asteroide y una explosión que sustituye al asteroide cuando el misil lo golpea. Se sugiere que sean tipo png para que los fondos sean transparentes.

Código en XAML

El programa en XAML cuenta con un Grid que incluye un Canvas para el área de juego, un conjunto de etiquetas para mostrar los puntos ganados por jugador y elementos multimedia para crear efectos de sonido. Hay que recordar que los elementos de sonido deben incluir la propiedad LoadedBehavior en manual y la propiedad Source con la trayectoria completa, dentro de la computadora, donde se encuentra el audio.

```
<Grid>
  <Canvas Name="MainCanvas" Grid.Row="1"
 HorizontalAlignment="Center"
    Height="480" Width="640">
   <Image Name="fondo" Canvas.Left="0" Canvas.Top="0"
   Height="480"
     Width="640" Source="espacio.jpg"/>
   <Image Name="Nave1" Canvas.Left="143" Canvas.Top="350"
   Height="60"
     Width="50" Source="F22.png"/>
   <Image Name="Nave2" Canvas.Left="463" Canvas.Top="350"
   Height="60"
     Width="50" Source="F22.png"/>
   <Image Name="Asteroide1" Canvas.Left="143" Canvas.Top="200"
   Height="50"
     Width="50" Source="asteroide.png"/>
   <Image Name="Asteroide2" Canvas.Left="463" Canvas.Top="200"
   Height="50"
     Width="50" Source="asteroide.png"/>
   <Image Name="Misil1" Canvas.Left="103" Canvas.Top="350"
```

```
    Height="30"
        Width="30" Source="rocket1.png"/>
    <Image Name="Misil2" Canvas.Left="423" Canvas.Top="350"
    Height="30"
        Width="30" Source="rocket1.png"/>
 </Canvas>
 <Label x:Name="L5" Content="Puntos:" FontSize="15"
FontWeight="Bold"
    Background="Aqua" HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="131,56,0,0"/>
 <Label x:Name="Puntaje1" Content="0" FontSize="15"
HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="207,56,0,0"/>
 <Label x:Name="L6" Content="Puntos:" FontSize="15"
FontWeight="Bold"
    Background="Aqua" HorizontalAlignment="Left"
    VerticalAlignment="Top"
    Margin="460,56,0,0"/>
 <Label x:Name="Puntaje2" Content="0" FontSize="15"
HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="538,56,0,0"/>
 <MediaElement x:Name="ExplosionS1" HorizontalAlignment="Left"
Height="50"
        Margin="381,597,0,0" VerticalAlignment="Top" Width="50"
        LoadedBehavior="Manual"
        Source="C:\BigExplosion1.mp3"/>
 <MediaElement x:Name="ExplosionS2" HorizontalAlignment="Left"
```

```
Height="50"
            Margin="445,597,0,0" VerticalAlignment="Top" Width="50"
            LoadedBehavior="Manual"
            Source="C:\BigExplosion2.mp3"/>
</Grid>
```

Figura 7.7. Muestra de cómo se podría ver el ejemplo.

Código en C#

Esta aplicación toma como base la plantilla para Body a la cual se le debieron realizar las modificaciones para la selección de dos personas (métodos **Kinect_FrameReady** y **usarBody**).

El programa emplea las siguientes bibliotecas.

```
using Microsoft.Kinect;
```

```
using System.IO;
using System.Windows.Threading;
```

En la parte superior de la pantalla se agregan las variables que se requieren para el manejo de dos jugadores, la contabilización de puntos y el manejo de la colisión entre el misil y el asteroide. Además se incluye un DispatcherTimer para el movimiento automático del asteroide y otros dos más para el control de los efectos de sonido. Finalmente, se agregan dos variables tipo BitmapImage que reemplazan al asteroide por la explosión.

```
double dMano_X; //Representa la coordenada X de la mano
private BodyFrameReader miBodyFrameReader; //FrameReader para recibir datos
private Body[] bodies = null; //Variable para almacenar datos de Body
Point joint_Point = new Point(); //Permite obtener los datos del Joint
ulong uBody_ID1 = 0; //ID primera persona
ulong uBody_ID2 = 0; //ID segunda persona
int iCasos_ID = 0; //Control de casos que se pueden presentar con los Ids.
Random numero = new Random(); //Generador de números al azar
DispatcherTimer timer;
DispatcherTimer timerSonido1; //Control sonido 1
DispatcherTimer timerSonido2; //Control sonido 2
int iStep = 20; //Pixeles a avanzar para simular moviento en monedas
int iPuntos1 = 0; //Contador puntos primera persona
int iPuntos2 = 0; //Contador puntos segunda persona
// Estructura que almacenará la información de objetos que colisionan
struct Imagenes
{
  public double dPosX;
  public double dPosY;
```

```
    public double dAncho;
    public double dAlto;
  }
  //Objetos que colisionan
  Imagenes asteroide_1, asteroide_2, misil_1, misil_2;
  // Imagenes para simular explosión del asteroide
  BitmapImage explosionBitmap = new BitmapImage(new
Uri(@"explosion.png",
                                    UriKind.RelativeOrAbsolute));
  BitmapImage asteroideBitmap = new BitmapImage(new
Uri(@"asteroide.png",
```

En el método **mainWindow** se dan valores iniciales a los objetos y se configuran los **timers**. El método queda de la siguiente forma:

```
public MainWindow() {
    InitializeComponent();
    //Realiza configuraciones e iniciar el Kinect
    Kinect_Config();
    // Asignar coordenadas iniciales Asteroide 1
    //Colocar el asteroide 1 de lado izquierdo de la parte superior de la
  ventana
    //y almacena la información requerida para el manejo de la colisión
    asteroide_1.dPosX = numero.Next(15, 260);
    Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
    asteroide_1.dPosY = 0.0;
    Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
    asteroide_1.dAncho = Asteroide1.Width;
    asteroide_1.dAlto = Asteroide1.Height;
```

```
//Colocar el asteroide 2 de lado derecho de la parte superior de la ventana
//y almacena la información requerida para el manejo de la colisión
asteroide_2.dPosX = numero.Next(330, 580);
Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
asteroide_2.dPosY = 0.0;
Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
asteroide_2.dAncho = Asteroide2.Width;
asteroide_2.dAlto = Asteroide2.Height;
//Coloca al misil 1 a una distancia apropiada a la nave 1 y
//almacena información requerida para el manejo de la colisión
Misil1.SetValue(Canvas.LeftProperty,
        (double)Nave1.GetValue(Canvas.LeftProperty) + 10);
misil_1.dPosY = (double)Nave1.GetValue(Canvas.TopProperty) - 40;
Misil1.SetValue(Canvas.TopProperty, misil_1.dPosY);
misil_1.dAncho = Misil1.Width;
misil_1.dAlto = Misil1.Height;
//Coloca al misil 2 a una distancia apropiada a la nave 2 y
//almacena información requerida para el manejo de la colisión
Misil2.SetValue(Canvas.LeftProperty,
        (double)Nave2.GetValue(Canvas.LeftProperty) + 10);
misil_2.dPosY = (double)Nave2.GetValue(Canvas.TopProperty) – 40;
Misil2.SetValue(Canvas.TopProperty, misil_2.dPosY);
misil_2.dAncho = Misil2.Width;
misil_2.dAlto = Misil2.Height;
// Configurar e iniciar timer que se encarga del juego
```

```
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 0, 100);
        timer.Tick += new EventHandler(timer_Tick);
        timer.IsEnabled = true;
        // Configurar timer que detiene el efecto de sonido para Jugador 1
        timerSonido1 = new DispatcherTimer();
        timerSonido1.Interval = new TimeSpan(0, 0, 0, 0, 550);
        timerSonido1.Tick += new EventHandler(timer_Tick2);
        // Configurar timer que detiene el efecto de sonido para Jugador 2
        timerSonido2 = new DispatcherTimer();
        timerSonido2.Interval = new TimeSpan(0, 0, 0, 0, 550);
        timerSonido2.Tick += new EventHandler(timer_Tick3);
    }
```

Al método **usarBody** solamente se le agregan las instrucciones para colocar las naves en la posición de la mano derecha de cada uno de los jugadores.

```
    //Recibe la información de un body y la utiliza para colocar la imagen □
//de las naves en la mismo posición que la mano derecha del jugador.
    private void usarBody(Body miBody) {
        Joint joint1 = miBody.Joints[JointType.HandRight];
        if (joint1.TrackingState == TrackingState.Tracked) {
         joint_Point = this.BodyPointToScreen(joint1);
         dMano_X = joint_Point.X;
            //PERSONA 1
            if (miBody.TrackingId == iBody_ID1) {
                //Coloca la imagen en la posición de la mano derecha siempre y
                cuando
                //ésta se encuentre a la izquierda de la ventana
```

```
            if (dMano_X < 261.0)
                Nave1.SetValue(Canvas.LeftProperty, dMano_X);
            return;
        }
        //PERSONA 2
        if (miBody.TrackingId == iBody_ID2) {
            //Coloca la imagen en la posición de la mano derecha siempre y
            cuando
            //esta se encuentre a la derecha de la ventana
            if (dMano_X < 321.0)
                Nave2.SetValue(Canvas.LeftProperty, dMano_X);
            return;
        }
        //Se establece el Caso 2 para que se realice nuevamente la selección
        //de otra persona ya que no se encontró información de ninguna de
        las dos
        //personas seleccionadas (una de las personas se salió)
        iCasos_ID = 2;
    }
}
```

El método **timer_Tick** se encarga de producir la animación en los elementos del juego. En él se encuentran las instrucciones para mover verticalmente los asteroides y determinar cuándo deben aparecer nuevamente en la parte superior en alguna otra posición dentro del eje x. También incluye el movimiento de los misiles, los cuales se trasladan verticalmente iniciando cerca de su nave y terminando cuando han llegado al límite superior de la ventana o han chocado con un asteroide. Los misiles están en un movimiento constante es decir no se requiere que el jugador realice alguna acción para dispararlo.

Cuando ocurre una colisión entre el misil y el asteroide, se habilita el **timer** que controlará el efecto de sonido de la explosión.

```
    //Método que realiza la animación de la aplicación, moviendo naves, asteroides y
    //misiles además del control de los puntos ganados por cada uno de los jugadores
    void timer_Tick(object sender, EventArgs e) {
      //Obtener coordenadas Asteroide y Misil para el Jugador 1
      asteroide_1.dPosX = (double)Asteroide1.GetValue(Canvas.LeftProperty);
      asteroide_1.dPosY = (double)Asteroide1.GetValue(Canvas.TopProperty);
      misil_1.dPosX = (double)Misil1.GetValue(Canvas.LeftProperty);
      misil_1.dPosY = (double)Misil1.GetValue(Canvas.TopProperty);
      //Obtener coordenadas Asteroide y Misil para el Jugador 2
      asteroide_2.dPosX = (double)Asteroide2.GetValue(Canvas.LeftProperty);
      asteroide_2.dPosY = (double)Asteroide2.GetValue(Canvas.TopProperty);
      misil_2.dPosX = (double)Misil2.GetValue(Canvas.LeftProperty);
      misil_2.dPosY = (double)Misil2.GetValue(Canvas.TopProperty);
      //JUGADOR 1
      if (asteroide_1.dPosY < 420) {
        //Colisión entre asteriode y misil
        if (checarColision(asteroide_1, misil_1)) {
          ExplosionS1.Play(); //Crear efecto de explución (sonido e imagen)
          Asteroide1.Source = explosionBitmap;
          asteroide_1.dPosX = numero.Next(15, 260); //Colocar asteroide al inicio
```

```
            Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
            asteroide_1.dPosY = 0.0;
            Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
            iPuntos1++; //Contabilizar puntos del jugador
            Puntaje1.Content = iPuntos1.ToString();
            timerSonido1.IsEnabled = true; //Habilitar el timer para efecto
            de sonido
        }
        else { //Continuar movimiento normal del asteroide
            Asteroide1.Source = asteroideBitmap;
            Asteroide1.SetValue(Canvas.TopProperty,     asteroide_1.dPosY
            += iStep);
        }
    }
    else { //Colocar asteroide al inicio en una posición x al azar
        asteroide_1.dPosX = numero.Next(15, 260);
        Asteroide1.SetValue(Canvas.LeftProperty, asteroide_1.dPosX);
        asteroide_1.dPosY = 0.0;
        Asteroide1.SetValue(Canvas.TopProperty, asteroide_1.dPosY);
    }
    // Actualizar posicion Misil
    if (misil_1.dPosY > 0) {
        Misil1.SetValue(Canvas.TopProperty, misil_1.dPosY -= 40);
    }
    else { //Coloca misil en posición original
        Misil1.SetValue(Canvas.LeftProperty,
```

```
                (double)Nave1.GetValue(Canvas.LeftProperty) + 10);
    Misil1.SetValue(Canvas.TopProperty,
                (double)Nave1.GetValue(Canvas.TopProperty) - 40);
    misil_1.dPosY = (double)Nave1.GetValue(Canvas.TopProperty) -
    40;
}
//JUGADOR 2
if (asteroide_2.dPosY < 420) {
    //Colisión entre asteriode y misil
    if (checarColision(asteroide_2, misil_2)) {
        ExplosionS2.Play(); //Crear efecto de explоción (sonido e
        imagen)
        Asteroide2.Source = explosionBitmap;
        asteroide_2.dPosX = numero.Next(330, 580); //Colocar
        asteroide al inicio
        Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
        asteroide_2.dPosY = 0.0;
        Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
        iPuntos2++; //Contabilizar puntos del jugador
        Puntaje2.Content = iPuntos2.ToString();
        timerSonido2.IsEnabled = true; //Habilitar el timer para efecto
        de sonido
    }
    else { //Continuar movimiento normal del asteroide
        Asteroide2.Source = asteroideBitmap;
        Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY
        += iStep);
```

```
        }
    }
    else { //Colocar asteroide al inicio en una posición x al azar
        asteroide_2.dPosX = numero.Next(330, 580);
        Asteroide2.SetValue(Canvas.LeftProperty, asteroide_2.dPosX);
        asteroide_2.dPosY = 0.0;
        Asteroide2.SetValue(Canvas.TopProperty, asteroide_2.dPosY);
    }
    // Actualizar posicion Misil
    if (misil_2.dPosY > 0) {
        Misil2.SetValue(Canvas.TopProperty, misil_2.dPosY -= 40);
    }
    else { //Coloca misil en posición original
        Misil2.SetValue(Canvas.LeftProperty,
                    (double)Nave2.GetValue(Canvas.LeftProperty) + 10);
        Misil2.SetValue(Canvas.TopProperty,
                    (double)Nave2.GetValue(Canvas.TopProperty) - 40);
        misil_2.dPosY = (double)Nave2.GetValue(Canvas.TopProperty) -
        40;
    }
}
```

Otros métodos nuevos son los **timer** que se encargan de controlar la reproducción de los sonidos. Cuando el tiempo especificado ha transcurrido aplican el método Stop a su respectivo audio y deshabilitan su **timer**. Esto con el fin de que se pueda volver a reproducir el sonido.

```
void timer_Tick2(object sender, EventArgs e)
{
```

```
        ExplosionS1.Stop();
        timerSonido1.IsEnabled = false;
    }
    void timer_Tick3(object sender, EventArgs e)
    {
        ExplosionS2.Stop();
        timerSonido2.IsEnabled = false;
    }
```

Finalmente, falta agregar el método para checar colisiones, este se ha empleado en otras ocasiones.

```
    private bool checarColision(Imagenes img1, Imagenes img2)
    {
      if (img1.dPosX + img1.dAncho < img2.dPosX) //Colisión por la izquierda de img2
        return false;
      if (img1.dPosY + img1.dAlto < img2.dPosY) //Colisión por arriba de img2
        return false;
      if (img1.dPosX > img2.dPosX + img2.dAncho) //Colisión por la derecha img2
        return false;
      if (img1.dPosY > img2.dPosY + img2.dAlto) //Colisión por abajo img2
        return false;
      return true;
    }
```

Ejecución

La siguiente imagen muestra cómo se ve la ejecución del proyecto. Al detectar a las dos personas, las naves se mueven de acuerdo al movimiento de las manos, y así, direccionan los misiles para destruir los asteroides.

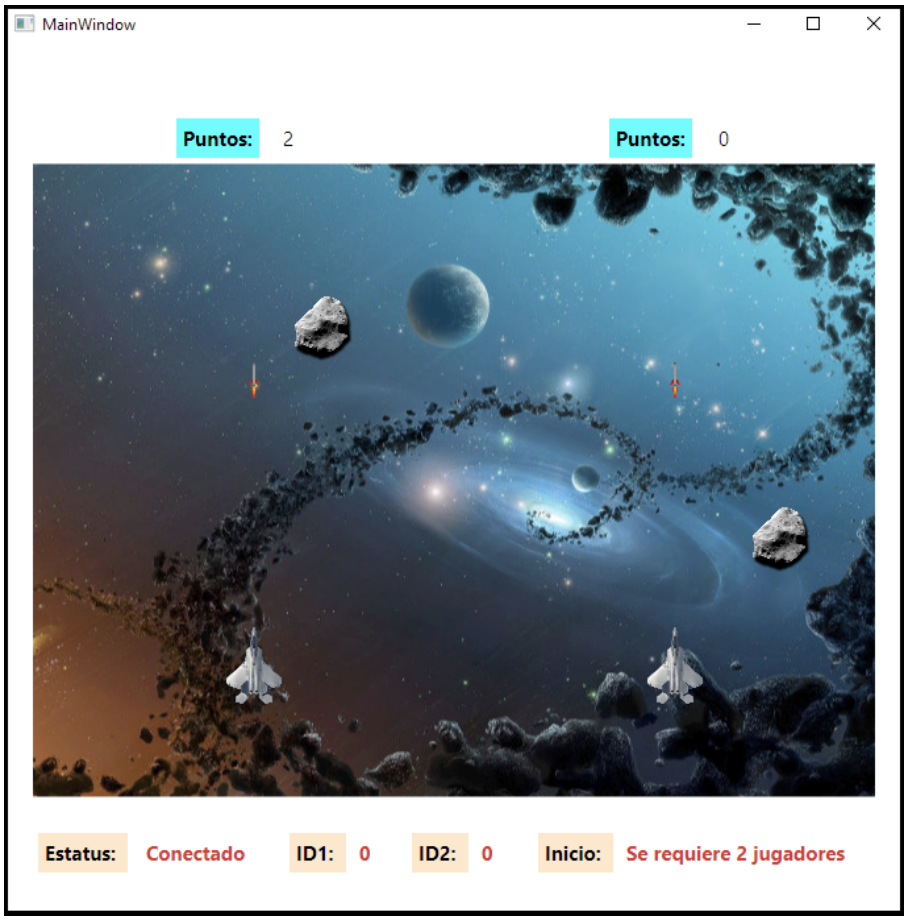


Figura 7.8. Muestra de cómo se podría ver el ejemplo.

Capítulo 8. Reacción a gestos faciales - Kinect V2

La versión 2 de Kinect cuenta con herramientas que permiten detectar una gran cantidad de puntos de la cara y algunas de las expresiones que más comúnmente realizan los humanos.

Para el desarrollo de este tipo de aplicaciones se requieren elementos de *software* adicionales (referencias) agregados a un proyecto de programación con una plantilla diferente a la que se ha estado empleando.

8.1 Biblioteca para la detección de la cara

Para realizar una aplicación que detecte gestos y expresiones es necesario incluir la biblioteca que contiene las herramientas de programación diseñadas para tal fin. Esta biblioteca está disponible después de añadir la referencia correspondiente.

```
using Microsoft.Kinect;
using Microsoft.Kinect.Face;
```

Esta biblioteca permite emplear las siguientes **FacePointType** y **FaceProperty** cuyo uso depende del desarrollo que se desee realizar.

8.2 Puntos de la cara (*FacePointType*)

La herramienta **FacePointType** permite localizar los principales puntos de la cara:

- Ojo izquierdo
- Ojo derecho
- Nariz
- Esquina izquierda de la boca
- Esquina derecha de la boca

8.3 Estado de los diferentes puntos (*FaceProperty*)

Esta propiedad muestra el estado en que se encuentran diferentes partes de la cara como los ojos o la boca, así como saber expresiones faciales y el uso de objetos.

Las diferentes condiciones que puede detectar son las siguientes.

- Ojo izquierdo cerrado
- Ojo derecho cerrado
- Boca abierta
- Boca movida
- Feliz
- Concentrado
- Distraído
- Usando lentes

8.4 Obtener información de la cara

Para obtener la información de la cara, es necesario incluir en la parte superior del programa las variables necesarias para depositar los datos que está capturando el Kinect.

```
KinectSensor MiKinect = null; //Nombre del dispositivo Kinect
ColorFrameReader ColorRead = null; //Encargada de la obtención de la imagen
BodyFrameReader BodyRead = null; //Inicializa lectura de cuerpos detectados
IList<Body> Bodies = null; //Almacena número de cuerpos detectados
FaceFrameSource FaceSource = null; //Indica dispositivo y lo que se estará leyendo
FaceFrameReader FaceReader = null; //Inicializa la lectura de la cara por los frames
```

Dentro del **MainWindow** se añaden las instrucciones que se requieren para activar al Kinect e iniciar la lectura de los bodys. Además, se incluyen las instrucciones para especificar los elementos de la cara que se detectarán y se activa el método que se encarga de recibir los *frames*.

```
public MainWindow() {
```

```
InitializeComponent();
MiKinect = KinectSensor.GetDefault();
if (MiKinect != null) {
    //Inicializar el Kinect
    MiKinect.Open();
    //Variable encargada para la deteccion de los cuerpos
    Bodies = new Body[MiKinect.BodyFrameSource.BodyCount];
    //Abrir lectura del video en formato color para que la aplicación pueda
    //mostrar lo que está visualizándose con la cámara.
    ColorRead = MiKinect.ColorFrameSource.OpenReader();
    ColorRead.FrameArrived += ColorReadFrameArrived;
    //Iniciar la lectura de los cuerpos que el Kinect pueda estar detectando
    BodyRead = MiKinect.BodyFrameSource.OpenReader();
    BodyRead.FrameArrived += BodyReadFrameArrived;
    //Indicar los estados de la cara que se detectarán
    FaceSource = new FaceFrameSource(MiKinect, 0,
                                     FaceFrameFeatures.BoundingBoxInColorSpace|
                                     FaceFrameFeatures.FaceEngagement |
                                     FaceFrameFeatures.Glasses |
                                     FaceFrameFeatures.Happy |
                                     FaceFrameFeatures.LeftEyeClosed |
                                     FaceFrameFeatures.MouthOpen |
                                     FaceFrameFeatures.PointsInColorSpace |
```

```
                                    FaceFrameFeatures.RightEyeClosed|
                                    FaceFrameFeatures.MouthMoved|
                                    FaceFrameFeatures.RotationOrientation);
            //Iniciar la lectura de la cara
            FaceReader = FaceSource.OpenReader();
            FaceReader.FrameArrived += FaceReadFrameArrived;
        }
    }
```

Lo siguiente debe acompañar al método anterior:

- **ToBitmap**: recibe la imagen que está capturando el Kinect con la cámara y la convierte a formato RGB (red-green-blue) además de darle un tamaño adecuado para que pueda ser desplegada en la ventana de la aplicación.
- **ColorReadFrameArrived**: despliega en la ventana la imagen que capta el Kinect. Antes de colocar la imagen en la ventana, utiliza la función ToBitmap para realizar la conversión al formato correcto.
- **BodyReadFrameArrived**: obtiene la información facial de la primera persona detectada.

```
    //Método encargado de darle formato RGB a la imagen con el tamaño adecuado para
    //que pueda ser desplegada.
    private ImageSource ToBitmap(ColorFrame Frame) {
        int width = Frame.FrameDescription.Width;
        int height = Frame.FrameDescription.Height;
        PixelFormat FormatoDePixel = PixelFormats.Bgr32;
        byte[] Pixeles = new byte[width * height * ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
        if (Frame.RawColorImageFormat == ColorImageFormat.Bgra) {
            Frame.CopyRawFrameDataToArray(Pixeles);
```

```
        }
        else {
            Frame.CopyConvertedFrameDataToArray(Pixeles,
            ColorImageFormat.Bgra);
        }
        int stride = width * FormatoDePixel.BitsPerPixel / 8;
        return BitmapSource.Create(width, height, 96, 96, FormatoDePixel,
      null, Pixeles, stride);
    }
    //Método encargado de desplegar la imagen que está capturando el
Kinect
    //dentro de la ventana de la aplicación
    void ColorReadFrameArrived(object sender,
ColorFrameArrivedEventArgs e){
        using (var frame = e.FrameReference.AcquireFrame()) {
            if (frame != null) {
                Imagen.Source = ToBitmap(frame);
            }
        }
    }
    //Método encargado de sacar del frame la información de la primera
persona detectada
    //y de esa obtener la información de la cara
    void BodyReadFrameArrived(object sender,
BodyFrameArrivedEventArgs e) {
        using (var frame = e.FrameReference.AcquireFrame()) {
            if (frame != null) {
```

```
            frame.GetAndRefreshBodyData(Bodies);
            Body Cuerpo = Bodies.Where(b => b.IsTracked).FirstOrDefault();
            if (!FaceSource.IsTrackingIdValid) {
                if (Cuerpo != null) {
                    FaceSource.TrackingId = Cuerpo.TrackingId;
                }
            }
        }
    }
}
```

También se incluye el método **FaceReadFrameArrived** el cual obtiene del *frame* la información de los elementos de la cara a través de las herramientas **FacePointType** y **FaceProperty**.

```
//Método que obtiene diferentes puntos de la cara así como sus estados.
void FaceReadFrameArrived(object sender, FaceFrameArrivedEventArgs e) {
    using (var Frame = e.FrameReference.AcquireFrame()) {
    if (Frame != null) {
        FaceFrameResult Resultado = Frame.FaceFrameResult;
        if (Resultado != null) {
            var ojoIzq = Resultado.FacePointsInColorSpace[FacePointType.EyeLeft];
            var ojoDer = Resultado.FacePointsInColorSpace[FacePointType.EyeRight];
            var Nariz = Resultado.FacePointsInColorSpace[FacePointType.Nose];
            var bocaIzq =
```

```
                Resultado.FacePointsInColorSpace[FacePointType.MouthCornerLeft];
                var bocaDer = Resultado.FacePointsInColorSpace[FacePointType.MouthCornerRight];
                var ojoIzqCerrado = Resultado.FaceProperties[FaceProperty.LeftEyeClosed];
                var ojoDerCerrado = Resultado.FaceProperties[FaceProperty.RightEyeClosed];
                var bocaAbierta = Resultado.FaceProperties[FaceProperty.MouthOpen];
                    //Método opcional al que se le envía la información de los elementos
                    //de la cara
                    ConfiguracionDelPrograma(ojoIzq,ojoDer,Nariz,bocaIzq, bocaDer,
                            ojoIzqCerrado,ojoDerCerrado,bocaAbierta);
                }
            }
        }
    }
    //Método opcional
    private void ConfiguracionDelPrograma(PointF ojoIzq, PointF ojoDer, PointF nariz,
                    PointF bocaIzq, PointF bocaDer, DetectionResult ojoIzqCerrado,
                    DetectionResult ojoDerCerrado, DetectionResult bocaAbierta) {
```

```
    }
```

Finalmente el método **Window_Closing** se utiliza para cerrar todos los canales que se abrieron para la lectura de información del Kinect.

```
    //Método que se ejecuta cuando se cierra la ventana de la aplicación y que tiene
    //como función la de cerrar los canales de comunicación abiertos con el Kinect.
    private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e) {
        if (ColorRead != null) {
            ColorRead.Dispose();
            ColorRead = null;
        }
        if (BodyRead != null) {
            BodyRead.Dispose();
            BodyRead = null;
        }
        if (FaceReader != null) {
            FaceReader.Dispose();
            FaceReader = null;
        }
        if (FaceSource != null) {
            FaceSource.Dispose();
            FaceSource = null;
        }
        if (MiKinect != null) {
            MiKinect.Close();
```

```
    }
}
```

8.5 Ejemplo: despertar al conductor (*FaceProperty*)

La aplicación emplea **FaceProperty** para determinar si los ojos de la persona detectada están cerrados y si esto se cumple, un mensaje saldrá en la pantalla junto con un sonido que darán aviso al conductor del peligro.

Requerimientos

- Proyecto **WPF**.
- Plantilla cara.
- Recursos: un sonido .wav
- Bibliotecas: System.Media, System.Windows.Threading, Microsoft Kinect, Microsoft.Kinect.Face.

Código en XAML

El archivo **MainWindow.xaml** del proyecto contiene la imagen de lo que está captando la cámara del Kinect y un **Canvas** que incluye el **textBlock** con el mensaje que aparecerá y desaparecerá según la condición del conductor.

```
<Viewbox>
  <Grid Width="1920" Height="1080">
    <Image Name="camera" />
    <Canvas Name="canvas">
      <TextBlock x:Name="textBlock" Canvas.Left="473"
      TextWrapping="Wrap"
        Text="¡CUIDADO!" Canvas.Top="416"
        FontSize="200"
        Foreground="#FFEC1515"/>
    </Canvas>
```

```
    </Grid>
</Viewbox>
```

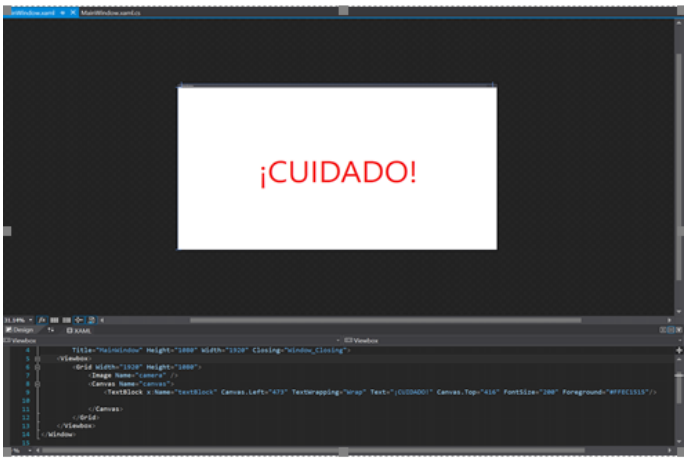


Figura 8.1

Código en C#

Dentro de las instrucciones del programa, se agrega la variable tipo **SoundPlayer** y la instrucción para almacenar el sonido tipo wav que se va a reproducir cuando el conductor se quede dormido. También se añade la variable para obtener la información de la cara y la variable y la activación del **timer** que permitirá verificar cada cierto tiempo el estado del conductor.

```
FaceFrameResult result; //Se emplea para obtener la información de la cara

DispatcherTimer timer; //Sirve para verificar el estado de conductor cada cierto tiempo

SoundPlayer Beep; //Almacena el sonido a reproducir cuando se duerme el conductor

public MainWindow() {
    Beep = new SoundPlayer(Properties.Resources.beep);
        timer = new DispatcherTimer();
        timer.Interval = new TimeSpan(0, 0, 0, 3);
        timer.Tick += new EventHandler(Timer_Tick);
```

```
        timer.IsEnabled = true;
```

Después hay que dirigirse al apartado para realizar la programación para emplear la aplicación, ahí se establecerá que si el ojo derecho e izquierdo están cerrados se habilite el **textBlock** y el sonido **Beep** se reproduzca dándole un **delay** de 500 milisegundos, en caso de que no la visibilidad del **textBlock** será nula. Esto simulará un mensaje de emergencia para que el conductor reaccione y abra los ojos.

```
private void Timer_Tick(object sender, EventArgs e) {
  var ojoIzquierdo = result.FacePointsInColorSpace[FacePointType.EyeLeft];
  var ojoDerecho = result.FacePointsInColorSpace[FacePointType.EyeRight];
  var nose = result.FacePointsInColorSpace[FacePointType.Nose];
  var bocaIzq = result.FacePointsInColorSpace[FacePointType.MouthCornerLeft];
  var bocaDer = result.FacePointsInColorSpace[FacePointType.MouthCornerRight];
  var bocaAbierta = result.FaceProperties[FaceProperty.MouthOpen];
  //Obtiene información del estado actual tanto del ojo izquierdo como del derecho
  var ojoDerCerrado = result.FaceProperties[FaceProperty.RightEyeClosed];
  var ojoIzqCerrado = result.FaceProperties[FaceProperty.LeftEyeClosed];
  //Si ambos ojos están cerrados, mostrar el mensaje “CUIDADO” y activar el audio.
  if ((ojoIzqCerrado & ojoDerCerrado) == DetectionResult.Yes) {
      textBlock.Visibility = Visibility.Visible;
      Beep.Play();
  }
```

```
    else { //Si los ojos están abiertos, quitar de la pantalla el mensaje "Cuidado"
        textBlock.Visibility = Visibility.Collapsed;
    }
}
```

8.6 Ejemplo: detectar emociones (*FaceProperty*)

La aplicación empleará **FaceProperty** para detectar tres de las expresiones faciales que pueden ser descubiertas por el Kinect.

Requerimientos

- Proyecto **WPF**.
- Plantilla cara.
- Biblioteca: Microsoft.Kinect y Microsoft.Kinect.Face

Código en XAML

El archivo **MainWindow.xaml** del proyecto contiene la imagen captada por la cámara del Kinect y un **Canvas** con tres diferentes **textBox**, uno de los cuales será manipulado desde el código para indicar la expresión que está realizando la persona frente al Kinect.

```
<Viewbox>
  <Grid Width="1920" Height="1080">
    <Image Name="camera" />
    <Canvas Name="canvas">
    <TextBox x:Name="textBox" Height="84" Canvas.Left="1183" TextWrapping="Wrap"
             Text="Condicion:" FontSize="50" Canvas.Top="232" Width="263"
             Background="White" />
    <TextBox x:Name="textBox2" Height="84" Canvas.Left="1451"
```

```
        TextWrapping="Wrap"
                Text="" FontSize="50" Canvas.Top="232" Width="444"
                Background="White"/>
        <TextBox x:Name="textBox3" Height="176" Canvas.Left="1262"
        TextWrapping="Wrap"
                Text="Condicion de la Cara" FontSize="70"
                Canvas.Top="10" Width="578"
                Background="White" />
        </Canvas>
    </Grid>
</Viewbox>
```

Figura 8.2

Código en C#

El programa solo estará atento a tres estados de la persona:

- **Engaged** (concentrado): se detecta cuando el sujeto observado se encuentre serio o mirando fijamente.
- **Happy** (feliz): si la persona se encuentra sonriente.
- **LookingAway** (mirada perdida): cuando está mirando hacia otro lugar que no sea la cámara.

Para lograr esto, se agregan al método **FaceReadFrameArrived** las instrucciones para

la detección de dichos estados y se llama al método **ConfiguracionDelPrograma** que coloca el mensaje apropiado en la ventana según lo que haya detectado.

```
//Método que obtiene diferentes puntos de la cara así como sus estados.
void FaceReadFrameArrived(object sender, FaceFrameArrivedEventArgs e) {
    using (var Frame = e.FrameReference.AcquireFrame()) {
    if (Frame != null) {
        FaceFrameResult Resultado = Frame.FaceFrameResult;
        if (Resultado != null) {
            var ojoIzq = Resultado.FacePointsInColorSpace[FacePointType.EyeLeft];
            var ojoDer = Resultado.FacePointsInColorSpace[FacePointType.EyeRight];
            var Nariz = Resultado.FacePointsInColorSpace[FacePointType.Nose];
            var bocaIzq = Resultado.FacePointsInColorSpace[FacePointType.MouthCornerLeft];
            var bocaDer = Resultado.FacePointsInColorSpace[FacePointType.MouthCornerRight];
            var ojoIzqCerrado = Resultado.FaceProperties[FaceProperty.LeftEyeClosed];
            var ojoDerCerrado = Resultado.FaceProperties[FaceProperty.RightEyeClosed];
            var bocaAbierta = Resultado.FaceProperties[FaceProperty.MouthOpen];
            var Concentrado = Resultado.FaceProperties[FaceProperty.Engaged];
```

```
            var Feliz = Resultado.FaceProperties[FaceProperty.Happy];
            var miradaPerdida = Resultado.FaceProperties[FaceProperty.LookingAway];
            //Método opcional al que se le envía la información de los elementos
            //de la cara
            ConfiguracionDelPrograma(Concentrado, Feliz, miradaPerdida);
            }
        }
    }
}
//Método opcional
private void ConfiguracionDelPrograma(DetectionResult Concentrado,
                DetectionResult Feliz, DetectionResult miradaPerdida) {
    if (Feliz == DetectionResult.Yes) {
        textBox2.Text = String.Format("Feliz");
    }
    else if (miradaPerdida == DetectionResult.Yes){
        textBox2.Text = String.Format("Distraido");
    }
    else if (Ocupado == DetectionResult.Yes) {
        textBox2.Text = String.Format("Concentrado");
    }
}
```

8.7 Ejemplo: poner una máscara (*FacePointType*)

La aplicación que emplea FacePointType para detectar expresiones faciales utilizara tres diferentes textBox que indicarán los gestos y condiciones en que se encuentra.

Requerimientos

- Proyecto WPF.
- Plantilla cara.
- Biblioteca: Microsoft.Kinect y Microsoft.Kinect.Face
- Recursos: la imagen que se colocará como máscara.

Código en XAML

El archivo **MainWindow.xaml** del proyecto incluye la imagen donde se proyecta el video capturado por el Kinect y un Canvas con una imagen la cual se posicionará en las coordenadas de la cara en el punto específico.

```
<Grid Width="1920" Height="1080">
  <Image x:Name="camera"/>
  <Canvas x:Name="canvas">
    <Image x:Name="Happyimg" Source="HappyFace.png"
    Canvas.Left="10"
      Canvas.Top="105" Height="250" Width="250" />
  </Canvas>
</Grid>
```

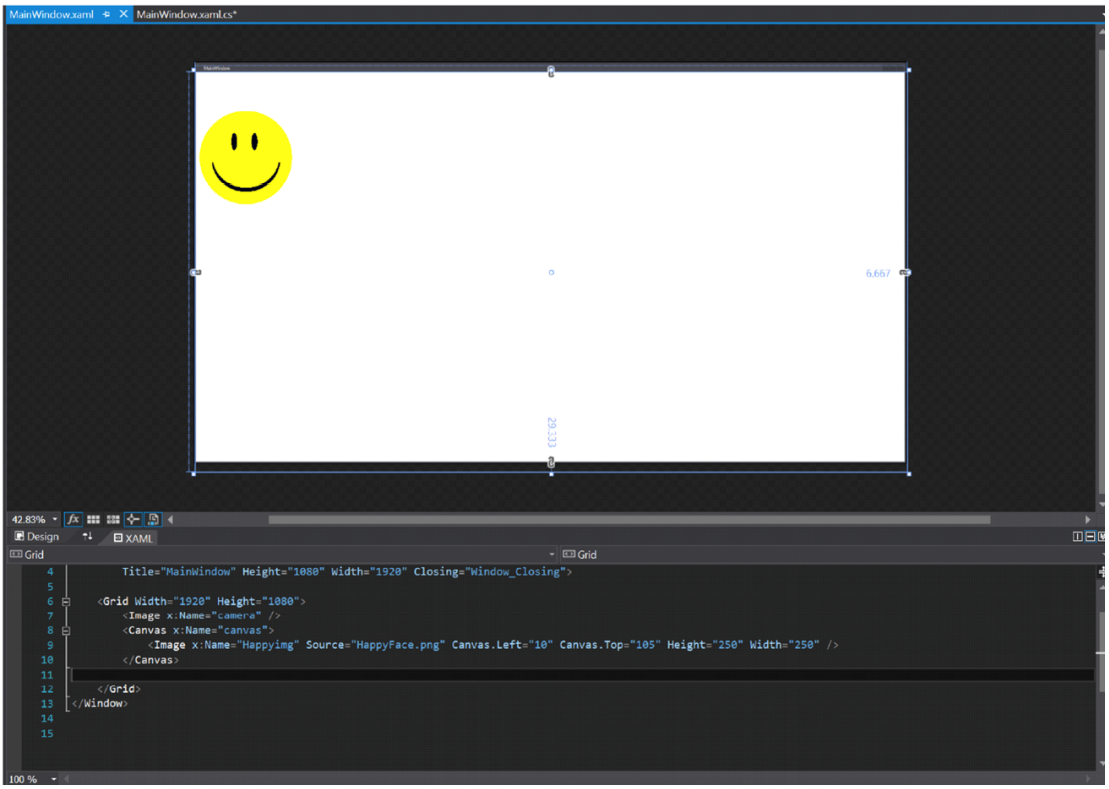


Figura 8.3

Código en C#

El programa emplea la variable **nariz** para almacenar la posición de esta frente al Kinect. La imagen se centra de acuerdo a dicha posición de la siguiente manera:

```
Canvas.SetTop(Happyimg, nariz.Y - Happyimg.Height / 2.0);
Canvas.SetLeft(Happyimg, nariz.X - Happyimg.Width / 2.0);
```

Ejecución

Se ejecuta el programa, la imagen se sitúa en la posición de la nariz de la persona detectada, simulando portar una máscara.

Capítulo 9. Comandos de voz – Kinect V2

El dispositivo Kinect v2, gracias a sus distintas capacidades, puede detectar comandos de voz, estos se pueden combinar con la detección del cuerpo para producir aplicaciones más versátiles.

Para el desarrollo de este tipo de aplicaciones se requieren elementos de software adicionales (referencias) agregados al proyecto de programación con una plantilla diferente a la que se ha utilizado.

9.1 Bibliotecas para el reconocimiento de voz

Para la realización de aplicaciones que reconozcan comandos de voz es necesario incluir las bibliotecas que contienen las herramientas de programación diseñadas para tal fin. Estas bibliotecas estarán disponibles una vez que se hayan añadido las referencias correspondientes.

```
using Microsoft.Kinect;
using Microsoft.Speech.AudioFormat;
using Microsoft.Speech.Recognition;
```

9.2 Configuración para reconocimiento de voz

Para que la aplicación pueda realizar el reconocimiento de voz, es necesario contar con una variable para el motor de reconocimiento de voz y otra para el flujo de datos de audio.

```
KinectSensor miKinect = null; //Nombre del dispositivo Kinect.
KinectAudioStream audioStr = null; //Flujo de datos de audio.
SpeechRecognitionEngine motorVoz = null; //Motor de reconocimiento de voz.
```

Dentro del **MainWindow** se añaden las instrucciones para activar y configurar al Kinect para el reconocimiento de la voz.

```
public MainWindow()
```

```
{
    InitializeComponent();
    Kinect_Config(); // Iniciar y configurar el Kinect
}
```

El método **Kinect_Config** además de activar al Kinect ahora se encargará de definir los comandos de voz que serán reconocidos. En la sección indicada por los comentarios se deben agregar todas las instrucciones de voz que a detectar.

```
private void Kinect_Config()
{
    //Buscar el primer Kinect conectado y lo asocia a miKinect
    miKinect = KinectSensor.GetDefault();
    if (miKinect != null) {
     //Obtener el flujo de audio
     IReadOnlyList<AudioBeam> audioBeamList = miKinect.AudioSource.AudioBeams;
     System.IO.Stream audioStream = audioBeamList[0].OpenInputStream();
     //Crear flujo de datos
     audioStr = new KinectAudioStream(audioStream);
    }
    else { }
    RecognizerInfo ri = KinectRecognizer();
    if (null != ri) {
     this.motorVoz = new SpeechRecognitionEngine(ri.Id);
  /*-------SECCIÓN DONDE SE DEFINEN LOS COMANDOS DE VOZ A UTILIZAR —------*/
    var Instrucciones = new Choices();
```

```
    Instrucciones.Add(new SemanticResultValue("play", "PLAY"));
    Instrucciones.Add(new SemanticResultValue("stop", "STOP"));
  /*-----------------------------------------------------------------*/
    //Crear la gramática
    var gb = new GrammarBuilder { Culture = ri.Culture };
    gb.Append(Instrucciones);
    var g = new Grammar(gb);
    this.motorVoz.LoadGrammar(g);
    //Activar el método para reconocer los comandos
    this.motorVoz.SpeechRecognized += this.SpeechRecognized;
    //Indicar que el Speech esta activo
    this.audioStr.SpeechActive = true;
    //Seleccionar el formato que utilizará el motor de voz
     this.motorVoz.SetInputToAudioStream(
     this.audioStr, new SpeechAudioFormatInfo(EncodingFormat.Pcm,
     16000, 16, 1, 32000, 2, null));
     this.motorVoz.RecognizeAsync(RecognizeMode.Multiple);
    }
    else { }
  //Definir el estatus del Kinect
    this.miKinect.Open();
  }
```

El método **SpeechRecognized**, que fue activado durante la configuración del Kinect, analiza los datos recibidos para determinar qué comandos de voz fueron expresado por el usuario y, en este caso en particular, se despliega en la consola el nombre del comando.

```
  private void SpeechRecognized(object sender,
SpeechRecognizedEventArgs e)
```

```
{
    //Especificar nivel de confidencia
    const double ConfidenceThreshold = 0.3;
    //Realizar acción de acuerdo al comando recibido
    if (e.Result.Confidence >= ConfidenceThreshold) {
        switch (e.Result.Semantics.Value.ToString()) {
            case "PLAY":
                Console.WriteLine("PLAY");
                break;
            case "STOP":
                Console.WriteLine("STOP");
                break;
            }
        }
}
```

Otro de los métodos que se ejecutan durante la configuración del Kinect es **KinectRecognizer** el cual se encarga de filtrar el flujo de datos recopilados en el idioma inglés ("en-US"), español (es-MX) o algún otro según se indique.

```
private static RecognizerInfo KinectRecognizer() {
    IEnumerable<RecognizerInfo> recognizers;
    try {
     recognizers = SpeechRecognitionEngine.InstalledRecognizers();
    }
    catch (COMException) {
     return null;
    }
```

```
    foreach (RecognizerInfo recognizer in recognizers) {
     string value;
     recognizer.AdditionalInfo.TryGetValue("Kinect", out value);
     if ("True".Equals(value, StringComparison.OrdinalIgnoreCase) &&
          "en-US".Equals(recognizer.Culture.Name,
          StringComparison.OrdinalIgnoreCase)) {
          return recognizer;
        }
    }
    return null;
  }
```

Por último, el siguiente método debe ser incluido en el programa para finalizar la detección de Kinect, el flujo de audio y el motor de voz.

```
  private void WindowClosing(object sender, CancelEventArgs e) {
    if(this.miKinect != null)
  {
        this.miKinect.Close();
        this.miKinect = null;
  }
  }
```

9.3 Propiedad ángulo de una imagen

Para que una aplicación pueda girar una imagen se le debe agregar (en el programa en XALM) una transformación de rotación que indique el ángulo (Angle) inicial y el nombre del ángulo (Name) tal y como se muestra en el siguiente ejemplo:

```
  <Image.RenderTransform>
    <TransformGroup>
```

```
        <RotateTransform Angle="0" x:Name="elementoRotation"/>
      </TransformGroup>
    </Image.RenderTransform>
```

Dentro del programa en C# se debe incluir la siguiente biblioteca:

```
using System.Runtime.InteropServices;
```

En el momento requerido, se cambia el ángulo de la imagen como en la siguiente instrucción que gira la imagen anterior 90º.

```
elementoRotation.Angle =+ 90;
```

9.4 Ejemplo: juego de la ruleta

Este programa da un ejemplo del uso de comandos de voz para hacer girar una ruleta.

Requerimientos

- Proyecto **WPF**.
- Plantilla speech.
- Bibliotecas: Microsoft.Kinect, Microsoft.Speech.AudioFormat, Microsoft.Speech.Recognition, System.Windows.Threading, System.Runtime.InteropServices.
- Imágenes: una ruleta y una flecha.

Nota: como anexo se incluye el proyecto WPF para el manejo de comandos de voz con las referencias necesarias. Se puede sacar una copia del mismo y modificarlo para realizar la aplicación. Si se decide iniciar desde cero, no olvidar añadir al proyecto las referencias. En caso de copiar el código y pegarlo en un nuevo proyecto es importante verificar que el nombre del proyecto creado sea el empleado tanto en el programa en XAML como en el programa en C#.

Código en XAML

El archivo **MainWindow.xaml** del proyecto incluye un Grid que, entre otras cosas, contiene la imagen de la ruleta a la cual se hará girar o detener con comandos de voz. Es importante observar que dicha imagen contiene la transformación requerida para poder modificar su ángulo (para girarla).

```
<Grid>
  <Image x:Name="background" Source="Resources/background.png"
      Margin="0,0,-51.571,0.143"/>
  <Image x:Name="Ruleta" Source="Resources/RuletaColores.png"
      Margin="160,57,167.429,60.143">
        <Image.RenderTransform>
            <TransformGroup>
                <RotateTransform                    Angle="0"
                x:Name="ruletaRotation"/>
            </TransformGroup>
        </Image.RenderTransform>
    </Image>
    <Image x:Name="Flecha" Source="Resources/flecha.png"
        Margin="86,146,359.429,140.143" />
</Grid>
```

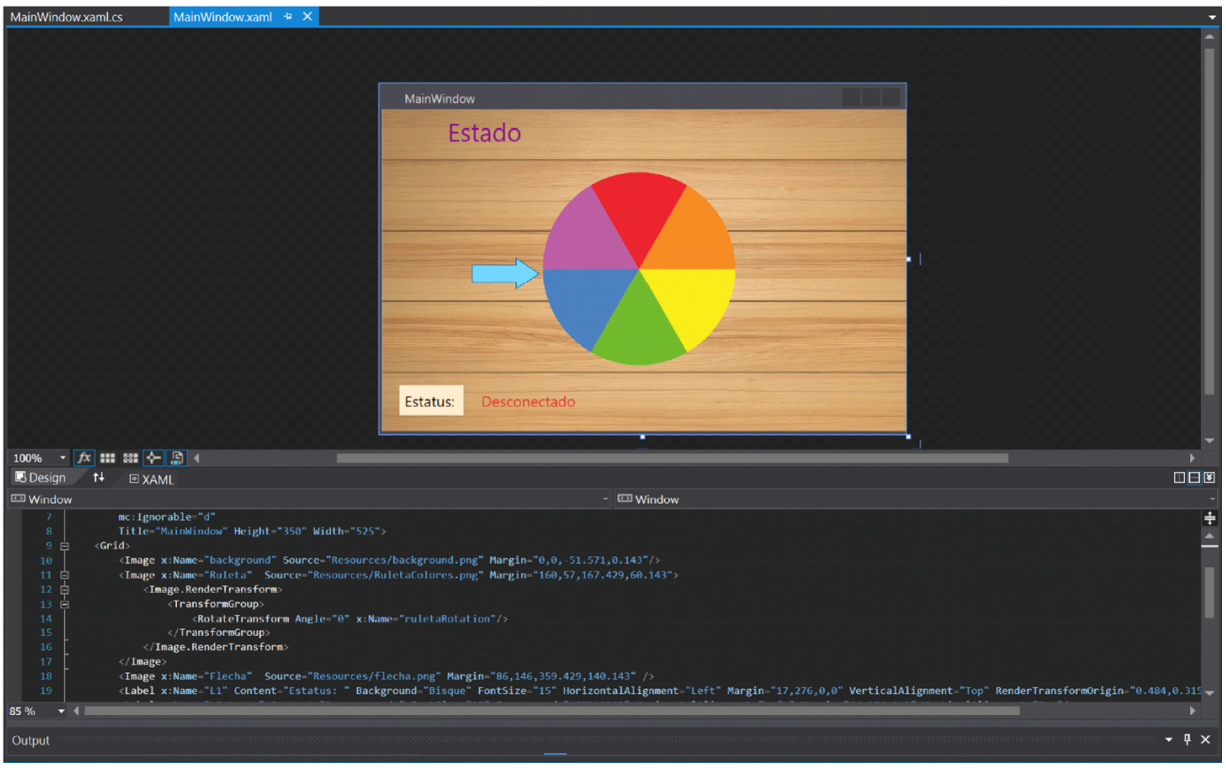


Figura 9.1

Código en C#

Para realizar la aplicación se efectuaron las siguientes modificaciones a la plantilla para el reconocimiento de comandos de voz.

Las bibliotecas que se emplean para el juego de la ruleta son:

```
using Microsoft.Kinect;
using System.IO;
using Microsoft.Speech.AudioFormat;
using Microsoft.Speech.Recognition;
using System.Runtime.InteropServices;
using System.Windows.Threading;
```

En la parte superior del programa, además de las variables para configurar el Kinect y la detección de comandos de voz, se incluyen variables para controlar el giro de la imagen de la ruleta.

```
KinectSensor miKinect = null; //Nombre del dispositivo Kinect.
```

```
KinectAudioStream audioStr = null; //Flujo de datos de audio.
SpeechRecognitionEngine motorVoz = null; //Motor de reconocimiento de voz.
DispatcherTimer timer; //Dispatchertimer.
double degrees = 10; //Grados de giro de la ruleta.
```

Posteriormente, dentro del **MainWindow** se realiza la configuración del Kinect y se activa el método **Timer_Tick** que se utiliza para simular el movimiento de la ruleta.

```
public MainWindow() {
    InitializeComponent();
    //Configurar e iniciar el Kinect
    Kinect_Config();
    //Iniciar el timer que gira la ruleta.
    //El método Timer_Tick se ejecuta cada segundo
    timer = new DispatcherTimer();
    timer.Interval = new TimeSpan(0, 0, 0, 1);
    timer.Tick += new EventHandler(Timer_Tick);
    timer.IsEnabled = true;
}
```

Cada segundo se ejecutará el método **Timer_Tick** para modificar el ángulo de la imagen de la ruleta en la cantidad especificada por la variable degrees. Si el valor de la variable degrees es igual a 10, entonces cada segundo se girará la imagen 10 grados. Por el contrario, si es igual a cero, la imagen no gira.

```
private void Timer_Tick(object sender, EventArgs e) {
    ruletaRotation.Angle =+ degrees;
}
```

El siguiente paso es modificar el método para la configuración del Kinect **Kinect_Config** al indicar los comandos de voz que deben ser detectados para detener y accionar el giro de la ruleta. En esta ocasión se emplearon comandos en inglés.

```
private void Kinect_Config()
{
  //Buscar el primer Kinect conectado y lo asocia a miKinect
  miKinect = KinectSensor.GetDefault();
  if (miKinect != null) {
   //Obtener el flujo de audio
   IReadOnlyList<AudioBeam> audioBeamList =
   miKinect.AudioSource.AudioBeams;
   System.IO.Stream audioStream =
   audioBeamList[0].OpenInputStream();
   //Crear flujo de datos
   audioStr = new KinectAudioStream(audioStream);
  }
  else { }
  RecognizerInfo ri = KinectRecognizer();
  if (null != ri) {
   this.motorVoz = new SpeechRecognitionEngine(ri.Id);
 /*-------SECCIÓN DONDE SE DEFINEN LOS COMANDOS DE VOZ
A UTILIZAR —------*/
    var Instrucciones = new Choices();
    Instrucciones.Add(new SemanticResultValue("play", "PLAY"));
    Instrucciones.Add(new SemanticResultValue("stop", "STOP"));
    Instrucciones.Add(new SemanticResultValue("continue", "PLAY"));
 /*-----------------------------------------------------------------*/
    //Crear la gramática
    var gb = new GrammarBuilder { Culture = ri.Culture };
```

```
            gb.Append(Instrucciones);
            var g = new Grammar(gb);
            this.motorVoz.LoadGrammar(g);
            //Activar el método para reconocer los comandos
            this.motorVoz.SpeechRecognized += this.SpeechRecognized;
            //Indicar que el Speech esta activo
            this.audioStr.SpeechActive = true;
            //Seleccionar el formato que utilizará el motor de voz
            this.motorVoz.SetInputToAudioStream(
            this.audioStr, new SpeechAudioFormatInfo(EncodingFormat.Pcm,
            16000, 16, 1, 32000, 2, null));
            this.motorVoz.RecognizeAsync(RecognizeMode.Multiple);
          }
          else { }
          //Definir el estatus del Kinect
          this.miKinect.Open();
      }
      ///<summary>
      /// Método que libera los recursos del Kinect cuando se termina la
aplicación
      ///</summary>
      private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
      {
          if( this.miKinect != null)
          {
```

```
        this.miKinect.Close();
        this.miKinect = null;
      }
    }
```

Finalmente, se modifica el método **SpeechRecognized** para cambiar el valor de la variable **degrees** para que la ruleta gire o se detenga según la orden dada. Como se observa, si el comando de voz detectado es "PLAY", la ruleta seguirá girando pero, si el comando es "STOP" se asignara a la variable **degrees** el valor de cero para detener el giro.

```
    private void SpeechRecognized(object sender, SpeechRecognizedEventArgs e) {
        //Especificar nivel de confidencia
        const double ConfidenceThreshold = 0.3;
        //Realizar acción de acuerdo al comando recibido
        if (e.Result.Confidence >= ConfidenceThreshold) {
            switch (e.Result.Semantics.Value.ToString()) {
                case "PLAY":
                    degrees = 10;
                    break;
                case "STOP":
                    degrees = 0;
                    break;
                }
            }
    }
```

Una vez indicado lo anterior ya es posible realizar la ejecución del programa.

Capítulo 10. Capturar el movimiento

Kinect es un dispositivo que cuenta con una cámara RGB con una resolución de 1280 x 960 pixeles para Kinect V1 y 1920 x 1080 pixeles en V2. También contiene un emisor de infrarrojos y un sensor de profundidad. Este capítulo incluye ejemplos sobre el uso de estas tecnologías.

10.1 Ejemplo: tomar fotos (Kinect V1)

Existen dos formas para obtener información del Kinect. La primera de ellas se logra al activar un evento que provoca que el Kinect envíe información al programa cada cierto tiempo (esta es la forma en la que funcionan los programas descritos en los capítulos anteriores).

La segunda manera de obtener información es solicitándola al Kinect cuando se necesita. La aplicación que se explicará a continuación utiliza este método para tomar una foto con el Kinect.

Requerimientos

- Proyecto **WPF**.
- Plantilla de cámara para Kinect V1 (ver anexo).

Código en XAML

Los elementos que se agregan al Grid en el archivo XAML son: el espacio donde se mostrará la foto y un botón que al dar clic ejecutará el método encargado de tomar la foto. Las dimensiones de la ventana son **Height**="770" y **Width**="770".

```
<Grid>
  <Image Name="Image" Width="640" Height="480"/>
  <Button x:Name="button" Content="Tomar Foto"
HorizontalAlignment="Left"
  VerticalAlignment="Top" Width="76" Click="tomar_Foto"/>
```

```
</Grid>
```

Código en C#

A través de la plantilla modificada para la cámara se agrega la siguiente biblioteca que se empleará para construir el nombre del archivo que almacenará la foto con la fecha y hora:

```
using System.Globalization;
```

También se debe añadir el método **tomar_Foto** que se ejecuta cada vez que el usuario presiona el botón colocado en XAML. Este toma la foto y la guarda en la computadora. Le solicita al Kinect un frame de datos utilizando la función **OpenNextFrame** (el valor 100 que se le envía a esta función es el número de milisegundos a esperar por el frame). Posteriormente, obtiene la fecha y hora del sistema para construir el nombre que se le dará al archivo en el que se almacenará la foto. Finalmente, la guarda en la computadora en formato JPG utilizando J**pegBitmapEncoder**. La imagen se almacena dentro del proyecto, en la subcarpeta **bin\debug**.

```
private void tomar_Foto(object sender, RoutedEventArgs e) {
  if (this.miKinect == null){
   return;
  }
  try {
     // Solicitar un frame de datos al Kinect
     using (ColorImageFrame frame =
     this.miKinect.ColorStream.OpenNextFrame(100)){
          if (frame != null) {
               frame.CopyPixelDataTo(this.cantidadPixeles);
               this.imagen.WritePixels(new Int32Rect(0, 0,
               this.imagen.PixelWidth,
                                this.imagen.PixelHeight),
                                this.cantidadPixeles,
                                this.imagen.PixelWidth * sizeof(int), 0);
```

```
            }
        }
    }
    catch (Exception ex) {
        //Reportar error
    }
    //Generar el nombre del archivo en el que se guardará la foto.
    //El nombre del archivo es la fecha-hora en que se tomó la foto.
    DateTime thisDay = DateTime.Now;
    string fecha = thisDay.ToString("dd-MM-yy",
  DateTimeFormatInfo.InvariantInfo);
    string time = thisDay.ToString("hh'-'mm'-'ss",
                        CultureInfo.CurrentUICulture.DateTimeFormat);
string fileName = fecha + '_' + time + ".jpg";
    if (File.Exists(fileName)) {
        File.Delete(fileName);
    }
    //Guardar la foto en format JPG en la carpeta del proyecto
    using (FileStream foto = new FileStream(fileName,
  FileMode.CreateNew)) {
        JpegBitmapEncoder jpgEncoder = new JpegBitmapEncoder();
        jpgEncoder.QualityLevel = 70;
        jpgEncoder.Frames.Add(BitmapFrame.Create(this.imagen));
        jpgEncoder.Save(foto);
        foto.Flush();
        foto.Close();
```

```
            foto.Dispose();
        }
    }
```

Ejecución

Al ejecutar el programa se muestra en la ventana la foto que se tomó al presionar el botón.

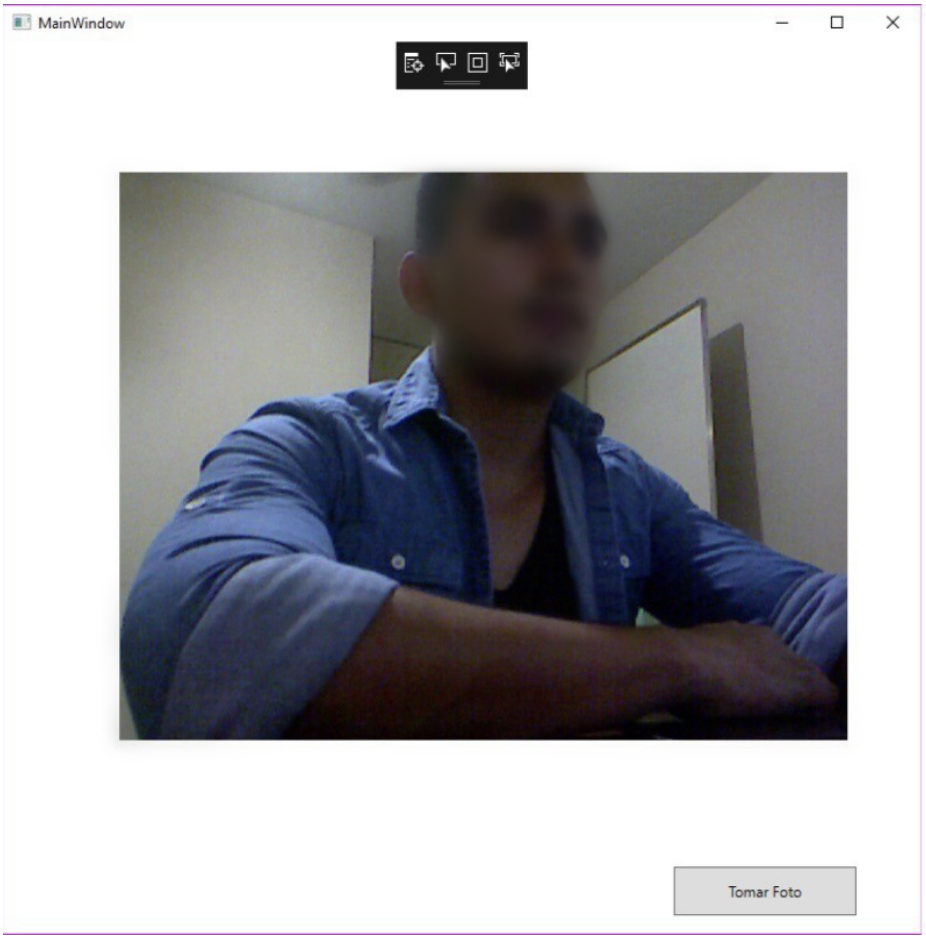


Figura 10.1

10.2 Ejemplo: tomar fotos (Kinect V2)

La siguiente aplicación permite utilizar el Kinect V2 como una cámara fotográfica.

Requerimientos

- Proyecto **WPF**.
- Plantilla de cámara para Kinect V2 (ver anexo).

Código en XAML

Los elementos que se agregan al Grid en el archivo XAML son: el espacio donde se mostrará la foto y un botón que ejecutará el método encargado de tomar la foto. Las dimensiones de la ventana son **Height**="770" y **Width**="770".

```
<Grid>
  <Image Name="Image" Width="640" Height="480"/>
  <Button x:Name="button" Content="Tomar Foto"
 HorizontalAlignment="Left"
  VerticalAlignment="Top" Width="76" Click="tomar_Foto"/>
</Grid>
```

Código en C#

Gracias a la plantilla modificada para la cámara se agregará la siguiente biblioteca con la que se construirá el nombre del archivo que almacenará la foto con la fecha y hora.

```
using System.Globalization;
```

También se debe añadir el método **tomar_Foto** el cual se ejecuta cada vez que el usuario presiona el botón colocado en XAML. Este toma la foto y la guarda en la computadora. Primero forma el nombre del archivo en el que almacenará la foto con fecha del día, después verifica si el archivo existe o no en la computadora (si existe lo borra). Finalmente, crea la imagen en formato JPG, mediante **JpegBitmapEncoder** y la almacena dentro del proyecto, en la subcarpeta **bin\debug**.

```
private void tomar_Foto(object sender, RoutedEventArgs e) {
  if (this.imagen != null) {
   //Construye el nombre del archive en el que se guarda la foto
   //El nombre del archivo es la fecha-hora en que se tomó la foto.
   DateTime thisDay = DateTime.Now;
   string fecha = thisDay.ToString("dd-MM-yy",
```

```
        DateTimeFormatInfo.InvariantInfo);
        string time = thisDay.ToString("hh'-'mm'-'ss",
                                CultureInfo.CurrentUICulture.DateTimeFormat);
        string fileName = fecha + '_' + time + ".jpg";
        //Si el archive ya existe lo borra
        if (File.Exists( ilename)) {
            File.Delete( ilename);
        }
        try {
            // Guardar la foto en format JPG
            using (FileStream foto = new FileStream(fileName,
            FileMode.CreateNew)) {
                JpegBitmapEncoder jpgEncoder = new JpegBitmapEncoder();
                jpgEncoder.QualityLevel = 70;
                jpgEncoder.Frames.Add(BitmapFrame.Create(this.imagen));
                jpgEncoder.Save(foto);
                foto.Flush();
                foto.Close();
                foto.Dispose();
            }
        }
        catch (IOException) {// Reportar error}
    }
}
```

Ejecución

En el programa se muestra en la ventana la foto que se tomó al presionar el botón.

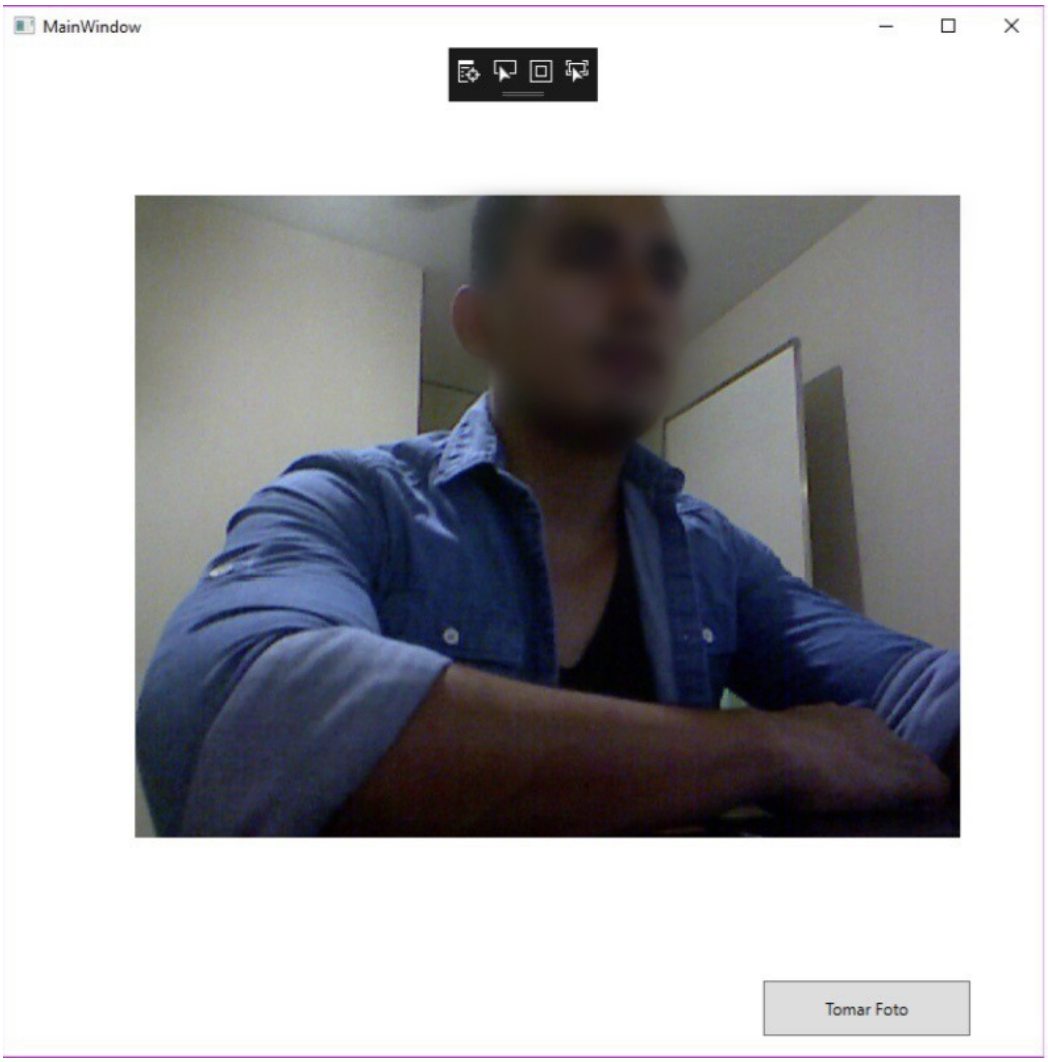


Figura 10.2

10.3 Ejemplo: cámara escondida (Kinect V1)

La aplicación que se desarrolla a continuación funciona como una cámara escondida; cada vez que detecta a una persona, le toma una foto y la guarda en la computadora.

Requerimientos

- Proyecto **WPF**.
- Plantilla Skeleton Kinect V1.

Código en XAML

El archivo MainWindow.xaml del proyecto incluye solo un elemento Image para visualizar la foto que se tome.

```
<Grid>
  <Image Name="Image" Width="640" Height="480"/>
</Grid>
```

Código en C#

A través de la plantilla del esqueleto se agrega la siguiente biblioteca que se empleará para construir el nombre del archivo que almacenará la foto con la fecha y hora.

```
using System.Globalization;
```

Esta aplicación utiliza como base la plantilla que detecta el esqueleto. A esta plantilla se le hacen cambios para habilitar el flujo **ColorStream** y guardar la foto. Las variables que se añaden en la parte superior del programa son **imagen** y cantidadPixeles para manipular los datos que envía el Kinect de la cámara y para configurar **ColorStream**. La última variable es **StopFoto**, que se emplea para tomar una sola foto por ocasión.

```
private KinectSensor miKinect; //Nombre del dispositivo Kinect.
private WriteableBitmap imagen; //Se utiliza para generar la imagen
private byte[] cantidadPixeles; //Recibe los bytes que envía el Kinect
bool StopFoto = false; //Se utiliza para generar la imagen la cantidad de fotos a tomar en cada ocasión
```

Dentro del método **Kinect_Config**, se agregarán las instrucciones para configurar **ColorStream** y la imagen, estas se muestran en el siguiente recuadro. Como se ve en el código, no se requiere un evento para adquirir los datos de la cámara ya que con **Skeleton** es suficiente. Lo relativo a la cámara se va a solicitar por demanda cuando se ejecute el método **tomar_Foto** al detectar a una persona.

```
//Configuración de ColorStream
this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
//Colocar la información enviada por el Kinect en la imagen de la ventana
this.cantidadPixeles = new
```

```
byte[this.miKinect.ColorStream.FramePixelDataLength];
   this.imagen = new
WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
                this.miKinect.ColorStream.FrameHeight,
                96.0, 96.0, PixelFormats.Bgr32, null);
    this.Image.Source = this.imagen;
    this.miKinect.SkeletonStream.Enable();
    this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady;
```

Finalmente, en el método **usarSkeleton** se llama al método **tomar_Foto** para tomar una foto cuando el Kinect ha detectado una persona.

```
   private void usarSkeleton(Skeleton skeleton) {
      //Se toma una foto
      if (!StopFoto)
          tomar_Foto();
   }
```

Al ejecutar el proyecto la ventana luce de la siguiente forma. Al inicio la imagen solo se verá en negro ya que la foto solo se toma en el momento en que hay una persona frente al Kinect. La carpeta donde se guarda la imagen es en la carpeta del proyecto dentro de la subcarpeta **bin\debug.**

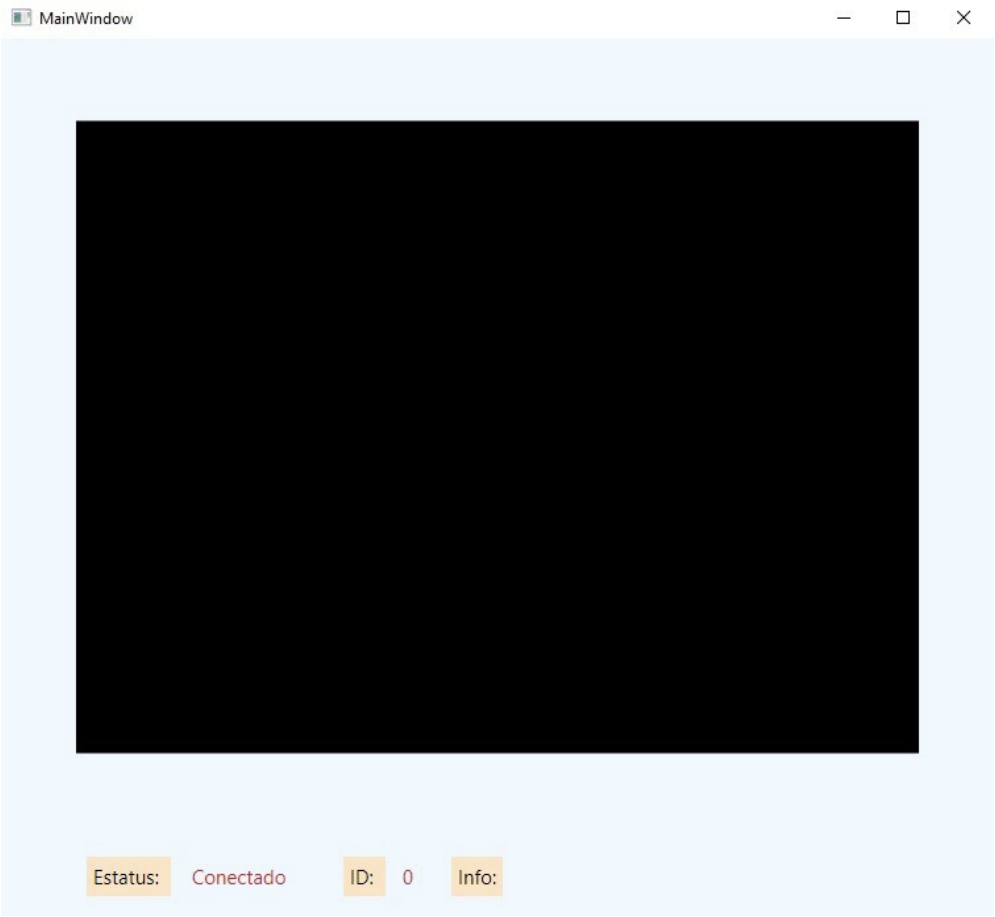


Figura 10.3

10.4 Ejemplo: cámara escondida Kinect V2

La aplicación que se desarrollará a continuación funciona como una cámara escondida; cada vez que detecta a una persona le toma una foto y la guarda en la computadora.

Requerimientos

- Proyecto **WPF**.
- Plantilla Body Kinect V2.

Código en XAML

El código incluye un elemento **Image** para visualizar la foto que se tomará cuando una persona se coloque frente al Kinect. Las dimensiones de la ventana son **Height**="770" y **Width**="770".

```
<Grid>
    <Image Name="Image" Width="640" Height="480" Stretch="Fill"/>
</Grid>
```

Código en C#

Para esta aplicación se emplearán dos tipos de datos vistos anteriormente, los referentes a la cámara y a Body. Los datos relacionados a Body sirven para detectar a la persona y los datos de la cámara toman la foto cuando se detecta a la persona. Para la obtención de los datos se utilizará un nuevo **FrameReader** al que se le denomina **MultisourceFrameReader** el cual podrá acceder a los datos de las dos tecnologías.

El programa utiliza como base la plantilla Kinect Body V2 que será editada para poder utilizar **MulsourceFrameReader**. Lo primero es agregar la librería **Globalization** que permitirá obtener la fecha y hora del día para dar nombre a las fotos que se guardarán.

```
using System.Globalization;
```

Así también, se agregan las variables que permite crear la imagen y controlar la toma de la foto. La primera es **multiReader**, la cual permite obtener los datos provenientes de la cámara y Body. Sigue un arreglo llamado bodies, que se usa para recibir los Body que envía el Kinect. Para la creación de la imagen, se emplean las variables: **imagen**, **iWidth**, **iHeight** y **cantidadPixeles**. Por último, la variable **bStopFoto** se usa para limitar a una la foto tomada.

```
MultiSourceFrameReader multiReader; //FrameReader que recibe datos de distintos tipos
private Body[] bodies = null; //Almacena datos de Body
private WriteableBitmap imagen = null; //Útil para generar la imagen
int iWidth = 0; //Ancho de la imagen
int iHeight = 0; //Alto de la imagen
byte[] cantidadPixeles; //Almacena los datos de la cámara (pixeles)
bool bStopFoto = false; //Para controlar la cantidad de fotos a tomar
```

El método **Kinect_Config** inicia al habilitar el **FrameReader** para recibir los datos del Body y la cámara. Se enlista la rutina **Kinect_FrameReady** al evento **MultisourceFrameArrived** para que sea llamada cada vez que se tienen datos. Posteriormente, se realiza la configuración de la imagen recibida del Kinect para

convertirla en un **bitmap** con el ancho y alto requerido para poder colocarlo en el objeto **image** creado en XAML.

```
private void Kinect_Config() {
  //Buscar el Kinect conectado
  miKinect = KinectSensor.GetDefault();
  /* ------------------- Configuración del Kinect ------------------- */
  //FrameReader para recibir datos de distintas fuentes
  multiReader = miKinect.OpenMultiSourceFrameReader(FrameSourceTypes.Color |
                                                    FrameSourceTypes.Body);
  //Enlistar la rutina que se llama cuando hay datos disponibles
  multiReader.MultiSourceFrameArrived += Kinect_FrameReady;
  //Crear FrameDescription desde ColorFrameSource usando un formato Bgra
  FrameDescription colorFrameDescription = this.miKinect.ColorFrameSource.
                                           CreateFrameDescription(ColorImageFormat.Bgra);
  iWidth = colorFrameDescription.Width;
  iHeight = colorFrameDescription.Height;
  cantidadPixeles = new byte[iWidth * iHeight * ((PixelFormats.Bgr32.BitsPerPixel
                                                  + 7) / 8)];
  //Crear el bitmap a mostrar (Imagen que se visualiza)
  this.imagen = new WriteableBitmap(colorFrameDescription.Width,
                                    colorFrameDescription.Height, 96.0, 96.0,
                                    PixelFormats.Bgr32, null);
```

```
        //Asignar la imagen como fuente para ser mostrada en la ventana
        this.Image.Source = this.imagen;
        /* ---------------------------------------------------------- */
        // Enlistar la rutina que se llama cuando cambia la disponibilidad del Kinect
        this.miKinect.IsAvailableChanged += this.Sensor_IsAvailableChanged;
        // Iniciar el Kinect
        this.miKinect.Open();
    }
```

El método **Kinect_FrameReady** tiene dos segmentos, uno para cada tipo de dato. Lo primero que hace es adquirir los datos del frame y luego se obtiene su contenido indicando como referencia el tipo de frame que se quiere utilizar (**ColorFrameReference** para cámara y **BodyFrameReference** para Body).

Una vez realizado lo anterior se valida que los frames no estén vacíos y se copian los datos a los respectivos arreglos.

```
    private void Kinect_FrameReady(object sender, MultiSourceFrameArrivedEventArgs e) {
        // Adquirir el Frame
        var reference = e.FrameReference.AcquireFrame();
        // Adquirir el ColorFrame
        using (var colorFrame = reference.ColorFrameReference.AcquireFrame()) {
            // Verificar contenido de datos
            if (colorFrame != null) {
                if (colorFrame.RawColorImageFormat == ColorImageFormat.Bgra) {
                    colorFrame.CopyRawFrameDataToArray(cantidadPixeles);
```

```
            }
            else {
                colorFrame.CopyConvertedFrameDataToArray(cantidadP
                ixeles,
                                                ColorImageFormat.Bgra);
            }
                // Llamar método que manipula los datos
                this.usarCamara();
        }
    }
    // Adquirir el BodyFrame
    using (var mibodyFrame =
reference.BodyFrameReference.AcquireFrame()) {
        // Verificar contenido de datos
        if (mibodyFrame != null) {
            // Crear arreglo de Bodies y copiar datos recibidos en el mismo
            this.bodies = new Body[mibodyFrame.BodyCount];
            mibodyFrame.GetAndRefreshBodyData(this.bodies);
            foreach (Body body in bodies) {
                if (body != null) {
                    // Llamar método que manipula los datos
                    this.usarBody(body);
                }
            }
        }
    }
```

```
}
```

El ultimo método es **Window_Closing** en el cual se liberan recursos utilizados.

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
  /* ------------------- Configuración del Kinect ------------------- */
  if (this.multiReader != null)
  {
   // BodyFrameReader is IDisposable
   this.multiReader.Dispose();
   this.multiReader = null;
  }
  /* ------------------------------------------------------------- */
  if (this.miKinect != null)
  {
   this.miKinect.Close();
   this.miKinect = null;
  }
}
```

Para finalizar, se deben agregar al proyecto tres métodos: **usarCamara**, **usarBody** y **tomar_Foto**. Los métodos **usarCamara** y **tomar_Foto** son los mismos que en ejemplos anteriores, el único que se modifica es el método **usarBody**, en el cual se valida que se ha detectado una persona mediante la propiedad **TrackingId** y con la variable **bStopFoto** se limita a tomar solo una foto.

```
private void usarBody(Body miBody) {
  //Tomar foto solamente cuando no se haya tomado
  if (!bStopFoto && (miBody.TrackingId != 0))
```

```
        tomar_Foto();
}
private void usarCamara() {
   // Escribir los datos en el Bitmap
   this.imagen.WritePixels(new Int32Rect(0, 0, this.imagen.PixelWidth,
                  this.imagen.PixelHeight), this.cantidadPixeles,
                  this.imagen.PixelWidth * sizeof(int), 0);
}
private void tomar_Foto() {
   if (this.imagen != null) {
      // Obtener fecha para construir el nombre del archivo que almacena
      la foto
      DateTime thisDay = DateTime.Now;
      string fecha = thisDay.ToString("dd-MM-yy",
      DateTimeFormatInfo.InvariantInfo);
      string time = thisDay.ToString("hh'-'mm'-'ss",
                              CultureInfo.CurrentUICulture.DateTimeF
                              ormat);
      string fileName = fecha + '_' + time + ".jpg";
      //Borrar el archivo de la foto si ya existe
      if (File.Exists(fileName)) {
            File.Delete(fileName);
      }
      try {
            // Guardar la foto
            using (FileStream foto = new FileStream(fileName,
```

```
                FileMode.CreateNew)) {
                    JpegBitmapEncoder jpgEncoder = new
                    JpegBitmapEncoder();
                    jpgEncoder.QualityLevel = 70;
                    jpgEncoder.Frames.Add(BitmapFrame.Create(this.imagen
                    ));
                    jpgEncoder.Save(foto);
                    foto.Flush();
                    foto.Close();
                    foto.Dispose();
                    // Indicar que se ha tomado una foto
                    bStopFoto = true;
                }
            }
            catch (IOException) { }
        }
    }
```

Ejecución

Al ejecutar el proyecto la ventana lucirá de la siguiente forma, la foto solo se toma en el momento en que hay una persona frente al Kinect. La imagen se guarda en la carpeta del proyecto, en la subcarpeta **bin\debug**.

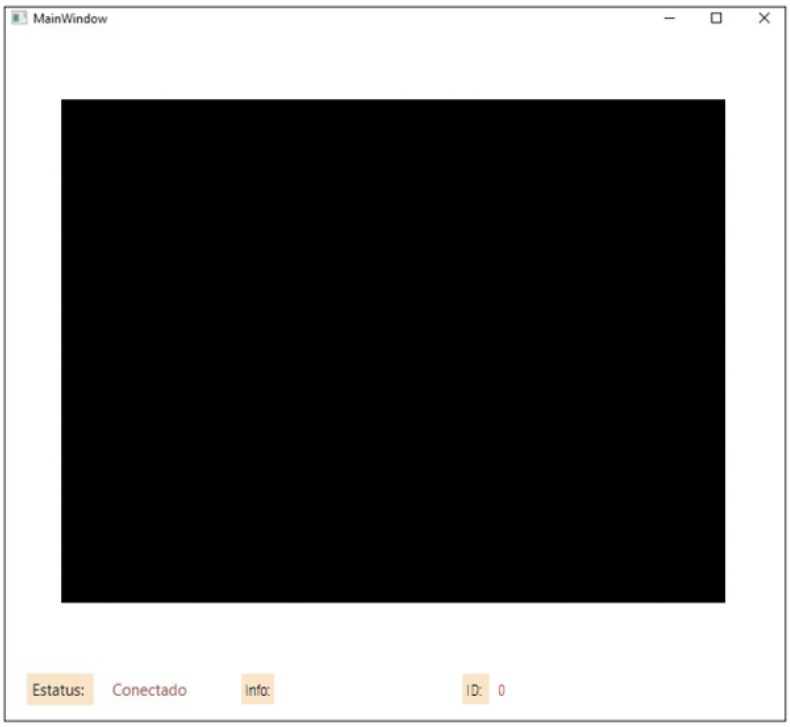


Figura 10.4

Capítulo 11. Extender el efecto del movimiento

Kinect nos permite un sinfín de aplicaciones pero se aprovecha más cuando se agregan otras plataformas como Arduino o Raspberry.

11.1 Arduino

Es un dispositivo electrónico con entradas y salidas tanto digitales como analógicas que permite realizar de una manera más fácil proyectos electrónicos con una gran variedad de compatibilidad en sensores y sistemas operativos.

11.2 Configuración del serial en el Arduino

La configuración para la comunicación entre el Arduino y Visual Studio por medio de serial requiere de distintos comandos, los cuales se presentan a continuación.

La variable **serialCharacter** servirá para almacenar los caracteres recibidos en la comunicación entre el Arduino y Visual Studio.

```
char serialCharacter;
```

Posteriormente, en la función **setup()** se debe inicializar el Serial para realizar la transferencia a 9,600 baudios y establecer el pin a 13 para salida (también se puede emplear para entrada).

```
void setup() {
  Serial.begin(9600);
  pinMode(13, OUTPUT);
}
```

Finalmente, dentro de la función **loop** se verifica si existe conexión serial disponible, de ser así, se leen y almacenan los datos recibidos en la variable **serialCharacter**.

```
void loop() {
  if(Serial.available()>0) {
      serialCharacter=Serial.read();
```

```
        //Escribir aquí las instrucciones que corresponden
        //al carácter recibido en la lectura
    }
}
```

Cabe destacar que así como el Arduino recibe información también puede enviar.

11.3 Configuración de serial en C#

Código en XAML

El programa en XAML incluye un **Canvas** con un botón que al ser presionado obtendrá todos los nombres de los puertos disponibles. También incluye un combo box en el cual se desplegarán los nombres de los puertos que se puede emplear para que el usuario seleccione el indicado.

```
<Grid Background="White">
    <Button x:Name="puertos" Content="Conexión"
  HorizontalAlignment="Left"
        Margin="104,131,0,0" VerticalAlignment="Top" Width="112"
        Height="36" Click="puertos_Click"/>
    <ComboBox x:Name="comboBox" HorizontalAlignment="Left"
        Margin="261,139,0,0" VerticalAlignment="Top" Width="120"
        SelectionChanged="comboBox_SelectionChanged"/>
</Grid>
```

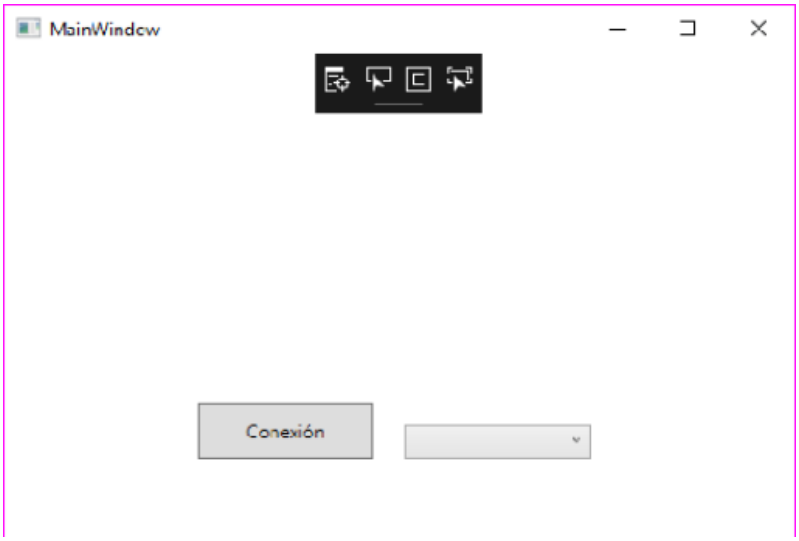


Figura 11.1

Código C#

Para la configuración del serial en Visual Studio C# se debe incluir la biblioteca que permite usar la comunicación de este tipo.

```
using System.IO.Ports;
```

En la parte superior del programa se declara la variable serial con la cual se especifica el puerto donde se conecta el Arduino por USB y la velocidad.

```
SerialPort serial = new SerialPort("COM4", 9600);
```

El método **puertos_Click** se ejecuta cuando se presiona el botón. Este método llena el combo box con los nombres de todos los puertos seriales disponibles para que el usuario pueda seleccionar uno de ellos.

```
private void puertos_Click(object sender, RoutedEventArgs e) {
   foreach (string s in SerialPort.GetPortNames()) {
      comboBox.Items.Add(s);
   }
}
```

El método **comboBox_SelectionChanged** se ejecuta automáticamente cada vez que el usuario abre el combo box y selecciona uno de los puertos COM. El método intenta realizar la conexión con el puerto seleccionado; si lo logra, despliega en la consola el mensaje “CONEXIÓN ESTABLECIDA” de lo contrario, se mostrará la leyenda “SIN

CONEXIÓN" (estos mensajes de consola solo son para verificar que todo esté funcionando de la manera correcta).

```
private void comboBox_SelectionChanged(object sender, SelectionChangedEventArgs e) {
    string portName = comboBox.Text;
    try {
     //Abrir el serial
     serial.Open();
     //Mensaje de consola
     Console.WriteLine("CONEXION ESTABLECIDA");
    }
    catch {
     //Mensaje de consola
     Console.WriteLine("SIN CONEXION");
    }
}
```

Una vez establecida la conexión con el puerto se pueden enviar caracteres al Arduino como se muestra en el siguiente ejemplo:

```
serial.Write("a");
```

11.4 Ejemplo: prender y apagar un LED

La siguiente aplicación usa comunicación **Serial** para establecer comunicación entre Visual Studio C# y Arduino con el objetivo de prender o apagar un foco LED según lo indique el usuario.

Requerimientos

- Proyecto **WPF**.
- Biblioteca: System.IO.Ports.

- Arduino Uno.
- Cable USB.

Código en XAML

El programa en XAML contiene un **Canvas** que incluye el botón buttonON que al ser presionado encenderá el LED y el botón buttonOFF que apaga el LED. Además da la definición de un botón (puertos) y un combo box (comboBox) que podrán ser empleados por el usuario para obtener el nombre de todos los puertos disponibles y seleccionar el puerto deseado para establecer la comunicación serial.

```
<Grid Background="White">
  <Button x:Name="buttonON" Content="LED ON" FontSize="25" HorizontalAlignment="Left"
        Margin="45,62,0,0" VerticalAlignment="Top" Width="184" Height="120"
        Click="buttonON_Click" Background="#FF19DB2B"/>
  <Button x:Name="buttonOFF" Content="LED OFF" FontSize="25" HorizontalAlignment="Left"
        Margin="268,62,0,0" VerticalAlignment="Top" Width="169" Height="120"
        Click="buttonOFF_Click" Background="#FFE72D13"/>
  <Button x:Name="puertos" Content="Conexión" HorizontalAlignment="Left"
        Margin="125,225,0,0" VerticalAlignment="Top" Width="112" Height="36"
        Click="puertos_Click"/>
  <ComboBox x:Name="comboBox" HorizontalAlignment="Left" Margin="258,239,0,0"
      VerticalAlignment="Top" Width="120" SelectionChanged="comboBox_SelectionChanged"/>
</Grid>
```

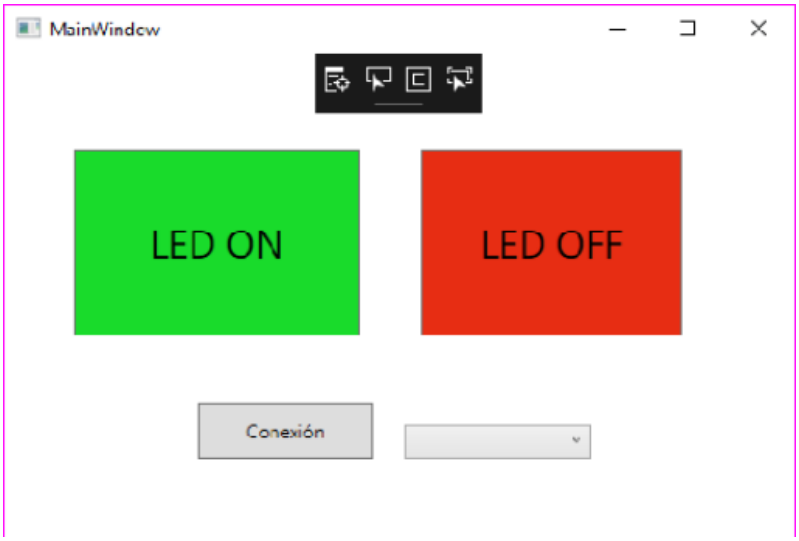


Figura 11.2

Código en C#

En la parte superior del programa se debe incluir la biblioteca que permite la comunicación serial.

```
using System.IO.Ports;
```

En esta parte también se declara la variable serial con la cual se especifica el puerto donde se conecta el Arduino por USB y la velocidad.

```
SerialPort serial = new SerialPort("COM4", 9600);
```

El método **buttonON_Click**, el cual es ejecutado cuando se presiona el botón **LED ON**, envía al Arduino el carácter "a" para indicarle que debe prender el **LED**.

```
private void buttonON_Click(object sender, RoutedEventArgs e) {
  serial.Write("a")
}
```

De la misma forma, el método **buttonOFF_Click**, que es ejecutado cuando se presiona el botón **LED OFF**, envía el carácter "b" al Arduino para indicar que debe apagar el LED.

```
private void buttonOFF_Click(object sender, RoutedEventArgs e) {
  serial.Write("b");
}
```

Código en C++ (Arduino)

El código inicia con la declaración de la variable **serialCharacter** que servirá para almacenar los caracteres recibidos en la comunicación entre el Arduino y Visual Studio.

```
char serialCharacter;
```

Posteriormente, en la función **setup** se debe inicializar el **Serial** para realizar la transferencia a 9,600 baudios y establecer el pin a 13 para salida (también se puede emplear para entrada).

```
void setup() {
  Serial.begin(9600);
  pinMode(13, OUTPUT);
}
```

Finalmente, dentro de la función **loop**, después de asegurar que existe una conexión con el Arduino, se prende o se apaga el foco LED (empleando la función **digitalWrite**) según sea el carácter recibido.

```
void loop() {
  if(Serial.available()>0) {
   serialCharacter=Serial.read();
   switch(serialCharacter)
   {
     case 'a':
          digitalWrite(13, HIGH);
          break;
     case 'b':
          digitalWrite(13,LOW);
          break;
   }
  }
```

}

Anexos

Plantillas

Para cada proyecto ejecutado durante el libro, se requiere una plantilla base, la cual ayudara al lector a realizar los distintos proyectos de manera más sencilla.

12.1 Plantillas Kinect V1

A continuación, se analizará y verificará la manera en que se estructura la plantilla para detectar el cuerpo en el Kinect versión 1. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés *Windows Presentation Foundation*) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - *Graphical User Interfaces*) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo este proceso.

Una vez realizado esto, el siguiente paso es la programación y estructuración.

Código XAML

Los siguientes *labels* serán usados para el monitoreo del dispositivo Kinect, si se encuentra conectado o desconectado.

```
<Window x:Class="PlantillaKV1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:PlantillaKV1"
        mc:Ignorable="d"
```

```
        Title="MainWindow"  Height="350"  Width="525"
        Closing="Window_Closing">
    <Grid>
        <Label x:Name="L1" Content="Estatus: " Background="Bisque"
        FontSize="15" HorizontalAlignment="Left" Margin="17,276,0,0"
        VerticalAlignment="Top"
        RenderTransformOrigin="0.484,0.315"/>
        <Label  x:Name="LEstatus"  Content="Desconectado"
        FontSize="15"  Foreground="#FFDA2828"
        HorizontalAlignment="Left"  Margin="94,276,0,0"
        VerticalAlignment="Top"/>
    </Grid>
</Window>
```

Código C#

Hay que agregar las siguientes bibliotecas que ayudarán a la configuración del Kinect.

```
using Microsoft.Kinect;
using System.IO;
```

Se declarará el *KinectSensor* para empezar con la ejecución del Kinect.

El método *MainWindow* se usará para utilizar los datos proporcionados por el Kinect y configuración del Kinect.

```
public MainWindow()
    {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
    }
```

Este método llamado *Kinect_Config* realizará las configuraciones necesarias en el Kinect. Asimismo, para la inicialización de envío de datos y se agregan las referencias que se desean poner: color, *depth*, entre otras.

```
private void Kinect_Config()
    {
        // Buscamos el Kinect conectado con la propiedad KinectSensors, al descubrir el primero con el estado Connected
        // se asigna a la variable miKinect que lo representará (KinectSensor miKinect)
        miKinect = KinectSensor.KinectSensors.FirstOrDefault(s => s.Status == KinectStatus.Connected);
        if (this.miKinect != null && !this.miKinect.IsRunning)
        {
            /* ------------------- Configuración del Kinect ------------------- */
            /* ---------------------------------------------------------------- */
            // Enlistamos el método que se llama cada vez que hay un cambio en el estado del Kinect
            KinectSensor.KinectSensors.StatusChanged += Kinect_StatusChanged;
            // Iniciar el Kinect
            try
            {
            this.miKinect.Start();
            }
            catch (IOException)
            this.miKinect = null;
```

```
            }
            LEstatus.Content = "Conectado";
        }
        else
        {
            // Enlistamos el método que se llama cada vez que hay un cambio en el estado del Kinect
            KinectSensor.KinectSensors.StatusChanged += Kinect_StatusChanged;
        }
    }
```

Ahora se declarará el método que adquiere los datos enviados al Kinect, su contenido varía según la tecnología que se esté usando (Cámara, Skeletontracking, DephtSensor, etc).

```
private void Kinect_FrameReady(object sender, EventArgs e)
        {
        }
```

El método *Kinect_StatusChanged* que se configura a partir del estado del Kinect (conectado, desconectado, etc).

```
private void Kinect_StatusChanged(object sender, StatusChangedEventArgs e)
    {
        switch (e.Status)
        {
```

```
            case KinectStatus.Connected:
                if (this.miKinect == null)
                {
                this.miKinect = e.Sensor;
                }
                if (this.miKinect != null && !this.miKinect.IsRunning)
                {
                    /* ------------------- Configuración del Kinect ----------
                    ------ */
                    /*  ------------------------------------------------------
                    */
                    // Iniciar el Kinect
                    try
                    {
                    this.miKinect.Start();
                    }
                    catch (IOException)
                    {
                    this.miKinect = null;
                    }
                    LEstatus.Content = "Conectado";
            }
            break;
        case KinectStatus.Disconnected:
            if (this.miKinect == e.Sensor)
            {
```

```
                    /* ------------------- Configuración del Kinect ----------
                    ------ */
                    /*  ------------------------------------------------------
                    */
                    this.miKinect.Stop();
                    this.miKinect = null;
                    LEstatus.Content = "Desconectado";
                    }
                    break;
    }
      }
```

El método *Window_Closing* es el encargado de liberar los recursos del Kinect cuando termina la aplicación.

```
    private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            if (this.miKinect != null && this.miKinect.IsRunning)
            {
                /* ------------------- Configuración del Kinect ------------------- */
                /* ------------------------------------------------------------ */
                this.miKinect.Stop();
            }
                }
```

Referencia frames de color

Se declararán las siguientes variables:

private WriteableBitmap imagen; //Se utiliza para generar la imagen a partir del arreglo de bytes recibidos

private byte[] cantidadPixeles; //Arreglo para recibir los bytes que envía el Kinect

---

Si se quiere realizar la configuración de color hay que poner el siguiente código en el método Kinect_StatusChanged.

```
/* ------------------- Configuración del Kinect ------------------- */

//Habilitar ColorStream con una resolución de 640x480 a una razón de 30 frames/seg

this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);

//Enlistar la función que se llamará cada vez que el Kinect tenga listo un frame de datos

this.miKinect.ColorFrameReady += this.Kinect_FrameReady;

// Crear el arreglo que recibe los datos de los pixeles, FramePixelDataLength es el número de bytes en el frame

this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];

// Crear el WriteableBitmap que tendrá la imagen

this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth, this.miKinect.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Bgr32, null);

// Asignar la imagen como fuente para ser mostrada en la ventana

this.Image.Source = this.imagen;
```

```
/* ---------------------------------------------------------- */
```

Se pondrá el siguiente código en el método Kinect_FrameReady.

```
using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
    {
     if (colorFrame != null)
     {
          // Copiar los datos(referentes a los pixeles) del frame a un arreglo
          colorFrame.CopyPixelDataTo(this.cantidadPixeles);
          // Manipular los bytes en el arreglo
          usarCamara();
     }
               }
```

En el método Kinect_Config se colocará el siguiente código donde dice configuración del Kinect.

```
// Habilitar ColorStream con una resolución de 640x480 a una razón de 30 frames / seg
this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
          // Enlistar la función que se llamará cada vez que el Kinect tiene listo un frame de datos
          this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
          // Crear el arreglo que recibe los datos de los pixeles, FramePixelDataLength es el número de bytes en el frame
```

```
            this.cantidadPixeles = new
    byte[this.miKinect.ColorStream.FramePixelDataLength];
            // Crear el WriteableBitmap que tendrá la imagen
            this.imagen = new
    WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
    this.miKinect.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Bgr32,
null);
            // Asignar la imagen como fuente para ser mostrada en la ventana
                this.Image.Source = this.imagen;
```

Referencia frames de infrarrojo

Se declararán las siguientes variables:

```
    private WriteableBitmap imagen; //Se utiliza para generar la imagen a
partir del arreglo de bytes recibidos
            private byte[] cantidadPixeles; //Arreglo para recibir los bytes que
            envía el Kinect
                int iColorFrameBytesPerPixel = 0; // Guarda la propiedad
                BytesPerPixel del frame de datos
```

Para realizar la configuración de color hay que poner el siguiente código en el método Kinect_StatusChanged.

```
    //Habilitar ColorStream con una resolución de 640x480 a una razón de 30
frames/seg
            this.miKinect.ColorStream.Enable(ColorImageFormat.InfraredResol
            ution640x480Fps30);
            //Enlistar la función que se llamará cada vez que el Kinect tenga
```

```
        listo un frame de datos
        this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
        // Crear el arreglo que recibe los datos de los pixeles, FramePixelDataLength es el número de bytes en el frame
        this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];
        // Crear el WriteableBitmap que tendrá la imagen
        this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
        this.miKinect.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Gray16, null);
        // Asignar la imagen como fuente para ser mostrada en la ventana
        this.Image.Source = this.imagen;
```

Poner el siguiente código en el método Kinect_FrameReady.

```
using (ColorImageFrame colorFrame = e.OpenColorImageFrame())
{
  if (colorFrame != null)
  {
    // Copiar los datos(referentes a los pixeles) del frame a un arreglo
    colorFrame.CopyPixelDataTo(this.cantidadPixeles);
    // Obtener datos referentes a las dimensiones del frame
    iColorFrameBytesPerPixel = colorFrame.BytesPerPixel;
    // Manipular los bytes en el arreglo
    usarCamara();
  }
```

```
}
```

En el método Kinect_Config se coloca el siguiente código donde dice configuración del Kinect.

```
// Habilitar ColorStream con una resolución de 640x480 a una razón de 30 frames / seg
this.miKinect.ColorStream.Enable(ColorImageFormat.InfraredResolution640x480Fps30);
// Enlistar la función que se llamará cada vez que el Kinect tiene listo un frame de datos
this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
// Crear el arreglo que recibe los datos de los pixeles, FramePixelDataLength es el número de bytes en el frame
this.cantidadPixeles = new byte[this.miKinect.ColorStream.FramePixelDataLength];
// Crear el WriteableBitmap que tendrá la imagen
this.imagen = new WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
this.miKinect.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Gray16, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
```

Referencia frames de profundidad

Se declararán las siguientes variables:

```
private WriteableBitmap colorBitmap; //Se utiliza para generar la imagen
```

```
private short[] dataPixels; //Arreglo que guarda lo bytes recibidos del Kinect
```

Para realizar la configuración de color hay que poner el siguiente código en el método Kinect_StatusChanged.

```
this.miKinect.DepthStream.Enable();
            this.miKinect.DepthFrameReady += Kinect_FrameReady;
            this.colorBitmap = new
            WriteableBitmap(this.miKinect.DepthStream.FrameWidth,
            this.miKinect.DepthStream.FrameHeight,
                  96.0, 96.0, PixelFormats.Gray16, null);
            this.dataPixels = new short[this.miKinect.DepthStream.FramePixelDataLength];
            // Asignar la imagen como fuente para ser mostrada en la ventana
                  this.DepthImageElement.Source = this.colorBitmap;
```

Se pone el siguiente código en el método Kinect_FrameReady.

```
// Abrir el frame y adquirir los datos y ceamos la imagen
   using (DepthImageFrame frame = e.OpenDepthImageFrame())
   {
      if (frame != null)
      {
            //Copiar datos al arreglo
            frame.CopyPixelDataTo(dataPixels);
```

```
            usarCamara();
        }
                }
```

En el método *Kinect_Config* se coloca el siguiente código donde dice configuración del Kinect.

```
this.miKinect.DepthStream.Enable();
            this.miKinect.DepthFrameReady += Kinect_FrameReady;
            this.colorBitmap = new
WriteableBitmap(this.miKinect.DepthStream.FrameWidth,
this.miKinect.DepthStream.FrameHeight,
                    96.0, 96.0, PixelFormats.Gray16, null);
this.dataPixels = new
short[this.miKinect.DepthStream.FramePixelDataLength];
        // Asignar imagen como fuente para ser mostrada en la ventana
            this.DepthImageElement.Source = this.colorBitmap;
```

Referencia frames de skeleton

Se declaran las siguientes variables:

```
double dMano_X; //Representa la coordenada X de la mano derecha
  double dMano_Y; //Representa la coordenada Y de la mano derecha
    Point joint_Point = new Point(); //Permite obtener los datos del Joint
```

Para realizar la configuración de color hay que poner el siguiente código en el método *Kinect_StatusChanged*.

```
// Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
            this.miKinect.SkeletonStream.Enable();
            // Enlistar al evento que se ejecuta cada vez que el Kinect tiene
            datos listos
            this.miKinect.SkeletonFrameReady                                    +=
            this.Kinect_FrameReady;
        this.miKinect.SkeletonFrameReady -= this.Kinect_FrameReady;
```

Poner el siguiente código en el método *Kinect_FrameReady*.

```
// Arreglo que recibe los datos
        Skeleton[] skeletons = new Skeleton[0];
        Skeleton skeleton;
        // Abrir el frame recibido y copiarlo al arreglo skeletons
        using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
        {
            if (skeletonFrame != null)
            {
                skeletons                    =                    new
                Skeleton[skeletonFrame.SkeletonArrayLength];
                skeletonFrame.CopySkeletonDataTo(skeletons);
            }
        }
        // Seleccionar el primer Skeleton trazado
        skeleton = (from trackSkeleton in skeletons where
        trackSkeleton.TrackingState    ==    SkeletonTrackingState.Tracked
```

```
        select trackSkeleton).FirstOrDefault();
        if (skeleton == null)
        {
        LID.Content = "0";
        return;
        }
        LID.Content = skeleton.TrackingId;
        // Enviar el Skelton a usar
              this.usarSkeleton(skeleton);
```

En el método *Kinect_Config* poner el siguiente código donde dice configuración del Kinect.

```
// Habilitar el SkeletonStream para permitir el trazo de "Skeleton"
        this.miKinect.SkeletonStream.Enable();
        // Enlistar al evento que se ejecuta cada vez que el Kinect tiene datos
        listos
        this.miKinect.SkeletonFrameReady += this.Kinect_FrameReady;
```

Y agregar los siguientes métodos en la sección de métodos nuevos.

```
/* -- Área para el método que utiliza los datos proporcionados por Kinect
-- */
    /// <summary>
    /// Método que realiza las manipulaciones necesarias sobre el Skeleton
    trazado
    /// </summary>
```

```
private void usarSkeleton(Skeleton skeleton)
{
    Joint joint1 = skeleton.Joints[JointType.HandRight];
    // Si el Joint está listo obtener las coordenadas
    if (joint1.TrackingState == JointTrackingState.Tracked)
    {
        // Obtener coordenadas
        joint_Point = this.SkeletonPointToScreen(joint1.Position);
        dMano_X = joint_Point.X;
        dMano_Y = joint_Point.Y;
        // Modificar coordenadas del indicador que refleja el
        movimiento (Ellipse rojo)
        Puntero.SetValue(Canvas.TopProperty, dMano_Y - 12.5);
        Puntero.SetValue(Canvas.LeftProperty, dMano_X - 12.5);
        // Indicar Id de la persona que es trazada
        LID.Content = skeleton.TrackingId;
    }
}
/* ------------------------------------------------------------------ */
/* ------------------------- Métodos Nuevos ---------------------------- */
/// <summary>
/// Metodo que convierte un "SkeletonPoint" a "DepthSpace", esto nos
permite poder representar las coordenadas de los Joints
/// en nuestra ventana en las dimensiones deseadas.
/// </summary>
private Point SkeletonPointToScreen(SkeletonPoint skelpoint)
```

```
{
    // Convertir un punto a "Depth Space" en una resolución de 640x480
    DepthImagePoint depthPoint = this.miKinect.CoordinateMapper.MapSkeletonPointToDepthPoint(skelpoint, DepthImageFormat.Resolution640x480Fps30);
    return new Point(depthPoint.X, depthPoint.Y);
        }
```

12.2 Plantillas Kinect V2

A continuación, se analizará y verificará la manera en que se estructura la plantilla para detectar el cuerpo para el Kinect versión 2. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés *Windows Presentation Foundation*) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - *Graphical User Interfaces*) muy vistosas y de una manera fácil ya que proporciona herramientas para hacer más sencillo dicho proceso.

Una vez realizado esto, el siguiente paso es la programación y estructuración.

Código XAML

Los siguientes *labels* serán usados para el monitoreo del dispositivo Kinect: si se encuentra conectado o desconectado.

```
<Window x:Class="PlantillaKV2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        "
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:PlantillaKV2"
```

```
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525"
        Closing="Window_Closing">
        <Grid>
        <Label x:Name="L1" Content="Estatus: " Background="Bisque"
        FontSize="15" HorizontalAlignment="Left" Margin="17,276,0,0"
        VerticalAlignment="Top" RenderTransformOrigin="0.484,0.315"/>
        <Label x:Name="LEstatus" Content="Desconectado"
        FontSize="15" Foreground="#FFDA2828"
        HorizontalAlignment="Left" Margin="94,276,0,0"
        VerticalAlignment="Top"/>
        </Grid>
    </Window>
```

Código C#

Hay que agregar las siguientes bibliotecas que ayudarán a la configuración del Kinect.

```
using Microsoft.Kinect;
using System.IO;
```

Se declarará el *KinectSensor* para empezar con la ejecución del Kinect.

El método *MainWindow* se usará para utilizar los datos proporcionados por el Kinect y configuración del Kinect.

```
public MainWindow()
    {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
```

```
}
```

Este método llamado *Kinect_Config* realizará las configuraciones necesarias en el Kinect. Asimismo, para la inicialización de envío de datos y se agregan las referencias de datos que se desean poner: color, *depth*, entre otras.

```
private void Kinect_Config()
    {
        // Buscar el Kinect conectado, al descubrir el primero se asigna a la variable miKinect que lo representará
        miKinect = KinectSensor.GetDefault();
        /* ------------------- Configuración del Kinect ------------------- */
        /* ------------------------------------------------------------ */
        // Enlistar la rutina que se llama cuando cambia la disponibilidad del Kinect
        this.miKinect.IsAvailableChanged += this.Kinect_StatusChanged;
        // Iniciar el Kinect
        this.miKinect.Open();
        // Verifica que el Kinect se inicio correctamente
        if (this.miKinect != null && this.miKinect.IsOpen)
        {
        // Notificar que se detectó correctamente
        LEstatus.Content = "Conectado";
        }
        else
        {
        // Notificar desconexión o error al iniciar
```

```
LEstatus.Content = "Desonectado";
}
}
```

Ahora se declarará el método que adquiere los datos enviados al Kinect, su contenido varía según la tecnología que se esté usando (Cámara, Skeletontracking, DephtSensor, etc).

```
private void Kinect_FrameReady(object sender, EventArgs e)
    {
    }
```

El método *Kinect_StatusChanged* que se configura a partir del estado del Kinect (conectado, desconectado, etc).

```
private void Kinect_StatusChanged(object sender,
IsAvailableChangedEventArgs e)
    {
        // Verificar el cambio que se efectuo en la disponibilidad del Kinect
        y notificarlo
        if (this.miKinect.IsAvailable)
        {
        // Notificar correcta conexión
        LEstatus.Content = "Conectado";
        }
        else
        {
        // Notificar desconexión
```

```
            LEstatus.Content = "Desconectado";
            }
                }
```

El método *Window_Closing* es el encargado de liberar los recursos del Kinect cuando termina la aplicación.

```
    private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            /* ------------------- Configuración del Kinect ------------------- */
            /* ----------------------------------------------------------------- */
            if (this.miKinect != null)
            {
            this.miKinect.Close();
            this.miKinect = null;
            }
                }
```

Referencia frames de color

Se declarará las siguientes variables:

```
    private ColorFrameReader colorFrameReader = null;       // FrameReader
para recibir datos referentes a la cámara
            private WriteableBitmap imagen = null;      // Permite crear la
            imagen a visualizar
            int iWidth = 0;         // Ancho de la imagen
```

```
int iHeight = 0;        // Alto de la imagen
byte[] cantidadPixeles;        // Arreglo para recibir los datos de la cámara (pixeles)
```

Si se desea realizar la configuración hay que poner el siguiente código en el método *Kinect_StatusChanged*.

```
// Verificar el cambio que se efectuo en la disponibilidad del Kinect y notificarlo
        if (this.miKinect.IsAvailable)
        {
        // Notificar correcta conexión
        LEstatus.Content = "Conectado";
        }
        else
        {
            // Notificar desconexión
            LEstatus.Content = "Desconectado";
                }
```

Se pondrá el siguiente código en el método *Kinect_FrameReady*.

```
// Adquirir el ColorFrame
        using (ColorFrame colorFrame = e.FrameReference.AcquireFrame())
        {
            // Verificar contenido de datos
```

```
if (colorFrame != null)
{
    if (colorFrame.RawColorImageFormat == ColorImageFormat.Bgra)
    {
    colorFrame.CopyRawFrameDataToArray(cantidadPixeles);
    }
    else
    {
    colorFrame.CopyConvertedFrameDataToArray(cantidadPixeles, ColorImageFormat.Bgra);
    }
    // Llamar método que manipula los datos
    this.usarCamara();
}
    }
```

En el método *Kinect_Config* se coloca el siguiente código donde dice configuración del Kinect.

```
// Abrir el FrameReader para poder recibir los datos referentes a la cámara
this.colorFrameReader = this.miKinect.ColorFrameSource.OpenReader();
// Enlistar la rutina que se llama cuando hay datos disponibles
this.colorFrameReader.FrameArrived += this.Kinect_FrameReady;
// Crear FrameDescription desde ColorFrameSource usando un formato
```

```
Bgra
 FrameDescription colorFrameDescription = this.miKinect.ColorFrameSource.CreateFrameDescription(ColorImageFormat.Bgra);
 iWidth = colorFrameDescription.Width;
 iHeight = colorFrameDescription.Height;
 cantidadPixeles = new byte[iWidth * iHeight * ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
 // Crear el bitmap a mostrar (Imagen que se visualiza)
 this.imagen = new WriteableBitmap(colorFrameDescription.Width, colorFrameDescription.Height, 96.0, 96.0, PixelFormats.Bgr32, null);
 // Asignar la imagen como fuente para ser mostrada en la ventana
        this.Image.Source = this.imagen;
```

Referencia frames de infrarrojo

Se declararán las siguientes variables:

```
private InfraredFrameReader infraredFrameReader = null; // FrameReader para recibir datos referntes al emisor de infrarojos
    private WriteableBitmap imagen = null; // Permite crear la imagen a visualizar
    int iWidth = 0; // Ancho de la imagen
    int iHeight = 0; // Alto de la imagen
    private byte[] cantidadPixeles; // Arreglo para recibir los datos de la cámara (pixeles)
    ushort[] infraredData; // Arreglo para recibir los datos de la cámara (pixeles)
```

Para realizar la configuración hay que poner el siguiente código en el método *Kinect_StatusChanged*.

```
// Verificar el cambio que se efectuo en la disponibilidad del Kinect y notificarlo
        if (this.miKinect.IsAvailable)
        {
        // Notificar correcta conexión
        LEstatus.Content = "Conectado";
        }
        else
        {
        // Notificar desconexión
        LEstatus.Content = "Desconectado";
            }
```

Poner el siguiente código en el método *Kinect_FrameReady*.

```
// Adquirir el ColorFrame
   using (InfraredFrame infraredFrame = e.FrameReference.AcquireFrame())
    {
        // Verificar contenido de datos
        if (infraredFrame != null)
        {
            infraredFrame.CopyFrameDataToArray(infraredData);
            // Llamar método que manipula los datos
```

```
            usarInfraredData();
        }
                }
```

En el método *Kinect_Config* se coloca el siguiente código donde dice configuración del Kinect.

```
/* ------------------- Configuración del Kinect ------------------- */
// Abrir el FrameReader para poder recibir los datos referentes al emisor de infrarojos
this.infraredFrameReader = this.miKinect.InfraredFrameSource.OpenReader();
// Enlistar la rutina que se llama cuando hay datos disponibles
this.infraredFrameReader.FrameArrived += this.Kinect_FrameReady;
// Crear FrameDescription desde InfraredFrameSource
FrameDescription infraredFrameDescription = this.miKinect.InfraredFrameSource.FrameDescription;
iWidth = infraredFrameDescription.Width;
iHeight = infraredFrameDescription.Height;
cantidadPixeles = new byte[iWidth * iHeight * ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
infraredData = new ushort[iWidth * iHeight];
// Crear el bitmap a mostrar (Imagen que se visualiza)
this.imagen = new WriteableBitmap(iWidth, iHeight, 96.0, 96.0, PixelFormats.Bgr32, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
```

```
/* ---------------------------------------------------------- */
```

Agregar el siguiente método que dará formato a los pixeles de la referencia infrarroja.

```
private void usarInfraredData()
  {
     int colorIndex = 0;
     for (int infraredIndex = 0; infraredIndex < infraredData.Length;
     ++infraredIndex)
     {
          ushort ir = infraredData[infraredIndex];
          byte intensity = (byte)(ir >> 8);
          cantidadPixeles[colorIndex++] = intensity; // Blue
          cantidadPixeles[colorIndex++] = intensity; // Green
          cantidadPixeles[colorIndex++] = intensity; // Red
          ++colorIndex;
     }
     int stride = iWidth * PixelFormats.Bgr32.BitsPerPixel / 8;
     this.imagen.WritePixels(new Int32Rect(0, 0,
     this.imagen.PixelWidth, this.imagen.PixelHeight),
     this.cantidadPixeles,
     stride, 0);
               }
```

Referencia frames de profundidad

Se declararán las siguientes variables:

```
private DepthFrameReader depthFrameReader = null; //FrameReader para recibir datos
        private WriteableBitmap imagen; // Se utiliza para generar la imagen
        int iWidth = 0; // Ancho de la imagen
        int iHeight = 0; // Alto de la imagen
        private byte[] cantidadPixeles = null; // Arreglo que guarda lo bytes recibidos del Kinect
        ushort[] depthData; // Arreglo para recibir los datos de la cámara (pixeles)
        ushort minDepth = 0; // Valor minimo del frame para Depth
            ushort maxDepth = 0; // Valor maximo del frame para Depth
```

Para realizar la configuración hay que poner el siguiente código en el método *Kinect_StatusChanged*.

```
// Verificar el cambio que se efectuo en la disponibilidad del Kinect y notificarlo
            if (this.miKinect.IsAvailable)
            {
            // Notificar correcta conexión
            LEstatus.Content = "Conectado";
            }
            else
            {
                // Notificar desconexión
                LEstatus.Content = "Desconectado";
                }
```

Se pone el siguiente código en el método *Kinect_FrameReady*.

```
using (DepthFrame depthFrame = e.FrameReference.AcquireFrame())
    {
        if (depthFrame != null)
        {
            depthFrame.CopyFrameDataToArray(depthData);
            // Obtener valores mínimos y máximos de la distancia
            minDepth = depthFrame.DepthMinReliableDistance;
            maxDepth = depthFrame.DepthMaxReliableDistance;
            // Llamar rutina para usa los datos
            usarDepthData();
        }
                }
```

En el método *Kinect_Config* se coloca el siguiente código donde dice configuración del Kinect.

```
/* ------------------- Configuración del Kinect ------------------- */
// Abrir el FrameReader para poder recibir los datos referentes a la cámara
        this.depthFrameReader =
        this.miKinect.DepthFrameSource.OpenReader();
        // Enlistar la rutina que se llama cuando hay datos disponibles
        this.depthFrameReader.FrameArrived += this.Kinect_FrameReady;
        // Crear FrameDescription desde ColorFrameSource usando un
```

```
formato Bgra
FrameDescription depthFrameDescription = this.miKinect.DepthFrameSource.FrameDescription;
iWidth = depthFrameDescription.Width;
iHeight = depthFrameDescription.Height;
cantidadPixeles = new byte[iWidth * iHeight * ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
depthData = new ushort[iWidth * iHeight];
// Crear el bitmap a mostrar (Imagen que se visualiza)
this.imagen = new WriteableBitmap(depthFrameDescription.Width, depthFrameDescription.Height, 96.0, 96.0, PixelFormats.Bgr32, null);
// Asignar la imagen como fuente para ser mostrada en la ventana
this.Image.Source = this.imagen;
/* ---------------------------------------------------------- */
```

Agregar el siguiente método para la configuración del sensor de profundidad:

```
private void usarDepthData()
  {
    int colorIndex = 0;
    for (int depthIndex = 0; depthIndex < depthData.Length; ++depthIndex)
    {
        ushort depth = depthData[depthIndex];
        byte intensity = (byte)(depth >= minDepth && depth <= maxDepth ? depth : 0);
        cantidadPixeles[colorIndex++] = intensity; // Blue
```

```
            cantidadPixeles[colorIndex++] = intensity; // Green
            cantidadPixeles[colorIndex++] = intensity; // Red
            ++colorIndex;
        }
        int stride = iWidth * PixelFormats.Bgr32.BitsPerPixel / 8;
        this.imagen.WritePixels(new Int32Rect(0, 0,
        this.imagen.PixelWidth, this.imagen.PixelHeight),
        this.cantidadPixeles,
        stride, 0);
                }
```

Referencia frames de skeleton

Se declaran las siguientes variables:

```
    private BodyFrameReader miBodyFrameReader; // FrameReader para recibir datos referentes a Body
        private Body[] bodies = null; // Arreglo para recibir datos de Body
        Point joint_Point = new Point(); // Permite obtener los datos de un Joint (Coordenadas)
            int iBodyID = 0;
```

Para realizar la configuración hay que poner el siguiente código en el método *Kinect_StatusChanged*.

```
    // Verificar el cambio que se efectuo en la disponibilidad del Kinect y notificarlo
        if (this.miKinect.IsAvailable)
```

```
        {
            // Notificar correcta conexión
            LEstatus.Content = "Conectado";
        }
        else
        {
            // Notificar desconexión
            LEstatus.Content = "Desconectado";
                }
```

Poner el siguiente código en el método *Kinect_FrameReady*.

```
// Adquirir el BodyFrame
   using (BodyFrame mibodyFrame = e.FrameReference.AcquireFrame())
   {
      // Verificar contenido de datos
      if (mibodyFrame != null)
      {
          // Crear arreglo de Bodies y copiar datos recibidos en el mismo
          this.bodies = new Body[mibodyFrame.BodyCount];
          mibodyFrame.GetAndRefreshBodyData(this.bodies);
          foreach (Body body in bodies)
          {
              if (body != null)
              {
                  // Llamar método que manipula los datos
```

```
                    this.usarBody(body);
                }
            }
        }
                }
```

En el método *Kinect_Config* poner el siguiente código donde dice configuración del Kinect.

```
// Abrir el FrameReader para poder recibir los datos referentes a Body
        this.miBodyFrameReader =
        this.miKinect.BodyFrameSource.OpenReader();
        // Enlistar la rutina que se llama cuando hay datos disponibles
            this.miBodyFrameReader.FrameArrived +=
            this.Kinect_FrameReady;
```

Y agregar los siguientes métodos en la sección de métodos nuevos.

```
/* -- Área para el método que utiliza los datos proporcionados por Kinect
-- */
    /// <summary>
    /// Método que realiza las manipulaciones necesarias sobre el Body
   recibido
    /// para obtener las coordenadas del Joint deseado
    /// </summary>
    private void usarBody(Body miBody)
    {
        // Crear Joint para obtener las propiedades del Joint deseado
```

```
        (Coordenadas)
        Joint miJoint = miBody.Joints[JointType.HandRight];
        // Verificar estado del Joint
        if (miJoint.TrackingState == TrackingState.Tracked)
        {
            // Obtener coordenadas
            joint_Point = this.BodyPointToScreen(miJoint);
            // Modificar coordenadas del indicador que refleja el movimiento (Ellipse rojo)
            Puntero.SetValue(Canvas.LeftProperty, (double)joint_Point.X);
            Puntero.SetValue(Canvas.TopProperty, (double)joint_Point.Y);
            // Indicar Id de la persona que es trazada
            LID.Content = miBody.TrackingId;
        }
    }
    /* ------------------------------------------------------------------ */
    /* ------------------------- Métodos Nuevos ---------------------------- */
    /// <summary>
    /// Método que convierte un "Body Point" a "DepthSpace", esto permite representar las coordenadas de los Joints
    /// en la ventana en las dimensiones correctas
    /// </summary>
    private Point BodyPointToScreen(Joint miJoint)
    {
        // Convertir un punto a "Depth Space"
        DepthSpacePoint                depthSpacePoint                =
```

```
        miKinect.CoordinateMapper.MapCameraPointToDepthSpace(miJoint.Position);
        return new Point(depthSpacePoint.X, depthSpacePoint.Y);
    }
    /* ---------------------------------------------------------------- */
```

12.3 Plantilla cara

A continuación, se analizará y verificará la manera en que se estructura la plantilla para detectar la cara en el Kinect versión 2. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés *Windows Presentation Foundation*) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - *Graphical User Interfaces*) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo este proceso.

Una vez realizado esto, el siguiente paso es la programación y estructuración.

Código XAML

Elemento image en el cual se desplegará el video capturado por el Kinect.

```
        <Window x:Class="PlantillaCara.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        "
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="1080" Width="1920"
        Closing="Window_Closing">
    <Viewbox>
        <Grid Width="1920" Height="1080">
            <Image Name="Imagen" />
            <Canvas Name="canvas">
```

```
        </Canvas>
    </Grid>
  </Viewbox>
</Window>
```

Código C#

Bibliotecas usadas para la configuración de esta plantilla.

```
using Microsoft.Kinect;
using Microsoft.Kinect.Face;
```

Primero, se deben de declarar las siguientes variables que serán colocadas después de que se abra el paréntesis del *Public Partial Class MainWindow* : Window, de las cuales *KinectSensor* ayudará a nombrar el dispositivo Kinect detectado. *ColorFrameReader* ayudará a la obtención de los frames de video en referencia a color. *BodyFrameReader* será la encargada de inicializar la lectura de los cuerpos detectados. *IList<Body>* almacenará la enumeración de los cuerpos detectados por el Kinect. *FaceFrameSource* ayudará a seleccionar la fuente de donde se obtendrán los frames de la cara del cuerpo detectado. *FaceFrameReader* inicializará con la lectura de los frames de la cara detectada y guardará los datos obtenidos.

```
public partial class MainWindow : Window
  {
  KinectSensor MiKinect = null;
  ColorFrameReader ColorRead = null;
  BodyFrameReader BodyRead = null;
  IList<Body> Bodies = null;
  FaceFrameSource FaceSource = null;
  FaceFrameReader FaceReader = null;
```

Posteriormente, en el evento MainWindow se colocarán las siguientes instrucciones las

cuales permitirán la realización de la configuración del Kinect inicializando las variables que se declararon anteriormente.

Se comienza con la instrucción *MiKinect = Kinect.GetDefault* para indicar que el primer sensor Kinect identificado será el que se usará, posteriormente se indica que empiece la detección de los cuerpos, así como poner que se tomen los frames de color, los cuales apoyarán posteriormente a crear video. Asimismo, se declaran los diferentes gestos a detectar, facilitando su localización y por último se relaciona al evento, el cual contendrá la información detectada por el Kinect.

```
public MainWindow()
{
    InitializeComponent();
    MiKinect = KinectSensor.GetDefault();
    if (MiKinect != null)
    {
        //Instruccion la cual nos permite inicializar el Kinect
        MiKinect.Open();
        //Variable encargada para la deteccion de los cuerpos
        Bodies = new Body[MiKinect.BodyFrameSource.BodyCount];
        //Se sabre la lectura del video en formato color para su posterior visualizacion
        ColorRead = MiKinect.ColorFrameSource.OpenReader();
        ColorRead.FrameArrived += ColorReadFrameArrived;
        //Se abre la lectura de los cuerpos que el Kinect pueda estar detectando
        BodyRead = MiKinect.BodyFrameSource.OpenReader();
        BodyRead.FrameArrived += BodyReadFrameArrived;
        //Indicamos los estados de la cara que detectaremos
        FaceSource = new FaceFrameSource(MiKinect, 0,
```

```
                    FaceFrameFeatures.BoundingBoxInColorSpace|
                    FaceFrameFeatures.FaceEngagement |
                    FaceFrameFeatures.Glasses |
                    FaceFrameFeatures.Happy |
                    FaceFrameFeatures.LeftEyeClosed |
                    FaceFrameFeatures.MouthOpen |
                    FaceFrameFeatures.PointsInColorSpace |
                    FaceFrameFeatures.RightEyeClosed|
                    FaceFrameFeatures.MouthMoved|
                    FaceFrameFeatures.RotationOrientation);
        //Se abre la lectura de la cara por cada frame que llega
        FaceReader = FaceSource.OpenReader();
        FaceReader.FrameArrived += FaceReadFrameArrived;
    }
}
```

Ahora se agregará un método que se encargará de dar formato al video que se presentará en forma RGB, además de ajustar el tamaño necesario (pixeles) para el despliegue del mismo, en calidad Full HD.

```
private ImageSource ToBitmap(ColorFrame Frame)
        {
            int width = Frame.FrameDescription.Width;
            int height = Frame.FrameDescription.Height;
            PixelFormat FormatoDePixel = PixelFormats.Bgr32;
            byte[] Pixeles = new byte[width * height *
            ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
```

```
            if (Frame.RawColorImageFormat == ColorImageFormat.Bgra)
            {
            Frame.CopyRawFrameDataToArray(Pixeles);
            }
            else
            {
            Frame.CopyConvertedFrameDataToArray(Pixeles,
            ColorImageFormat.Bgra);
            }
            int stride = width * FormatoDePixel.BitsPerPixel / 8;
            return BitmapSource.Create(width, height, 96, 96,
            FormatoDePixel, null, Pixeles, stride);
        }
```

Después, se agregará el método *ColorReadFrameArrived* que se encarga de desplegar la imagen dentro del grid para poder visualizar la imagen al momento de la ejecución.

```
        void ColorReadFrameArrived(object sender,
        ColorFrameArrivedEventArgs e)
    {
      using (var frame = e.FrameReference.AcquireFrame())
      {
        if (frame != null)
        {
            Imagen.Source = ToBitmap(frame);
        }
      }
```

```
}
```

Posteriormente, se declarará el método *BodyReadFrameArrived* encargado de obtener los cuerpos, hacer un filtro de personas detectada y asignar un identificador a cada uno. Se toma para la selección al primer cuerpo detectado por el Kinect para obtener información de él.

```
void BodyReadFrameArrived(object sender, BodyFrameArrivedEventArgs e)
    {
        using (var frame = e.FrameReference.AcquireFrame())
        {
            if (frame != null)
            {
                frame.GetAndRefreshBodyData(Bodies);
                Body Cuerpo = Bodies.Where(b => b.IsTracked).FirstOrDefault();
                if (!FaceSource.IsTrackingIdValid)
                {
                    if (Cuerpo != null)
                    {
                        FaceSource.TrackingId = Cuerpo.TrackingId;
                    }
                }
            }
        }
    }
```

Una vez realizado lo anterior, se declara el método *FaceReadFrameArrived*, este adquiere los frames del cuerpo detectado de la referencia color, del cual se obtiene la cara. Se declaran las variables y los puntos faciales y algunas expresiones que ayudan a la recopilación de información más fácilmente.

```
void FaceReadFrameArrived(object sender, FaceFrameArrivedEventArgs
e)
    {
        using (var Frame = e.FrameReference.AcquireFrame())
        {
            if (Frame != null)
            {
                FaceFrameResult Resultado = Frame.FaceFrameResult;
                if (Resultado != null)
                {
                    var ojoIzq =
                    Resultado.FacePointsInColorSpace[FacePointType.Eye
                    Left];
                    var ojoDer =
                    Resultado.FacePointsInColorSpace[FacePointType.Eye
                    Right];
                    var Nariz =
                    Resultado.FacePointsInColorSpace[FacePointType.Nos
                    e];
                    var bocaIzq =
                    Resultado.FacePointsInColorSpace[FacePointType.Mou
                    thCornerLeft];
                    var bocaDer =
                    Resultado.FacePointsInColorSpace[FacePointType.Mou
                    thCornerRight];
```

```
                var ojoIzqCerrado =
                Resultado.FaceProperties[FaceProperty.LeftEyeClosed]
                ;
                var ojoDerCerrado =
                Resultado.FaceProperties[FaceProperty.RightEyeClose
                d];
                var bocaAbierta =
                Resultado.FaceProperties[FaceProperty.MouthOpen];
    ConfiguracionDelPrograma(ojoIzq,ojoDer,Nariz,bocaIzq,bocaDer,ojoIzq
Cerrado,ojoDerCerrado,bocaAbierta);
                    }
                }
            }
        }
```

A partir del método de arriba, se podrán manejar los datos agregando otro método dentro de la función *FaceReadFrameArrived*, el cual se llama *ConfiguracionDelPrograma*. En este se agregaron todos los puntos y expresiones que el Kinect puede detectar, manipulando la información para la realizar proyectos que reaccionen a gestos o movimiento de puntos en la cara.

```
    private void ConfiguracionDelPrograma(PointF ojoIzq, PointF ojoDer,
PointF nariz, PointF bocaIzq, PointF bocaDer, DetectionResult
ojoIzqCerrado, DetectionResult ojoDerCerrado, DetectionResult
bocaAbierta)
        {
        }
```

Por último, se agrega el método Windows_Closing , el cual se ejecutará al momento de detener al WPF Application, parando la lectura de los datos obtenidos por el Kinect y apagando el dispositivo, lo cual ayudará a tener una finalización del programa adecuado.

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
    {
        if (ColorRead != null)
        {
        ColorRead.Dispose(); ColorRead = null;
        }
        if (BodyRead != null)
        {
        BodyRead.Dispose(); BodyRead = null;
        }
        if (FaceReader != null)
        {
        FaceReader.Dispose(); FaceReader = null;
        }
        if (FaceSource != null)
        {
        FaceSource.Dispose(); FaceSource = null;
        }
        if (MiKinect != null)
        {
        MiKinect.Close();
        }
            }
```

12.4 Plantilla audio

A continuación, se analizará y verificará la manera en que se estructura la plantilla para detectar el audio en el Kinect versión 2. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés Windows *Presentation Foundation*) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - *Graphical User Interfaces*) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo este proceso.

Una vez realizado esto, el siguiente paso es la programación y estructuración.

Código XAML

En este caso no es necesario poner ningún elemento dentro del grid de trabajo, por lo cual la siguiente estructura aparece por *default* dentro del código XAML.

```
<Window x:Class="PlantillaVoz.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        "
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:PlantillaVoz"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
    </Grid>
</Window>
```

Código C#

Bibliotecas usadas para la configuración de esta plantilla.

```
using Microsoft.Kinect;
using System.IO;
using Microsoft.Speech.AudioFormat;
using Microsoft.Speech.Recognition;
```

Primero se deben de declarar las siguientes variables que serán colocadas después de abrir el paréntesis del *Public Partial Class MainWindow* : *Window*, de las cuales *KinectSensor* ayudará a nombrar el dispositivo Kinect detectado, *KinectAudioStream* se encargará del flujo de datos de audios recopilados por el dispositivo Kinect y *SpeechRecognitionEngine* manejará el motor de reconocimiento de vos, tal como su idioma y demás.

```
namespace PlantillaVoz
{
  public partial class MainWindow : Window
  {
    KinectSensor miKinect = null;
    KinectAudioStream audioStr = null
    SpeechRecognitionEngine motorVoz = null
```

Dentro del método public MainWondow se asignan las siguientes instrucciones que se ejecutarán al momento de inicializar el proyecto, dentro de estas va la ejecución de un método llamado Kinect_Config .

```
public MainWindow()
    {
        InitializeComponent();
        // Realizar configuraciones e iniciar el Kinect
        Kinect_Config();
```

```
        }
```

El metodo Kinect_Config se encarga de la configuración de todo el flujo de datos de audio del Kinect V2 y filtraje de la información de este para obtener mejor calidad de audio. En esta parte del código se indican los comandos de voz que se desea sean detectados.

```
private void Kinect_Config()
{
  // Buscamos el Kinect conectado con la propiedad KinectSensors, al descubrir el primero con el estado Connected
  // se asigna a la variable miKinect que lo representará (KinectSensor miKinect)
  miKinect = KinectSensor.GetDefault();
  if (miKinect != null)
  {
     // Obtenemos el flujo de audio a partir del sensor Kinect
     IReadOnlyList<AudioBeam> audioBeamList = miKinect.AudioSource.AudioBeams;
     System.IO.Stream audioStream = audioBeamList[0].OpenInputStream();
     // Crea una nueva peticion para el flujo de datos
     audioStr = new KinectAudioStream(audioStream);
  }
  else
  {
  }
  RecognizerInfo ri = KinectRecognizer();
  if (null != ri)
```

```
{
    this.motorVoz = new SpeechRecognitionEngine(ri.Id);
    ////////////////////////////////////////////////////////
    //Agregamos comandos para que sean reconocidos mas adelante. //
    ////////////////////////////////////////////////////////
    /*----------------COMANDOS----------------*/
    var Instrucciones = new Choices();
    Instrucciones.Add(new SemanticResultValue("play", "PLAY"));
    Instrucciones.Add(new SemanticResultValue("stop", "STOP"));
    /*----------------------------------------*/
    var gb = new GrammarBuilder { Culture = ri.Culture };
    gb.Append(Instrucciones);
    var g = new Grammar(gb);
    this.motorVoz.LoadGrammar(g);
    this.motorVoz.SpeechRecognized += this.SpeechRecognized;
    // Indica a audioStr que el Speech esta activo
    this.audioStr.SpeechActive = true;
    //Seleccionamos el formato
    this.motorVoz.SetInputToAudioStream(
        this.audioStr,                                    new
        SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1,
        32000, 2, null));
    this.motorVoz.RecognizeAsync(RecognizeMode.Multiple);
}
else
{
```

```
    }
    // open the sensor
    this.miKinect.Open();
  }
```

La clase *KinectAudioStream* será la estructura encargada de la obtención de los datos, con los cuales, se permitirá construir el audio a partir de la obtención de los datos de onda que genera el sonido, tales como la longitud, amplitud, frecuencia, entre otros parámetros, convirtiendo estas señales en información digital que se puede manipular.

```
  internal class KinectAudioStream : Stream
  {
    private Stream kinect32BitStream;
    public KinectAudioStream(Stream input)
    {
    this.kinect32BitStream = input;
    }
    public bool SpeechActive { get; set; }
    public override bool CanRead
    {
    get { return true; }
    }
    public override bool CanWrite
    {
    get { return false; }
    }
    public override bool CanSeek
```

```
{
get { return false; }
}
public override long Position
{
get { return 0; }
set { throw new NotImplementedException(); }
}
public override long Length
{
get { throw new NotImplementedException(); }
}
public override void Flush()
{
throw new NotImplementedException();
}
public override long Seek(long offset, SeekOrigin origin)
{
return 0;
}
public override void SetLength(long value)
{
throw new NotImplementedException();
}
public override void Write(byte[] buffer, int offset, int count)
```

```
{
throw new NotImplementedException();
}
public override int Read(byte[] buffer, int offset, int count)
{
    // Kinect gives 32-bit float samples. Speech asks for 16-bit integer samples.
    const int SampleSizeRatio = sizeof(float) / sizeof(short); // = 2.
    // Speech reads at high frequency - allow some wait period between reads (in msec)
    const int SleepDuration = 50;
    // Allocate buffer for receiving 32-bit float from Kinect
    int readcount = count * SampleSizeRatio;
    byte[] kinectBuffer = new byte[readcount];
    int bytesremaining = readcount;
    // Speech expects all requested bytes to be returned
    while (bytesremaining > 0)
    {
        // If we are no longer processing speech commands, exit
        if (!this.SpeechActive)
        {
        return 0;
        }
        int result = this.kinect32BitStream.Read(kinectBuffer, readcount - bytesremaining, bytesremaining);
        bytesremaining -= result;
```

```
        // Speech will read faster than realtime - wait for more data to
        arrive
        if (bytesremaining > 0)
        {
        System.Threading.Thread.Sleep(SleepDuration);
        }
}
// Convert each float audio sample to short
for (int i = 0; i < count / sizeof(short); i++)
{
        // Extract a single 32-bit IEEE value from the byte array
        float sample = BitConverter.ToSingle(kinectBuffer, i *
        sizeof(float));
        // Make sure it is in the range [-1, +1]
        if (sample > 1.0f)
        {
        sample = 1.0f;
        }
        else if (sample < -1.0f)
        {
        sample = -1.0f;
        }
        // Scale float to the range (short.MinValue, short.MaxValue] and
        then
        // convert to 16-bit signed with proper rounding
        short convertedSample = Convert.ToInt16(sample *
        short.MaxValue);
```

```
            // Place the resulting 16-bit sample in the output byte array
            byte[] local = BitConverter.GetBytes(convertedSample);
            System.Buffer.BlockCopy(local, 0, buffer, offset + (i * sizeof(short)), sizeof(short));
        }
        return count;
    }
}
```

El método *KinectRecognizer* permitirá indicar el idioma con el cual se realiza la detección, es muy importante ya que el Kinect se apoyará en una base de datos predeterminados para poder compararlo y empezar a obtener el texto dictado al dispositivo.

```
private static RecognizerInfo KinectRecognizer()
    {
        IEnumerable<RecognizerInfo> recognizers;
        try
        {
            recognizers = SpeechRecognitionEngine.InstalledRecognizers();
        }
        catch (COMException)
        {
            return null;
            }
            foreach (RecognizerInfo recognizer in recognizers)
            {
```

```
            string value;
            recognizer.AdditionalInfo.TryGetValue("Kinect", out
            value);
            if ("True".Equals(value,
            StringComparison.OrdinalIgnoreCase) && "en-
            US".Equals(recognizer.Culture.Name,
            StringComparison.OrdinalIgnoreCase))
            {
                return recognizer;
            }
        }
            return null;
                }
```

Después se declarará el método *SpeechRecognized* el cual identificará los comandos recibidos y los mismos comandos de voz, aquí es donde se inicia con la programación y se evaluan los resultados de los comandos.

```
private void SpeechRecognized(object sender,
SpeechRecognizedEventArgs e)
    {
        // Se especifica un nivel de confidencia
        const double ConfidenceThreshold = 0.3;
        /// Aquí se analiza la señal recibida y se checa que sea
        fidedigno.
        if (e.Result.Confidence >= ConfidenceThreshold)
        {
            switch (e.Result.Semantics.Value.ToString())
```

```
                {
                    case "PLAY":
                        Console.WriteLine("PLAY");
                        break;
                    case "STOP":
                        Console.WriteLine("STOP");
                        break;
                }
            }
                }
```

Por último, se agrega el método Windows_Closing, el cual se ejecutará al momento de detener al WPF Application, parando la lectura de los datos obtenidos por el Kinect y apagando al dispositivo, lo cual ayudará a tener una finalización del programa adecuado.

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
    {
        if (this.miKinect != null)
        {
            this.miKinect.Close();
            this.miKinect = null;
        }
    }
```

12.5 Plantilla cámara

12.5.1 Kinect V1

A continuación, se analizará y verificará la manera en que se estructura la plantilla para usar de distintas maneras la cámara en el Kinect versión 1. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés *Windows Presentation Foundation*) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - *Graphical User Interfaces*) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo dicho proceso.

Una vez realizado esto, el siguiente paso es la programación y estructuración de esta.

Código XAML

En este caso se colocará un elemento *Image*, el cual ayudará al despliegue del video.

```
<Window x:Class="PlantillaCamaraKV1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace: PlantillaCamaraKV1"
        mc:Ignorable="d"
        Title="MainWindow" Height="770" Width="770" Closing="Window_Closing">
    <Grid>
        <Image Name="Image" Width="640" Height="480" Stretch="Fill"/>
    </Grid>
</Window>
```

Código C#

Bibliotecas usadas para la configuración de esta plantilla:

- using Microsoft.Kinect
- using System.IO
- using System.Globalization

Primero que nada se deben declarar las siguientes variables que serán colocadas después de que se abra el paréntesis del *Public Partial Class MainWindow* : *Window*, de las cuales *KinectSensor* ayudará a nombrar el dispositivo Kinect detectado, **WriteableBitmap** será la encargada de generar la imagen a partir de un arreglo de *bytes* recibidos y *byte*[] será encargada de almacenarlos en un arreglo.

```
namespace PlantillaCamaraKV1
    {
      public partial class MainWindow : Window
      {
          private KinectSensor miKinect;
          private WriteableBitmap imagen;
          private byte[] cantidadPixeles;
```

Dentro del método *public MainWindow* asignaremos las siguientes instrucciones que se ejecutarán al momento de inicializar el proyecto, dentro del cual va la ejecución de un método llamado *Kinect_Config*.

```
public MainWindow()
      {
          InitializeComponent();
          Kinect_Config();
      }
```

El método *Kinect_Config* se encarga de la configuración de todo el flujo de datos de video del Kinect V1 y filtro de la información de este para obtener mejor calidad. Asimismo, se habilita la referencia en la cual se desean captar los datos, la cual es *ColorStream* y la inicialización de la lectura de Kinect.

```
private void Kinect_Config()
```

```
        {
            miKinect = KinectSensor.KinectSensors.FirstOrDefault(s =>
            s.Status == KinectStatus.Connected);
            if (this.miKinect != null && !this.miKinect.IsRunning)
            {
  this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640
x480Fps30);
                this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
                this.cantidadPixeles = new
                byte[this.miKinect.ColorStream.FramePixelDataLength];
  this.imagen = new
WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
this.miKinect.ColorStream.FrameHeight, 96.0, 96.0, PixelFormats.Bgr32,
null);
                this.Image.Source = this.imagen;
                KinectSensor.KinectSensors.StatusChanged +=
                Kinect_StatusChanged;
                 try
                 {
                  this.miKinect.Start();
                 }
                  catch (IOException)
                 {
                  this.miKinect = null;
                 }
            }
            else
            {
```

```
            KinectSensor.KinectSensors.StatusChanged +=
            Kinect_StatusChanged;
        }
    }
```

El método *Kinect_StatusChanged* será encargado de verificar el estado del Kinect, si este se encuentra conectado o no, así como su funcionamiento. Además, dentro de este método se da formato al arreglo de *bytes* el cual será encargado de almacenarlos para su debida manipulación, dándole formato a los mismos.

```
private void Kinect_StatusChanged(object sender,
StatusChangedEventArgs e)
    {
        switch (e.Status)
        {
         case KinectStatus.Connected:
           if (this.miKinect == null)
           {
            this.miKinect = e.Sensor;
           }
           if (this.miKinect != null && !this.miKinect.IsRunning)
           {
this.miKinect.ColorStream.Enable(ColorImageFormat.RgbResolution640
x480Fps30);
this.miKinect.ColorFrameReady += this.Kinect_FrameReady;
this.cantidadPixeles = new
byte[this.miKinect.ColorStream.FramePixelDataLength];
            this.imagen = new
            WriteableBitmap(this.miKinect.ColorStream.FrameWidth,
            this.miKinect.ColorStream.FrameHeight, 96.0, 96.0,
            PixelFormats.Bgr32, null);
```

```
            this.Image.Source = this.imagen;
            try
            {
            this.miKinect.Start();
            }
             catch (IOException)
            {
             this.miKinect = null;
            }
          }
          break;
        case KinectStatus.Disconnected:
          if (this.miKinect == e.Sensor)
          {
            this.miKinect.ColorFrameReady -= this.Kinect_FrameReady;
            this.miKinect.Stop();
            this.miKinect = null;
          }
          break;
      }
  }
```

La clase Kinect_FrameReady será el método encargado de direccionar los frames procesados por el Kinect para posteriormente desplegarlos en el elemento Image, por el cual tiene que pasar un proceso de verificación de que estos datos hayan llegado de manera correcta.

```
private void Kinect_FrameReady(object sender, ColorImageFrameReadyEventArgs e)
```

```
        {
            using (ColorImageFrame colorFrame =
            e.OpenColorImageFrame())
            {
                if (colorFrame != null)
                {
                    colorFrame.CopyPixelDataTo(this.cantidadPixeles);
                    usarCamara();
                }
            }
        }
```

El método usarCamara será el método encargado de implementar los frames obtenidos por el Kinect, dándole formato de tamaños para que esta se despliegue de manera óptima dentro de la ventana MainWindow.

```
    private void usarCamara()
        {
            this.imagen.WritePixels(new Int32Rect(0, 0,
    this.imagen.PixelWidth, this.imagen.PixelHeight), this.cantidadPixeles,
    this.imagen.PixelWidth * sizeof(int), 0);
        }
```

Por último, el método Window_Closing realizará que el Kinect se finalice de manera correcta, reduciendo el uso de los recursos para de esta manera se apague el Kinect para que deje de leer y adquirir datos.

```
    private void Window_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
        {
            if (this.miKinect != null && this.miKinect.IsRunning)
```

```
{
        this.miKinect.ColorFrameReady                    -=
        this.Kinect_FrameReady;
        this.miKinect.Stop();
}
        }
```

12.5.2 Kinect V2

A continuación, se analizará y verificará la manera en que se estructura la plantilla para usar de distintas maneras la cámara en el Kinect versión 2. Para construir una aplicación en el lenguaje C# lo primero que se debe hacer es seleccionar el tipo de proyecto a emplear. **WPF** (del inglés Windows Presentation Foundation) permite desarrollar aplicaciones con interfaces de usuario gráficas (GUI - Graphical User Interfaces) muy vistosas, de una manera fácil ya que proporciona herramientas para hacer más sencillo dicho proceso.

Una vez realizado esto, nuestro siguiente paso es la programación y estructuración de esta.

Código XAML

En este caso se colocará un elemento Image, el cual ayudará al despliegue del video.

```
<Window x:Class="PlantillaCamaraKV2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
        "
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace: PlantillaCamaraKV2"
        mc:Ignorable="d"
        Title="MainWindow"          Height="770"          Width="770"
        Closing="Window_Closing">
```

```
    <Grid>
        <Image Name="Image" Width="640" Height="480"
        Stretch="Fill"/>
</Window>
```

Código C#

Bibliotecas usadas para la configuración de esta plantilla:

- using Microsoft.Kinect
- using System.IO
- using System.Globalization

Primero se deben declarar las siguientes variables que serán colocadas después de que se abra el paréntesis del Public Partial Class MainWindow : Window, de las cuales KinectSensor ayudará a nombrar el dispositivo Kinect detectado. WriteableBitmap será la encargada de generar la imagen a partir de un arreglo de bytes recibidos, ColorFrameReader es la variable encargada del procesamiento de los frames de Kinect recibidos pero de la referencia Color, iWidth y iHeight almacenan el ancho y alto de la imagen del Kinect; y por último, byte[] será encargada de almacenar los bytes dentro de un arreglo.

```
namespace PlantillaCamaraV2
{
  public partial class MainWindow : Window
  {
      private KinectSensor miKinect;
      private ColorFrameReader colorFrameReader = null;
      private WriteableBitmap imagen = null;
      int iWidth = 0;
      int iHeight = 0;
      byte[] cantidadPixeles;
```

Dentro del método public MainWindow asignaremos las siguientes instrucciones que se ejecutarán al momento de inicializar el proyecto, dentro del cual va la ejecución de un

método llamado Kinect_Config.

```
public MainWindow()
    {
        InitializeComponent();
        Kinect_Config();
    }
```

El método Kinect_Config se encarga de la configuración de todo el flujo de datos de video del Kinect V2 y filtro de la información de este para obtener mejor calidad. Asimismo, se habilita la referencia en la cual se quieren captar los datos, la cual es ColorFrame y la inicialización de la lectura de Kinect. También hay una ejecución de la configuración de la imagen que el Kinect capta, para que este mismo pueda ser desplegado a la ventana MainWindow.

```
        private void Kinect_Config()
        {
            miKinect = KinectSensor.GetDefault();
            this.colorFrameReader =
    this.miKinect.ColorFrameSource.OpenReader();
            this.colorFrameReader.FrameArrived                    +=
            this.Kinect_FrameReady;
            FrameDescription colorFrameDescription =
    this.miKinect.ColorFrameSource.CreateFrameDescription(ColorImageFo
rmat.Bgra);
            iWidth = colorFrameDescription.Width;
            iHeight = colorFrameDescription.Height;
            cantidadPixeles = new byte[iWidth * iHeight *
    ((PixelFormats.Bgr32.BitsPerPixel + 7) / 8)];
            this.imagen                    =                    new
            WriteableBitmap(colorFrameDescription.Width,
```

```
colorFrameDescription.Height, 96.0, 96.0, PixelFormats.Bgr32, null);
            this.Image.Source = this.imagen;
            this.miKinect.Open();
        }
```

La clase Kinect_FrameReady será el método encargado de direccionar a los frames procesados por el Kinect para posteriormente desplegarlos en el elemento Image, por el cual tiene que pasar un proceso de verificación de que estos datos hayan llegado de manera correcta, así como el uso de la variable que almacena los pixeles en un arreglo de datos.

```
    private void Kinect_FrameReady(object sender,
ColorFrameArrivedEventArgs e)
        {
            // Adquirir el ColorFrame
            using (ColorFrame colorFrame =
            e.FrameReference.AcquireFrame())
            {
                // Verificar contenido de datos
                if (colorFrame != null)
                {
                    if (colorFrame.RawColorImageFormat ==
    ColorImageFormat.Bgra)
                    {
                        colorFrame.CopyRawFrameDataToArray(cantidadP
                        ixeles);
                    }
                    else
                    {
    colorFrame.CopyConvertedFrameDataToArray(cantidadPixeles,
```

```
ColorImageFormat.Bgra);
                    }
                    // Llamar método que manipula los datos
                    this.usarCamara();
                }
            }
        }
```

El método usarCamara será el método encargado de implementar los frames obtenidos por el Kinect, dándole formato de tamaños para que esta se despliegue de manera óptima dentro de la ventana MainWindow.

```
private void usarCamara()
        {
            this.imagen.WritePixels(new Int32Rect(0, 0,
this.imagen.PixelWidth, this.imagen.PixelHeight), this.cantidadPixeles,
this.imagen.PixelWidth * sizeof(int), 0);
                    }
```

Por último, el método Window_Closing realizará que el Kinect se finalice de manera correcta, reduciendo el uso de los recursos para que se apague el Kinect para que deje de leer y adquirir datos.

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            if (this.colorFrameReader != null)
            {
                // ColorFrameReder is IDisposable
                this.colorFrameReader.Dispose();
                this.colorFrameReader = null;
```

```
}
if (this.miKinect != null)
{
        this.miKinect.Close();
        this.miKinect = null;
}
        }
```

Aviso legal

Leal Flores, Armandina Juana.
Desarrollo de aplicaciones inteligentes al movimiento del cuerpo humano / Armandina Juana Leal Flores, Eduardo Alan Torres Alejandro, Eduardo Luis Vázquez Nieto.

471 p. cm.

1. Software de aplicación—Desarrollo
I. Torres Alejandro, Eduardo Alan
II. Vázquez Nieto, Eduardo Luis

LC: QA76.76.A65 Dewey: 005.3

Producción editorial

Alejandra González Barranco. Dirección de la Editorial Digital.

Elizabeth López Corolla. Coordinación editorial.

Producción audiovisual

Carolina Ramírez García. Coordinación de diseño.

Martha Espinosa. Administración de proyecto.

María Isabel Zendejas Morales / Fernanda Mesta Pichardo. Diseño editorial.

eBook editado, diseñado y publicado por el Instituto Tecnológico y de Estudios Superiores de Monterrey.

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 | Monterrey, Nuevo León | México.

ISBN en trámite.

Primera edición: junio de 2018.